The engineering of generic requirements for failure management

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 consider the failure detection and management function for engine control systems as an application domain where product line engineering is indicated. 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 this safety-critical domain, subject to avionics industry standards. We present our case study experience in this area as a candidate methodology for the engineering, validation and verification of generic requirements using domain engineering and Formal Methods techniques and tools. For a defined class of systems, the case study produces a generic requirement set in UML and an example instantiation in tabular form. Domain analysis and engineering produce a model which is integrated with the formal specification/ verification method B by the use of our UML-B profile. The formal verification both of the generic requirement set, and of a simple system instance, is demonstrated using our U2B and ProB tools.
This work is a demonstrator for a tool-supported method which will be an output of EU project RODIN[3]. The method, based in the dominant UML standard, will exploit formal verification technology largely as a "black box" for this novel combination of product line, failure management and safety-critical engineering.

1 Introduction

The notion of software *product line* (also known as *system family*) engineering became well established [14], after Parnas' proposal [17] in the 70's of information hiding and modularization as techniques that would support the handling of program families. Product line engineering arises where multiple variants of essentially the same software system are required, to meet a variety of platform, functional, or other requirements. This kind of generic systems engineering is well known in the avionics industry; e.g. [12, 9] describe the reuse of generic sets of requirements in engine control and flight control systems.

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

Domain analysis and object oriented frameworks are among numerous solutions proposed to product line technology. In Domain-Specific Software Architecture [23] for example, the domain engineering of a set of general, domain-specific requirements for the product line is followed by its successive refinement, in a series of system engineering cycles, into specific product requirements. On the other hand [10] describes the Object-Oriented Framework as a "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 domain and product line engineering.

Developers in the avionics industry are interested in the use of object-oriented and UML technology (OOT) [6, 15] as a way to increase productivity. Concepts such as inheritance and design patterns facilitate the reuse of requirements and designs. UML has emerged as the de-facto standard modelling language for object-oriented design and analysis, supported by a wide variety of tools. Due to concerns over safety certification issues however, OOT has not seen widespread use in avionics applications. The controlling standards used in the industry, such as RTCA DO-178B [11] and its European equivalent, EUROCAE ED-12B [1], do not consider OOT, although this is under review.

It is widely recognized that formal methods (FM) technology makes a strong contribution to the verification required for safety-critical systems; indeed, DefStan 00-55 [16] as well as the avionics standards above recommend the use of FM for critical systems. It is further recognized that FM will need to be integrated [4] - in as "black-box" as possible a manner - with OOT in order to achieve serious industry penetration. The combination of UML and formal methods therefore offers the promise of improved safety and flexibility in the design of software for aviation certification.

One approach to the integration of FM and OOT is to enhance - at the abstract modelling stage - UML with the minimum amount of textual formal specification required to completely express functional, safety and other requirements. A tool will convert the customer-oriented abstract UML model to a fully textual model as input to FM tools such as model-checkers and theorem provers. With suitable tool support for configuration and projct management, this approach will facilitate the reuse of verified software specifications and consequently code components.

Adoption of formal methods in the safety-critical industry has been limited partly due to the need for industrial strength tool support. The B method of J.-R. Abrial [2, 19] is a formal method with good tool support, and a good industrial track record, e.g. [8]. At Southampton, we have for some years been developing an approach of integrating formal specification and verification in B, with UML-based design methods, in the above style. The UML-B [22] is a profile of UML that defines a formal modelling notation combining UML and B. It is supported by the U2B tool [20], which translates UML-B models into B, for subsequent formal verification. This verification includes model-checking with the ProB model-checker [13] 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 environmental inputs to the control system, providing the best information possible whilst determining whether a component has failed or not. The role of failure management in an embedded control system is shown in Fig. 1.

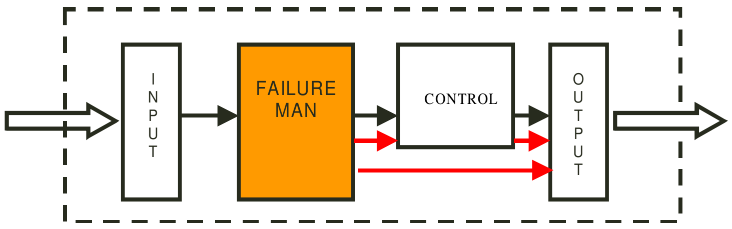


Fig. 1. Context diagram for failure management subsystem

Inputs may tested for signal magnitude and/or rate of change being within defined bounds, and for consistency with other inputs. If no failure is detected, some input values are passed on to the control subsystem; others are only used for failure detection and management. When a failure is detected it is managed in order to maintain a usable set of input values for the control subsystem. This may involve substituting values, and taking alternative actions. To prevent over-reaction to isolated transient values, 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 oversensitivity 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.

1.2 Contribution

Failure detection and management (FDM) in engine control systems is a demanding application area, see e.g. [5]. It gives rise to far more than a trivial parameterizable product line situation, although there will indeed be small-delta system variants fitted out with component sets which differ only on operational parameters. In general, component sets will differ more significantly between system variants. The domain engineering of our requirements covers all variants of interest to AT Engine Controls.

Our approach contributes to the failure detection and management domain by presenting 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 [12, 9], 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.

A further complicating factor is the instability of the FDM requirements domain, which is often subject to late change. This is because the failure characteristics of dynamic controlled systems are usually dependent on interaction with the control systems being developed and can only be fully determined via prototyping. Our generic requirements formulation accommodates this ongoing requirements change process.

Our approach contributes methodologically to product-line requirements engineering in its integration of informal domain analysis with domain engineering that exploits both UML and Formal Methods technology. The application of product-line engineering to failure detection and management is also novel. We have developed a generic requirement set for the failure detection and management function for a class of systems supported by AT Engine Controls; this is modelled in UML-B and in a tabular data schema. We present the process of domain engineering, validating and verifying a generic model and an example instance model. The UML-B is translated to B with the U2B tool, and then verified by model-checking with ProB; this verifies both the generic requirement set and the system instance.

1.3 Structure of the paper

The paper now proceeds as follows. Section 2 introduces formal specification and verification in B, and our approach in Southampton. Section 3 gives an overview of our methodology. Sections 4 - 5 discuss the domain analysis and engineering activities. Section 6 discusses the testing of an instantiated example to verify the generic model. Section 7 concludes with a discussion of future work.

2 Formal specification and verification with B

The B language [2] of J.-R. Abrial is a wide-spectrum language supporting a full formal development lifecycle from specification through refinement to programming. It is supported by full-lifecycle verification toolkits such as *Atelier B* [3], and has been instrumental in successful safety-critical system developments such as signalling on the

Paris Metro [8]. Specification in B gives a concrete description of requirements that can be formally validated by proof tools. The refinement of this specification, and the realization of the requirements into code, are subject to formal verification by these tools.

The Abstract Machine Notation (AMN) of B describes specification in terms of an abstract view of state and behaviour. Simply put, state is described in terms of sets, and behaviour in terms of relations on those sets. Nondeterministic choice is the formal mechanism by which simple, abstract expressions of requirements are made, subject to implementation freedom. An invariant clause captures required properties of the system that must hold at all times, defining the meaning of the data and the integrity of its structure.

The B verification tools [7] generate proof obligations (POs) that initialisation and all operations maintain the invariant; this is called *operation consistency*. Automatic and interactive provers are supplied; an *interactive* prover is really a GUI interface to the automated prover, allowing the user to guide and decompose a proof process which does not work automatically first time. Discharging POs with the interactive prover provides a form of mathematical "debugging" of the specification. Discharging proof obligations can be difficult and time consuming, but once complete the specification is known to be consistent.

A refinement step involves the transformation of a more abstract specification into a more concrete one[4], by elaboration with more data and algorithmic structure, thus reducing nondeterminism. The nature of a valid refinement is that it always satisfies the abstract specification; proof guarantees that the refinement reflects the behaviour of the abstract specification it refines.

2.1 The Southampton approach

In Southampton we have developed two tools to support formal system development in B: ProB and U2B. ProB [13] is a model checker that searches the full abstract state machine model of a B specification for invariant violations, returning any counterexample found. The state model can be presented graphically. Model checking avoids the effort of proof debugging in early development stages, serving as a preliminary validation of the specification before commencing proof. ProB furthermore provides a limited from of temporal model-checking. ProB is also an animator for B machines. In this mode, 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 (not all operations are enabled, or valid, all the time) in order to explore the behaviour of the specification, thus providing a form of informal prototyping.

The UML-B [22] 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

[4] The concrete specification in this context is often called the *refinement*.

entities to provide formal modelling details that have no counterpart in UML. UML-B provides a diagrammatic, formal modelling notation based on UML. The popularity of the UML enables UML-B to overcome some of the barriers to the acceptance of formal methods in industry. Its familiar diagrammatic notations make specifications accessible to domain experts who may not be familiar with formal notations. UML-B hides B's infrastructure by packaging mathematical constraints and action specifications into small sections within the context of an owning UML entity. The U2B [20] translator converts UML-B models into B components (abstract machines and their refinements), thus enabling B verification and validation technology to be exploited.

3 Overview of methodology

We first give an overview of the methodology which is then discussed in more detail in the following sections - see Fig. 2. The first stage is an informal domain analysis (section 4) which is based on prior experience of developing products for the application domain of failure detection and management in engine control. A taxonomy of the kind of generic requirements found in the application domain is developed and, from this, a *first-cut* generic model is formed, in entity-relationship terms, naming and relating the generic requirements.

The identification of a useful generic model is a difficult process and therefore further exploration of the model is warranted. This is done in the domain engineering stage (section 5) where a more rigorous examination of the first-cut model is undertaken, using the B-method and the Southampton tools. The model is animated by creating typical instances of its generic requirement entities, 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 entities 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.

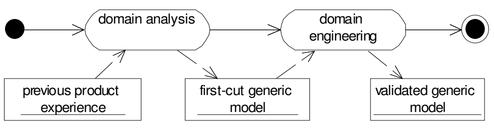


Fig. 2. Process for obtaining the generic model

For each new application instance, the requirements are expressed as instances of the relevant generic requirement entities and their relationships, in an *instance* model - see Fig. 3. The ProB model checker is then used to automatically verify that the 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 (desired) set of requirements, in terms of system behaviour that will result.

The final stage, 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. This final stage of the process - “validate instantiation” in Fig. 3 - is work in progress.

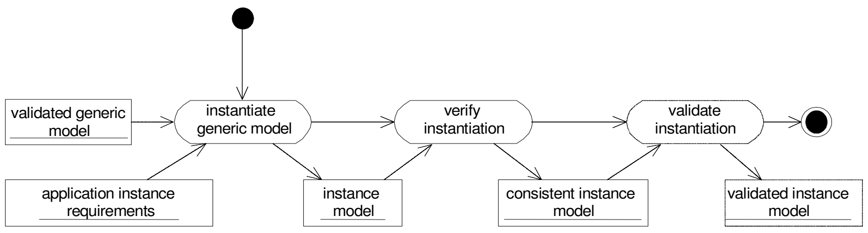


Fig. 3. Process for using the generic model in a specific application

4 Domain analysis

The strategy adopted to reach the first-cut generic requirement model (Fig. 2) was to apply domain analysis in a style similar to that used by Lam [12]. Prieto-Diaz [18] defines domain analysis as “a process by which information used in developing software systems is identified, captured and organised with the purpose of making it reusable when creating new systems”. The first step was to define the scope of the domain in discussion with the engine controller experts. An early synthesis of the requirements and key issues were formed, giving due attention to the rationale for the requirements; elements of this early draft are given in the introduction. Considering the requirements’ rationale is useful in reasoning about requirements in the domain [12]. For example, the rationale for confirmation of failure before permanent action is taken, is for the system to be tolerant to noise. 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 key property would be that the 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 generic 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.

This 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 type is usually required in a number of instances in a particular system. This is in contrast to the engine start domain addressed by Lam [12], where a single instance of each reusable function exists in a particular product. The methodology described in this paper 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. The following requirements taxonomy was derived from this stage of the domain analysis:

INP Identification of the inputs to be tested.
COND Condition under which a test is performed. A predicate based on the values and/or failure states of other inputs. Produces a number of actions.
DET Detection of a failure state. A predicate that compares the value of an expression involving the input to be tested against a limit value.
CONF Confirmation of a (persistent) 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 failures, 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 the outputs to be used in an action

The analysis then considered the relationships between the common elements. This was used to form the first-cut generic domain model of Fig. 2, which is elaborated in UML in Fig. 4. An input (INP) instance contains a sensor value input from the environment. It may have many associated tests, and a test instance may utilise many other inputs. A test is composed of a detection method (DET) and confirmation mechanism (CONF) pair[5]. For example an engine speed input, which is tested for out of range magnitude as well as excessive rate of change, has a detection and associated confirmation for the magnitude test and a different detection and different confirmation for the rate of change test. Each test also has a collection of conditions (COND) that must be satisfied for the test to be applied. For example, the engine speed magnitude test that is be applied when the engine is running is different from the engine speed magnitude test applied while starting the engine. A confirmation mechanism is associated with three different sets of actions (ACT), the healthy actions, the temporary actions (taken while a test is confirming failure) and the permanent actions (taken when a test has confirmed failure). Each action is associated with at least one output (OUT) that it modifies.

The detection mechanism DET of a test can be further decomposed as (i) a check that signal magnitude is within defined bounds, (ii) a check that signal rate of change is within defined bounds or (iii) a comparison with a predicted value.

The generic requirement set was recorded into a traceable requirements document for the case study. The document had several features which assisted in presenting the

[5] Consequently TEST is not a category in the requirements taxonomy above.

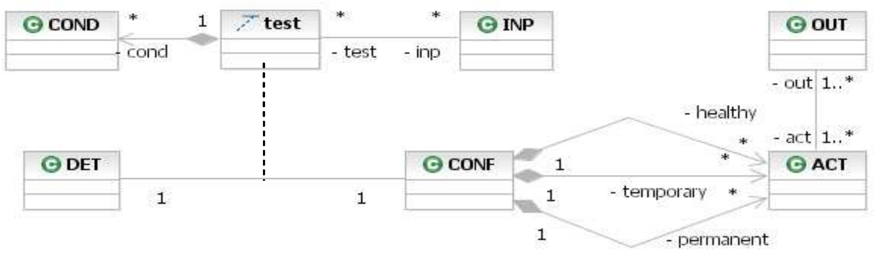


Fig. 4. Overview of common types of elements of functionality and their interrelationships

requirements in a way suitable for further analysis, in particular a *generic* section and an example *instantiation* of it in tabular form.

The generic section includes

1. The taxonomy of requirements
2. The model (Fig.4) of the generic requirement domain
3. For each generic requirement: a formal statement, and a narrative explanation and rationale, where relevant

The example instantiation section is in tabular form and consists of

1. Uniquely identified instances of the elements in the generic model
2. References from each instance to other instances as described by the relationships in the generic model

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 as per Fig. 2. At this stage this is essentially an entity relationship model, omitting any dynamic features (except for temporary ones added for validation purposes). The example presented in this paper (Fig.5) was derived from the domain analysis (we will illustrate this derivation as we go) by trying it with specific example scenarios taken from existing applications. In this way, we develop our understanding of the common, reusable elements within the domain by testing the relationships between them. Again, we rely on our knowledge of existing systems.

The model was converted to the UML-B notation (Fig.5) by adding stereotypes and UML-B clauses (tagged values) as defined in the UML-B profile [22]. This allows the model to be converted into the B notation where validation and verification tools are available. The B model contains the *invariant* property, which defines each association, and ensures that every instance is a member of its class. To validate the first-cut model we needed to be able to build up the instances it holds in steps. For this stage, all classes

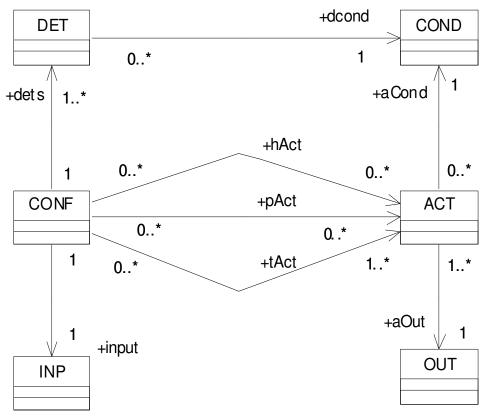
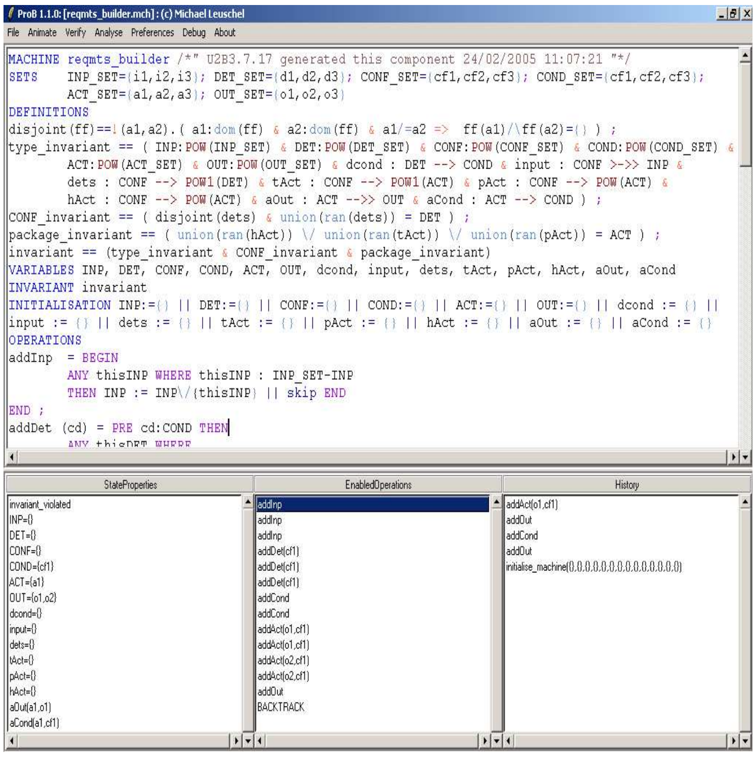


Fig. 5. Final UML-B version of generic model of failure management requirements

were given variable cardinality (there is a UML-B clause to define the cardinality and variability of classes) and a constructor was added to each class so that the model could be populated with instances. The constructor was endowed with behaviour (written in μB, the action and constraint language of UML-B) to set any associations belonging to that class to values (i.e. instances of other classes) supplied as parameters.

The developing model was then 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. For example an invariant is constructed to show that the type of the association input is a relation from CONF to INP but the multiplicity constraint (1-1) strengthens this to a total surjection. Initially, an instance of CONF cannot be added because there are no instances of INP with which to parameterize it. This forces a partial ordering on the population of the model but still does not prevent the invariant from being violated. The INP constructor is available initially because the INP class has no outgoing associations, but as soon as an instance of INP is added the multiplicity constraint of input 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 at least as important as knowing that it accepts consistent ones.

Figure 6 shows ProB being used to validate the generic model. The top pane shows the B version of the model generated automatically from the UML-B version (Fig. 5). The centre bottom pane shows the currently available operations (i.e. the constructors we added to the generic model for testing purposes). An operation is only enabled when its preconditions and guards are true. In our model this is when there are unused 'possible' instances left to create and values available to supply as parameters. (Note that, for practical reasons, we limit a class' set of possible instances to a small enumerated set). An available operation is invoked by selecting it from this pane. Each available operation is repeated in the list for each possible external (parameter) or internal (non-determinism) choice. The left pane shows the current value of each variable as well as an indication of whether the invariant has been violated. The right pane shows the sequence of operations that have already been taken. In the state shown, two outputs and a condition were created. This enabled the constructors for detections and actions. An action, a1, associated with output, o1, and condition, cf1, has been created and can be seen in the variables aout and acond representing the associations from the action class. The invariant is (correctly) violated in this state because output o2 is not used by an action, disobeying the surjection property of the association.


ProB 1.1.0: [reqmts_builder.mch] : (c) Michael Leuschel
File Animate Verify Analyse Preferences Debug About

```
MACHINE reqmts_builder /*" U2B3.7.17 generated this component 24/02/2005 11:07:21 "*/
SETS    INP_SET={i1,i2,i3}; DET_SET={d1,d2,d3}; CONF_SET={cf1,cf2,cf3}; COND_SET={cf1,cf2,cf3};
        ACT_SET={a1,a2,a3}; OUT_SET={o1,o2,o3}
DEFINITIONS
disjoint(ff)==!(a1,a2).( a1:dom(ff) & a2:dom(ff) & a1/=a2 =>  ff(a1)/\ff(a2)={} ) ;
type_invariant == ( INP:POW(INP_SET) & DET:POW(DET_SET) & CONF:POW(CONF_SET) & COND:POW(COND_SET) &
        ACT:POW(ACT_SET) & OUT:POW(OUT_SET) & dcond : DET --> COND & input : CONF >->> INP &
        dets : CONF --> POW1(DET) & tAct : CONF --> POW1(ACT) & pAct : CONF --> POW(ACT) &
        hAct : CONF --> POW(ACT) & aOut : ACT -->> OUT & aCond : ACT --> COND ) ;
CONF_invariant == ( disjoint(dets) & union(ran(dets)) = DET ) ;
package_invariant == ( union(ran(hAct)) \/ union(ran(tAct)) \/ union(ran(pAct)) = ACT ) ;
invariant == (type_invariant & CONF_invariant & package_invariant)
VARIABLES INP, DET, CONF, COND, ACT, OUT, dcond, input, dets, tAct, pAct, hAct, aOut, aCond
INVARIANT invariant
INITIALISATION INP:={} || DET:={} || CONF:={} || COND:={} || ACT:={} || OUT:={} || dcond := {} ||
input := {} || dets := {} || tAct := {} || pAct := {} || hAct := {} || aOut := {} || aCond := {}
OPERATIONS
addInp  = BEGIN
        ANY thisINP WHERE thisINP : INP_SET-INP
        THEN INP := INP\/{thisINP} || skip END
END ;
addDet (cd) = PRE cd:COND THEN
        ANY thisDET WHERE
```

StateProperties	EnabledOperations	History
invariant_violated	addInp	addAct(o1,cf1)
INP={}	addInp	addOut
DET={}	addInp	addCond
CONF={}	addDet(cf1)	addOut
COND={cf1}	addDet(cf1)	initialise_machine({},{},{},{},{},{},{},{},{},{},{},{},{},{})
ACT={a1}	addDet(cf1)	
OUT={o1,o2}	addCond	
dcond={}	addCond	
input={}	addAct(o1,cf1)	
dets={}	addAct(o1,cf1)	
tAct={}	addAct(o2,cf1)	
pAct={}	addAct(o2,cf1)	
hAct={}	addOut	
aOut(a1,o1)	BACKTRACK	
aCond(a1,cf1)		

Fig. 6. ProB being used to validate the generic model

The model was re-arranged substantially during this phase as the animation revealed problems. Firstly, our initial model associated each detection with a confirmation. During animation we discovered that this was a mistake since many related detections may be used on a single input all of which should have the same confirmation mechanism. We re-arranged the model to associate inputs with confirmations (hence losing the association class, test). We also discovered that actions were often conditional, so we added an association from ACT to COND. Finally we found that, since we had associated sets of actions with confirmations, we did not need a further multiplicity from actions and outputs, thus simplifying the model a little. Hence, ProB animation provides a useful feedback tool for validation while domain engineering a reusable model in UML-B. The final version of the generic model is shown in Fig. 5. A confirmation (CONF) is the key item in the model. Each and every confirmation has an associated input (input:INP) to be tested and a number of detections (dets:DET) are performed on that input. Each detection has exactly one enabling condition (dcond:COND). A confirmation may also have a number of actions (hact:ACT) to be taken while healthy, a number to be taken while confirming (tAct:ACT) and a number to be taken once confirmed (pAct:ACT). Each action acts upon exactly one output (aOut:OUT).

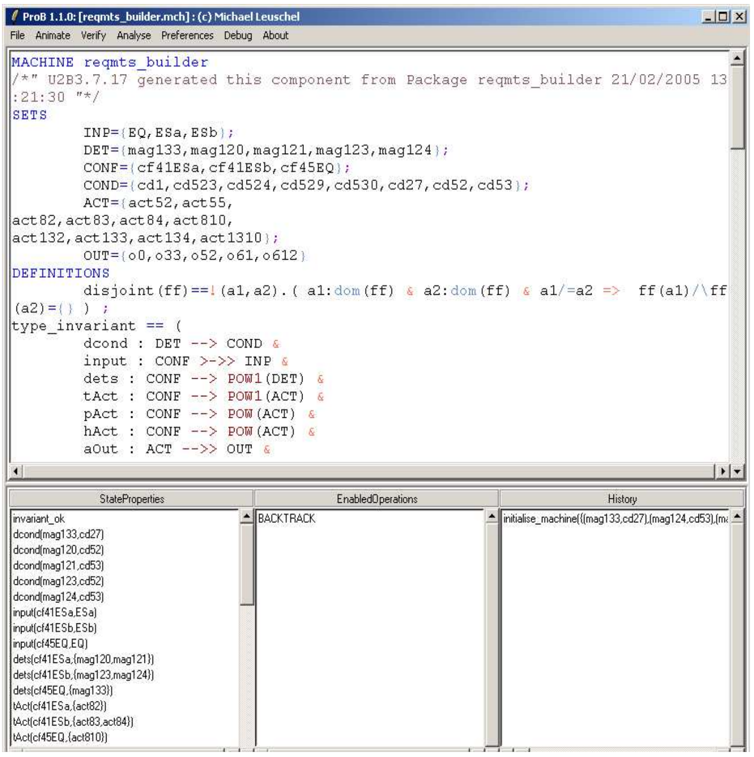
Once we were satisfied that the model was suitable, we converted the classes to fixed instances and removed the constructor operations. This simplifies the corresponding B model and the next stage. The associations (representing the relationships between class instances) are the part of the model that will constrain a specific configuration. These are still modelled as variables so that they can be described by an invariant and particular values of them verified by ProB.

6 Testing a specific example satisfies the generic model

Having arrived at a useful model we then specified an example application - creating an *instance* model - and used ProB to check the example was consistent with the properties expressed in the generic model. This produced a *verified* instance model as per Fig. 3. This verification is a similar process to the previous validation but the focus is on possible errors in the example rather than in the model. The example specification was 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 with properties for each entry in the table representing the associations owned by that class. The tabular form is useful as an accessible documentation of the specific example but is not directly useful for verification. To verify its consistency, the tabular form was translated into class instance enumerations and association initialisation clauses attached to the UML-B class model. This was done manually, which was tedious and error prone, but automation via a tool is envisaged.

ProB was then used to check which conjuncts of the invariant were violated by the example. Several iterations were necessary to eliminate errors in the tables before the invariant was satisfied. Fig. 7 shows ProB being used to verify the example. The initialisation has been invoked (as seen in the history pane) leaving no operations enabled (since they have all been removed). The state variable values in the left-hand pane embody the example specification which satisfies the invariant.

Initially, testing of the specific instantiation caused an invariant violation. The ProB ‘analyse invariant’ facility provides information about the invariant in the following form (each line represents a conjunct in the invariant). Only a few conjuncts are shown:



ProB 1.1.0: [reqmts_builder.mch] : (c) Michael Leuschel
File Animate Verify Analyse Preferences Debug About
StateProperties
EnabledOperations
History

```
MACHINE reqmts_builder
/*" U2B3.7.17 generated this component from Package reqmts_builder 21/02/2005 13
:21:30 "*/
SETS
        INP={EQ,ESa,ESb};
        DET={mag133,mag120,mag121,mag123,mag124};
        CONF={cf41ESa,cf41ESb,cf45EQ};
        COND={cd1,cd523,cd524,cd529,cd530,cd27,cd52,cd53};
        ACT={act52,act55,
act82,act83,act84,act810,
act132,act133,act134,act1310};
        OUT={o0,o33,o52,o61,o612}
DEFINITIONS
        disjoint(ff)==!(a1,a2).( a1:dom(ff) & a2:dom(ff) & a1/=a2 =>  ff(a1)/\ff
(a2)={} ) ;
type_invariant == (
        dcond : DET --> COND &
        input : CONF >->> INP &
        dets : CONF --> POW1(DET) &
        tAct : CONF --> POW1(ACT) &
        pAct : CONF --> POW(ACT) &
        hAct : CONF --> POW(ACT) &
        aOut : ACT -->> OUT &
```

```
invariant_ok
dcond(mag133,cd27)
dcond(mag120,cd52)
dcond(mag121,cd53)
dcond(mag123,cd52)
dcond(mag124,cd53)
input(cf41ESa,ESa)
input(cf41ESb,ESb)
input(cf45EQ,EQ)
dets(cf41ESa,{mag120,mag121})
dets(cf41ESb,{mag123,mag124})
dets(cf45EQ,{mag133})
tAct(cf41ESa,{act82})
tAct(cf41ESb,{act83,act84})
tAct(cf45EQ,{act810})
```

```
BACKTRACK
```

```
initialise_machine({(mag133,cd27),(mag124,cd53),(m
```

Fig. 7. ProB being used to verify the example application

```
 (COND:POW(COND_SET))                 == TRUE
 (ACT:POW(ACT_SET))                   == TRUE
 (OUT:POW(OUT_SET))                   == TRUE
 (aOut:TotalSurjection(ACT,OUT))      == false
 (aCond:(ACT-->COND))                 == false
(((union(ran(hAct))\/union(ran(tAct)))
   \/union(ran(pAct)))=ACT)           == false
```

We found that the analyse invariant facility provided some indication of where the invariant was inviolated (i.e. which conjunct) 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). This is another area for potential tool support.

7 Conclusion

In this paper we have illustrated a product-line approach to the rigorous engineering, validation and verification of generic requirements for critical systems such as 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 amortises the effort involved in formal validation and verification over many instance applications.

As indicated before, the methodology and tools presented are work in progress. 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. At the moment, however, these are not enforced by the generic model. Key issues are higher level requirements that could be expressed at a more abstract level from which the (already validated) 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 [21] which gives an example of refinement of UML-B models in the failure management domain. The domain analysis process of Fig. 2 would then be elaborated as shown in Fig. 8.

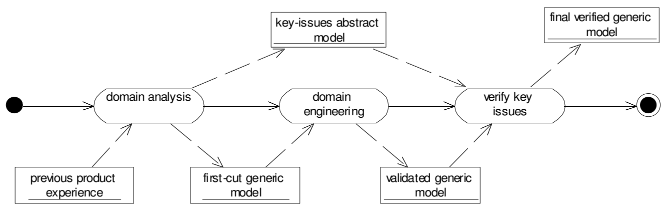


Fig. 8. Elaboration of domain analysis process to show refinement of key issues

Further development is required to validate instance models, as per section 3, Fig. 3. Whilst an instance model can be verified against the constraints embodied within the generic model (i.e. that it is a valid instantiation of the generic model), it may be the wrong configuration. That is, it may specify the wrong run-time behaviour. A further stage to validate the specific configuration is envisaged. Formal refinement could be utilised to add new variables and events to represent the dynamic behaviour of the system. This would allow the specific configuration to be validated via animation.

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. ProB could easily be enhanced to provide, for example, data counterexamples explaining invariant violations.

References

[1] *EUROCAE ED12B - Software considerations in Airborne Systems and Equipment Certification.* http://www.eurocae.org.

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

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

[4] 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.

[5] 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.

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

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

[8] 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.

[9] 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.

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

[11] Radio Technical Commission for Aeronautics. *RTCA DO 178B -Software considerations in Airborne Systems and Equipment Certification*. http://www.rtca.org.

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

[13] M. Leuschel and M. Butler. ProB: a model checker for B. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *Proc. FME2003: Formal Methods*, volume 2805 of *LNCS*, pages 855–874, Pisa, Italy, September 2003. Springer.

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

[15] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1997.

[16] 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.

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

[18] R. Prieto-Diaz. Domain analysis: An introduction. *ACM SIGSOFT Software Engineering Notes*, 15(2):47–54, 1990.

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

[20] 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.

[21] 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.

[22] 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.

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