Towards Automated Unit Testing of Statechart Implementations

Simon Burton

Simon.Burton@cs.york.ac.uk

Department of Computer Science, University of York, UK

August 2, 1999

Abstract

This report describes an automated method of unit test design based on requirements specified in a subset of the statechart notation. The behaviour under test is first extracted from the requirements and specified in the Z notation. Existing methods and tools are then applied to this specification to derive the tests. Using Z to model the requirements and specify the tests allows for a deductive approach to verifying test satisfiability, test result correctness and certain properties of the requirements. An examination of the specification coverage achieved by the tests is provided and the report concludes with an evaluation of the work to date and a set of directions for future work.

Contents

1 Introduction **3**

2 A subset of statecharts **4**
2.1 States . . . 5
2.2 Transitions . . . 6
2.3 Statechart computations . . . 6

3 Modelling the behaviour to be tested **7**
3.1 Making the behaviour under test explicit . . . 7
3.1.1 Introduction . . . 7
3.1.2 Specifying transitions terminating at non-basic states . . . 8
3.1.3 Specifying transitions originating from non-basic states . . . 8
3.1.4 Specifying the completeness assumption . . . 9
3.1.5 Specifying priority resolution . . . 11
3.2 Specifying the behaviour under test in Z . . . 12
3.3 Remaining specification looseness . . . 17
3.4 Checking state completeness and determinism . . . 17

4 Generating the tests **18**
4.1 Testing goals and assumptions . . . 18
4.2 The category partitioning method . . . 19
4.3 Tool support . . . 21
4.3.1 Automated test partitioning . . . 21
4.3.2 Automated proof tactics . . . 21

5 Test coverage and adequacy **23**
5.1 Specification statement coverage . . . 23
5.2 Specification MC/DC coverage . . . 23
5.3 Beyond structural coverage . . . 25

6 Conclusions and future work **26**

7 Acknowledgements **27**

8 Appendix: Z specification for ThrustLimitation **30**
8.1 Test Summary . . . 30
8.2 Z Specification . . . 30
8.2.1 Auxiliary definitions . . . 30
8.2.2 Status . . . 32
8.2.3 Transition Operations/Test cases . . . 33

1 Introduction

Statecharts [1] are becoming increasingly popular as a means of specifying safety–critical systems. It is therefore necessary to develop a systematic and rigorous approach to verifying that these systems conform to their specifications. Unit testing is performed on safety–critical software to verify its functional behaviour and as a means of obtaining the structural coverage metrics mandated by certification standards and guidelines such as DO-178B [2] and Defence Standard 00–55 [3]. Although commercially available tools exist [4, 5] to automate the collection of structural coverage metrics, unit testing still consumes a large proportion of total development costs. Furthermore, the current approach to unit testing is often *ad-hoc* and the necessity to check the functional behaviour of the system against its specification can be overshadowed by the more quantifiable targets of structural coverage. As a result, confidence in the quality of the software is compromised and the significance of unit testing within the context of demonstrating safety is reduced. This report describes work in progress on a Rolls–Royce funded project to increase the value of unit testing by reducing costs through automation and increasing the confidence in the tests through a more rigorous approach to unit test design.

Statecharts are a rich notation that allow complex system behaviour to be specified in concise diagrams. This efficiency is made possible through a complicated syntax and semantics. As a result, much of the system behaviour is not immediately obvious from the diagrams. This poses a problem when designing automated functional testing techniques based on statechart specifications. They would require an understanding of the complex statechart semantics in order to interpret the behaviour implied by the diagrams and generate tests based on this behaviour. Although statecharts are expressive, it is unlikely that in practical situations they will be the only notation used to specify a system or even that they will be used consistently across projects. It is therefore undesirable to invent and implement testing mechanisms for each notation and associated semantics. Instead, a means of allowing existing general purpose testing techniques to be applied to statecharts (and other notations) is required. In this report we show how existing testing techniques can be applied to statechart specifications by explicitly specifying the behaviour under test in an intermediate notation thus abstracting from the statechart semantics. Notation specific techniques are then only needed to provide the translation between the graphical specifications and the intermediate notation.

The formal notation Z [6] is used to specify the statechart behaviour (the intermediate notation mentioned above). The use of a formal notation allows proof to be used as a means of reasoning about properties of the specification and tests. Furthermore, techniques exist [7, 8] for the rigorous derivation of tests from Z specifications with the potential for automation. Restricting the testing procedures to a particular domain (here, embedded safety–critical systems) increases the potential for automation. Commonly occurring constraints in the specification can be identified and dedicated heuristics (e.g. for proof or test data generation) can be developed. The proposed process is summarised in figure 1.

This report illustrates through example our method of generating test cases from statechart specifications via an intermediate formal specification. It describes the re-expression

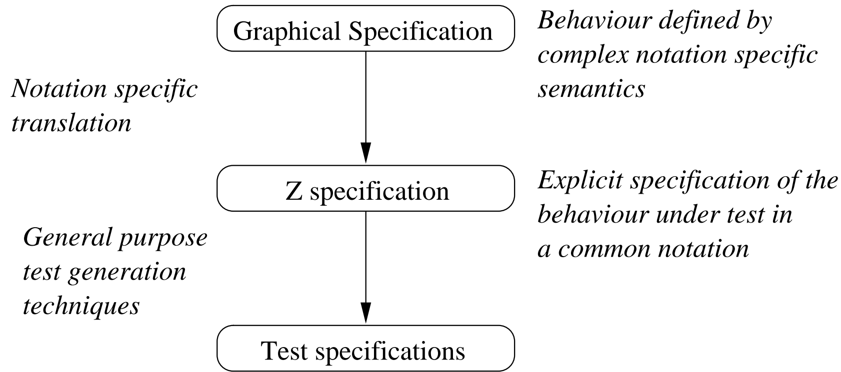

Figure 1: Testing complex graphical notations

of a statechart specification in Z, the application of generic test generation procedures to the Z specification and discusses adequacy criteria for testing statechart implementations. The report also includes a short discussion on the extension of the techniques to handle the problem of test sequence generation to be addressed as part of future work.

The report is structured as follows. The next section introduces the subset of statecharts considered in this report. Section three describes how the behaviour under test is extracted from the statechart specification and described as a formal specification. Section four presents the automated method of generating tests from the specification and the use of automated proof tactics to verify properties of the tests and test data. Section five addresses issues of test adequacy and coverage. The final section presents an evaluation of the results so far and summarises directions for future work.

2 A subset of statecharts

Statecharts allow complex reactive systems to be described in concise diagrams. This is made possible through the use of hierarchical state machines, an extended transition operation syntax and the ability to model concurrency. This section describes those parts of the notation that are relevant to this report. At present only a subset of the statechart notation is covered by the techniques described here. Although as the techniques develop this subset will be expanded, it is unrealistic to expect the whole of the language to be handled by such techniques. Also, it may be desirable to apply restrictions on the use of the certain constructs to prevent misuse of the expressiveness of the language (e.g. Safecharts [9]).

An example statechart is given in figure 2 and describes a portion of a jet engine electronic controller specification for a commercial airline application (a typical use of statecharts in the embedded control systems domain). In particular the statechart describes the cancellation of the thrust limitation mechanism following an inadvertent deployment (e.g due to a mechanical fault) of the engine thrust reverser doors. The thrust limitation is

removed once the thrust reverser doors are locked back in place and the pilot returns the throttle to the idle position. The limitation system can then return to its idle state. If the aircraft is on the ground, the thrust limitation is removed immediately to allow for reverse thrust if needed. The limitation system is not reset until the doors are finally locked.

2.1 States

Each statechart has a state hierarchy. There is one state which is highest in the hierarchy and each state may contain a set of sub-states which are lower in the hierarchy than their surrounding superstate. In the example in figure 2, *ThrustLimitation* is the highest (root) state in the hierarchy, *Idle* and *InadvertentDeploy* are its immediate sub-states. States which contain no sub-states are basic states (e.g. *Idle*, *Unstowed* and *InAir*).

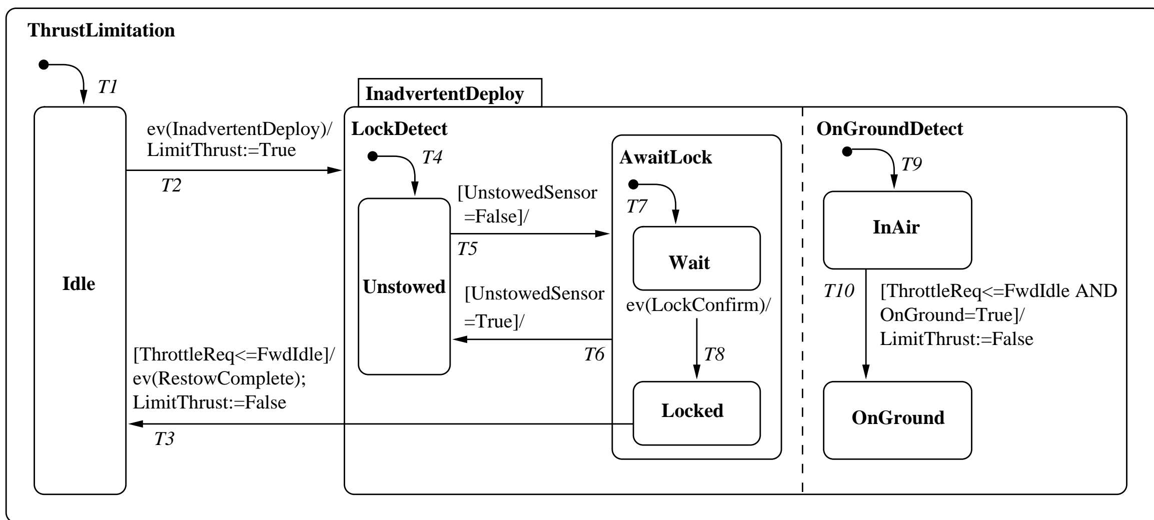

Figure 2: Example Statechart - Thrust Limitation

Superstates can be either AND-states or OR-states. When an OR-state is active, one and only one of its immediate sub-states is active at any time. The system moves between these sub-states sequentially according to the order in which the transitions connecting the states are triggered. In figure 2, *ThrustLimitation* and *LockDetect* are both examples of OR-states. If a transition terminates at a basic state, then that state as well as all of its ancestors in the state hierarchy are active. If the transition terminates at the border of an OR-state then that state and the destination of the default transition within that state become active. *T7* is an example of a default transition in figure 2.

AND-states are used to model concurrency and allow sets of transitions in parallel components to be simultaneously enabled. Whenever an AND-state is active all of its immediate sub-states are also active. *InadvertentDeploy* is the only AND-state in the example. Whenever *InadvertentDeploy* is active both *LockDetect* and *OnGroundDetect* are active.

2.2 Transitions

Control is passed between the states via the transitions. Each transition consists of an event expression, a guarding condition and an action, each of which is optional. An event expression is a set of predicates combined using standard logical operators. Each predicate evaluates to true if the event that it represents has been generated either by the environment or by one of the system elements in the previous execution step. Guarding conditions are formed by logical and arithmetic conditions on data-items and conditions. Actions consist of a combination of event broadcast expressions and assignments to conditions and data-items. The syntax used for transition expressions here is: *ev(event) [guarding conditions]/ev(event);actions.* The values of data-items and conditions can be either stored as local variables of the statechart or passed in and out via a mechanism akin to procedure parameter passing.

If two transitions are enabled simultaneously, the transition with the highest scope is given priority. The scope of a transition is defined as the lowest common OR-state that is an ancestor of all its source and target states. E.g. in figure 2, transitions *T3* and *T6* may become simultaneously enabled. The scope of transition *T3* is *ThrustLimitation*, higher than *T6*'s scope (*LockDetect*) and therefore transition *T3* would be taken.

2.3 Statechart computations

The computation that is performed by the statechart is represented by a series of statuses. A status consists of a set of currently active states, the set of events generated in the last computation step, and the set of values for the conditions and data-items local to the statechart (analogous to variables local to a procedure in imperative programming languages). When we come to test the implementation of the statechart we will verify that these computations are performed correctly.

The statuses defining the computation are collected into sequences of distinct steps. During each step the status is transformed by a set of transitions. When a set of events is received from the environment or data values are changed by the environment, a set of non-conflicting transitions activated as a result of the changes in the status is taken. The statechart semantics used here (relating to the implementation of the statechart semantics in the Statemate tool from i-Logix [10]) contains two alternative models of time, each of which has an effect on the set of transitions that are triggered during each step. The time model used must be taken into account when modelling the behaviour of the statechart and designing tests based on this behaviour.

The synchronous time model assumes that the system executes a single step every time unit, reacting to changes made to the status either by the environment or by the system itself since the last step. The changes to the status caused by the transitions enabled by the current status are processed in the next step along with any external changes to the status that occurred during the step.

The asynchronous time model assumes that the system reacts whenever an external change occurs allowing for several steps to be taken at a single point in time to form a

superstep. Within a superstep, the transitions that are triggered as a result of the external changes to the status may change the status in such a way that a new set of transitions become enabled and are spontaneously taken. A sequence of steps are taken until no more transitions are enabled; the system has reached a stable state. In both models, the execution of a step is assumed to take zero time since external changes to the status have no effect during a step.

3 Modelling the behaviour to be tested

3.1 Making the behaviour under test explicit

3.1.1 Introduction

The semantics of statechart transitions is rather involved and imposes constraints on the behaviour which are left implicit in the diagrams. However in order to mechanically derive accurate and complete tests using generic test automation techniques this behaviour must be made explicit in any formal specification on which we base our tests. For the subset of statecharts under consideration here, this implicit behaviour occurs in the following four situations.

- **Transitions terminating at a non-basic state.** If a transition terminates at a non-basic state a compound transition is formed by linking the original transition with the default transition of the target state. The composition is continued until a default transition terminates at a basic state.

- **Transitions originating from non-basic states.** If a transition originating at a non–basic state is enabled, then it must be possible to take the transition if any one of the sub-states of its source state is enabled.

- **Priority resolution mechanism.** If two transitions are enabled simultaneously, the transition with the higher scope is given priority. The scope of a transition is defined as the lowest common OR-state that is an ancestor of all its source and target states.

- **Completeness assumption.** The operational semantics of statecharts specify that if a state is active and no transitions are enabled in a step, events generated in the previous step are consumed and the active state is maintained.

Extended Hierarchical Automata (EHA) [11] are intended to serve as an intermediate formalism for tools linking to the Statemate tool [12], a commercially available tool from i-Logix for specifying, analysing and animating statechart specifications. They resolve some of the issues presented above by employing a simpler operational semantics whilst maintaining the expressiveness of the original notation. EHA are used here to demonstrate

a method for eliciting the behaviour to be tested and a subsequent translation of this behaviour to the Z notation.

EHA are a useful notation for visualising the complex behaviour of the statecharts but still leave certain areas of the operational semantics implicit and are therefore not sufficient for our automated test generation purposes. In EHA, each transition has associated with it a source and target restriction set. However, these sets do not fully specify which states may form the source and target sets of the transitions. Some changes are therefore made to the original definition of EHA [11] to make more of the behaviour explicit and simplify even further the semantics of the graphical notation.

3.1.2 Specifying transitions terminating at non-basic states

Although in the graphical notation of statecharts, transitions may terminate at non-basic states, the execution of a *step* always terminates in a set of basic states. These states are determined by default transitions (see section 2). The behavioural semantics of statecharts describes this behaviour as a compound transition, combining the original transitions (terminating at the state boundary) and the default transition of the target state. The actions of the original transition and default are combined and assumed to be performed simultaneously. It may be necessary to combine several default transitions depending on whether the default terminates at another non-basic state or whether an AND-state appears somewhere in the combination chain (in which case the default of each component state of the AND-state is considered).

The combination of defaults is modelled in the extension of EHA used here by conjoining any actions on the defaults with the action of the original transition and adding the targets of all defaults to the target restriction set. If this process is repeated for each transition terminating at a state boundary the default behaviour is completely specified within the updated transition specifications and the default transitions need no longer be considered.

3.1.3 Specifying transitions originating from non-basic states

In standard EHA, the source restriction set is empty for transitions whose source is a super-state. This is not only counter-intuitive to the use of the set in other cases (the set usually has the meaning: the transition can trigger only when these states are active) but this also leaves the transition loosely specified without the intervention of an interpretation of the semantics yet again. For example, transition *T6* in figure 2 would be given an empty source restriction set. This is not an adequate input to mechanical test generators as it is not clear whether it is sufficient to test the transition when just one of its sub-states is enabled or whether to test both cases. This behaviour can be completely specified by creating a new transition for each possible sub-state from which the transition can originate (these sub-states are added to the source restriction set).

The EHA for figure 2 is given in figure 3, the dashed arrows indicate state refinement (as in the statechart sub-state mechanism). Where a state is refined by more than one automaton they are considered to run in parallel with one another. Transitions are shown

only in the lowest common OR-state that is an ancestor of all the transition's source and destination state. The full set of source and destination states are specified in the source and target restrictions. The source and target restrictions are given in table 1. For ease of reading, basic states are given in bold text. Note, that the source and target sets are fully specified with the exception of parallel state behaviour[1].

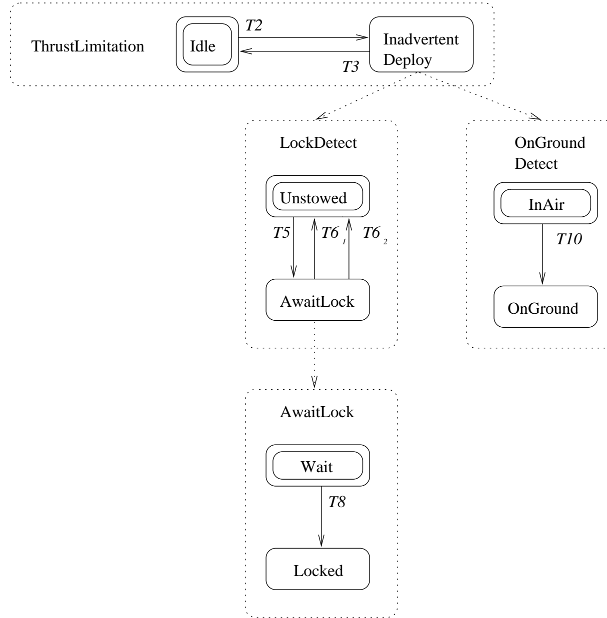

Figure 3: Extended Hierarchical Automaton for figure 2

3.1.4 Specifying the completeness assumption

The paragraphs above have shown how to specify all possible ways of exiting a basic state but not what happens if none of the transitions is chosen. This is implied in the operational semantics of statecharts by the completeness assumption. The completeness assumption can be modelled by adding a transition to each basic state. The guard of the new transition should be satisfied only when the guard of no other transition out of the state can be, i.e. it is the negation of the disjunction of the guards of all transitions originating at the state. The transition has no actions other than the consumption of all present events (from the statechart operational semantics – events are present for one step only, unless generated by the current step). The target restriction of the completing transition is the same as

[1] Values of parallel states are not specified at the moment. We assume that, in the example implementation under test, the state of a parallel component does not affect transitions in other components unless explicitly specified. By not specifying the possible values of the parallel components' states, this behaviour will not be propagated into the tests. Techniques such as Hieron's semi-independent communicating finite-state machine method [13] will be applied as part of future work to test interactions between parallel components.

Transition	**Source Restriction**	**Target Restriction**
T2	**Idle**	InadvertentDeploy, LockDetect, OnGroundDetect, **Unstowed, InAir**
T3	InadvertentDeploy, LockDetect, OnGroundDetect, AwaitLock, **Locked**	**Idle**
T5	InadvertentDeploy, LockDetect, OnGroundDetect, **Unstowed**	InadvertentDeploy LockDetect, OnGroundDetect, AwaitLock, **Wait**
$T6_1$	InadvertentDeploy, LockDetect, OnGroundDetect, AwaitLock, **Wait**	InadvertentDeploy LockDetect, OnGroundDetect, **Unstowed**
$T6_2$	InadvertentDeploy, LockDetect, OnGroundDetect, AwaitLock, **Locked**	InadvertentDeploy LockDetect, OnGroundDetect, **Unstowed**
T8	InadvertentDeploy, LockDetect, OnGroundDetect, AwaitLock, **Wait**	InadvertentDeploy, LockDetect, OnGroundDetect, AwaitLock, **Locked**
T10	InadvertentDeploy, LockDetect, OnGroundDetect, **InAir**	InadvertentDeploy, LockDetect, OnGroundDetect, **OnGround**

Table 1: Source and target restriction sets for the EHA of figure 2

its source restriction. Completing transitions only need to be added to basic states – an OR-state is modelled as the disjunction of the behaviours of its sub-states, if all sub-states are fully specified, then the disjunction of all its sub-states must by construction also be fully specified. If there are no transitions that exit the state the pre-condition is simply the source restriction set.

The pre–conditions of the completing transitions for the running example are given in table 2. Completing transitions must be added to the EHA before the priority resolution mechanism is specified, as the completing transitions themselves can be pre-empted by higher priority transitions. For example, the specification must include the declaration that the system remains in state *OnGround* unless transition *T3* (from the parallel component) is enabled. The method for specifying this behaviour is described next.

Transition	**Precondition**
CompleteIdle	$\neg pre\ T2$
CompleteUnstowed	$\neg pre\ T5$
CompleteWait	$\neg(pre\ T6_1 \vee pre\ T8)$ $\equiv \neg pre\ T6_1 \wedge \neg pre\ T8$
CompleteLocked	$\neg(pre\ T3 \vee pre\ T6_2)$ $\equiv \neg pre\ T3 \wedge \neg \text{pre}\ T6_2$
CompleteInAir	$\neg pre\ T10$
CompleteOnGround	Source restriction set for OnGround $\equiv$ {InadvertentDeploy, LockDetect, OnGroundDetect, OnGround}

Table 2: Completing transitions for the EHA of figure 2

3.1.5 Specifying priority resolution

Another drawback of EHA is that some priority resolution mechanism (albeit simplified) is still required. We would like our tests to verify that this part of the semantics is also maintained in the implementation and therefore the priority resolution mechanism will also be made explicit in the specification of the transitions. Where a transition can be pre-empted by another higher priority transition we strengthen the pre-condition (source restriction, guarding condition and event expression) on the lower priority transition with the negation of the pre-condition of the higher priority transition. This specifies that a transition can only be taken if its own guard is enabled and no guards of transitions of higher priority are also enabled. Conflicting transitions can be straightforwardly identified from the source restriction sets. A transition is in conflict with another if the elements (belonging to a particular parallel component) of one of the source restrictions are a subset of the others and both set of event expressions and guards can be enabled simultaneously. The priority is determined by the level in EHA hierarchy in which the transitions appear.

The guarding condition of the lower priority transition is updated accordingly. The final changes to the pre-conditions that must be made to specify the priority mechanism are given in table 3.

Transition	**Pre-condition**
T2	$pre\ T2$
T3	$pre\ T3$
T5	$pre\ T5$
$T6_1$	$pre\ T6_1$
$T6_2$	$pre\ T6_2 \land \neg pre\ T3$
T8	$pre\ T8 \land \neg pre\ T6_1$
T10	$pre\ T10 \land \neg pre\ T3$
CompleteIdle	$pre\ CompleteIdle$
CompleteUnstowed	$pre\ CompleteUnstowed$
CompleteWait	$pre\ CompleteWait \land \neg pre\ T6_1$
CompleteLocked	$pre\ CompleteLocked \land \neg pre\ T3$
CompleteInAir	$pre\ CompleteInAir \land \neg pre\ T3$
CompleteOnGround	$pre\ CompleteOnGround \land \neg pre\ T3$

Table 3: Priority resolution calculations for the EHA of figure 2

3.2 Specifying the behaviour under test in Z

As mentioned earlier in the report, the intermediate language used as input to the automated test generation techniques must be capable of explicitly representing the statechart transition behaviour (i.e. the behaviour summarised above in figure 2 and tables 1, 2 and 3). We have found Z [6] a convenient language for this purpose. Z is a model-based formal notation based on set theory and predicate logic. Furthermore, theorem proving tools exist that can be exploited to generate tests and verify properties of the tests and specification. Z has also been used by other authors to complement statechart specifications [14, 15] and to specify the statechart semantics [16, 17]. Here we shall use Z to formally specify the behaviour under test.

Statechart transitions can be modelled as operations over the status (see section 2). The operation is defined in terms of a pre-condition (a constraint over the set of source states, events, data values and inputs that form the transition trigger) and a post-condition (a constraint over the set of target states, generated events, updated data values and outputs that form the transition action).

The template for transition operation schemas is given in figure 4. *Status* is a schema defining currently active states (configuration), events, values of local conditions and data-items. *Parameters* is a schema defining the input parameters (as events, conditions or data-items) and the output parameters (as events, conditions or data-items). *SourceRestriction*

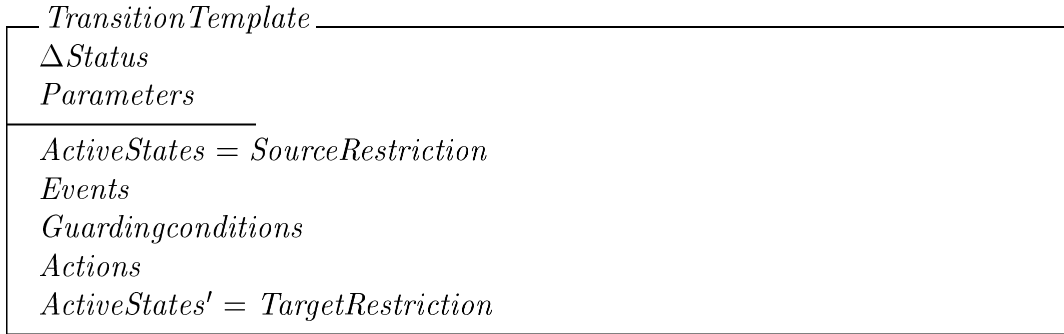

Figure 4: Template schema operation for a statechart transition.

and *TargetRestriction* are analogous to the source and target restriction sets of EHA with the recommended extensions described above to ensure they are fully specified. *Events, Guarding conditions,* and *Actions* are equivalent to the transition label elements defined in section 2. $\Delta Status$ is used to specify that the contents of status will be updated by the operation and the $'$ decoration is used to refer to "after" values of the status.

The translation from statechart specification to Z specification has been automated using a prototype tool (StateZ) written for the project following the same process of behaviour elicitation as described above. The tool is based on foundations [18] common to a general purpose Z type checker and theorem prover and the Statemate [12] Application Programming Interface (API). The translation program produces self-contained specifications with the following components.

- **Auxiliary definitions.** These may include definitions of types, constants and relations used to constrain the operation semantics of the system.
- **Status.** Contains all information relating to the global state of the system. This may include a set of current states, active events and the values of conditions and data-items.
- **Operations over the status.** The transition behaviour of the system is specified as a set of operations. Each operation defines a transformation (or class of transformations) of the status in terms of a pre-condition (possible values of the status that cause the transformations to occur) and a post-condition (constraints on the values of the status following the operation).

This section will be completed by giving some example transition specifications to illustrate the main points discussed so far. Figure 5 gives the definition of the *Status* and *Parameters* for our running example. Note that the the conditions *UnstowedSensor* and *OnGround*, the data-item *ThrottleReq* and events *InadvertentDeploy* and *LockConfirm* are defined as input parameters (hence the *?* decoration). Likewise the condition *LimitThrust*

$$\begin{array}{|l}
\hline
\textit{Status} \\
ThrustLimitationState : ThrustLimitationStates \\
LockDetectState : LockDetectStates \\
AwaitLockState : AwaitLockStates \\
OnGroundDetectState : OnGroundDetectStates \\
\hline
SemanticRelation \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline
\textit{Parameters} \\
InadvertentDeploy? : Event \\
LockConfirm? : Event \\
RestowComplete! : Event \\
LimitThrust! : Boolean \\
UnstowedSensor? : Boolean \\
OnGround? : Boolean \\
ThrottleReq? : \mathbb{Z} \\
\hline
\end{array}$$

Figure 5: Z definition of the Status and Parameter schemas for figure 2.

and the event *RestowComplete* are defined as output parameters (hence the *!* decoration). The parallel nature of *LockDetect* and *OnGroundDetect* (immediate sub-states of the AND-state *InadvertentDeploy* is made explicit in the semantic relation which specifies that if *InadvertentDeploy* is active both *LockDetect* and *OnGroundDetect* must have defined values.

A more concise and elegant specification could have been used (e.g. where the current events and active states are defined as sets) however the simplistic representation given here was used as it aids the automation of proof tactics and test data generation procedures. Such a representation is reasonable as the specification is designed to be used purely as an intermediate representation and therefore attributes that aid automatic manipulation are more desirable than elegance or conciseness. Once the specification and tests have been verified and test data has been generated, the specification and data can be converted to a more suitable specification style for presentation if desired.

The mechanical generation of the formal specification should ensure that the semantics of the original statechart are upheld. However additional trust in the specification can be obtained by representing the statechart semantic definitions as relations in Z (see [16, 17]). These relations can then used as state invariants and the operations can be verified (either through proof or counter-example generation) that they maintain these state invariants. *SemanticRelation* in figure 4 is a place holder for such semantic relations.

Figure 6 gives the operation schema for transition *T3* of figure 2. *FwdIdle* is a constant defined in the auxiliary definitions section. This transition cannot be pre-empted by any transitions in the system of a higher priority. Parts of *Status* and *Parameters* that are undefined (e.g. the parallel component *OnGroundDetectState*) indicate that they are not relevant to the success of the tests and can therefore take any suitable value (where a value may be judged suitable if it conforms to the semantic relation and type definitions).

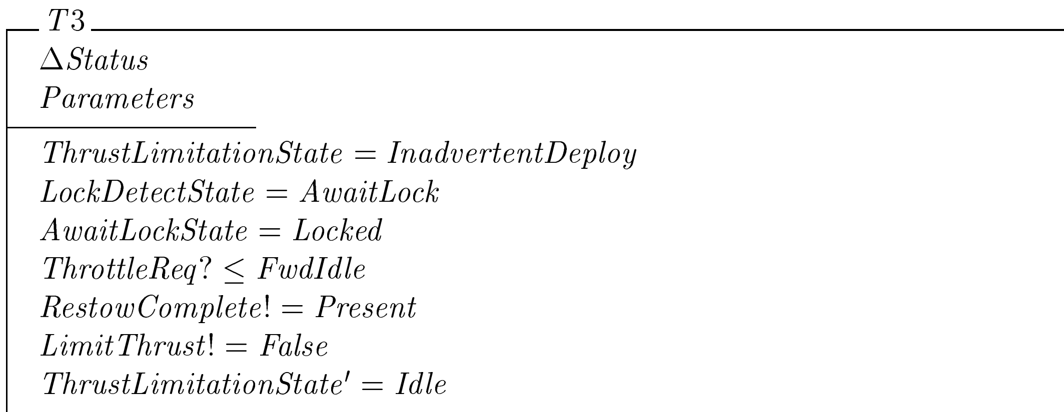

Figure 6: Z definition of transition T3

Figure 7 gives the operation schema for transition $T6_2$. The pre-condition of the

operation contains the negation of the pre-condition for transition *T3* to specify that the transition can only be taken if not pre-empted by the higher priority *T3*.

$$
\begin{array}{l}
\textbf{schema } T6_2 \\
\Delta Status \\
Parameters \\
\hline
(ThrustLimitationState = InadvertentDeploy \\
LockDetectState = AwaitLock \\
AwaitLockState = Locked \\
UnstowedSensor? = True) \wedge \\
\neg(ThrustLimitationState = InadvertentDeploy \\
LockDetectState = AwaitLock \\
AwaitLockState = Locked \\
ThrottleReq? \leq FwdIdle) \\
ThrustLimitationState' = InadvertentDeploy \\
LockDetectState' = Unstowed
\end{array}
$$

Figure 7: Z definition of transition $T6_2$

The completing transition for state *InAir* is given in figure 8. $\Xi Status$ indicates that the values of the elements of *Status* are unchanged by the operation. Again, this transition can be pre-empted by *T3* and therefore contains *T3*'s negated pre-condition.

$$
\begin{array}{l}
\textbf{schema } CompleteInAir \\
\Xi Status \\
Parameters \\
\hline
(ThrustLimitationState = InadvertentDeploy \\
OnGroundDetectState = InAir \\
\neg(ThrottleReq? \leq FwdIdle \wedge \\
OnGround? = True)) \\
\neg(ThrustLimitationState = InadvertentDeploy \\
LockDetectState = AwaitLock \\
AwaitLockState = Locked \\
ThrottleReq? \leq FwdIdle)
\end{array}
$$

Figure 8: Z definition of completing transition for InAir

3.3 Remaining specification looseness

So far we have covered the issues of parallelism (i.e. sets of transitions firing simultaneously) only lightly. This is because, due to our convenient testing method, we are able to test each transition in isolation. However, for more general testing techniques this issue must be resolved more formally. Note that the values of the conditions, data-items and events are not fully specified in the above transitions, i.e. if a parameter is not updated by a transition it is left undefined. This is to allow simultaneously firing transitions that are not the subject of the current test to update these variables without affecting the current test results. If the testing method were extended to testing the parallel behaviour of statecharts our current translation process would leave the behaviour loosely defined. This can be resolved in the following way.

Assume that if elements of *Status* or *Parameters* are not updated by any of the transitions forming a test, they are set to default values. For elements of *Status* (states or local variables) the values are unchanged from the previous step. The default values for events are that they are absent. This behaviour can be modelled in Z in the following way:

$$Test == T_1 \wedge T_2 \wedge ... \wedge T_n \wedge (Default \setminus \{UpdatedElmts\})$$

Where the test is specified as the conjunction of the transitions under test ($T_1 .. T_n$) and the *Default* operation (sets all elements in *Status* and *Parameters* to their default values except for that set of elements updated by any of the transitions - $\{UpdatedElmts\}$). For such a test to be valid the resulting schema expression must be satisfiable by at least on set of values for *Status* and *Parameters*.

3.4 Checking state completeness and determinism

Given the formal specification of the statechart it is possible to ascertain whether certain desirable properties are satisfied by the statechart specification. In doing so, errors in the specification can be eliminated before they propagate into the implementation. As an example, Heimdahl and Leveson [19] describe a number of checks that can be made on hierarchical state-based requirements. These include checks on state determinism (at most one transition out of a state is enabled at any one time) and completeness (the behaviour when in a state is always fully defined). In our formalism, the behaviour of a (possibly non-deterministic) state can be defined as the disjunction of the transitions leaving it, i.e.

$$State == T_1 \vee T_2 \vee ... \vee T_n$$

A theorem on the completeness of the state (the pre-condition of at least one transition out of a state can be satisfied at any time) can be specified as follows:

$$\vdash \forall Status, Parameters \mid LocalState = State \bullet$$
$$pre T_1 \vee pre T_2 \vee ... pre T_n$$

Where $LocalState = State$ constrains the completeness theorem to a particular state in the statechart. In the method presented above we have already given a method for deriving

complete state specifications. However, such a proof can be used as an additional check on the calculation of the completeness transitions or it may be considered good practice for the system engineers to specify all possible behaviour themselves in the original specification (obviating the need for the automatic addition of completing transitions).

A similar theorem can be defined to demonstrate determinism (disjointness of the transition pre-conditions). Although it is not always necessary that specifications are deterministic, determinism affects the way that test results are evaluated and therefore it is at least desirable to know where non-determinism exists in a specification.

$$\vdash \forall\, Status, Parameters \mid LocalState = State \bullet$$
$$\forall\, i, j : 1 \mathinner{..} n \mid i \neq j \bullet preT_i \wedge preT_j = false$$

These theorems can be shown to be valid through proof, or a counter-example can be found to demonstrate that they are invalid.

4 Generating the tests

4.1 Testing goals and assumptions

A clear definition of the testing goals is essential to the success of any testing process and the goals for our example process are stated here. The tests will verify that a particular software module correctly implements the transition mechanism as described in the statechart of figure 2 given that the synchronous time model is used in the implementation. The operation of the Module Under Test (MUT) can be briefly described as follows. At each step the MUT is invoked with the current status (states, events and data variables) and input parameters and the module returns an updated status and output parameters corresponding to a step change in the system. The tests will be performed in a unit test context. As a result, the testing environment has full observability and controllability over the status and parameters. This is a valid assumption given the context and allows each transition to be tested in isolation (the status is set directly before the test and is checked directly after the test). In the general case some state information will be hidden from the testing environment. Test sequences will then be required to move the system into an initial state and to check that the system has moved to a correct state after each test step. Test sequence generation either for initialisation sequences or state checking shall be addressed by further work and will be only briefly introduced in this report.[2]

The MUT will be verified by exercising it with tests derived from the formal specification of the transition behaviour and the results of the tests will be compared against the specification to judge whether the implementation passed or failed each test. The tests should achieve (as near as possible) 100% boolean coverage of the MUT (as suggested in the DO–178B [2] guidelines). In our experiments the code coverage was assessed using the

[2] A considerable amount of work already exists in the areas of test generation from both finite state automata and model-based specifications. However due to the expressiveness of statechart transition operations it appears that statechart testing requires a combination and extension of both these techniques.

AdaTest [4] test harness from IPL. An equivalent metric for specification coverage will be calculated during the construction of the tests. Where coverage of the specification results in shortfalls in code coverage, the unexercised code can be reviewed and additional tests written. The generation of the original test set (i.e. to achieve specification coverage) should be automated to as high a degree as possible. Note, for practical testing situations it is also important that tests are well documented. Automatic test documentation is an extension to the current work that will need to be considered at a later date before the method can be used in a practical context.

4.2 The category partitioning method

In the translation to Z, we have abstracted from the statechart notation and therefore generic test generation techniques can be applied to the resulting formal specification. Furthermore, the fixed and regular structure of the specification has allowed many parts of the test design process to be automated. This section describes the category partitioning method and prototypical tools that have been developed to automate the application of the method to Z specifications.

The category partition method [20] is based on the theory of equivalence classes, introduced by Goodenough and Gerhart in 1975 [21]. The input domain is partitioned into sets of data that, for testing purposes, exhibit the same behaviour in the specification. Therefore only a selection of data from each equivalence class is needed to show that the implementation performs correctly for all data in that class, assuming the particular equivalence class hypothesis holds in the implementation. The equivalence classes are derived by decomposing the specification into functional units that can be independently tested. These functional units are then partitioned further according to the classes of input that can affect the operation and heuristics based on typical implementation faults. This theory of testing has been developed and applied to Z specifications [7]. Work of particular importance has been produced by Stocks and Carrington [8] who described a method (the Test Template Framework) by which the refinement of the specification into individual test cases can be specified in the Z notation itself. This method results in test cases and test oracles specified as Z schemas. A similar approach is followed here, differing from the test template framework in that the prototypical tool support used does not currently specify the test derivation hierarchy in Z, only the generated partitions. Our approach also delays the restriction of the schemas to their pre-condition part until test data is generated. This allows the test specifications to be used directly as test oracles and allows partitioning strategies to be applied to the post-condition of the operation. Similar motivation is given by MacColl and Carrington [22] in their extension of the Test Template Framework.

The decomposition of the specification into functional units was achieved by the translation of the statechart to Z. Each transition specification forms the basis of a set of tests, generated by repeated identification and partitioning of equivalence classes in the specification. Partitioning strategies have been developed for many of the operators that occur in model-based languages [**?**, 23]. Consider as an example, the specification of transition *T3* given in figure 6. One common testing heuristic is boundary value analysis [24], based upon

the hypothesis that programming errors are often made around the boundaries of equivalence classes. Taking $ThrottleReq? \leq FwdIdle$ as defining an equivalence class, boundary value analysis can be used to derive the following three partitions of *T3*. The partitioned parts of the operation are highlighted.

$T3_1$

$\Delta Status$
$Parameters$

$ThrustLimitationState = InadvertentDeploy$
$LockDetectState = AwaitLock$
$AwaitLockState = Locked$
$\boxed{\textbf{ThrottleReq? = FwdIdle}}$
$RestowComplete! = Present$
$LimitThrust! = False$
$ThrustLimitationState' = Idle$

$T3_2$

$\Delta Status$
$Parameters$

$ThrustLimitationState = InadvertentDeploy$
$LockDetectState = AwaitLock$
$AwaitLockState = Locked$
$\boxed{\textbf{ThrottleReq? = FwdIdle - 1}}$
$RestowComplete! = Present$
$LimitThrust! = False$
$ThrustLimitationState' = Idle$

$T3_3$

$\Delta Status$
$Parameters$

$ThrustLimitationState = InadvertentDeploy$
$LockDetectState = AwaitLock$
$AwaitLockState = Locked$
$\boxed{\textbf{ThrottleReq?} < \textbf{FwdIdle - 1}}$
$RestowComplete! = Present$
$LimitThrust! = False$
$ThrustLimitationState' = Idle$

In order to generate test data, the test case specifications are restricted to the pre-condition parts only. The full specification is used as the test oracle (to evaluate the results of applying the test data).

4.3 Tool support

4.3.1 Automated test partitioning

CADiZ [25] is a general purpose Z type checker and theorem prover allowing a user to interactively browse, type check and perform proofs upon a Z specification. The functionality of CADiZ has been extended to make use of its theorem proving abilities to partition a specification according to some pre-defined testing heuristics. The partitioning strategies are specified in a separate Z section referenced by the auto-generated output of the statecharts to Z translation tool StateZ. One such strategy for boundary value analysis of the integer $\leq$ operator is given here:

$$\begin{array}{l} \vdash \forall\, x, y : \mathbb{Z} \mid \mathit{true} \bullet x \leq y \Leftrightarrow \\ x = y \vee \\ x = y - 1 \vee \\ x < y - 1 \end{array}$$

A partitioning strategy is applied by selecting the operation within the test case to be partitioned and invoking the *partition* menu command. A partitioning heuristic matching the selected operation is then instantiated with the operands from the test case and conjoined with the test case predicate. The new predicate is then converted to disjunctive normal form which is in turn partitioned into a test specification for each disjunct.

There are several advantages to specifying and invoking the partitioning heuristics in this way. By specifying the heuristics as Z theorems they themselves can be proven down to their axioms (for the boundary value analysis theorems this is an automatic step within CADiZ) and users may add more partitioning heuristics at their will without the need to change the CADiZ source code. The partitioning heuristics are applied as a proof step within CADiZ which in turn can be inspected for additional validation.

4.3.2 Automated proof tactics

Formal methods can be further exploited by checking that the tests relate to the original specification, that there exists a set of test data that satisfies the tests and to verify the results of applying the tests.

If a set of test data can be shown to satisfy a particular test specification then that test specification is also shown to be satisfiable and also demonstrates that the test data generation (whether manual or automatic) was performed correctly. The test data satisfiability check has now been automated using the CADiZ theorem prover. CADiZ allows

general purpose proof tactics to be written in a lazy functional notation [26] that can be invoked from within CADiZ and applied to the specification.

Due to the consistent structure of the test specifications, general purpose proof tactics can be written to prove that test input data satisfies the test specification. Similar tactics can be written to prove that the test inputs and outputs satisfy the test oracle specification. The same procedure is followed for both types of check. Note, this is not generally possible for all test cases specified in Z but could be achieved in this context due to knowledge of typical constraints that appear in the statechart–derived test cases and the "shape" of the proofs needed to discharge these obligations. The procedure for verifying the test input data and test specification satisfiability is as follows:

1. Use the CADiZ *exists conjecture* command to convert the test specification schema into an exists conjecture. For example, the second boundary value analysis test case for transition *T3* restricted to its pre-condition for test input generation purposes,

   $T3_2$
   $$\begin{array}{l} Status \\ Parameters \\ \hline ThrustLimitationState = InadvertentDeploy \\ LockDetectState = AwaitLock \\ AwaitLockState = Locked \\ ThrottleReq? = FwdIdle - 1 \end{array}$$

   is converted to:

   $$\exists\, Status;\ Parameters \mid ThrustLimitationState = InadvertentDeploy \wedge \\ LockDetectState = AwaitLock \wedge AwaitLockState = Locked \wedge \\ ThrottleReq? = FwdIdle - 1$$

2. Select the exists conjecture and apply the test specification satisfiability proof tactic. The tactic will then ask the user to input values for each element in *Status* and *Parameters*, i.e. the test data. Elements not relevant to the test can be entered using a wild-card character (interpreted as meaning "any value").

3. The proof tactic quantifies the exists conjecture with the data and attempts to reduce the resulting expression to true (to show the satisfiability of the test specification and the validity of the test data). If this is not possible, the proof tactic fails. If no valid test data can be found the test specification can be shown to be unsatisfiable through further proof efforts and removed from the test suite.

The same procedure would be followed to verify the test results and oracle satisfiability with the exception that the test case would not be restricted to its pre–condition and the

test oracle satisfiability tactic would be applied. Ongoing work is looking to develop the tactics further in order to automatically generate the test data itself using the constraint solving facilities of CADiZ. The same proof tactic that demonstrates the satisfiability of a test would then also provide a set of test data for that test.

5 Test coverage and adequacy

5.1 Specification statement coverage

In section 3 we defined one of the testing criteria as achieving the equivalent of 100% boolean coverage in the specification. In this section we show how this criterion is satisfied by application of the automated category partitioning method described in the previous section and demonstrate how, in practice, it is desirable to test beyond this coverage.

The result of the translating each statechart transition to a Z operation can be seen as a rudimentary set of test cases. Each transition operation is a test that exercises a transition. Applying these tests can be compared to achieving statement coverage of the specification, where each transition is equivalent to a statement in the specification and the combination of state and conditions for triggering a transition are equivalent to branch points. This observation was supported by the structural coverage achieved by running these tests against the implementation. Statement coverage (at either the specification or code level) is not considered sufficient for high integrity systems. Statement coverage can be shown to be insufficient by giving examples of simple errors that may be overlooked by the tests. Consider, for example that in implementing transition *T6* of the specification the engineer mistook the source state as *Wait* instead of *AwaitLock* and as a result also did not consider the fact that *T3* can take priority over this transition. A statement covering test case of the *specification* of *T6* (see section 3) may choose the value *AwaitLockState = Wait* rather than *AwaitLockState = Locked* and would therefore miss the error. More rigorous coverage of the specification is therefore required.

5.2 Specification MC/DC coverage

Our target coverage criteria is 100% boolean operand effectiveness (also known as Modified Condition/Decision Coverage - MC/DC). MC/DC is achieved by showing that each condition within a decision can independently affect that decision's outcome by varying just that condition while holding all other possible conditions fixed [2]. The conditions for satisfying MC/DC coverage are summarised in table 4 from [27]. In our specifications the pre-condition is the decision to be satisfied and the individual terms of the pre-condition are its component conditions.

To show how MC/DC coverage can be satisfied, consider first the problem of generating test cases for decisions of the form $A \vee B$. Dick and Faivre [?] suggested a partitioning tactic for such expressions which resulted in the following three test cases: $A \wedge B$, $A \wedge \neg B$ and $\neg A \wedge B$. This partitioning tactic, combined with the effect of ensuring that each state

$A \vee B$	
$A \vee B = true$	1
$A \vee B = false$	2
$A = true$	3
$A = false$	4
$B = true$	5
$B = false$	6
show that each condition independently affects the decision outcome	
$A \vee false = A \vee B$	7
$B \vee false = A \vee B$	8

$A \wedge B$	
$A \wedge B = true$	9
$A \wedge B = false$	10
$A = true$	11
$A = false$	12
$B = true$	13
$B = false$	14
show that each condition independently affects the decision outcome	
$A \wedge true = A \wedge B$	15
$B \wedge true = A \wedge B$	16

Table 4: Conditions for MC/DC coverage

Test Case	A	B	$A \vee B$
$A \wedge B$	A	B	true
$\neg A \wedge B$	$\neg A$	B	true
$A \wedge \neg B$	A	$\neg B$	true
$\neg(A \vee B) \equiv \neg A \wedge \neg B$	$\neg A$	$\neg B$	false

Table 5: Logic table for $A \vee B$ tests

Test Case	A	B	$A \wedge B$
$A \wedge B$	A	B	true
$\neg(A \wedge B) \equiv$ $\neg A \wedge \neg B$	$\neg A$	$\neg B$	false
$A \wedge \neg B$	A	$\neg B$	false
$\neg A \wedge B$	$\neg A$	B	false

Table 6: Logic table for $A \wedge B$ tests

is complete, i.e. the other transitions out of the state must have the combined effect of satisfying $\neg(A \vee B)$, is sufficient to ensure that the conditions given in the top half of table 4 are satisfied. The proof of this statement can be demonstrated by examining the combined logic table (figure 5) for test cases resulting from the disjunct partitioning (first three rows) and state completion (last row). The last column demonstrates that the first two conditions are met (row 1 and 2 in table 4). The second and third column demonstrate that all values of A and B have been exercised (rows 3-6 in table 4). The table also shows that each condition is tested independently of the other (rows 7-8). For example, A is tested in combination with both B and $\neg B$, therefore regardless of the actual value of B, the expression $A \vee false$ is always exercised and similarly for $B \vee false$.

The same reasoning can be applied for triggering expressions of the form $A \wedge B$, the truth table for which is given below. The same table is produced whether the state contains one original transition and one completing transition of the form $\neg(A \wedge B) \equiv \neg A \vee \neg B$ (which is subsequently partitioned into $\neg A \wedge \neg B$, $A \wedge \neg B$ and $\neg A \wedge B$) or whether a combination of several other transitions complete the state. The logic table for the test cases is given in table 6. 26 test cases (see Appendix) were required to achieve MC/DC of the statechart given in figure 2.

5.3 Beyond structural coverage

Although the equivalent of MC/DC coverage can be achieved at the specification level (depending on the refinement this can result in similar coverage of the code) and it is likely that these tests would detect some errors, they cannot be guaranteed to detect all errors (nothing short of exhaustive testing can). In our running example, if the constant *FwdIdle* was given the wrong value in the implementation, then the equivalence class boundary $ThrottleReq? \leq FwdIdle$ in the operation *T3* will be incorrect. Boundary value analysis can be used to ensure that at least boundary errors are detected.

Additional testing heuristics can be added in this way until the cost of applying the tests out-weighs the savings made by detecting the errors. Naturally if the testing process can be automated, a far greater number of tests can be applied, thus increasing the end confidence in the implementation. Applying boundary value analysis to the MC/DC covering test cases for figure 2 would raise the total number of test cases to 51.

To conclude the discussion of test coverage, we point out that by making all behaviour

in the statechart explicit we have increased the number of tests generated compared to how many would be generated if each transition on the original statechart diagram was used as the basis for the tests. This suggests that greater coverage of the system behaviour has been achieved.

6 Conclusions and future work

Statecharts are a rich graphical notation that allow for the specification of complex control systems. The apparent simplicity of the specifications is deceptive since the actual behaviour they specify can be extremely involved. When testing an implementation of a statechart specification, it is the detailed behaviour *implied* by the statecharts that must be verified. It is clear to see from the structure of this report that much of the testing problem reduces to eliciting an *explicit* definition of the statechart behaviour from which suitable tests can be identified.

The objective of the current project is to automate the testing process. Due to the complexity of the task, careless automation could not only lead to errors appearing in the process, but due to the very nature of automation these errors may be hidden. Confidence must be obtained in the outputs (a set of test data and result checking mechanism) of any automated test environment. This report has shown how formal methods can be used to support the integrity of an automated testing process. The behaviour under test is first elicited (via a mechanical process defined in section 4) and specified in the formal specification notation Z. Proof-based checks can be applied to this specification to ensure that certain properties hold (e.g. completeness, satisfiability).

Existing techniques based on the theory of equivalence classes can be applied to the Z specification. Furthermore, due to the restricted subset of Z used and common structures in the specification (that arise due to the automated translation process), the test generation process and much of the proof effort can also be automated. The tests themselves are specified in Z and as such proof can be used to demonstrate the relationship to the original specification, the satisfiability of the tests and the validity of test results. The integration of formal and mechanical methods supports the project objective of achieving "trusted test automation".

Evaluating test effectiveness is not straightforward. The report described how a minimum criterion of MC/DC coverage of the specification can be automatically achieved. This coverage alone is not guaranteed to detect all errors. Further tests can be automatically generated based on typical errors that appear in the implementation. The work so far can be considered to be successful in that unit tests can be automatically generated based on boolean coverage of the specifications and error-directed testing heuristics. The structural code coverage achieved by applying these tests will vary depending on the amount of refinement between specification and implementation.

This work can be seen as the first step to a complete and practical testing solution for statecharts. Further work must address several key issues. Test data generation can be automated by applying either heuristic optimisation [28] or constraint logic programming

[23] techniques. These techniques have been shown to be successful at generating test data for constraints typical of the specifications under consideration. Preliminary work has also shown that such techniques can be applied successfully from with an automated proof environment allowing test data generation to be performed as part of test case satisfiability checks. Future work will continue to look at integrating these techniques into the CADiZ proof environment.

Future work should also look at reducing some of the testing assumptions made in this report. In particular, the techniques should allow parts of the status to be hidden (neither controllable nor observable) from the testing environment – as would occur in typical integration testing contexts. Test sequences would then be required to move the system into an initial state and to check the current state. The proposed line of work will construct a finite state automaton from the partitioned test cases based on techniques suggested by Dick and Faivre [**?**]. Test sequence generation techniques (e.g. [29, 13]) will then be applied to this finite state automaton to generate state checking sequences. In particular the work will investigate the affect that the rich transition operation syntax and state concurrency inherent in statechart specifications will have on the application of these techniques.

7 Acknowledgements

This work was funded by the Rolls–Royce High Integrity Systems and Software Centre (HISSC). The author would like to thank Rolls-Royce for continued support throughout the project. The author would also like to thank his supervisors, John McDermid and John Clark for their invaluable support and advice, Steve King for his useful review comments and Ian Toyn and Sam Valentine for their support in using CADiZ.

References

[1] D. Harel, "Statecharts: A visual formalism for complex systems," *Science of Computer Programming*, vol. 8, pp. 231–274, 1987.

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

[3] Ministry of Defence, UK, *Requirements For the Procurement Of Safety Critical Software In Defence Equipment (00-55/Issue 2)*, August 1997.

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

[5] Liverpool Data Research Associates Ltd.,Liverpool, UK, *LDRA Testbed Technical Description*, 1993.

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

[7] N. Amla and P. Ammann, "Using Z specifications in category partition testing," *Proceeding of COMPASS 1992, Seventh Annual Conference On Computer Assurance*, pp. 3–10, 1992.

[8] P. Stocks and D. Carrington, "A framework for specification-based testing," *IEEE Transactions On Software Engineering*, vol. 22, pp. 777–793, November 1996.

[9] J. Armstrong, *Safecharts: Safer And More Structured Statecharts.* British Aerospace DCSC, Department of Computer Science, University of York, March 1996.

[10] D. Harel and A. Naamad, "The Statemate semantics of statecharts," *IEEE Transactions On Software Engineering And Methodology*, vol. 5, pp. 293–33, Oct 1996.

[11] E. Mikk, Y. Lakhnech, and M. Siegel, "Hierarchical automata as model for statecharts," in *Proceedings of Advances in Computing Science - ASIAN'97: third annual Computing Science Conference, Kathmandu, Nepal (LNCS 1345)*, December 1997.

[12] D. Harel, H. Lachover, A. Naamad, A. Pnueli, R. Sherman, A. Shtull-Truaring, and M. Trakhenbrot, "Statemate, a working environment for the development of complex reactive systems," *IEEE Transactions On Software Engineering*, vol. 16, pp. 403–414, 1988.

[13] R. M. Hierons, "Testing from semi-independent communicating finite state machines," *IEEE Proceedings In Software Engineering*, vol. 144, no. 5-6, pp. 291–295, 1997.

[14] R. Buessow, R. Geisler, and M. Klar, "Specifying safety-critical embedded systems with statecharts and Z: A case study," in *Proceedings of Fundamental Approaches to Software Engineering (FASE'98), Lisbon, Portugal*, March/April 1998.

[15] W. Grieskamp, M. Heisel, and H. Doerr, "Specifying safety-critical embedded systems with statecharts and Z: An agenda for cyclic software components," in *Proceedings of Fundamental Approaches to Software Engineering (FASE'98), Lisbon, Portugal*, March/April 1998.

[16] E. Mikk, Y. Lakhnech, C. Petersohn, and M. Siegel, "On the formal semantics of statecharts as supported by Statemate," *Proceedings Of The Second Northern Formal Methods Workshop*, July 1997.

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

[18] I. Toyn, D. M. Cattrall, J. A. McDermid, and J. L. Jacob, "A practical language and toolkit for high-integrity tools," *Journal of Systems and Software*, vol. 41, pp. 161–173, June 1998.

[19] M. Heimdahl and N. Leveson, "Completeness and consistency in heirarchical state-based requirements," *IEEE Transactions On Software Engineering*, vol. 22, pp. 363–377, June 1996.

[20] T. J. Ostrand and M. J. Balcer, "The category-partition method for specifying and generating functional tests," *Communications of the ACM*, vol. 31, pp. 676–686, June 1988.

[21] J. B. Goodenough and S. L. Gerhart, "Towards a theory of test data selection," *IEEE Transactions On Software Engineering*, vol. 1, pp. 156–173, June 1975.

[22] I. MacColl and D. Carrington, "Extending the test template framework," *Proceedings of the third northern formal methods workshop, Ilkley, UK*, sep 1998.

[23] C. Meudec, *Automatic Generation of Software Test Cases From Formal Specifications.* PhD thesis, The Queen's University of Belfast, 1998.

[24] B. Beizer, *Software Testing Techniques.* Thomson Computer Press, 1990.

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

[26] I. Toyn, "A tactic language for reasoning about Z specifications," in *Proceedings of the Third Northern Formal Methods Workshop, Ilkley, UK*, September 1998.

[27] J. Voas, J. Payne, and K. Miller, *Automating Test Case Generation For Coverages Required by FAA Standard DO-178B.* RST Corporation, 1993.

[28] N. Tracey, J. Clark, and K. Mander, "Automated program flaw finding using simulated annealing," in *Software Engineering Notes, Proceedings Of The International Symposium On Software Testing And Analysis*, vol. 23, pp. 73–81, ACM/SIGSOFT, March 1998.

[29] T. S. Chow, "Testing software design modeled by finite-state machines," *IEEE Transactions On Software Engineering*, vol. SE-4, pp. 178–187, May 1978.

8 Appendix: Z specification for ThrustLimitation

8.1 Test Summary

The derivation of the tests for figure 2, specified in the remainder of this document is summarised in table 7. The tests that appear to the furthest right of the table on each row are used for generating test data. In total, 26 test cases were required to achieve MC/DC coverage of the statechart, 51 test cases were produced when the boundary value analysis tactic was applied. The acronyms used for the partitioning heuristics are; D+F (Dick and Faivre disjunct partitioning) and BVA (Boundary Value Analysis).

8.2 Z Specification

8.2.1 Auxiliary definitions

section $casestudy2$ parents $partitions$

Auxiliary definitions: types, constants and the semantic relation.

$Boolean ::= True \mid False$

$Event ::= Absent \mid Present$

$ThrustLimitationStates ::= Idle \mid InadvertentDeploy$

$LockDetectStates ::= Unstowed \mid AwaitLock$

$OnGroundDetectStates ::= InAir \mid OnGround$

$AwaitLockStates ::= Wait \mid Locked$

$$\begin{array}{|l} FwdIdle : \mathbb{Z} \\ \hline FwdIdle = 1 \end{array}$$

Test case	Heuristic	Test case	Heuristic	Test case
Init				
$T2$				
$T3$	BVA	$T3bva1$		
		$T3bva2$		
		$T3bva3$		
$T5$				
$T6_1$				
$T6_2$	BVA	$T6_2bva1$		
		$T6_2bva2$		
$T8$				
$T10$	D+F	$T10df1$	BVA	3 further tests
		$T10df2$	BVA	3 further tests
CompleteIdle				
CompleteUnstowed				
CompleteWait				
CompleteLocked	BVA	*CompleteLockedbva*1		
		*CompleteLockedbva*2		
CompleteInAir	D+F	*CompleteInAirdf*1	BVA	2 further tests
		*CompleteInAirdf*2	BVA	2 further tests
		*CompleteInAirdf*3	BVA	2 further tests
		*CompleteInAirdf*4	BVA	3 further tests
		*CompleteInAirdf*5	BVA	3 further tests
		*CompleteInAirdf*6	BVA	2 further tests
		*CompleteInAirdf*7	BVA	2 further tests
		*CompleteInAirdf*8	BVA	2 further tests
CompleteOnGround	D+F	*CompleteOnGrounddf*1	BVA	2 further tests
		*CompleteOnGrounddf*2	BVA	3 further tests
		*CompleteOnGrounddf*3	BVA	2 further tests
		*CompleteOnGrounddf*4	BVA	3 further tests
		*CompleteOnGrounddf*5	BVA	2 further tests

Table 7: Test summary

8.2.2 Status

The semantic relation is used as an invariant for the transition operations. It ensures that the system is in a valid combination of states at the beginning and end of each transition (due to its inclusion in the predicate part of the status schema). The semantic relation as defined here ensures the disjointness or concurrency properties of OR and AND-states hold and also models the parent relation. The exclusivity of or-states (only one sub-state can be active at any time) is implied by the disjointness properties of the enumerated type specification. The semantic relation as described here is specific to the particular statechart specification.

$$relation(SemanticRelation_)$$

SemanticRelation _ ==

$$\begin{array}{l}
\{ ThrustLimitationState : ThrustLimitationStates; \\
LockDetectState : LockDetectStates; \\
AwaitLockState : AwaitLockStates; \\
OnGroundDetectState : OnGroundDetectStates \mid \\
(ThrustLimitationState = Idle \lor ThrustLimitationState = InadvertentDeploy) \land \\
(ThrustLimitationState = InadvertentDeploy \Leftrightarrow \\
(LockDetectState = Unstowed \lor LockDetectState = AwaitLock) \land \\
(OnGroundDetectState = InAir \lor OnGroundDetectState = OnGround)) \land \\
(LockDetectState = AwaitLock \Leftrightarrow \\
AwaitLockState = Wait \lor AwaitLockState = Locked)\}
\end{array}$$

Status

$$\begin{array}{l}
ThrustLimitationState : ThrustLimitationStates \\
LockDetectState : LockDetectStates \\
AwaitLockState : AwaitLockStates \\
OnGroundDetectState : OnGroundDetectStates \\
\hline
SemanticRelation(ThrustLimitationState, LockDetectState, \\
AwaitLockState, OnGroundDetectState)
\end{array}$$

Parameters

$$\begin{array}{l}
InadvertentDeploy? : Event \\
LockConfirm? : Event \\
RestowComplete! : Event \\
LimitThrust! : Boolean \\
UnstowedSensor? : Boolean \\
OnGround? : Boolean \\
ThrottleReq? : \mathbb{Z}
\end{array}$$

8.2.3 Transition Operations/Test cases

Init

$\Delta Status$

$ThrustLimitationState' = Idle$

T2

$\Delta Status$
$Parameters$

$ThrustLimitationState = Idle$
$InadvertentDeploy? = Present$
$LimitThrust! = True$
$ThrustLimitationState' = InadvertentDeploy$
$LockDetectState' = Unstowed$
$OnGroundDetectState' = InAir$

Transition *T2* is not partitioned further and therefore is used directly as a test case.

T3

$\Delta Status$
$Parameters$

$ThrustLimitationState = InadvertentDeploy$
$LockDetectState = AwaitLock$
$AwaitLockState = Locked$
$ThrottleReq? \leq FwdIdle$
$RestowComplete! = Present$
$LimitThrust! = False$
$ThrustLimitationState' = Idle$

Using boundary value analysis, *T3* can be partitioned into *T3bva1*, *T3bva2* and *T3bva3*.

$T3bva1$

$\Delta Status$
$Parameters$

$ThrustLimitationState = InadvertentDeploy$
$LockDetectState = AwaitLock$
$AwaitLockState = Locked$
$(ThrottleReq? < FwdIdle - 1$
$ThrottleReq? \leq FwdIdle)$
$RestowComplete! = Present$
$LimitThrust! = False$
$ThrustLimitationState' = Idle$

$T3bva2$

$\Delta Status$
$Parameters$

$ThrustLimitationState = InadvertentDeploy$
$LockDetectState = AwaitLock$
$AwaitLockState = Locked$
$(ThrottleReq? = FwdIdle - 1$
$ThrottleReq? \leq FwdIdle)$
$RestowComplete! = Present$
$LimitThrust! = False$
$ThrustLimitationState' = Idle$

$T3bva3$

$\Delta Status$
$Parameters$

$ThrustLimitationState = InadvertentDeploy$
$LockDetectState = AwaitLock$
$AwaitLockState = Locked$
$(ThrottleReq? = FwdIdle$
$ThrottleReq? \leq FwdIdle)$
$RestowComplete! = Present$
$LimitThrust! = False$
$ThrustLimitationState' = Idle$

$$
\begin{array}{|l}
\multicolumn{1}{l}{T5} \\
\Delta Status \\
Parameters \\
\hline
ThrustLimitationState = InadvertentDeploy \\
LockDetectState = Unstowed \\
UnstowedSensor? = False \\
ThrustLimitationState' = InadvertentDeploy \\
LockDetectState' = AwaitLock \\
AwaitLockState' = Wait
\end{array}
$$

Transition *T5* is not partitioned further and therefore is used directly as a test case.

$$
\begin{array}{|l}
\multicolumn{1}{l}{T6_1} \\
\Delta Status \\
Parameters \\
\hline
ThrustLimitationState = InadvertentDeploy \\
LockDetectState = AwaitLock \\
AwaitLockState = Wait \\
UnstowedSensor? = True \\
ThrustLimitationState' = InadvertentDeploy \\
LockDetectState' = Unstowed
\end{array}
$$

Transition $T6_1$ is not partitioned further and therefore is used directly as a test case.

$$
\begin{array}{|l}
\multicolumn{1}{l}{T6_2} \\
\Delta Status \\
Parameters \\
\hline
(ThrustLimitationState = InadvertentDeploy \\
LockDetectState = AwaitLock \\
AwaitLockState = Locked \\
UnstowedSensor? = True) \wedge \\
\neg(ThrottleReq? \leq FwdIdle) \\
ThrustLimitationState' = InadvertentDeploy \\
LockDetectState' = Unstowed
\end{array}
$$

Boundary value analysis can be applied to $T6_2$ to reveal two further test cases; $T6_2bva1$ and $T6_2bva2$.

$$T6_2bva1$$

$$\Delta Status$$
$$Parameters$$

$$ThrustLimitationState = InadvertentDeploy$$
$$LockDetectState = AwaitLock$$
$$AwaitLockState = Locked$$
$$UnstowedSensor? = True$$
$$(ThrottleReq? > FwdIdle + 1$$
$$ThrottleReq? > FwdIdle)$$
$$ThrustLimitationState' = InadvertentDeploy$$
$$LockDetectState' = Unstowed$$

$$T6_2bva2$$

$$\Delta Status$$
$$Parameters$$

$$ThrustLimitationState = InadvertentDeploy$$
$$LockDetectState = AwaitLock$$
$$AwaitLockState = Locked$$
$$UnstowedSensor? = True$$
$$(ThrottleReq? = FwdIdle + 1$$
$$ThrottleReq? > FwdIdle)$$
$$ThrustLimitationState' = InadvertentDeploy$$
$$LockDetectState' = Unstowed$$

$$T8$$

$$\Delta Status$$
$$Parameters$$

$$(ThrustLimitationState = InadvertentDeploy$$
$$LockDetectState = AwaitLock$$
$$AwaitLockState = Wait$$
$$LockConfirm? = Present) \wedge$$
$$\neg UnstowedSensor? = True$$
$$ThrustLimitationState' = InadvertentDeploy$$
$$LockDetectState' = AwaitLock$$
$$AwaitLockState' = Locked$$

$T8$ cannot be partitioned further and is therefore used directly as a test case.

$$
\begin{array}{|l}
\hline
T10 \\
\Delta Status \\
Parameters \\
\hline
(ThrustLimitationState = InadvertentDeploy \\
OnGroundDetectState = InAir \\
ThrottleReq? \leq FwdIdle \land OnGround? = True) \land \\
\neg(LockDetectState = AwaitLock \\
AwaitLockState = Locked \\
ThrottleReq? \leq FwdIdle) \\
LimitThrust! = False \\
ThrustLimitationState' = InadvertentDeploy \\
OnGroundDetectState' = OnGround \\
\hline
\end{array}
$$

$T10$ is simplified and Dick and Faivre's disjunction partitioning heuristic is applied to reveal the following test cases.

$$
\begin{array}{|l}
\hline
T10df1 \\
\Delta Status \\
Parameters \\
\hline
(ThrustLimitationState = InadvertentDeploy \\
OnGroundDetectState = InAir \\
ThrottleReq? \leq FwdIdle \land OnGround? = True) \land \\
(\neg LockDetectState = AwaitLock \\
\neg AwaitLockState = Locked) \\
LimitThrust! = False \\
ThrustLimitationState' = InadvertentDeploy \\
OnGroundDetectState' = OnGround \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
T10df2 \\
\Delta Status \\
Parameters \\
\hline
(ThrustLimitationState = InadvertentDeploy \\
OnGroundDetectState = InAir \\
ThrottleReq? \leq FwdIdle \land OnGround? = True) \land \\
(LockDetectState = AwaitLock \\
\neg AwaitLockState = Locked) \\
LimitThrust! = False \\
ThrustLimitationState' = InadvertentDeploy \\
OnGroundDetectState' = OnGround \\
\hline
\end{array}
$$

Boundary value analysis could be applied to each of the above tests for transition T10 resulting in a further 6 partitions.

$$
\begin{array}{|l}
\hline
CompleteIdle \\
\Xi Status \\
Parameters \\
\hline
ThrustLimitationState = Idle \\
\neg InadvertentDeploy? = Present \\
\hline
\end{array}
$$

CompleteIdle cannot be partitioned further and is used directly as a test case.

$$
\begin{array}{|l}
\hline
CompleteUnstowed \\
\Xi Status \\
Parameters \\
\hline
ThrustLimitationState = InadvertentDeploy \\
LockDetectState = Unstowed \\
\neg UnstowedSensor? = False \\
\hline
\end{array}
$$

CompleteUnstowed cannot be partitioned further and is used directly as a test case.

$$
\begin{array}{|l}
\hline
CompleteWait \\
\Xi Status \\
Parameters \\
\hline
ThrustLimitationState = InadvertentDeploy \\
LockDetectState = AwaitLock \\
AwaitLockState = Wait \\
\neg UnstowedSensor? = True \\
\neg LockConfirm? = Present \\
\hline
\end{array}
$$

CompleteWait cannot be partitioned further and is used directly as a test case.

$$
\begin{array}{l}
\textbf{CompleteLocked} \\
\hline
\Xi Status \\
Parameters \\
\hline
ThrustLimitationState = InadvertentDeploy \\
LockDetectState = AwaitLock \\
AwaitLockState = Locked \\
ThrottleReq? > FwdIdle \\
\neg UnstowedSensor? = True \\
\hline
\end{array}
$$

Boundary value analysis is applied to *CompleteLocked* to reveal the following partitions.

$$
\begin{array}{l}
\textbf{CompleteLockedbva1} \\
\hline
\Xi Status \\
Parameters \\
\hline
ThrustLimitationState = InadvertentDeploy \\
LockDetectState = AwaitLock \\
AwaitLockState = Locked \\
(ThrottleReq? > FwdIdle + 1 \\
ThrottleReq? > FwdIdle) \\
\neg UnstowedSensor? = True \\
\hline
\end{array}
$$

$$
\begin{array}{l}
\textbf{CompleteLockedbva2} \\
\hline
\Xi Status \\
Parameters \\
\hline
ThrustLimitationState = InadvertentDeploy \\
LockDetectState = AwaitLock \\
AwaitLockState = Locked \\
(ThrottleReq? = FwdIdle + 1 \\
ThrottleReq? > FwdIdle) \\
\neg UnstowedSensor? = True \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{CompleteInAir} \\
\Xi Status \\
Parameters \\
\hline
ThrustLimitationState = InadvertentDeploy \\
OnGroundDetectState = InAir \\
\neg(ThrottleReq? \leq FwdIdle \land OnGround? = True) \\
\neg(LockDetectState = AwaitLock \\
AwaitLockState = Locked \\
ThrottleReq? \leq FwdIdle) \\
\hline
\end{array}
$$

CompleteInAir is simplified and Dick and Faivre's disjunction heuristic applied to $\neg(ThrottleReq? \leq FwdIdle \land OnGround? = True)$ and $\neg(LockDetectState = AwaitLock \land AwaitLockState = Locked \land ThrottleReq? \leq FwdIdle)$ in turn to reveal the following satisfiable partitions.

$$
\begin{array}{|l}
\hline
\textit{CompleteInAirdf1} \\
\Xi Status \\
Parameters \\
\hline
ThrustLimitationState = InadvertentDeploy \\
OnGroundDetectState = InAir \\
(ThrottleReq? > FwdIdle \land OnGround? = True) \\
(\neg LockDetectState = AwaitLock \\
\neg AwaitLockState = Locked) \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{CompleteInAirdf2} \\
\Xi Status \\
Parameters \\
\hline
ThrustLimitationState = InadvertentDeploy \\
OnGroundDetectState = InAir \\
(ThrottleReq? > FwdIdle \land OnGround? = True) \\
(LockDetectState = AwaitLock \\
\neg AwaitLockState = Locked) \\
\hline
\end{array}
$$

CompleteInAirdf3

$$\Xi Status$$
$$Parameters$$

$$ThrustLimitationState = InadvertentDeploy$$
$$OnGroundDetectState = InAir$$
$$(ThrottleReq? > FwdIdle \land OnGround? = True)$$
$$(LockDetectState = AwaitLock$$
$$AwaitLockState = Locked)$$

CompleteInAirdf4

$$\Xi Status$$
$$Parameters$$

$$ThrustLimitationState = InadvertentDeploy$$
$$OnGroundDetectState = InAir$$
$$(ThrottleReq? \leq FwdIdle \land \neg(OnGround? = True))$$
$$(\neg LockDetectState = AwaitLock$$
$$\neg AwaitLockState = Locked)$$

CompleteInAirdf5

$$\Xi Status$$
$$Parameters$$

$$ThrustLimitationState = InadvertentDeploy$$
$$OnGroundDetectState = InAir$$
$$(ThrottleReq? \leq FwdIdle \land \neg(OnGround? = True))$$
$$(LockDetectState = AwaitLock$$
$$\neg AwaitLockState = Locked)$$

CompleteInAirdf6

$$\Xi Status$$
$$Parameters$$

$$ThrustLimitationState = InadvertentDeploy$$
$$OnGroundDetectState = InAir$$
$$(ThrottleReq? > FwdIdle \land \neg(OnGround? = True))$$
$$(\neg LockDetectState = AwaitLock$$
$$\neg AwaitLockState = Locked)$$

$$
\begin{array}{l}
\textit{CompleteInAirdf7} \\
\hline
\Xi Status \\
Parameters \\
\hline
ThrustLimitationState = InadvertentDeploy \\
OnGroundDetectState = InAir \\
(ThrottleReq? > FwdIdle \land \neg(OnGround? = True)) \\
(LockDetectState = AwaitLock \\
\neg AwaitLockState = Locked) \\
\hline
\end{array}
$$

$$
\begin{array}{l}
\textit{CompleteInAirdf8} \\
\hline
\Xi Status \\
Parameters \\
\hline
ThrustLimitationState = InadvertentDeploy \\
OnGroundDetectState = InAir \\
(ThrottleReq? > FwdIdle \land \neg(OnGround? = True)) \\
(LockDetectState = AwaitLock \\
AwaitLockState = Locked) \\
\hline
\end{array}
$$

Boundary value analysis could be applied to each of the above test cases for *CompleteInAir* to reveal a further 18 partitions.

$$
\begin{array}{l}
\textit{CompleteOnGround} \\
\hline
\Xi Status \\
Parameters \\
\hline
ThrustLimitationState = InadvertentDeploy \\
OnGroundDetectState = OnGround \\
\neg(LockDetectState = AwaitLock \\
AwaitLockState = Locked \\
ThrottleReq? \leq FwdIdle) \\
\hline
\end{array}
$$

CompleteOnGround is simplified and Dick and Faivre's disjunction heuristic applied to $\neg(LockDetectState = AwaitLock \land AwaitLockState = Locked \land ThrottleReq? \leq FwdIdle)$ in turn to reveal the following satisfiable partitions.

$$
\begin{array}{l}
\textbf{CompleteOnGrounddf1} \\
\Xi Status \\
Parameters \\
\hline
ThrustLimitationState = InadvertentDeploy \\
OnGroundDetectState = OnGround \\
(\neg LockDetectState = AwaitLock \\
\neg AwaitLockState = Locked \\
ThrottleReq? > FwdIdle)
\end{array}
$$

$$
\begin{array}{l}
\textbf{CompleteOnGrounddf2} \\
\Xi Status \\
Parameters \\
\hline
ThrustLimitationState = InadvertentDeploy \\
OnGroundDetectState = OnGround \\
(\neg LockDetectState = AwaitLock \\
\neg AwaitLockState = Locked \\
ThrottleReq? \leq FwdIdle)
\end{array}
$$

$$
\begin{array}{l}
\textbf{CompleteOnGrounddf3} \\
\Xi Status \\
Parameters \\
\hline
ThrustLimitationState = InadvertentDeploy \\
OnGroundDetectState = OnGround \\
(LockDetectState = AwaitLock \\
\neg AwaitLockState = Locked \\
ThrottleReq? > FwdIdle)
\end{array}
$$

$$
\begin{array}{l}
\textbf{CompleteOnGrounddf4} \\
\Xi Status \\
Parameters \\
\hline
ThrustLimitationState = InadvertentDeploy \\
OnGroundDetectState = OnGround \\
(LockDetectState = AwaitLock \\
\neg AwaitLockState = Locked \\
ThrottleReq? \leq FwdIdle)
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{CompleteOnGrounddf5} \\
\Xi Status \\
Parameters \\
\hline
ThrustLimitationState = InadvertentDeploy \\
OnGroundDetectState = OnGround \\
(LockDetectState = AwaitLock \\
AwaitLockState = Locked \\
ThrottleReq? > FwdIdle) \\
\hline
\end{array}
$$

Boundary value analysis could be applied to each of the above tests for *CompleteOnGround* to reveal a further 12 test cases.