Rigorous development of reusable, domain-specific components, for complex applications*

I. Johnson[1], C. Snook[2], A Edmunds[2] & M. Butler[2]

[1]AT Engine Controls Ltd., Portsmouth, UK.
[2] University of Southampton, Southampton, UK.

Abstract. The reuse of reliable, domain-specific software components is a strategy commonly used in the avionics industry to develop safety critical airborne systems. One method of achieving reuse is to use domain specific languages that map closely onto abstractions in the problem domain. While this works well for control algorithms, it is less successful for some complex ancillary functions such as failure management. The characteristics of device failures are often difficult to predict resulting in late requirements changes. Hence a small semantic gap is especially desirable but difficult to achieve. Object-oriented design techniques include mechanisms, such as inheritance, that cater well for variations in behaviour. However, object-oriented notations such as the UML lack the precision, and rigor, needed for safety critical software. UML-B is a profile of the UML for formal modelling. In this paper we show how UML-B can be used to model failure management systems via progressive refinement, and indicate how this approach could utilise UML concepts to cope with high variability, while providing rigorous verification.

1 Introduction

Developers in the avionics industry are interested in the use of object-orientated technology (OOT) [1, 2] as a way to increase productivity. In particular, concepts, such as inheritance and design patterns, enable more flexible reuse of software. The emergence of the UML [3] as the de-facto standard modelling language for object-oriented design and analysis, and the subsequent development of supporting tools, has promoted the modelling and design of applications in the UML. Due to concerns over safety certification issues, OOT has not seen widespread use in avionics applications. One reason for this is that the controlling standards used in the industry, such as RTCA DO-178B [6] and its European equivalent, EUROCAE ED-12B [7], do not consider OOT. In order to address this, a new version of the standard, DO-178C/ED 12C, is planned. A significant contribution to this new standard will come from the findings of the Object Orientated Technology in Aviation (OOTiA) initiative [8], which was set up by the FAA and NASA to develop guidelines for the safe use of object-oriented technology in avionics software development.

* This research is being carried out as part of the EU funded research project: IST 511599 RODIN (Rigorous Open Development Environment for Complex Systems).

Application development based on a combination of UML and formal methods will improve safety and provide flexibility in the design of software for aviation certification. The combination of UML and formal methods at an abstract modelling stage will enable the reuse of reliable software components both at the specification and code levels. We indicate how we can exploit the reuse features of the UML and the reliability provided by formal methods. The development process will benefit from a reduction in the semantic gap by defining a vocabulary of entities that maps closely onto abstractions in the problem domain. UML class diagrams assist greatly with this, especially in complex application domains where the use of features such as inheritance caters for variation of subtypes. The use of formal methods to address the rigorous verification required for safety critical applications has been advocated before [5] but adoption within industry has been limited partly due to the need for industrial strength tool support. One formal method that has been developed for practical use in industry and enjoys good tool support is the B method [9].

1.1 B and B tools

The B method is based on a set theoretic approach and provides the ability to perform rigorous proof, thus ensuring a self-consistent specification. The B method's Abstract Machine Notation (AMN) is used to describe the state and behaviour. Under-specification in assignments or choices is possible via non-deterministic constructs, which must be resolved in later refinements prior to implementation. An invariant clause describes properties of the system that must hold at all times. The B verification tools [11] generate proof obligations (POs) and then attempt to automatically discharge (prove) them. Invariably there are a number of PO's that remain to be discharged semi-automatically using the interactive prover [10]. The user guides the proof by suggesting strategies and sub-goals. Discharging POs with the interactive prover often leads to a greater insight into the specification and inaccuracies can be identified and addressed at an early stage of development. Discharging proof obligations can be difficult and time consuming, but once complete the specification is known to be self-consistent. A model checker, ProB [13], that searches for deadlocks and invariant violations may be used as a preliminary verification of the specification before commencing proof. Refinements are related to the previous level of abstraction in such a way that a valid refinement always satisfies the abstract specification. Proof provides confidence that the refinement is not only self-consistent but also reflects the behaviour of the abstract specification it refines.
Event B [12] is a derivation of the original B Method that uses the notion of predicate guards that enable or disable events. The event driven approach of Event B begins with the abstraction of the observable events that 'may' occur in a system. The abstraction is refined in a number of steps, adding new events and state information at each iteration. The aim is to move towards a consistent model, with enough detail to fully describe the behaviour of the system. There are a number of additional requirements for the event driven approach, firstly, the added events of a refinement are not allowed to permanently take control of the system so that the events of the more abstract model are eventually enabled; secondly, a concrete event must not be enabled more often than its abstract counterpart; but the abstract event must not be

enabled more than the disjunction of the concrete event together with the new events. That is, the abstract event is replaced by a sequence of new events culminating in the refined event.
Validation of specifications is just as important as verification. ProB [13] can also be used to animate B machines. The list of currently enabled operations is displayed in a pane. The current state of variables is displayed in another pane. The user may choose sequences of enabled operations in order to explore the behaviour of the specification.

2 Failure management

A common functionality required of many systems is to manage failure of its inputs. This is particularly pertinent in aviation applications where lack of tolerance to failed system inputs could lead to loss of aircraft. The role of failure management in an embedded control system is shown in Fig.1.

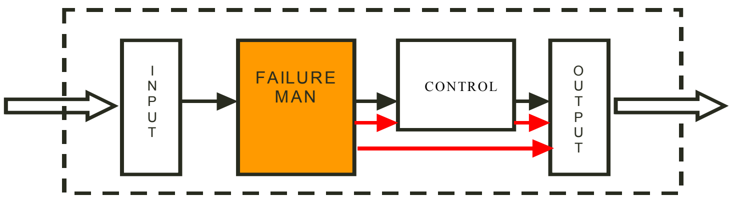

Fig. 1. Context diagram for failure management subsystem

The inputs are tested and, if good, are passed unaltered to the control subsystem; otherwise the failure of the input is managed. This may involve substituting values, and taking alternative actions. There are two aspects to the subsystem; detection and remedial action. Failure detection involves checking for input validity, including out of range checks, rate of change checks, and comparison with other conditions in the system. A failed condition must persist for a period of time before a failure is confirmed. If the invalid condition is not confirmed the input recovers and is used again. When setting the persistence conditions for confirmation of a failure, a balance must be sought between achieving a fast response to failures and over sensitivity to spurious interference. Once a failure is confirmed it is latched until power is reset. Remedial actions vary, depending on the input’s function and importance within the system, and the state of the system when the failure occurred. Temporary remedial actions, such as relying on the last good value, or suppressing control behaviour, may be taken while a failure is being confirmed. Once a failure is confirmed, more permanent actions are taken such as switching to an alternative source, altering or degrading the method of control, engaging a backup system or freezing the controlled equipment. Tables 1 and 2, show some typical failure management activities.

Table 1. Detection

Signal	High	Low	Rate	Conditions for test
ESa	120%	0%	100%/sec	Engine Stood
	120%	10%	100%/sec	Engine Starting
	120%	50%	100%/sec	Engine Running
ESb	120%	0%	100%/sec	Engine Stood
	120%	10%	100%/sec	Engine Starting
	120%	50%	100%/sec	Engine Running
ESa - ESb	5%	-5%	-	ESa or ESb >30%
ESa – Engine speed (main input) ESb – Engine speed (alternative input)				

Table 2. Remedial Actions

Signal	Procedure	code
ESa	Select ESb if available	ES1
	else Switch to backup system	
ESb	ESb not available	ES2
ESa/ESb diff	Use highest value sensor	ES3

Experience has shown that failure management systems can be difficult, and expensive, to develop and maintain due to their complexity and vulnerability to change. Changes often occur late in the development cycle, since requirements are redefined based on empirical performance under failure conditions. The semantic gap between control algorithm design notations and coding constructs has been addressed successfully by the development of domain specific languages. Unlike control algorithms failure management has no successful domain specific language. The nature of failure management is that different control actions and behaviour are required, dependent on the outcome of conditional logic for each of many inputs; this can result in complex overall behaviour. Failure management can become functionally complex due to the following: the rate of decay of an input depends on sensor device characteristics, the application of a test depends on engine and input conditions, a test may depend on the sampling rates of inputs, a test may vary according to outcomes of other tests, the detection of a failure may require hysteresis to avoid oscillation, the sequence of tests may depend on temporal interdependence of input sampling, remedial actions depend on the system state.

One approach, to improve flexibility, is to model a failure management subsystem using the UML; this will improve configurability and, if combined with formal methods to ensure consistency, will be particularly suited to the safety critical applications found in aviation. Modelling functional behaviour will provide the ontology to convey functional understanding and, through formal techniques, provide a way to map this to the code, reducing the semantic gap.

3 Overview of the UML-B profile and U2B translator

The UML-B [17] is a profile of UML that defines a formal modelling notation. It has a mapping to, and is therefore suitable for translation into, the B language. UML-B consists of class diagrams with attached statecharts, and an integrated constraint and action language based on the B AMN notation. The UML-B profile uses stereotypes to specialise the meaning of UML entities to enhance correspondence with B concepts. The profile also defines tagged values (UML-B clauses) that may be attached to UML entities to provide formal modelling details that have no counterpart in UML. Several styles of modelling are available within UML-B. Here we use its event systems mode, which corresponds with the Event B modelling paradigm. In event systems mode, UML operations represent spontaneous events. Since events are parameterless, operation parameters represent non-deterministic selection of local variables. UML-B provides a diagrammatic, formal modelling notation based on UML. It has a well defined semantics, as a direct result of its mapping to B. There are barriers to the acceptance of formal methods in industry. The popularity of the UML enables UML-B to overcome some of these barriers. Its familiar diagrammatic notations make specifications accessible to domain experts who may not be familiar with formal notations. UML-B hides B's infrastructure, it packages mathematical constraints and action specifications into small sections, each being set in the context of its owning UML entity. The U2B [18] translator converts UML-B models into B components (abstract machines and their refinements). Translation from UML-B into B enables verification and validation tools to be utilised.
In many respects B components resemble an encapsulation and modularisation mechanism suitable for representing classes. A component encapsulates variables that may only be modified by the operations of the component. However, to ensure compositionality of proof, B imposes restrictions on the way variables can be modified by other components (even via local operations). Translating classes into B components imposes corresponding restrictions on the relationships between classes. Therefore we translate a complete UML package (i.e. many classes and their relationships) into a single B component. This option allows unconstrained (non-hierarchical) class relationship structures to be modelled. Since the B language is not object-oriented, class instances must be modelled explicitly. Attributes and associations are translated into variables whose type is a function from the class instances to the attribute type or associated class. For example a class A with attribute x of type X would generate the following B:

```
SETS          A
VARIABLES            x
INVARIANT            x : A --> (X)
```

Operation behaviour may be represented textually in a notation based on B, as a state chart attached to the class, or as a simultaneous combination of both. Further details of UML-B are given in [17]. Examples of previous case studies using UML-B and U2B are given in [14,15,16 and 19].

4 UML-B model of failure management

As an example of using UML-B to develop failure management systems we show a simplified model and its verification. Our first abstract model captures the overall states of the system. in subsequent refinements we model the stages in confirming a failure, the mechanism for freezing the system and the relationship between individual inputs and the collective state of the system. In these early stages we leave many aspects of the system under specified, saying only, for example, that an input may be detected as an unconfirmed failure and then may either recover or become a confirmed failure; but saying nothing about how or why these choices are made. Despite this (non-deterministic) under-specification the model embodies important properties about the interaction of the states of inputs that we verify by proof. In practice, inputs have differing levels of sensitiveness and importance to the control system operation. However, to simplify the example we only consider inputs to which the controller is sensitive (i.e. freezes while a failure is unconfirmed) and can not continue to control without (i.e. hard fault).

4.1 Machine fman_a

This first abstract model of failure management considers the overall state of the system. It defines the three main states of the controller in response to input validity conditions, which are; a) normal operation, b) frozen while attempting to confirm a possible input error, and c) hardfaulted when an input error has been confirmed. Note that once the system has hard faulted no further events may occur (the model is intentionally deadlocked). The following state diagram is attached to a class utility (within the fman_a <<machine>> package) and hence represents a simple variable.

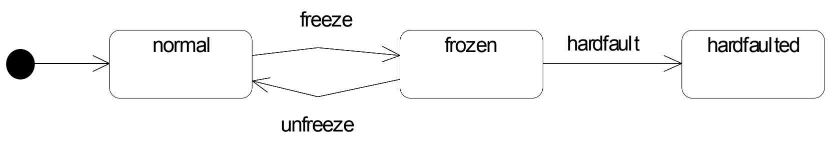

Fig. 2. Statechart diagram of the abstract machine

Since this level is a simple expression of the permitted transitions between the three states of the system, the only verification is that there are no other states. The B produced by U2B for this model is shown in the appendix.

4.2 Refinement fman_r1

In this refinement we recognise that the system state is actually an abstraction of the states of many instances of input failure management (Fig.4). Each input has three possible states; `ok`, `suspect`, and `confirmed`. Each input can have a `good` event (corresponding to a valid input value being detected) or a `bad` event (when an invalid value is detected). Some of these good and bad events (depending on the state of the full collection of inputs) refine the freeze, unfreeze and hardfault events from the abstract model. These are first_bad (the first input to enter the suspect state), last_good and confirm respectively. When an input has confirmed detection of an invalid value, a guard on each transition prevents further events from being enabled. This models the intentional deadlock in the abstract model. The refinement relation (Fig. 6.) specified in a REFINEMENT_RELATION clause attached to the package, fman_r1, gives the correspondence between the equivalent states of the two models. The system is normal when no inputs have detected invalid values (confirmed or suspect), frozen when at least one input has detected an invalid value but no inputs have been confirmed invalid, and hardfaulted when an input has detected and confirmed an invalid value. Note that we use a 'Petri' style interpretation of the state model (where each state is a variable whose value is the set of instances in that state) since this makes it easier to specify the collective state of the class in transition guards. Verification proves that the collective state of the inputs behaves in accordance with the overall system states; normal, frozen and hardfaulted. The B produced by U2B for this model is shown in the appendix.

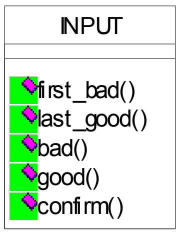

Fig. 3. Class diagram of the first refinement

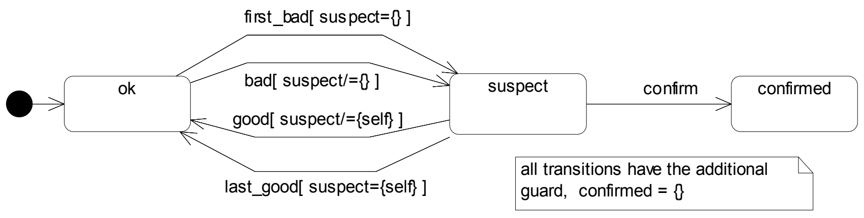

Fig. 4. Statechart diagram of the INPUT class

```
REFINEMENT_RELATION
((control_state=normal) <=> (ok=INPUT)) &
((control_state=frozen) <=> (suspect/={} & confirmed={})) &
((control_state=hardfaulted) <=> (confirmed/={}))
```

Fig. 5. Refinement relation between abstraction and first refinement.

Table 3. - Event refinement in first refinment

event in refinement r1	refines event in abstraction a
first_bad	freeze
bad	(new event)
first_good	unfreeze
good	(new event)
confirm	hardfault

4.3 Refinement fman_r2

In this refinement we introduce the idea that each input consists of several tests and each test is in a passed, failed or latched state. This is modelled as a class `TEST` which has a state model, and an association to its owning input. `INPUT` no longer has any events or a state model. Its state is derived from its associated collection of `TEST`s. The state of an input is `ok` when all its tests are `passed`, `confirmed` when one of its tests is `latched` and `suspect` otherwise.

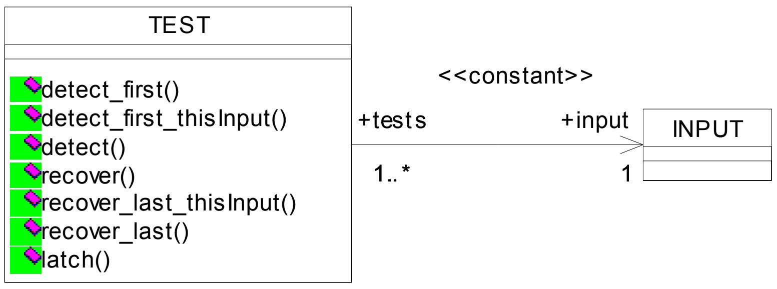

Fig. 6. Class diagram of the second refinement

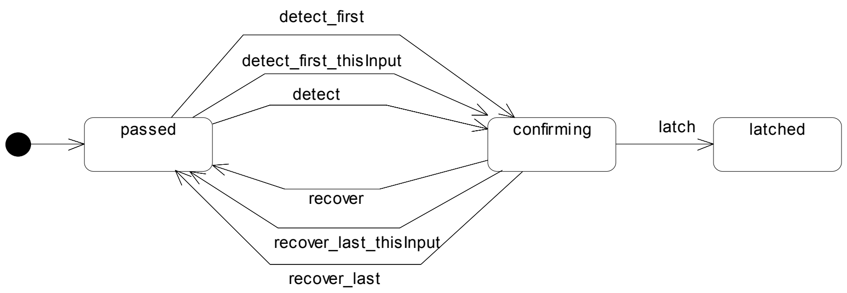

Fig. 7. state diagram of the TEST class

The guards for the transitions (events) are given in Fig 9. (Where `a|>s` restricts the association `a` to links whose targets belong to the set `s`. E.g. `(tests |> confirming)[{i}]` is the set of instances of `TEST` that are associated with the input, `i`, that are in the state `confirming`).

```
detect_first:              confirming={}
detect_first_thisInput:    confirming/={} &
                           (tests|>confirming)[{input}]={}
detect:                    (tests|>confirming)[{input}]/={}
recover_last:              confirming={self}
recover_last_thisInput:    confirming/={self} &
                           (tests|>confirming)[{input}]={self}
recover:                   (tests|>confirming)[{input}]/={self}
latch:                     latched={}
   (all events have the additional guard, latched={})
```

Fig. 8. Guards on events in second refinement

The refinement relation defines the set of inputs in each of the r1 level states based on the state of its collection of tests. (where `s<<|a` restricts the association a to links whose source do not belong to the set `s`).

```
REFINES fman_r1
REFINEMENT_RELATION
ok =         INPUT - ran(confirming\/latched <| input) &
suspect =    ran(confirming <| input) -ran(latched <| input) &
confirmed =  ran(latched <| input)
```

Fig. 9. Refinement relation between first and second refinement.

The events of the class `INPUT` are re-specified as events of the class `TEST` and in terms of the conditions of the collection of tests belonging to the input. Two new events, `detect` and `recover`, are added to model the transitions of subsequent tests detecting failures when another test on the same input has already done so. These new events had no effect in the previous level of refinement.

Table 4. Event refinement in second refinement

event in refinement r2	**event in refinement r1**	**event in abstraction a**
detect_first	first_bad	freeze
detect_first_thisInput	bad	-
detect	-	-
recover_last	last_good	unfreeze
recover_last_thisInput	good	-
recover	-	-
latch	confirm	hardfault

4.4 Adding further details to Test

In subsequent refinements we introduce further detail to the model in many stages that we summarise here. This includes events and a counter attribute for confirming (or recovery of) a test. We add a parameter, `pval`, and a corresponding `limit` for deciding when `detect`, `recover` and `latch` events occur for a particular test. Having verified the previous refinement stages we no longer need to distinguish separate events for the differing conditions when a test detects an invalid input. We therefore merge `detect`, `detect_first` and `detect_first_thisInput` using a three branch guarded choice (SELECT substitution). Subsequently we also merge in the confirmation counting events and latching event with further guarded branches of the same event. In this way the correspondence of actions taken under different conditions is verified to represent the abstract event model before being merged into a conditional single event, as the model is refined towards an implementation. The B produced by U2B for the merged operation, `test`, is shown in the appendix.

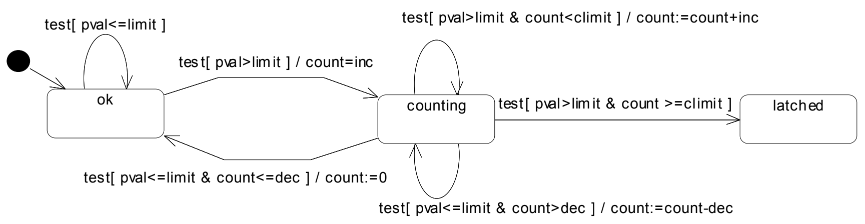
test[pval<=limit]
ok
test[pval>limit] / count=inc
test[pval>limit & count<climit] / count:=count+inc
counting
test[pval>limit & count >=climit]
latched
test[pval<=limit & count<=dec] / count:=0
test[pval<=limit & count>dec] / count:=count-dec

Fig. 10. Statechart diagram of refinement with counting and merged events

4.5 Defining subtypes of Test

Having rigorously developed a generic test class this can be specialised in further refinements to perform several sub-types of test, such as magnitude tests, rate of change tests and difference comparison tests. There are now three subclasses of `TEST`. `TEST` is an abstract class having no instances other than those that belong to one of its subclasses. The inheritance is modelled in the B produced by U2B as disjoint subsets of `TEST`. `MAG` represents a magnitude test that reuses the behaviour of its superclass. `DIFF` is a test that compares the associated input with another input (represented by association, `comp`). It overrides the test event by 'calling' the superclass' `test` passing it the difference in the values of the 2 inputs. To achieve this we add a `value` attribute to the class `INPUT` with an `update` event to change its value. `RATE` is a rate of change test that compares the current value of its associated input with the last value it tested. It has an additional attribute to record the `last` value and overrides the `test` operation as shown.

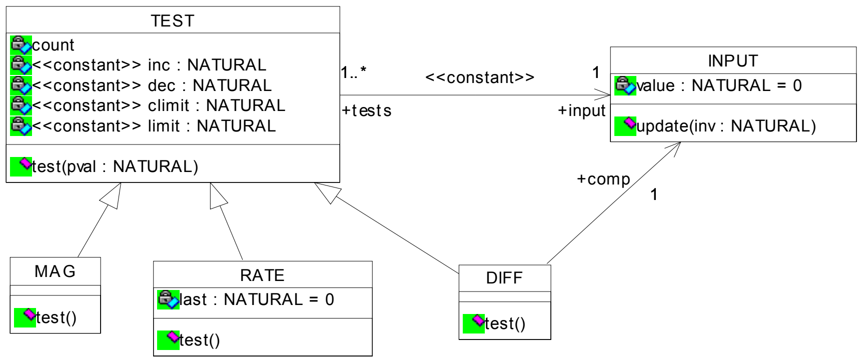

Fig. 11. Class diagram for refinement with subclasses of test

The overriding of the `test` event is shown below. This is modelled in the corresponding B by collating the three specialised test events into a single event with three guarded branches. The guard for each branch tests the instance for membership of a subclass.

```
MAG:test   =  super.test(input.value)
DIFF:test  =  super.test(abs(input.value-comp.value))
RATE:test  =  last:=input.value ||
                       super.test(abs(input.value-last))

   where abs(i,j)= max(i-j,j-i)
```

Fig. 12. Overriding of the event, test

5 Discussion

To verify an event refinement it is required to show that any new events lead (eventually) to the enabling of one of the original events. When we attempted to verify our first refinement we found that this wasn't possible. Our initial attempt at a model as presented above doesn't ensure that if the system enters the freeze state it will ever be able to leave it, (a requirement imposed by Event B). The problem is that, although an individual input must leave the unconfirmed state after entering it, another input might also enter the unconfirmed state before the original one leaves it. In this way, inputs could take it in turns to be unconfirmed leaving the system permanently frozen. This would be an undesirable outcome since the frozen state is intended to be a temporary stage before confirming or recovering. A limiting mechanism that forced the system into the hardfaulted state after a certain number of unconfirmed inputs would prevent this and enable us to prove the event refinement. Even with this grossly oversimplified example the event modelling approach increased our understanding of the problem at the earliest stages of development.

6 Future work

This paper describes our first attempt using an event based modelling approach, with UML-B, to improve the reusability and portability of failure management systems. We are in the preliminary stages of a three-year research programme that aims to investigate this area. Within the project we will develop UML-B, to better support the use of UML features such as inheritance, and to provide modelling mechanisms that aid the refinement and transformation of UML-B models. We plan to re-implement the U2B translator within eclipse in order to achieve better integration with the B validation and verification tools, which will also be ported to eclipse. We will test and develop the ideas presented in this paper, on a larger scale example, in the following stages. Model a small but realistic, imaginary failure management application. The model may consist of several levels of refinement but be platform independent. We will then validate and verify the model via translation to B. Following validation and verification; we will investigate methods for abstracting away from the specific example to obtain a generic UML-B model that is application independent. New ideas for the development, using B# [20], may be used in this stage. Further development of UML-B and U2B may be needed to support this kind of model. Finally, we will investigate mechanisms for model transformation to obtain application and domain specific models from the generic model.

7 Conclusions

In this paper we have illustrated an approach to rigorous development of critical complex systems, such as failure management, in a manner that results in a high degree of re-usable verified components. The approach provides rigorous consistency verification through a sequence of refinement steps starting from a very abstract expression of overall system behaviour. This process of refinement can be continued through requirements development, design and into implementation. The process involves a, UML based, formal modelling notation, UML-B and utilises the tools available for the formal notation B. The refinement process inherently provides a high degree of reuse of verified specifications due to the deferment of specific application details. The genericity features of the UML may be used to provide re-usable common components within each refinement level. In this way we hope to provide a library of component classes that have a flexible but simple mapping with application domain concepts. The hierarchy of class models can then be instantiated with an object model for different applications thus achieving a specification and implementation language with small semantic gap that is suitable for the target complex problem domain; in this case, failure management systems. This approach could similarly be used for other complex systems problems.

References

1. Bertrand Meyer, *Object-Oriented Software Construction*, Prentice Hall PTR, Upper Saddle River, NJ 1997.
2. Grady Booch, *Object Oriented Analysis and Design with Applications* , The Benjamin/Cummings Publishing Company, Inc., Redwood City, CA 1994.
3. Object Management Group, *OMG Unified Modeling Language Specification*, version 1.3, June 1999, from http://www.omg.org/technology/documents/modeling_spec_catalog.htm.
4. Jishnu Mukerji, Joaquin Miller, *MDA guide version 1.0*, available from Object Management Group website http://www.omg.org.
5. Ministry of Defence (1997) *Def Stan 00-55: Requirements for safety related software in defence equipment, Issue 2.* http://www.dstan.mod.uk/data/00/055/02000200.pdf
6. Radio Technical Commission for Aeronautics, *RTCA DO 178B -Software considerations in Airborne Systems and Equipment Certification,* from http://www.rtca.org.
7. Eurocae ED12B, *Software considerations in Airborne Systems and Equipment Certification*, from http://www.eurocae.org.
8. FAA/NASA, *OOTiA - Object Orientated Technology in Aviation Program*, from http://shemesh.larc.nasa.gov/foot/.
9. J-R Abrial, *The B-Method*, Cambridge University Press, 1996.
10. J-R Abrial_ and D Cansell, *Click'n Prove: Interactive Proofs Within Set Theory,* from http://www.loria.fr/~cansell/cnp.html.
11. B4free is a set of tools for the development of B models from http://www.b4free.com.
12. J-R Abrial, *Event Driven Construction*, 1999, from http://www.atelierb.com/documents.htm
13. M.Butler, and M. Leuschel, *ProB: A Model-Checker for B,* Proceedings of FME 2003: Formal Methods - LNCS 2805, from http://www.ecs.soton.ac.uk/~mal/systems/prob.html
14. C. Snook, K. Sandstrom, *Using UML-B and U2B for formal refinement of digital components*, Proceedings of Forum on specification & design languages, Frankfurt, 2003.
15. C. Snook and M. Butler, *Using a Graphical Design Tool for Formal Specification,* Proceedings of the 13th Annual Workshop of the Psychology of Programming Interest Group (PPIG).
16. M. Butler, C. Snook, *Verifying Dynamic Properties of UML Models by Translation to the B Language and Toolkit*, Proceedings of UML 2000 Workshop, Dynamic Behaviour in UML Models: Semantic Questions.
17. C. Snook, I. Oliver and M. Butler, The UML-B profile for formal systems modelling in UML, In *UML-B Specification for Proven Embedded Systems Design*, Springer (In press 2004)
18. C. Snook, and M. Butler, U2B - A tool for translating UML-B models into B, *In UML-B Specification for Proven Embedded Systems Design*, Springer (In press 2004)
19. Mermet, J. (ed.) *UML-B Specification for Proven Embedded Systems Design*, Springer (In press 2004)
20. J-R Abrial, B#: Toward a synthesis between Z and B, In D.Bert, J.Bowen, S.King, M.Walden, editors, *ZB 2003: Formal Specification and Development in Z and B. Third International Conference of B and Z Users,* Lecture Notes in Computer Science, Vol.2651, Springer, pp.168-178.

Appendix - B produced by U2B

B machine for first abstract level

```
MACHINE     fman_a
SETS       CONTROL_STATE={normal,frozen,hardfaulted}
DEFINITIONS
   type_invariant == ( control_state : CONTROL_STATE ) ;
   invariant == (type_invariant)
VARIABLES   control_state
INVARIANT   invariant
INITIALISATION
   control_state :(invariant & control_state = normal )
OPERATIONS /*EVENTS*/
   hardfault =
    SELECT control_state=frozen THEN
     control_state:=hardfaulted
    END ;
   freeze =
    SELECT control_state=normal THEN
     control_state:=frozen
    END ;
   unfreeze =
    SELECT control_state=frozen  THEN
     control_state:=normal
    END
END
```

B refinement for the first refinement:

```
REFINEMENT  fman_r1
REFINES     fman_a
SETS       INPUT
DEFINITIONS
   tests == input~ ;
   type_invariant == (
    ok : POW(INPUT) &
    suspect : POW(INPUT) &
    confirmed : POW(INPUT) ) ;
   INPUT_invariant == (
    ok /\ suspect={} & ok /\ confirmed={} &
    suspect /\ confirmed={} &
    ok \/ suspect \/ confirmed = INPUT ) ;

   invariant == (type_invariant & INPUT_invariant) ;
   refinement_relation == (
    ((control_state=normal) <=> (ok=INPUT)) &
    ((control_state=frozen) <=> (suspect/={} &
                   confirmed={})) &
    ((control_state=hardfaulted) <=> (confirmed/={})) )
VARIABLES  ok, suspect, confirmed
INVARIANT  invariant & refinement_relation
INITIALISATION
   ok, suspect, confirmed :(invariant &
        ok=INPUT & suspect={} & confirmed={} )
OPERATIONS /*EVENTS*/
   first_bad =
    ANY thisINPUT WHERE thisINPUT:INPUT THEN
     SELECT confirmed={} THEN
```

```
        SELECT thisINPUT : ok & suspect={} THEN
           ok:=ok-{thisINPUT} ||
         suspect:=suspect\/{thisINPUT}
        END
      END
   END ;
etc.
```

B for the operation test:

```
ANY thisTEST,pval WHERE thisTEST:TEST & pval:NATURAL
THEN
   SELECT thisTEST : ok & pval>limit(thisTEST)
   THEN   ok:=ok-{thisTEST} ||
       counting:=counting\/{thisTEST} ||
       count(thisTEST)=inc(thisTEST)
   WHEN   thisTEST : ok & pval<=limit(thisTEST)
   THEN   skip
   WHEN   thisTEST : counting & pval>limit(thisTEST) &
       count(thisTEST)<climit(thisTEST)
   THEN   count(thisTEST):=count(thisTEST)+inc(thisTEST)
   WHEN   thisTEST : counting & pval>limit(thisTEST) &
       count(thisTEST) >=climit(thisTEST)
   THEN   counting:=counting-{thisTEST} ||
       latched:=latched\/{thisTEST}
   WHEN   thisTEST : counting & pval<=limit(thisTEST) &
       count(thisTEST)<=dec(thisTEST)
   THEN   counting:=counting-{thisTEST} ||
       ok:=ok\/{thisTEST} || count(thisTEST):=0
   WHEN   thisTEST : counting & pval<=limit(thisTEST) &
       count(thisTEST)>dec(thisTEST)
   THEN   count(thisTEST):=count(thisTEST)- dec(thisTEST)
   END
END
```