Automated V&V for high integrity systems, a targeted formal methods approach

Simon Burton, John Clark, Andy Galloway, John McDermid
Department of Computer Science.
University of York,
Heslington, York.
Y010 5DD, England.
+44 1904 432749
{burton, jac, andyg, jam}@cs.york.ac.uk

Abstract

This paper describes the intermediate results of a project to develop automated, high integrity, software verification and validation techniques for aerospace applications. Automated specification validation and test case generation are made possible by the targeted use of formal methods. Specifically, the restricted domain of use is exploited to reduce the set of mathematical problems to those that can be solved using constraint solvers, model checkers and automated proof tactics. The practicality of the techniques is enhanced by the tight integration of the formal methods to intuitive specification notations, existing specification modelling tools and a traditional software development process.

This paper presents evidence to support an emerging appreciation amongst the software engineering community that, for the benefits of formal methods to be widely exploited in industry, an approach must be taken that integrates formal analysis with intuitive engineering notations, traditional engineering approaches and powerful tool support.

1. Introduction

It is widely accepted that verification and validation (V&V) activities for high integrity systems are expensive (typically over 50% of total software development costs [2]). The requirements for such systems are often subject to change throughout the project so the high V&V costs are normally incurred not only once, but many times. Also, the cost of fixing errors later in the development life-cycle can be many times more than if they were identified during the phase in which they were introduced. Additionally, commercial pressures to reduce time to market, technological conservatism and the need to meet standard test metrics make the software V&V process a highly fragile and risky component of system development.

The use of formal methods has long been advocated as a means of improving the development of high integrity systems. Despite evidence to support this claim, e.g. [14, 17], formal methods have still to gain widespread use in the software industry. Industrial acceptance of formal methods requires the development of powerful tools to support formal analysis, pragmatic approaches to using these tools within a software process and more industrially applicable examples of the successful use of formal methods [6, 15]. Also, for the engineers with the system domain knowledge to be able to perform V&V there is a need, as Ould [22] put it, to "disguise" the formality so that an impractical amount of formal methods skill is not a pre-requisite to effective V&V.

This paper describes the results of a project that has achieved practical integration of automated formal methods for V&V into an industrially applicable software development process. The paper is structured as follows: Section 2 introduces the background and objectives of the work reported here. Section 3 describes the translation of domain specific, intuitive engineering notations into formal specifications. Section 4 describes how these intermediate formal specifications can be used to automatically analyse certain properties of the requirements specification. Section 5 describes a method of automated test case design and test data generation, based again on the intermediate formal specification. Section 6 presents some results of applying the techniques in practice and gives an evaluation of the work so far. Section 7 presents some conclusions and suggests directions for future work.

2. Background and objectives

The work reported here is being undertaken as part of a process improvement programme to demonstrate a "better, faster, cheaper" software process for developing Electronic Engine Controllers (EECs) for aircraft engines. These are real-time, safety critical, fault-tolerant computer systems embedded in complex engineering products. The contribution of the V&V strand of the process improvement work (the subject of this paper) is to develop efficient and effective V&V techniques that can be smoothly integrated into a practical engineering process.

The use of formal methods is intended to increase the integrity of engineering activities already performed such as specification and testing. These improvements must be implemented within a process that engineers can use with the minimum amount of re-training. Therefore intuitive engineering notations have been retained as a means of software specification and techniques have been developed to increase the integrity of these specifications through the use of automated formal analysis.

Although this research is targeted towards specification validation and software testing, it is acknowledged that significant benefits in these areas can not be attained without improving the rigour and consistency of the requirements specifications. Specification notations are therefore used that are both "engineer friendly" and amenable to formal analysis. The savings demonstrated in the validation and testing phases serve as drivers to encourage investment in these improved specification activities. It is expected that the most significant cost-benefits can be achieved by capturing more requirements and software specification errors at the specification validation phase (therefore reducing the number of iterations of the software design, coding and testing phases) and by automating test case generation (one of the most time consuming parts of the present process).

3. Translating engineering notations into formal specifications

Domain specific graphical engineering notations are popular with engineers, but their semantics are often unclear from inspection of the diagrams alone. In reality, it is also unlikely that only one notation will be used to specify a system, or indeed that notations will be used consistently between projects. The resulting loose specification and inconsistency complicates the task of automating specification validation and test case generation based on these notations. Indeed as the notations change so frequently (as a result of commercial trends, new or outdated tool-support etc.) it may not even be cost effective to invest in automated V&V tool support which may in practice only have a limited life-span and audience.

Translation of the graphical requirements into a core formal notation removes the vagueness of the original notations and makes the behaviour implied by the specification explicit for the purposes of V&V. Validation may thus be supported by rigorous (or formal) reasoning using the formal representation. Also, by explicitly rendering all specified behaviour, the intermediate representation is a strong basis for automated test case design. The use of a common formal notation to model several engineering notations facilitates re-use of the analysis and test techniques. The introduction of a new notation requires only a translation to the formal notation, after which the previously developed tools and heuristics can be re-used. A strict translation process allows a fixed structure to be enforced on the resulting formal specification that can be exploited in the development of automated heuristics, e.g. test data generation procedures and proof tactics.

The work reported here focuses on the specification, validation and verification of the discrete aspects of engine controllers (well-established mathematically based processes were already in place for modelling and validating the continuous aspects of the control software – e.g. the control laws). The Practical Formal Specification (PFS) notation [7, 19] is used to specify the functional software requirements. The PFS notation consists of hierarchical state machines (in particular, a dialect of statecharts[1] [9]) and tabular forms, such as those employed in SCR (Software Cost Reduction) e.g.[12]. The notation has so far proven popular with the engineers introduced to PFS[2]. PFS also provides a theory for combining components specified in the notation – based on weakest precondition reasoning – and a set of suggested "healthiness properties" that specifications should display to be considered valid.

One of the cornerstones of the PFS approach is that engineers are not only required to specify the intended behaviour of components of the system, they are also obliged to state explicitly the assumptions on which each component relies, i.e. its domain of applicability. Healthiness conditions can then be stated and discharged to demonstrate that, for instance, within the assumptions of each component the behaviour is completely and unambiguously defined. Additionally, healthiness conditions are stated to demonstrate that where behaviour is scoped by assumptions, it is only ever used when the required assumptions hold.

[1]The state-based notation employed in this paper, a sub-set of Statecharts, differs slightly from that usually employed in PFS.

[2]Who have stated that they find the notation valuable *even without* the added rigour provided by a formal underpinning.

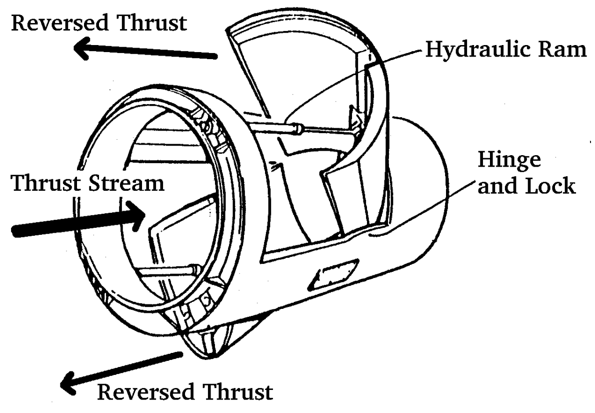

Figure 1. Thrust Reverser System (Deployed)

Due to the restricted domain, the specifications shared some common attributes which reduced the set of (mathematical) problems that needed to be solved when applying the formal analysis:

- The software requirements did not involve the storage and maintenance of complex information structures, typically only fixed-point numbers, conditions and enumerated types were used.
- In the control software domain, non-determinism (looseness) as a means of abstraction is difficult to apply[3]. In the example given here, the requirements were tightly specified, only one outcome was to be specified for each situation[4].

PFS components are either reactive (the outputs depend only on the current set of inputs) and specified using tables, or else state-based (the outputs depend on both the current inputs and the current state of the system) and specified using annotated state-machines.

The example used to illustrate the techniques reported in this paper is the specification of a thrust reverser deployment mechanism. The thrust reverser provides part of the retarding force for an aircraft on landing (see figure 1). It slows the aircraft by using pivoting doors to redirect the engine thrust. For the purpose of clarity and brevity, we will present a much simplified version of the specifications, although their essence is retained. A real thrust reverser system was used as the primary case study for evaluating the techniques described here. The software specification used for the case study consisted of 70 pages of PFS tables and statecharts, component combination diagrams and supporting text.

Examples of software requirements written in PFS are given in figures 2 and 3. Figure 2 describes a function that returns a boolean value (*DoorDeployed*) corresponding to whether a door is locked into its deployed position or not. The assumption defines the context in which the component may be safely used (in this case, the conditions under which sensor values may be deemed to be valid). The guard/definition pairings define the conditions (guards) under which the function returns particular values (definitions). A state-machine that specifies which commands should be sent to the door actuators based on the pilot actions and current door position is given in figure 3. The part of the transition labels before the '/' defines the condition under which the transition is taken. The rest of the label defines the action to be performed on taking the transition. The values for *DoorDeployed*, *DoorStowed* and *PilotCommand* are calculated based on functions defined in the reactive notation. Likewise the *DoorActuators* command would be transformed into actuator signals based on the command and a number of environmental and positional inputs. This function would also be specified in the reactive notation.

Both the reactive and state-based components are translated into formal specifications (we use the model-based notation Z [24] due to the large amount of local experience and existing in-house tool support). The semantics of PFS notations has been formally specified also using Z and this is used to define the translation from the reactive components into Z. The state-based components are modelled using Statemate [10] (a commercially available tool that allows Statecharts to be entered and animated via a mouse-driven interface). The semantics used by the tool are well-documented [11] and have also been formally specified [20, 29]. These semantics are used to define the translation from the Statecharts into Z. The formal specifications for both notation types are structured as follows:

- *Auxiliary definitions:* These may include definitions of types, constants and relations used to constrain the system according to the static semantics of the engineering notation.
- *Global State (for Statecharts only):* Contains all information relating to the persistent state of the system. This may include a set of currently active behavioural states, active events and the values of all data variables local to the statechart. The global state is constrained by semantic relations specified

[3]The controlled environment is understood in terms of the great many relationships between physical quantities, as a result the expression of requirements are highly explicit and deterministic.

[4]Although PFS notation does allow some non-deterministic abstractions to be used in certain situations [7].

Function:	DoorDeployed
Assumption:	FullyRetracted $\leq$ RamPosition $\leq$ FullyExtended **AND** $0 \leq$ Hinge ≤ 90
Guard 1:	RamPosition $>$ DeployedPosition **AND** Hinge $>$ 80 **AND** DeployLock $=$ Activated
Definition 1:	True
Guard 2:	RamPosition $<$ DeployedPosition **OR** Hinge $<$ 80 **OR** DeployLock $=$ Deactivated
Definition 2:	False

Figure 2. Reactive component for sensing thrust reverser door deployment

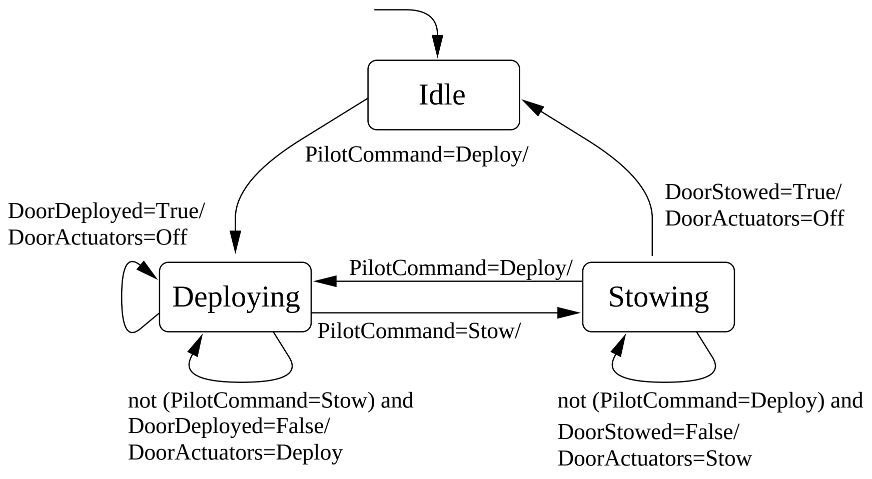

Figure 3. State-based component for controlling door deployment

in the auxiliary definitions (defined in terms of a state invariant).

- *Operations:* The dynamic behaviour of the system is specified as a set of operations. Each operation defines a transformation of the global state (for statechart operations only) and inputs to the component in terms of a pre-condition. A post-condition constrains the next value of the global state (for Statecharts operations only) and a definition of the outputs of the component. One operation is specified for each reactive definition and for each statechart transition. These operations form the basis of the specification validation and test case generation activities described in the following sections.

The Statemate [10] tool provides an "Application Programming Interface" (API), that allows direct access to the internal form of the specification. An interfacing tool, called StateZ, was written by the authors, that takes this internal form and, based on an understanding of the formal semantics of the Statecharts, directly generates a formal specification of the statechart in Z. Included in this specification are the accompanying proof conjectures required to discharge particular healthiness checks of the specifications (see section 4) and automatically generated English language annotations. This informal text provides the traceability between the formal operations and their corresponding part in the original requirements or a description of the property which the conjectures are used to prove. These annotations not only allow the generated Z document to be reviewed for correctness with respect to the original requirements (verifying the automated translation) but also form the basis of the test descriptions which are used to associate each test with the property in the requirements being checked. The addition of the informal text generation to the translation tool was found to greatly increase the readability and usability of the formal specification and associated tests.

The Statemate and StateZ tools can be run in parallel, allowing the Z to be re-generated whenever a change is made to the Statecharts. Coupled with the automated theorem proving described below, this allows the formal analysis to be used as a development aid rather than a separate post development activity.

At present, no tool support exists for translating the PFS reactive notation into Z (this step is done by hand)

and therefore the checking of the reactive components was not as tightly integrated into the specification process as for the Statecharts, however we predict that this should not present any technical difficulties, given a suitable method of electronically recording and managing the reactive tables.

4. Specification validation

In current industrial practice, many requirements errors are only found once the system has been implemented. Detecting them at an earlier stage in the development would greatly reduce the cost of (both implementation and V&V) re-work. This can best be achieved by applying a variety of diverse methods to validate the requirements specifications. These can include peer review, model animation (as supported by tools such as Statemate) and automated formal analysis. The use of intuitive engineering notations would normally exclude the possibility of applying formal analysis. However, based on the same mapping used to generate the formal specification, specification healthiness conditions can be couched as formal constraints. Formal analysis can then be used to show the truth (or otherwise) of these constraints.

Completeness[5] and determinism[6] are two of the healthiness conditions suggested by the PFS approach. If the behaviour of a component is defined as a set of operations $\{Op_1, Op_2, ...Op_n\}$ over the inputs and state, then a conjecture on the completeness of the specification of that component can be formulated as follows:

$$\begin{array}{l}\vdash \forall \mathit{GlobalState}, \mathit{Inputs} \bullet \mathit{Assumptions} \Rightarrow \\ \quad \text{pre}\, Op_1 \lor \text{pre}\, Op_2 \lor \ldots \text{pre}\, Op_n\end{array}$$

Informally, for each possible value of the global state (if there is one) and each combination of inputs that satisfy the validity assumptions of the component, the precondition of *at least one* operation is satisfied.

A similar conjecture can be defined to show the determinism of the operations. Each combination of global state and inputs that satisfies the component validity assumption must satisfy *at most one* operation.

$$\begin{array}{l}\vdash \forall \mathit{GlobalState}, \mathit{Inputs} \bullet \mathit{Assumptions} \Rightarrow \\ \quad \forall i : 1..n-1 \bullet \forall j : i+1..n \bullet \\ \qquad \neg(\text{pre}\, Op_i \land \text{pre}\, Op_j)\end{array}$$

[5]The behaviour of a system is defined for each combination of inputs and current state.

[6]The behaviour of a system is unambiguously defined for each combination of inputs and current state.

Completeness and determinism conjectures for the example reactive definition and state-based component are shown in figures 4 and 5 respectively.

$$\begin{array}{l}\vdash \forall \mathit{RamPosition} : \mathbb{N};\ \mathit{Hinge} : \mathbb{N}; \\ \mathit{DeployLock} : \mathit{Activated} \mid \mathit{Deactivated} \bullet \\ (\mathit{FullyRetracted} \leq \mathit{RamPosition} \leq \mathit{FullyExtended} \\ \land\ 0 \leq \mathit{Hinge} \leq 90) \Rightarrow \\ \quad (\mathit{RamPosition} > \mathit{DeployedPosition} \land \\ \quad \mathit{Hinge} > 80 \land \mathit{DeployLock} = \mathit{Activated}) \lor \\ \quad (\mathit{RamPosition} < \mathit{DeployedPosition} \lor \\ \quad \mathit{Hinge} < 80 \lor \mathit{DeployLock} = \mathit{Deactivated})\end{array}$$

Figure 4. Completeness conjecture for DoorDeployed

$$\begin{array}{l}\vdash \forall \mathit{State} : \mathit{Idle} \mid \mathit{Deploying} \mid \mathit{Stowing}; \\ \mathit{PilotCommand} : \mathit{Off} \mid \mathit{Deploy} \mid \mathit{Stow}; \\ \mathit{DoorDeployed} : \mathit{Boolean}; \\ \mathit{DoorStowed} : \mathit{Boolean} \bullet \\ \mathit{State} = \mathit{Stowing} \Rightarrow \\ \quad \neg(\mathit{DoorStowed} = \mathit{True} \land \\ \quad \neg \mathit{PilotCommand} = \mathit{Deploy} \land \\ \quad \mathit{DoorStowed} = \mathit{False}) \land \\ \quad \neg(\mathit{DoorStowed} = \mathit{True} \land \\ \quad \mathit{PilotCommand} = \mathit{Deploy}) \land \\ \quad \neg(\neg \mathit{PilotCommand} = \mathit{Deploy} \land \\ \quad \mathit{DoorStowed} = \mathit{False} \land \\ \quad \mathit{PilotCommand} = \mathit{Deploy})\end{array}$$

Figure 5. Determinism conjecture for Stowing

Closer inspection of these two conjectures show that they are invalid. For the reactive component, no outcome is specified if the hydraulic ram is exactly at the deployed position or the hinge is at exactly 80 degrees. For the state-based component, it is not clear to which state the machine should move while in the *Stowing* state if the pilot requests deployment at the same moment as the doors become stowed. Depending on the behaviour specified within the *Idle* and *Deploying* states (these could be super-states encapsulating more detailed behaviour) taking one transition over another may have a serious impact on the behaviour of the system.

The conjectures that arose from the case studies were proven using CADi$\mathbb{Z}$ [26, 28]. CADi$\mathbb{Z}$ is a general purpose Z type checker and theorem prover that allows a user to interactively browse, type check and perform

$$
\begin{array}{|l}
\hline
\textit{DoorDeployedOperation} \\
\textit{RamPosition}? : \mathbb{N} \\
\textit{Hinge}? : \mathbb{N} \\
\textit{DeployLock}? : \textit{Activated} \mid \textit{Deactivated} \\
\textit{DoorDeployed}! : \textit{Boolean} \\
\hline
(\textit{FullyRetracted} \leq \textit{RamPosition}? \leq \textit{FullyExtended} \land 0 \leq \textit{Hinge}? \leq 90) \Rightarrow \\
((\textit{RamPosition}? \geq \textit{DeployedPosition} \land \textit{Hinge}? \geq 80 \land \textit{DeployLock}? = \textit{Activated}) \land \\
\quad \textit{DoorDeployed}! = \textit{True}) \lor \\
((\textit{RamPosition}? < \textit{DeployedPosition} \lor \textit{Hinge}? < 80 \lor \textit{DeployLock}? = \textit{Deactivated}) \land \\
\quad \textit{DoorDeployed}! = \textit{False}) \\
\hline
\end{array}
$$

Figure 6. Z operation schema for DoorDeployed

proofs upon a Z specification. CADi$\mathbb{Z}$ allows general purpose proof tactics to be written in a lazy functional notation [27], these can be invoked from within a CADi$\mathbb{Z}$ window and applied to any proof obligation on the screen. This level of proof tactic re-use is possible because of the consistent structure of the completeness and determinism conjectures. A proof tactic has been written to prove the determinism and completeness conjectures. The tactic first simplifies the constraint and then calls either the SMV [3] model checker (most suitable for predicates involving finite types) or a simulated annealing based constraint solver [4] (used for counter-example generation for predicates involving mixed numeric types including integers and reals). If the check fails, a counter-example is given. This information has been found to be extremely valuable when tracking the error in the specification. Conjectures that can not be automatically discharged in this way involve a mixture of enumerated and infinite numeric types. This combination is not currently supported by the constraint solvers. Restricting the numeric types to sensible finite ranges allows these constraints to be checked automatically.

The healthiness checks that failed have been found to be due to areas of omission or ambiguity in the original system requirements that were not detected through review or animation. This illustrates that there is much benefit to be obtained by verifying relatively simple properties of the specifications and the high level of automation ensures that the only additional work required is that of locating the errors in the specification based on the counter-examples. This work would otherwise be done at a later stage with perhaps less illuminating data to work from.

The high level of automation allows the analysis to be re-run each time the specification is changed, further reducing the cost of rework. Although the interactive version of CADi$\mathbb{Z}$ allows the proof effort to be automated it still requires some repetitive work from the user to load the generated Z file and select each proof obligation in turn to apply the proof tactics. Work is underway to encapsulate the functionality of CADi$\mathbb{Z}$ within an API. This will provide the opportunity to fully integrate the formal analysis into specification modelling tools. Instant feedback on the properties being analysed can then be presented to the user using the same format as the original specification. The details of the analysis would be recorded (as the intermediate specification and proofs in Z) for review by engineers with the relevant formal methods skills.

5. Automatic test case generation

The formal specification describes each atomic action defined by the requirements specification. These operations can be used as basic test specifications. If data can be found to satisfy these constraints, the results of applying the data to the implementation can be used to gain confidence in its correctness with respect to the specification. The success of testing depends on the ability to select data that demonstrates the presence of a fault in the program. Category-partitioning [21] is a method of deriving tests based on a formal specification and testing heuristics based on common error types. Test data generated for the partitioned specification is then assumed to have a greater chance of detecting errors in the implementation (at least errors of the type used to formulate the testing heuristic). This approach was first applied to formal specifications by Ostrand and Balcer [21] and has been developed and applied to the formal specification notation Z by Stocks and Carrington [25].

The category-partition method is based on the theory of equivalence classes [8]. The input domain of the test specification is partitioned into sets of data that exhibit the same behaviour in the specification. If the equiva-

$$\begin{array}{l}
\textit{DoorDeployedOperationPartition1} \\
\hline
RamPosition? : \mathbb{N} \\
Hinge? : \mathbb{N} \\
DeployLock? : Activated \mid Deactivated \\
DoorDeployed! : Boolean \\
\hline
(FullyRetracted \leq RamPosition? \leq FullyExtended \land 0 \leq Hinge? \leq 90) \Rightarrow \\
(\boxed{\mathbf{RamPosition? = DeployedPosition}} \land \\
(RamPosition? \geq DeployedPosition \land Hinge? \geq 80 \land DeployLock? = Activated) \land \\
\quad DoorDeployed! = True) \lor \\
((RamPosition? < DeployedPosition \lor Hinge? < 80 \lor DeployLock? = Deactivated) \land \\
\quad DoorDeployed! = True)
\end{array}$$

$$\begin{array}{l}
\textit{DoorDeployedOperationPartition2} \\
\hline
RamPosition? : \mathbb{N} \\
Hinge? : \mathbb{N} \\
DeployLock? : Activated \mid Deactivated \\
DoorDeployed! : Boolean \\
\hline
(FullyRetracted \leq RamPosition? \leq FullyExtended \land 0 \leq Hinge? \leq 90) \Rightarrow \\
(\boxed{\mathbf{RamPosition? = DeployedPosition + 1}} \land \\
(RamPosition? \geq DeployedPosition \land Hinge? \geq 80 \land DeployLock? = Activated) \land \\
\quad DoorDeployed! = True) \lor \\
((RamPosition? < DeployedPosition \lor Hinge? < 80 \lor DeployLock? = Deactivated) \land \\
\quad DoorDeployed! = True)
\end{array}$$

$$\begin{array}{l}
\textit{DoorDeployedOperationPartition3} \\
\hline
RamPosition? : \mathbb{N} \\
Hinge? : \mathbb{N} \\
DeployLock? : Activated \mid Deactivated \\
DoorDeployed! : Boolean \\
\hline
(FullyRetracted \leq RamPosition? \leq FullyExtended \land 0 \leq Hinge? \leq 90) \Rightarrow \\
(\boxed{\mathbf{RamPosition? > DeployedPosition + 1}} \land \\
(RamPosition? \geq DeployedPosition \land Hinge? \geq 80 \land DeployLock? = Activated) \land \\
\quad DoorDeployed! = True) \lor \\
((RamPosition? < DeployedPosition \lor Hinge? < 80 \lor DeployLock? = Deactivated) \land \\
\quad DoorDeployed! = True)
\end{array}$$

Figure 7. Test partitions for DoorDeployed

lence class hypothesis is assumed to hold in the implementation, only a selection of data from each equivalence class is needed to show that the implementation satisfies the specification for all data in that class.

As an example, the operation defining the *DoorDeployed* function (from figure 2, but corrected based on the completeness analysis described above) will now be partitioned to verify that the boundary used to define when the hydraulic ram is in the deployed position is correctly implemented in the code. The Z specification

$$\begin{aligned}&\forall X, Y : \mathbb{N} \bullet X \geq Y \Leftrightarrow \\ &\quad X = Y \vee \\ &\quad X = Y + 1 \vee \\ &\quad X > Y + 1\end{aligned}$$

Figure 8. Generic test heuristic for $\geq$

of the operation is given in figure 6. The *?* and *!* decorations are used to distinguish between the input and output parameters to the schema. Based on the assumption that errors often occur on or around boundaries, applying a boundary value analysis partitioning strategy would result in the partitions shown in figure 7. The additional constraints added by the partitioning are shown in bold. These partitions together with those generated for the condition where the hydraulic ram is not in the deployed position, from the second guard in figure 2, fully test the boundary.

The category-partition method has been automated as extensions to the CADi$\mathbb{Z}$ theorem prover. Partitioning heuristics are specified as lemmas and general-purpose proof tactics are used to apply the heuristics via the graphical user interface. The predicate to be partitioned is highlighted and a proof tactic invoked via a menu option which automatically introduces the partitioning heuristic into the operation conjecture, instantiates the generic heuristic with the operands of the predicate and simplifies the whole conjecture to reveal a disjunction of partitions. Each partition is then converted into a separate schema operation. The lemma used to create the partitions shown in figure 7 is given in figure 8. The user is also given the opportunity to amend the supporting English language description of the operation being partitioned, to include for example the rationale behind using the particular partitioning strategy.

The method of specifying the heuristics as lemmas, stored in a separate Z library file, which are then 'cut' into the operation has several important advantages. Properties of the heuristics themselves can be proven (e.g. that the partitions together maintain the state-space of the operation). If more heuristics are required (e.g. based on common errors specific to the system under development), they can be added without making a change to the software itself. The test specifications can be instantiated with test data via a similar mechanism to the test partitioning. The test specification is highlighted and an option called from within a CADi$\mathbb{Z}$ menu. A proof tactic is then automatically applied that simplifies the constraint and applies either the SMV model checker or simulated annealing constraint solver to generate a set of data satisfying the test specification.

Once the test data has been generated, CADi$\mathbb{Z}$ produces a corresponding AdaTEST [16] test script. AdaTEST provides a harness for automating the execution, checking and documentation of tests for software written in the Ada language. AdaTEST can also record the structural code coverage achieved by running the tests. Manually producing these test scripts, consumes a large proportion of the test engineers time. By automating this step, effort that was previously required for test implementation can now be redirected towards more rigorous test design. The generated test scripts also include the informal text derived from the original requirements and annotated with the test rationale during partitioning. This text is automatically included in the AdaTEST test results file and provides the traceability between any suspected fault in the program, the requirement under test and the heuristic used in designing the test.

The test specifications for the case study were first partitioned to give Modified Condition/Decision Coverage[7] (MC/DC) of each operation. Additional tests were then generated based on other heuristics, such as boundary value analysis. If full MC/DC (as mandated by certification standards such as D0-178B [23]) was not achieved by running these tests, it was assumed that the untested code represented refinements in the design or potential errors. Additional manual test effort then concentrated on writing tests for and reviewing these potentially problematic areas of code. The targeting of testing resources in this way was made possible by the high amount of automation achieved in generating the requirements covering tests.

6. Results and Evaluation

A summary of the specification validation and testing work performed for the thrust reverser case study is shown in figure 9. The numbers include only automatically generated proof obligations and tests and the requirements errors found by discharging the proofs. An activity is said to be automated if it requires at most a single interaction from the user to perform (e.g. a proof is discharged by selecting a completeness conjecture and choosing "Completeness Check" from the on-screen menu). Consequently these activities take very little time to perform. Many of the proof obligations stretched over 4 pages of formal text. Each of these would have taken an engineer a significant amount of time to prove or disprove. On a Pentium II 400 MHz computer running the linux operating system, the largest of these proofs took no more than a second to discharge.

The activities in the process described in this paper that have so far been automated are: automatic generation of a formal specification and associated healthiness

[7]MC/DC is achieved by showing that each condition within a decision can independently affect that decision's outcome[23].

State-based components:	
State-machines	9
States	48
Transitions	112
Z operations	112
Specification validation proofs	74
Automatically discharged proofs	74
Requirements errors found	18
Automatically generated tests	262
Reactive components:	
Tables	34
Definition/Guard pairings	84
Z operations	34
Specification validation proofs	62
Automatically discharged proofs	52
Requirements errors found	18
Automatically generated tests	237

Figure 9. Summary of results

property proof obligations from a Statechart modelled using STATEMATE, automatic proof or generation of a counter-example for PFS (Statechart and tabular) completeness and determinism healthiness properties, automatic partitioning of formally specified operations (derived from Statecharts of tabular requirements) into test cases based on pre-defined heuristics and the automatic generation of test data for the partitions and associated AdaTEST test script. Activities that we believe can be automated or are already in the process of being automated are; automatic generation of a formal specification and associated healthiness proof obligations from PFS tabular requirements (given a consistent form of recording and managing these requirements), automatic identification of the healthiness property proof obligations within the Z specification and application of the appropriate proof tactics and the automatic selection of partitioning strategies to generate test sets to satisfy particular structural specification coverage criteria.

The results show that a significant number of requirements errors were detected for little additional effort. All the requirements errors detected using this method manifested themselves as either non-determinism or incompleteness of the specification (as would be expected based on the nature of the checks). On analysis of these errors we discovered two distinct causes. The first type of error was the result of a mis-interpretation of some higher level requirements that resulted in an incorrect specification with respect to these requirements. These errors accounted for the greater proportion of total requirements errors found. The second type of error was non-determinism or incompleteness as the result of some omission or ambiguity in the higher level requirements. Although these errors were less frequent (potentially because the analysis was not specifically targeted at validating the higher level requirements) they were deemed to be very valuable.

A far greater number of tests were generated than would have been written for a specification of this size using the traditional process. The number of tests that can be written are typically restricted by the time it takes to design, implement and evaluate each test, in the process described here much of this effort has been automated, greatly reducing the amount of effort per test case. When the analysis or test revealed an error, the time taken to review and rework the error varied according to the nature of the problem. However, the impression amongst those involved was that the counter-example information and supporting informal text greatly contributed to the process of tracking down the errors in the requirements. It was also noted that as the case study progressed the number of requirements errors being detected decreased significantly. The feedback from the formal analysis was thought to have influenced the style of requirements specification, i.e. the author of the requirements was consciously writing specifications to meet the healthiness conditions specified by the PFS methodology.[8]

In [18], Knight presented the following criteria for industrial acceptance of formal methods. Based on the evidence from the case study and experience of working with our industrial partners we can now assess the industrial suitability of our work along similar lines.

1. Formal methods must not detract from the accomplishments achieved by current methods

2. Formal methods must augment current methods so as to permit industry to build “better” software

3. Formal methods must be consistent with those current methods with which they must be integrated

4. They must be compatible with the tools and techniques that are currently in use

These criteria emphasise the need to develop formal methods for the types of practical tools and notations used in industry and also for formal methods to complement and not preclude existing practices. It is the authors’ opinion that the work described in this paper has gone some way to satisfying these criteria, although admittedly for a particular domain and set of V&V activities. This was accomplished by basing the formal

[8]The final validation of the system will tell whether the requirements indeed improved or whether errors were instead being introduced into areas not covered by the healthiness conditions.

analysis and test case generation activities on an automatically generated formal representation of the intuitive requirements specifications, written using existing modelling tools. In addition, the activities performed here complement methods already in use such as review and animation. As such they can be seen as natural extensions to the existing modelling process.

Formal specifications are typically very sensitive to change. However, due to automation, the formal specifications can be re-generated and verified whenever a change in the requirements occurred, at little extra cost. The intuitive engineering requirements remained the first class citizens of the process and the standard interface to the engineers. The ongoing extensions to CADiℤ to provide a ‘silent’ interface via an API will allow modelling environments such as Statemate to exploit an intermediate formal representation of the requirements to perform checks and generate tests while hiding the details of the analysis from the engineer. This would further encourage an iterative development of the requirements (i.e. do not pass onto the coding phase until the requirements have been properly validated) and increase the efficiency of the test generation process.

7. Conclusions and Future Work

In this paper we described our experiences in integrating formal methods into an industrial software development process. Intuitive engineering notations were translated into intermediate formal specifications which formed the basis of automated proof and test case generation activities. The high level of automation was made possible by restricting the work to a particular domain (discrete engine control requirements) and a tight subset of an otherwise highly expressive formal notation (Z). The automated analysis and tests allowed a significant amount of errors to be detected earlier than would have been possible had a manual, ad-hoc approach been taken.

The findings support other work [5, 13, 1] that has similarly used formal semantics of engineering notations to facilitate an effective approach to verification. The work presented here contributes to this field by showing that general purpose formal analysis tools such as CADiℤ and SMV can be used to support automated analysis and test generation based on different engineering notations given a suitable translation from the notations to a formal specification.

In developing these techniques we have identified the following generic process for applying formal methods to engineering notations for automated V&V.

1. **Use intuitive Engineering notations with fixed semantics:** Record and maintain the requirements specifications in notations most suitable for the domain and whose semantics can be formally specified.

2. **Couch healthiness conditions as mathematical constraints:** Identify “healthiness properties” that should be common to all specifications e.g. completeness and determinism. Based on the formal semantics of the notations, couch these properties as mathematical constraints. Automate the translation if possible.

3. **Formally specify the behaviour under test:** Define a translation between the original notation and a formal specification of the properties under test based on the formal semantics. Automate the translation if possible.

4. **Exploit existing tool-support:** Apply a combination of automated proof, model checking and constraint solving to analyse the healthiness properties, to generate test specifications and to generate test data.

5. **Complement the formal specification and tests with informal text:** To aid the review and documentation of the analysis and test results, informal text descriptions should be generated and maintained to describe the formal specification of the healthiness properties and tests.

Ongoing work aims to increase the level of automation and integration of the formal techniques into existing specification modelling environments with the development of a CADiℤ API. In addition to this, we also hope to expand the generic process and the toolset to cover more areas of the verification process. In particular, the refinement of the intermediate formal specification into program annotations that can be used to discharge correctness proofs on the code and the automatic efficient sequencing of test cases to reduce the amount of effort required to physically run the generated tests. Other work will concentrate on developing further the constraint solving abilities of the CADiℤ theorem prover. This will allow a wider range of specifications and properties to be automatically verified than the current system.

8. Acknowledgements

This work was funded by the High Integrity Systems and Software Centre, Rolls-Royce Plc. The PFS project is funded by the UK Ministry of Defence. The authors would also like to thank Steve King for his valuable review comments.

References

[1] Jim Armstrong. Industrial integration of graphical and formal specifications. *Systems Software*, 40:211–225, 1998.

[2] Boris Beizer. *Software Testing Techniques*. Thomson Computer Press, 1990.

[3] Sergey Berezin. The SMV web site. http://www.cs.cmu.edu/~modelcheck/smv.html/, 1999. The latest version of SMV and its documentation may be downloaded from this site.

[4] John Clark and Nigel Tracey. Solving constraints in LAW. LAW/D5.1.1(E), European Commission - DG III Industry, 1997. Legacy Assessment WorkBench Feasibility Assessment.

[5] Nancy Day, Jeffrey Joyce, and Gerry Pelletier. Formalization and analysis of the seperation minima for aircraft in the north atlantic region. *Proceedings of the Fourth NASA Langley Formal Methods Workshop*, pages 35–49, September 1997.

[6] David Dill and John Rushby. Acceptance of formal methods: Lessons from hardware design. *IEEE Computer*, 29(4):23–24, April 1996.

[7] Andy Galloway, Trevor Cockram, and John McDermid. Experiences with the application of discrete formal methods to the development of engine control software. *Proceedings of DCCS '98. IFAC*, 1998.

[8] John B Goodenough and Susan L Gerhart. Towards a theory of test data selection. *IEEE Transactions On Software Engineering*, 1(2):156–173, June 1975.

[9] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[10] David Harel, Hagi Lachover, Amnon Naamad, Amir Pnueli, , Michael Politi, Rivi Sherman, Aharon Shtull-Truaring, and Mark Trakhenbrot. Statemate, a working environment for the development of complex reactive systems. *IEEE Transactions On Software Engineering*, 16:403–414, 1988.

[11] David Harel and Amnon Naamad. The Statemate semantics of statecharts. *IEEE Transactions On Software Engineering And Methodology*, 5(4):293–33, Oct 1996.

[12] Kathryn L. Heninger. Specifying software requirements for complex systems: New techniques and their applications. *IEEE Transactions on Software Engineering*, 6(1), 1980.

[13] Constance Hietmeyer, Bruce Labaw, and Daniel Kiskis. Consistency checking of SCR-style requirements specifications. *In Proceedings of International Symposium on Requirements Engineering*, pages 56–63, 1995.

[14] M.G. Hinchey and J. P. Bowen (editors). *Applications of formal methods*. Prentice-Hall, 1995.

[15] C. Michael Holloway and Ricky W. Butler. Impediments to industrial use of formal methods. *IEEE Computer*, 29(4):25–26, April 1996.

[16] Information Processing Ltd., Bath, UK. *AdaTEST 95 User Manual*, June 1997.

[17] Steve King, Jonathon Hammond, Rod Chapman, and Andy Pryor. The value of verification: Positive experience of industrial proof. In J. Wing, J. Woodcock, and J. Davies, editors, *FM'99: Formal Methods*, volume 1709 of *Lecture Notes in Computer Science*, pages 1527–1545. Springer-Verlag, September 1999.

[18] John Knight, Colleen DeJong, Matthew Gibble, and Luis Nakano. Why are formal methods not used more widely? *Proceedings of the Fourth NASA Langley Formal Methods Workshop*, pages 1–12, September 1997.

[19] John McDermid, Andy Galloway, Simon Burton, John Clark, Ian Toyn, Nigel Tracey, and Samuel Valentine. Towards industrially applicable formal methods: Three small steps, one giant leap. *Proceedings of the International Conference on Formal Engineering Methods*, October 1998.

[20] Erich Mikk, Yassine Lakhnech, Carsta Petersohn, and Michael Siegel. On the formal semantics of statecharts as supported by Statemate. *Proceedings Of The Second Northern Formal Methods Workshop*, July 1997.

[21] Thomas J Ostrand and Marc J Balcer. The category-partition method for specifying and generating functional tests. *Communications of the ACM*, 31(6):676–686, June 1988.

[22] Martyn Ould. Testing - a challenge to method and tool developers. *Software Engineering Journal*, 39:59–64, March 1991.

[23] RTCA. *RTCA DO-178B, Software Considerations in Airborne Systems and Equipment Certification*, 1992.

[24] J. M. Spivey. *The Z Notation: A Reference Manual, second edition*. Prentice Hall, 1992.

[25] Phil Stocks and David Carrington. Test template framework: A specification-based case study. *Proceedings Of The International Symposium On Software Testing And Analysis (ISSTA'93)*, pages 11–18, 1993.

[26] I. Toyn. Formal reasoning in the Z notation using CADiℤ. *2nd International Workshop on User Interface Design for Theorem Proving Systems*, July 1996.

[27] Ian Toyn. A tactic language for reasoning about Z specifications. In *Proceedings of the Third Northern Formal Methods Workshop, Ilkley, UK*, September 1998.

[28] Ian Toyn. The CADiℤ web site. http://www.cs.york.ac.uk/~ian/cadiz/, 1999. The latest version of CADiℤ and its documentation may be downloaded from this site.

[29] Sam Valentine. *Modeling Statemate Statecharts In Z (DCSC/TR/98/15)*. Dependable Computer Systems Centre (DCSC), University of York, October 1998.