Towards a method for rigorous development of generic requirements patterns

Colin Snook[1], Michael Poppleton[1], and Ian Johnson[2]

[1] School of Electronics and Computer Science,
University of Southampton, Highfield,
Southampton SO17 1BJ, UK,
cfs,mrp@ecs.soton.ac.uk
[2] AT Engine Controls, Portsmouth
IJohnson@atenginecontrols.com

Abstract. We present work in progress[3] on a method for the engineering, validation and verification of generic requirements using domain engineering and formal methods. The need to develop a generic requirement set for subsequent system instantiation is complicated by the addition of the high levels of verification demanded by safety-critical domains such as avionics. Our chosen application domain is the failure detection and management function for engine control systems: here generic requirements drive a software product line of target systems.
A pilot formal specification and design exercise is undertaken on a small (two-sensor) system element. This exercise has a number of aims: to support the domain analysis, to gain a view of appropriate design abstractions, for a B novice to gain experience in the B method and tools, and to evaluate the usability and utility of that method. We also present a prototype method for the production and verification of a generic requirement set in our UML-based formal notation, UML-B, and tooling developed in support. The formal verification both of the structural generic requirement set, and of a particular application, is achieved via translation to the formal specification language, B, using our U2B and ProB tools.

1 Introduction

The need for generic approaches to support reuse in systems engineering is well known; in the avionics industry, for example, [16, 11] describe the reuse of generic sets of requirements in engine control and flight control systems. The need for reuse arises in many contexts, such as in system evolution, adaptation, or component-based construction. In this paper we are concerned with formal, generic requirements engineering to address the need for software product lines in the failure management domain in avionics.

A *software product line* (SPL) is a collection of variant implementations of a generic software requirement specification, to meet a variety of platform, environmental, functional, or other requirements. In avionics, the generic requirement specification for an

[3] This work is part of the EU funded research project IST 511599 - RODIN (Rigorous Open Development Environment for Complex Systems).

engine control system is implemented in a different variant in each manufacturer airframe; [Op.Cit.] describe SPL solutions. The notion of software product line engineering became well established [18], after Parnas' prescient proposal [22] in the 70's.

Domain analysis and object oriented frameworks are among numerous vehicles proposed to support product line development. In Domain-Specific Software Architecture [29] for example, the production of a set of generic, domain-specific requirements through domain engineering is followed by its successive refinement, in a series of system engineering cycles, into specific product instance requirements. On the other hand [12] describes the Object-Oriented Framework as "a reusable, semi-complete application that can be specialized to produce custom applications". Here the domain engineering produces an object-oriented model that must be instantiated, in some systematic way, for each specific product required. In this work we combine object-oriented and formal techniques and tools in the domain analysis and engineering of generic requirements.

It is widely recognized that formal methods (FM) technology makes a strong contribution to the verification required for safety-critical systems [19]. It is further recognized that FM will need to be integrated [3] in as "black-box" as possible a manner in order to achieve serious industry penetration. The B method of J.-R. Abrial [1, 23] is a formal method with good tool support [2, 9] and a good industrial track record, e.g. [10]. At Southampton, we have for some years been developing an approach of integrating formal specification and verification in B, with the UML [8]. The UML-B [26] is a specialisation of UML that defines a formal modelling notation combining UML and B. It is supported by the U2B tool [24], which translates UML-B models into B, for subsequent formal verification. This verification includes model-checking with the ProB model-checker [17] for B. These tools have all been developed at Southampton, and continue to be extended in current work.

1.1 Failure detection and management for engine control

A common functionality required of many systems is to detect and manage the failure of its inputs. This is particularly pertinent in aviation applications where lack of tolerance to failed system inputs could have severe consequences. The failure manager filters inputs from the controlled system, providing the best information possible and determining whether a transducer or system component has failed or not.

Inputs may be tested for magnitude, rate of change and consistency with other inputs. When a failure is detected it is managed in order to maintain a usable set of input values for the control subsystem and provide 'graceful degradation'. To prevent overreaction to isolated transient values, a failed condition must be confirmed as persistent before irreversible action is taken. Failure detection and management (FDM) in engine control systems is a demanding application area, see e.g. [7], giving rise to far more than a simple parameterizable product line situation.

Our approach contributes to the failure detection and management domain by proposing a method for the engineering, validation and verification of generic requirements for product-line engineering purposes. The approach exploits genericity both within as well as between target system variants. Although product-line engineering has been applied in engine and flight control systems [16, 11], we are not aware of any such work in the

FDM domain. We define generic classes of failure-detection test for sensors and variables in the system environment, such as rate-of-change, limit, and multiple-redundant-sensor, which are simply instantiated by parameter. Multiple instances of these classes occur in any given system. Failure confirmation is then a generic abstraction over these test classes: it constitutes a configurable process of execution of specified tests over a number of system cycles, that will determine whether a failure of the component under test has occurred. Our approach is focussed on the genericity of this highly variable process.

1.2 Fault Tolerance

This application domain (and our approach to it) includes fault tolerant design in two senses: tolerance to faults in the environment, and in the control system itself. The FDM application is precisely about maximizing tolerance to faults in the sensed engine and airframe environment. The control system (including the FDM function) is supported by a backup control system in a dynamically redundant design. This backup system with dissimilar hardware/software design, with a reduced-functionality sensing fit can be switched in by a watchdog mechanism if the main system has failed.

In the narrower (and more usual) sense, we will be examining various schemes for designing fault tolerance into the FDM software subsystem. Work to date has specified and validated a generic requirements specification for FDM. As we apply refinement techniques and technology to construct the design, we will consider various relevant approaches, such as driving the specification of a control system from environmental requirements [13], or the use of fault-tolerant patterns for B specifications [14] and their refinements [15].

1.3 The paper

We present the results of a pilot formal specification and design exercise. This was undertaken on a small (two-sensor) element of a typical system from our partner ATEC's domain. This exercise was intended to support the domain analysis, and to gain a view of appropriate design abstractions for the full exercise of developing and validating the generic requirements. Furthermore, since the ATEC engineer (and co-author of this paper) was a novice to the B method, the exercise would also enable him to gain experience in the B method and tools, and to evaluate the usability and utility of that method to an engineer in the target domain.

The pilot exercise took place in the context of our development of a prototype method for the specification and verification of generic requirements sets for systems of this type. The method is briefly presented here; for a fuller discussion see [27, 28]. We also report briefly on tooling subsequently developed to support the method.

The paper proceeds as follows. The pilot study and its evaluation is presented in section 2. Sections 3 and 4 review our prototype method and our domain analysis activity. Section 5 describes the domain engineering of the generic model, followed by the engineering of a sample application instance in section 6. A brief taxonomy of validation/verification problems is then presented in section 7. Section 8 concludes.

2 Pilot study

To explore the FDM domain in more detail we carried out a pilot study that modeled and verified in B a very small example consisting of two sensors used to measure one environment variable. The aim of the pilot study was to gain a better understanding of the stages and processes involved, before embarking on the search for generic re-usable modeling abstractions. The pilot study was carried out by our co-author, an engineer with our industrial partner company, who provided the domain expertise. Since he was a novice to formal specification and the B method, the pilot study also provided insight into the reaction we might expect from industrial users that adopt our method. Thus a secondary aim was to evaluate model development using B and the existing B tools from the point of view of adoption in an industrial setting.

The model was intended to be both analytical and specificational, i.e. the aim was to explore the requirements as well as develop a specification. The dual redundant engine speed (ES) functionality was selected as it includes behaviour representative of other control inputs and includes interaction between sensors. The ES value is normally taken from the ESa input but if this is not healthy, the ESb input is used instead. If both inputs are unhealthy the ES signal is not updated. The ES failure management requirements include input magnitude tests, comparative (difference) tests between two given inputs, and confirmation mechanisms that select appropriate output values and failure flag settings.

2.1 Approach

The engineer initially explored the requirements by developing the model as a series of specifications adding functionality and validating new behaviour in stages using the ProB animation tool. The stages were 'idealisations' rather than true abstractions because they omitted to allow for the effects of events added in later stages. Hence they give only an approximate indication of behaviour corresponding to that obtained by ignoring some of the details to be added later. (Analogous to idealisation in physics, such as ideal gases, where some phenomena that affect the system are simply ignored even if they effect the the variables of the model). This approach, although less rigorous than refinement, was chosen as it allowed a quick exploration of the requirements by avoiding the difficult process of finding useful abstractions and proving their refinements. The ProB model checker was used to check internal consistency within the requirements at each stage.

The next stage was to revise the specifications to achieve refinement consistency. The ProB model checker [17] and the prover tool Click'N'Prove/B4free [9] were used to verify the refinements. A final stage to refine the model towards code implementation is ongoing.

To obtain a refinement chain, sufficient abstract detail was added to the idealised specification to satisfy the proof obligations. In practice this meant that, when adding new events, the abstract versions in previous levels were not `skip`, but non-deterministic alterations to variables at that level. Informally, this is a generalisation of event refinement where the effect of new events is, in a loose sense, not significant to the old variables. It is interesting to note that a proven refinement chain can be constructed in this

manner with little experience, using the proof obligations as a guide to find the weakest appropriate abstraction. However, the constructed refinement chain is only useful if it is used in some way to validate the refinements. Hence the abstract level was reconsidered to ensure that the effect of the new events on the old variables is acceptable and does not invalidate the model.

Hence, the approach utilises idealisation (which seems to be easier than abstraction), superimposing detail (which is easy because it is not required to refine anything) and 'upwards' addition of suitable detail to obtain refinement (which is fairly easy because it is led by the proof tools). We present a sanitized version of the final specifications with notes to explain the initial intentions from which they derived.

2.2 Abstract Model

The most abstract specification is shown below in machine `Engine_speed_0`. The intention of this stage was to represent the functionality as a 'black box' that allows given output combinations as a result of under-specified environment changes. The only constraint is that the outputs have some interlocking relationships. For example, if a flag indicates that the ESa signal healthy action was taken, then the output should be equal to the ESa sensor value until the environment changes the ESa sensor value. The idealisation is that the output is only the same as a snapshot reading of the sensor value. In later refinements we would like to introduce the read value of the sensor and specify that the output is equal to this.

Two sensors, `esa` and `esb` are defined. Each sensor has a change event which alters the sensor value non-deterministically. Each sensor has a flag which is set (by the outcome operations) to record when it has failed. There are four alternative outcome events corresponding to the health or failure combinations of the two sensors (e.g. `hh` is the outcome taken when both `esa` and `esb` are healthy). These outcomes take the appropriate failure action by setting the output to either `esa`, `esb` or its previous value. In this stage, the selection of the appropriate outcome, from those available, is left to chance and not related to the input values in any way. Initially all 4 outcomes are available but, since failed sensors are not allowed to recover, `hh` will be disabled thereafter if either of the sensors is subjected to a failing outcome. This 'latching out' of a possible outcome is represented by the latch flags `esalatch` and `esblatch`. Eventually only the `ff` outcome will be available. An invariant specifies the alternative properties that should be achieved on the output in terms of the sensor values (until the environment changes the sensor values again).

```
MACHINE Engine_speed_0
...
INVARIANT              ... &
       (newVal = TRUE or
       (esalatch = UNSET & output = esavalue) or
       (esblatch = UNSET & output = esbvalue) or
       output = previous)


OPERATIONS /*EVENTS*/
esaChange =
BEGIN
       esavalue :: NATURAL || newVal := TRUE
END;
```

```
hh =
        SELECT
             esalatch = UNSET & esblatch = UNSET
        THEN
             output , previous := esavalue, esavalue ||
             newVal := FALSE
        END;
...
ff =
        BEGIN
             output := previous ||
             esalatch, esblatch := SET, SET ||
             newVal := FALSE
        END
```

2.3 First Refinement

In the first refinement of this abstract model more detail is added by introducing the events, `esavalidate` and `esbvalidate`. These events are responsible for determining which of the alternative outcomes will be taken. They do this by setting the `esaresult` and `esbresult` flags, which are now used in the guards of the outcome events. The guards of the outcome events are also strengthed to ensure that both `esa` and `esb` sensors have been newly validated before the outcome is taken. The means by which `validate` decides which outcome to activate remains underspecified. This is a valid refinement because the outcome event guards are strengthened and new data (result) is superimposed. However, the specification of validation is idealised because it omits the difference test which, in the next refinement, is added as a new event that also modifies the variable `esbresult`. To satisfy the prover and achieve the next refinement we later revisit this stage to de-idealise it by adding an abstract version of the `difftest` event that non-deterministically alters `esaresult` and `esbresult`.

This refinement stage led to consideration of what types and level of failure detection should be addressed and whether they should indicate different actions. The ordering of the sequence of events from failure detections to actions was considered. The refinement raised the issue of test scheduling and input sampling and whether a series of tests could easily be accommodated in the design by an appropriate sequencing mechanism

```
REFINEMENT Engine_speed_1
REFINES Engine_speed_0
...
;
esavalidate =
        SELECT
             esavalidated=FALSE
        THEN
             esaresult :: PASS_FAIL ||
             esavalidated := TRUE
        END
;
hh =
        SELECT
             esavalidated = TRUE & esbvalidated = TRUE &
             esaresult = PASS & esbresult = PASS &
             esalatch = UNSET & esblatch = UNSET &
        THEN
             esavalidated, esbvalidated := FALSE, FALSE ||
             output, previous := esavalue, esavalue ||
             newVal := FALSE
        END;
```

```
hf =
       SELECT
            esavalidated = TRUE & esbvalidated = TRUE &
            esaresult = PASS & esbresult = FAIL &
            esalatch = UNSET
       THEN
            esavalidated, esbvalidated := FALSE, FALSE ||
            output, previous := esavalue, esavalue ||
            esblatch := SET ||
            newVal := FALSE
       END;
...
```

2.4 Second and third Refinements

In the second refinement, a new event that can also affect the selected outcome, is added. This event, `difftest`, represents a comparison between the two sensor values. It must happen after both `esa` and `esb` have been validated and before an outcome event occurs. Note that it can only change sensor results from `pass` to `fail` not vice versa. Again the mechanism by which it decides this is left under-specified.

```
difftest =
       SELECT
            esavalidated = TRUE & esbvalidated = TRUE &
            esdiffvalidated = FALSE
       THEN
            IF   esaresult = PASS & esbresult = PASS
            THEN esbresult :: PASS_FAIL
            END ||
            esdiffvalidated := TRUE
       END;
```

Since this stage added a new event that alters a variable (`esbresult`) introduced in the previous refinement stage, an abstract version of this event must be added to the first refinement to allow the second refinement to be a refinement of the first. The most abstract version of the event would be a simple non-deterministic assignment of the variable to any value from its type (`esbresult :: PASS_FAIL`). However, although this would ensure the refinement, it would be a pointless exercise since the previous level would no longer describe a desired behaviour. That is, the `difftest` event could unlatch some failure results which is one of the main features that were embodied in the abstract level. A slightly more constrained abstract version is obtained by also retaining any conjuncts from guards or conditions that are based on variables in the previous level. This restricts the effect of the event in a way that corresponds with the refinement. Hence it stands a better chance of being an acceptable specification for the refinement. Its consistency and validity can then be examined using the ProB model checker and animator. In our case, this method produced abstract specifications that were consistent with the existing invariants and valid but we are not convinced that this will always be the case. It may be necessary to reconsider the way the feature is introduced if the abstract model does not behave as desired or violates the invariant.

```
difftest =
       SELECT
            esavalidated = TRUE & esbvalidated = TRUE
       THEN
            IF   esaresult = PASS & esbresult = PASS
            THEN esbresult :: PASS_FAIL
```

```
        END
    END;
```

Similarly, the previous level abstract model must have an even more abstract version of the new `difftest` event added to it. This is obtained by a similar process, retaining only the guards, conditions and non-deterministic versions of assignments that utilise variables in the abstract specification.

```
difftest =
    IF esaresult = PASS & esbresult = PASS
    THEN esbresult :: PASS_FAIL
    END;
```

In the third refinement, the details of how `validate` and `difftest` set result (and hence select an outcome) is provided. This entails comparing the sensor value against fixed limits and against each other. A confirmation counter mechanism was also introduced to model the requirement to not be oversensitive to sensor noise. This entailed adding a third, intermediate state `FAILING` to the possible values of result with corresponding new outcome events. The refinement was proven by adding a gluing invariant to match the states with the abstract version.

2.5 Evaluation

The pilot study was carried out and proven in B and the refinement chain was fully proven by automatic proof. Since choosing useful abstractions and appropriate refinements is difficult, requiring considerable experience and understanding of the refinement process, it surprised us that the novice engineer was able to achieve this, even for the simple example. To complicate matters, an event style of B (used in the Rodin project) was used but proof had to be achieved using the existing proof tools which are not event B based. For example, if a new event is added at a refinement, an abstract version of the event, with body `skip` (i.e. do nothing) must be added to all previous levels. The prover was able to automatically prove all of the proof obligations when the specifications were correct. This meant that the interactive prover was only used to identify corrections to the specifications.

The ease of proof was due to two factors. Firstly, only simple data types were used. We tried rewriting the specifications using a set of sensors with two elements `esa` and `esb` and relations from this set for the sensor data. With this data representation, the prover was no longer able to complete automatically. The almost exclusive use of superposition refinement (i.e. lack of data refinement) also probably assisted the prover.

The engineer recognised that it was difficult to find early abstract models that allowed for future refinements. The modeller's perception of what is important may change through experience with the model and understanding of the domain. The process is therefore iterative in nature. The novice engineer found the animation facility of the ProB tool particularly useful to quickly validate and explore model behaviour. He recognised that in larger scale problems the use of invariant checking with the tools will be an invaluable aid to verification and validation where it may be quite onerous to exercise the equivalent assurance using other methods. However the effectiveness of

invariants in models relies on how well they can be created and it was recognised that weak or incorrect invariants can be generated by lack of experience, which may be a hindrance to development and verification.

The syntax of the B notation did not present significant difficulties in this development as functionality could be expressed using simple constructs. Most proof obligations were discharged automatically. Where they were not, the proof goals were used to identify where the specifications need to be corrected or enhanced to achieve automatic proof.

The pilot study provided a better understanding of several issues in the FDM domain. In particular, a better understanding of the reaction between sensor values, tests and outcomes was gained.

3 Methodology

The process for obtaining a generic model of requirements is illustrated in Fig. 1. The first stage is an informal domain analysis which is based on prior experience of developing products for the application domain of failure detection and management in engine control. A taxonomy of generic requirements found in the application domain is developed. For example, class INP includes generic requirements INP2, INP5 and their sample instances below:

INP2	The subsystem input variables represent either sensor values or other subsystem variables.
INP5	Some "other subsystem" input variables represent the controller state.

Ref	**Name**	**Type**	**Range**	**Res**	**Description**	**Freq**
INP2.5	ET5	digitised	-200 to 2000F	0.1F	Engine Temp. sensor 5	24
INP2.10	ESa	digitised	0-200 %	0.01	Engine Speed (main)	24
INP5.1	CYCLE_NO	digital	1..16	1	Execution cycle counter	24

The instance requirements in each generic requirement class are thus expressed as data in tabular form. Thus a *first-cut generic entity-relationship model* can be constructed by relating these generic requirement entities, or classes. This generic model is represented as a UML-B class diagram, and a corresponding B specification is generated.

The identification of a useful generic model is a difficult process warranting further exploration. This is done in the domain engineering stage where a more rigorous examination of the first-cut model is undertaken, using UML-B, U2B and ProB. The model is animated by creating typical instances of its generic requirement classes, to test when it is and is not consistent. This stage is model validation by animation, using the ProB and U2B tools, to show that it is capable of holding the kind of information that is found in the application domain. During this stage the relationships between the classes are likely to be adjusted as a better understanding of the domain is developed. This stage results in a *validated generic model* of requirements that can be instantiated for each new application.

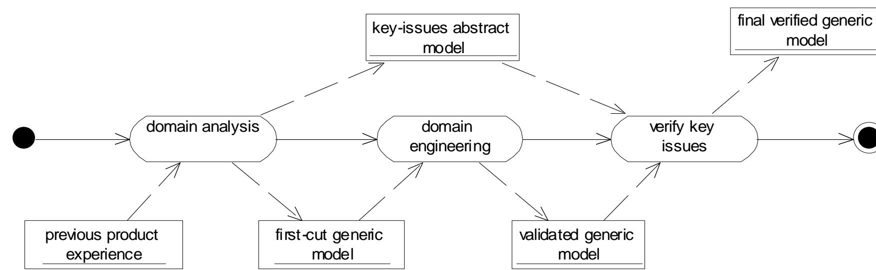

Fig. 1. Process for obtaining the generic model

For each new application instance, the generic requirement classes are instantiated from product instance data, producing an instance model. The relationship between the generic and the instance model is analogous to that between a class and and object model in UML. Instantiation is done by our prototype *Requirements Manager* tool, which reads instance requirement data from a database (see sec. 6), and uses that data to instantiate the generic UML-B model. The ProB model checker is then used to automatically verify that the instantiated application is consistent with the relationship constraints embodied in the generic model. This stage, producing a *consistent instance model*, shows that the requirements are a consistent set of requirements for the domain. It does not, however, show that they are the right set of requirements that will give the desired system behaviour.

Our aim in future work, therefore, is to add dynamic features to the instantiated model in the form of variables and operations that model the behaviour of the entities in the domain and to animate this behaviour so that the instantiated requirements can be validated. The ultimate goal is to specify this behaviour in the generic model in order to maximize reuse at the instantiation stage.

During the domain analysis phase we found that considering the rationale for requirements revealed key issues, which are properties that an instantiated model should possess. Key issues are higher level requirements that could be expressed at a more abstract level from which the generic model is a refinement. The generic model could then be verified to satisfy the key issue properties by proof or model checking. This matter is considered in [25] which gives an example of refinement of UML-B models in the failure management domain.

The final stage is to validate a specific, instantiated configuration. This would be done by providing actual values to generic behaviours when the generic model is instantiated. The resulting specific model could then be animated to validate its behaviour.

Finally, we recognize the need for tools to support uploading of bulk system instance definition data, as well as the efficient and user-friendly validation/ debugging of said data. The Requirements Manager prototype provides database storage and some validation; ProB could easily be enhanced to provide, for example, data counterexamples explaining invariant violations.

4 Domain Analysis

Domain analysis such as used by Lam [16] is the study of the application domain, with domain specialists, with the intention of capturing its characteristics, processes and requirements in textual and diagrammatic form. The first step was to define the scope of the domain in discussion with engine controller experts. An early synthesis of the requirements and key issues were formed, giving due attention to the rationale for the requirements. Considering the requirements rationale is useful in reasoning about requirements in the domain [Op.Cit.]. For example, the rationale for confirming a failure before taking action is that the system should not be generate false positive failure results from transient interference on its inputs. From the consideration of requirements rationale, key issues were identified which served as higher level properties required of the system. An example of such a property would be that the failure management system must not be held in a transient action state indefinitely. The rationale from which it has been derived is that a transient state is temporary and actions associated with this state may only be valid for a limited time.

A core set of requirements were identified from several representative failure management engine systems. For example, the identification of magnitude tests with variable limits and associated conditions established several magnitude test types; these types have been further subsumed into a general detection type. This type structure provided a taxonomy for classification of the requirements.

Domain analysis showed that failure management systems are characterised by a high degree of fairly simple similar units made complex by a large number of minor variations and interdependencies. The domain presents opportunities for a high degree of reuse within a single product as well as between products. For example, a magnitude test is usually required in a number of instances in a particular system. This is in contrast to the engine start domain addressed by Lam [16], where a single instance of each reusable function exists in a particular product. Our method is targeted at domains such as failure management where a few simple units are reused many times and a particular configuration depends on the relationships between the instances of these simple units. A first-cut entity relationship model was constructed from the units identified during the domain analysis stage. The entities identified during domain analysis were:

- **INP** Identification of an input sensor and its characteristics to be tested
- **COND** Condition under which a test is performed or an action is taken. (A predicate based on the values and/or failure states of other inputs)
- **DET** Detection of a failure state. A predicate that compares the value of an expression to be tested against a limit value. There are specialized versions of detection, e.g. **DET_MAG** for magnitude tests and **DET_RATE** for rate-of-change tests
- **CONF** Confirmation of a failure state. An iterative algorithm performed for each invocation of a detection, used to establish whether a detected failure state is genuine or transitory
- **ACT** Action taken either normally or in response to a failure, possibly subject to a condition. Assigns the value of an expression, which may involve inputs and/or other output values, to an output
- **OUT** Identification of an output to be used by an action

Figure 2 shows the final class diagram resulting from this early entity-relationship model of generic requirements.

5 Domain Engineering

The aim of the domain engineering stage is to explore, develop and validate the first-cut generic model of the requirements into a validated generic model, using suitable technology. At this stage this is essentially an entity relationship model, omitting any behaviours (except temporary ones added for validation purposes). The model indicates the necessary and permitted configurations of the various functional requirements without detailing the behaviour involved in those requirements. For example, that there must be one confirmation mechanism for each input and that a configuration must have at least one detection mechanism.

The first-cut model from the domain analysis stage was converted to the UML-B notation (Fig. 2) by adding stereotypes and UML-B clauses (tagged values) as defined in the UML-B profile [26]. This allows the model to be converted into the B notation where validation and verification tools are available. The model contains invariant properties, which constrain the associations, and ensures that every instance is a member of its class. As well as these diagrammatic invariants, additional textual invariants may be added where the diagram notation is unable to express constraints. For example, the invariant in Fig. 2 expresses the fact that every action (instance of class `ACT`) must be linked at least once to a confirmation (instance of class `CONF`) via one of the three associations, `hAct`, `pAct` and `tAct`. To validate the model we needed to be able to build up the instances it holds in steps. For this stage a constructor was added to each class so that the model could be populated with instances. The constructor was defined to set any associations belonging to that class according to values supplied as parameters.

The model was tested by adding example instances using the animation facility of ProB and examining the values of the B variables representing the classes and associations in the model to see that they developed as expected. ProB provides an indicator to show when the invariant is violated. Due to the 'required' (i.e. multiplicity greater than 0) constraints in our model, the only way to populate it without violating the invariant would be to add instances of several classes simultaneously. However, we found that observing the invariant violations was a useful part of the feedback during validation of the model. Knowing that the model recognises inconsistent states, is just as important as knowing that it accepts consistent ones. The model was rearranged substantially during this phase as the animation revealed problems. Once we were satisfied that the model was suitable, we removed the constructor operations to simplify the corresponding B model for the next stage.

The next stage is to add behaviour to the generic model by giving the classes operations. In future work we will investigate the best way to introduce this behaviour during the process. It may be possible to add the behaviour after the static model has been validated as described above. Alternatively, perhaps the behaviour will affect the static structure and should be added earlier. In either case, we aim to formalise the rationale described in the domain analysis and derive the behaviour as a refinement from this.

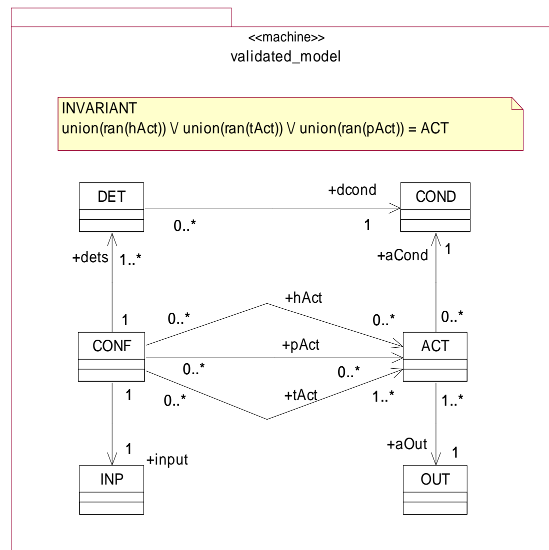

Fig. 2. Final UML-B version of generic model of failure management requirements

6 Requirements for a specific application

Having arrived at a useful model we then use it to specify the requirements for an instance application by populating it with instance requirements for each of the generic requirement classes. For a particular application, the instances are not created and destroyed dynamically but are defined as a static configuration consistent with the generic model. Thus we do not use constructors to populate the model; we define each class to have a fixed set of instances.

At first we used ProB to check the application is consistent with the properties expressed in the generic model. This verification is a similar process to the previous validation but the focus is on possible errors in the instantiation rather than in the model. The application is first described in tabular form. The generic model provides a template for the construction of the tables. Each class is represented by a separate table; foreign key links represent the associations owned by that class. The tabular form is useful as documentation of the application but is not directly useful for verification. To verify its consistency, the tabular form is translated into class instance enumerations and association initialisation clauses attached to the UML-B class model. We found that doing this manually was tedious and error prone. Therefore we automated the translation by implementing a ‘Requirements Manager’ tool. The tool was developed as an IBM eclipse plug-in by a student group[4]. The Requirements Manager (RM) tool loads application

[4] See acknowledgements

configuration data from an Excel file and populates the relevant fields in the UML-B class model.

Initially, we used ProB to check which conjuncts of the invariant are violated. For our FDM example, several iterations were necessary to eliminate errors in the tables before the invariant was satisfied. The ProB 'analyse invariant' facility provides information about which conjuncts of the invariant are violated but, in a data intensive model such as this, it is still not easy to see which part of the data is at fault. It would be useful to show a data counterexample to the conjunct (analogous to an event sequence counterexample in model checking). The RM tool verifies the application data against the class structure and association constraints of the UML-B class model, when that data is first loaded into the database. RM then reports any violations, identifying the specific data that caused the violation. Figure 3 is a screenshot of the RM tool in use, showing the generic requirements structure in two views, and two detail views (on lower right) of data verification errors. Note that a limitation of the tool (inherited from its underlying database representation) is that many to many associations cannot be represented. This is circumvented by inserting an intermediate class into the association (e.g. HACT, PACT and TACT). The RM tool is described in more detail in [28].

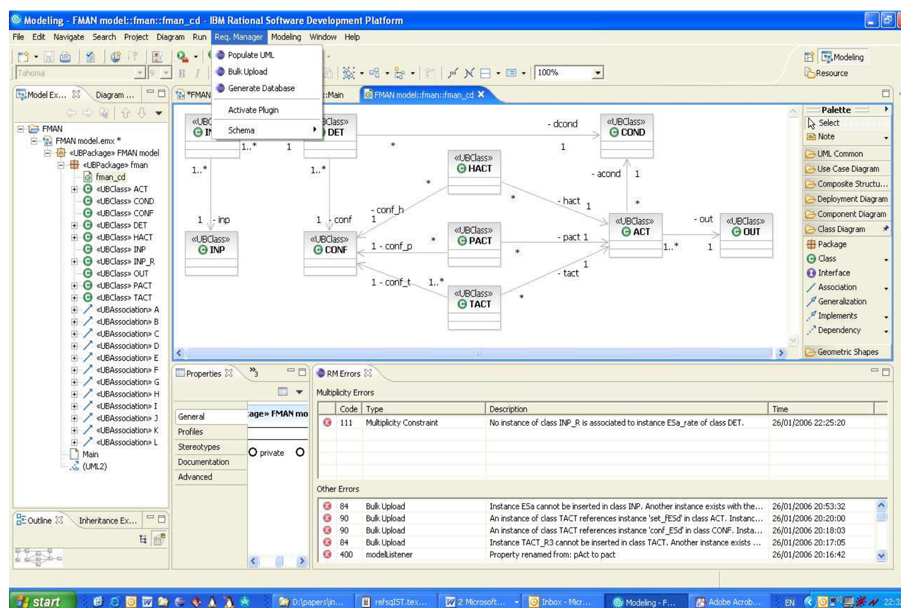

Fig. 3. Screenshot of Requirements Manager being used to populate the generic model

7 Classification of problems

It is useful to classify the kinds of problems found during animation and verification in order to better understand the source of problems and improve the requirements engineering process. So far, we have found that problems can be classified on a methodological stage basis. Possible categories on this basis, some of which we have experienced, are as follows.

- Verification of generic model - the generic model is inconsistent or incorrect
- Validation of generic model - the generic model is correct and consistent but does not reflect the generic requirements
- Validation of generic requirement - the generic model works as expected but animation leads expert to review generic requirements
- Verification of instantiation - the instantiation is inconsistent with the generic model because of an incorrect instantiation
- Verification of instantiation (generic model) - the instantiation is inconsistent with the generic model because the generic model is inadequate
- Validation of instantiation - the instantiation is consistent with the generic model but does not reflect the specific requirements
- Validation of specific requirements - the instantiation is consistent with the generic model but animation leads expert to review specific requirements

In the future, when behavioural features are modelled, we expect to find other ways of classifying problems. For example we may be able to distinguish functional areas that are prone to incorrect specification.

8 Conclusion

In this paper we have discussed a formal, object-oriented approach to the rigorous engineering, validation and verification of generic requirements for a product line of critical systems. Our case study is in the domain of failure management and detection for engine control. The approach can be generalised to any relatively complex system component where repetitions of similar units indicate an opportunity for parameterised reuse but the extent of differences and interrelations between units makes this non-trivial to achieve. The product-line approach to application instance production amortises the effort involved in formal validation and verification over many instances. So far we have considered the static entity-relationship, or class-association aspects of the requirements. In future work we aim to extend the approach to consider also the detailed meaning (i.e. dynamic behaviour) of these classes of requirements.

We have also undertaken a pilot study on a small subset of the requirements, i.e. the dual redundant engine speed functionality ESa and ESb. This exercise had a number of aims: (i) to support the domain analysis at a level of fine detail, (ii) to gain an early view of appropriate design abstractions, (iii) for a B novice to gain experience in the B method and tools, (iv) and to evaluate the usability and utility of method and tools. In particular, the novice engineer's independent development of an approach to refinement by what we have called 'idealisation' and 'de-idealisation' may be a promising

methodological contribution. Thus invaluable input has been provided to the ongoing exercise of developing the new Event-B method and tools.

Two broad areas of future work are indicated by the case study, both linking to related work on Product Line Engineering (PLE). The first concerns instance data management, the second variability vs. commonality in the generic model.

For a product family such as FDM at ATEC as currently envisaged, instance data management is in principle straightforward. This is because no system instance/variant requirements are defined at the generic level - all structure and behaviour is specified in terms of a single generic model. Instance/variant requirements are captured completely by instance-level data. This means that all instance data structures are defined in terms of the generic class definitions. Therefore, the data for a system instance is simply defined as a subset of the database of all required instance specifications; tooling is thus a straightforward database application, as we have demonstrated with our new Requirements Manager tool.

Instance management becomes more complex when variability is required in the generic model. This is the usual state of affairs in PLE. The mobile phone scenario of [20] is typical, where each system instance is defined by a distinct set of functional features, aimed at a specific market segment and target price. We might define a *feature* to be a small coherent group of requirements respresenting some system goal; examples in telephony include CH (call hold), CD (caller divert), CC (conference call). Features are not in general simply composable, and the totality of features cannot in general be specified in one generic model: variability specification is required in the generic model. To date approaches to this (such as [20]) have been in the obvious syntactic form: in ATEC for example, variants on the generic model for other engine manufacturers might be described as extra colour-coded classes, associations, states, events etc. A system variant (or sub-family) would thus be defined in terms of some colour-combination submodel. A more sophisticated metamodelling approach to variability specification, based on the Model-Driven Architecture of the OMG, has recently been proposed [21].

Future work will investigate developing such variability and tooling issues in the ATEC context, using the UML-B and refinement approaches and the RM tool discussed in this paper. The application of refinement approaches to PLE to date has been modest, e.g. [6, 30], and has, in our view, much potential. An obvious unit for modelling variabilities is the feature. Investigations are ongoing into the development of refinement decomposition and generic instantiation in Event-B, and their deployment on variability specification via features. Retrenchment, a generalizing theory for refinement, has been investigated in a feature engineering context [5], and may well also be useful in PLE.

Acknowledgements

We are grateful to ECS students Ledina Hido, Robert Stops and Martin Ross for their work developing the Requirements Manager tool, and to Ledina for her work on the tool specification and verification in B.

References

[1] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[2] J.-R. Abrial. http://www.atelierb.societe.com/index_uk.html, 1998. Atelier-B.

[3] P. Amey. Dear sir, Yours faithfully: an everyday story of formality. In F. Redmill and T. Anderson, editors, *Proc. 12th Safety-Critical Systems Symposium*, pages 3–18, Birmingham, 2004. Springer.

[4] K. Araki, S. Gnesi, and D. Mandrioli, editors. *International Symposium of Formal Methods Europe*, volume 2805 of *LNCS*, Pisa, Italy, September 2003. Springer.

[5] R. Banach and M. Poppleton. Retrenching partial requirements into system definitions: A simple feature interaction case study. *Requirements Engineering Journal*, 8(2), 2003. 22pp.

[6] D. Batory, J. Sarvela, and A. RauschMayer. Scaling step-wise refinement. *IEEE Transactions on Software Engineering*, 30(6):355–371, June 2004.

[7] C.M. Belcastro. Application of failure detection, identification, and accomodation methods for improved aircraft safety. In *Proc. American Control Conference*, volume 4, pages 2623–2624. IEEE, June 2001.

[8] G. Booch, I. Jacobson, and J. Rumbaugh. *The Unified Modeling Language - a Reference Manual*. Addison-Wesley, 1998.

[9] D. Cansell, J.-R. Abrial, et al. B4free. A set of tools for B development, from http://www.b4free.com, 2004.

[10] B. Dehbonei and F. Mejia. Formal development of safety-critical software systems in railway signalling. In M.G. Hinchey and J.P. Bowen, editors, *Applications of Formal Methods*, chapter 10, pages 227–252. Prentice-Hall, 1995.

[11] S.R. Faulk. Product-line requirements specification (PRS): an approach and case study. In *Proc. Fifth IEEE International Symposium on Requirements Engineering*. IEEE Comput. Soc, Aug. 2000.

[12] M. Fayad and D. Schmidt. Object-oriented application frameworks. *Communications of the ACM*, 40(10):32–38, Oct. 1997.

[13] I.J. Hayes, M. A. Jackson, and C. B. Jones. Determining the specification of a control system from that of its environment. In Araki et al. [4], pages 154–169.

[14] L. Laibinis and E. Troubitsyna. Fault tolerance in a layered architecture: a general specification pattern in B. In *Proc. 2nd Int. Conf. on Software Engineering and Formal Methods*, pages 346–355. IEEE Computer Society, Sep 2004.

[15] L. Laibinis and E. Troubitsyna. Refinement of fault tolerant control systems in B. In *Proc. SAFECOMP 2004*, volume 3219 of *LNCS*, pages 254–268. Springer, 2004.

[16] W. Lam. Achieving requirements reuse: a domain-specific approach from avionics. *Journal of Systems and Software*, 38(3):197–209, Sept. 1997.

[17] M. Leuschel and M. Butler. ProB: a model checker for B. In Araki et al. [4], pages 855–874.

[18] R. Macala, L. Jr. Stuckey, and D. Gross. Managing domain-specific, product-line development. *IEEE Software*, pages 57–67, May 1996.

[19] UK Ministry of Defence. Def Stan 00-55: Requirements for safety related software in defence equipment, issue 2. http://www.dstan.mod.uk/data/00/055/02000200.pdf, 1997.

[20] D. Muthig. GoPhone - a software product line in the mobile phone domain. Technical Report IESE-Report No. 025.04/E, Fraunhofer Institut Experimentelles Software Engineering, 2004.

[21] D. Muthig and C. Atkinson. Model-driven product line architectures. In G.J. Chastel, editor, *Proc. Software Product Lines, Second International Conference, SPLC 2002*, volume 2379 of *LNCS*, pages 110–129. Springer, 2002.

[22] D. L. Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, SE-2, March 1976.

[23] S. Schneider. *The B-Method*. Palgrave Press, 2001.

[24] C. Snook and M. Butler. U2B - a tool for translating UML-B models into B. In J. Mermet, editor, *UML-B Specification for Proven Embedded Systems Design*, chapter 5. Springer, 2004.

[25] C. Snook, M. Butler, A. Edmunds, and I. Johnson. Rigorous development of reusable, domain-specific components, for complex applications. In J. Jurgens and R. France, editors, *Proc. 3rd Intl. Workshop on Critical Systems Development with UML*, pages 115–129, Lisbon, 2004.

[26] C. Snook, I. Oliver, and M. Butler. The UML-B profile for formal systems modelling in UML. In J. Mermet, editor, *UML-B Specification for Proven Embedded Systems*, chapter 5. Springer, 2004.

[27] C. Snook, M. Poppleton, and I. Johnson. The engineering of generic requirements for failure management. In E. Kamsties, V. Gervasi, and P. Sawyer, editors, *Proc. 11th Int. Workshop on Requirements Engineering: Foundation for Software Quality*, pages 145–160, Oporto, June 2005. Essener Informatik Beitrage.

[28] C. Snook, M. Poppleton, and I. Johnson. Rigorous engineering of product-line requirements: a case study in failure management. *submitted*, 2006.

[29] W. Tracz. DSSA (Domain-Specific Software Architecture) pedagogical example. *ACM Software Engineering Notes*, pages 49–62, July 1995.

[30] A. Wasowski. Automatic generation of program families by model restrictions. In R.L. Nord, editor, *Software Product Lines, Third International Conference, SPLC 2004, Proceedings*, volume 3154 of *LNCS*, pages 73–89. Springer, 2004.