Automated Generation of High Integrity Test Suites from Graphical Specifications

Simon Burton

Submitted for the degree of Doctor of Philosophy

University of York

Department of Computer Science

March, 2002

Abstract:

There is a strong need to ensure that the software controlling safety–critical systems is of the highest possible integrity. Present approaches to achieving this integrity through Verification and Validation (V&V) are expensive and predominantly reliant on manual processes. Techniques that have been proposed to improve the V&V of safety–critical systems – formal methods, graphical specifications and automated testing – do not fully address the needs of the industry. Formal methods have still to gain widespread use, graphical specifications are generally not well supported by rigorous V&V techniques and automation has not yet successfully addressed the problem of specification–based test design in a rigorous yet flexible manner.

This thesis describes the integration of graphical specification notations, formal methods and automated testing to achieve efficient and effective V&V. A framework is presented within which formal and automated techniques can be applied to the validation of graphical specifications and the automated generation of effective tests from these specifications.

A formalisation of two notations of proven practical use (Statecharts and PFS tables) is presented. An intermediate formal representation of the specifications then forms the basis of a set of generic automated V&V techniques. Common structures in the formal specification are exploited to develop automated specification validation techniques based on automated theorem proving and model checking. Automated test case design is achieved by formally specifying testing heuristics, based on either partition testing strategies or particular hypothesised faults. These heuristics are then automatically applied to the formal representation to produce the test cases. In some situations, test cases must be sequenced in order to indirectly infer the correctness of parts of the system not directly measurable at the testing interface. The formalisation is extended to provide an abstraction that allows existing finite state machine-based techniques for selecting test sequences to be exploited. Methods for optimising the effectiveness and efficiency of the testing heuristics are described as are industrially applicable case studies that validate the techniques.

The thesis concludes that the formalisation of specifications and testing heuristics can lead to a practical level of automation that not only has the potential to reduce the costs involved in V&V but can also increase its effectiveness, allowing a greater number of defects to be detected earlier in the software development.

Contents

Acknowledgements:

I would like to thank Rolls-Royce Plc. who have supported this research by providing many of the case studies and a forum in which the applicability of the techniques to practical processes and toolsets could be explored. I would also like to thank my supervisors, Professor John McDermid and John Clark for their support and encouragement. Thank you too, to Ian Toyn who proof read much of the Z contained within the thesis. Finally, huge thanks to my wife, Carolin and my parents for all their support over the years.

Declaration:

The work presented in this thesis has been drawn from research undertaken between August 1997 and July 2001 at the Department of Computer Science, University of York. Much of the work has been published elsewhere as follows:

- Simon Burton. Towards automated unit testing of Statechart implementations, *Technical Report (YCS 319)*, Department of Computer Science, University of York, 1999.
- Simon Burton, John Clark and John McDermid. Testing, proof and automation, an integrated approach, *Proceedings of the 1st International Workshop on Automated Program Analysis, Testing and Verification*, 2000.
- Simon Burton, John Clark, Andy Galloway and John McDermid. Automated V&V for high integrity systems, A targeted formal methods approach, *Proceedings of the 5th NASA Langley Formal Methods Workshop*, 2000.
- Simon Burton. Automated testing from Z specifications, *Technical Report (YCS 329)*, Department of Computer Science, University of York, 2000.
- Simon Burton, John Clark and John McDermid. Automated generation of tests from Statechart specifications, *Proceedings of Formal Approaches to Testing Software*, 2001.
- Karen Allenby, Simon Burton, Darren Buttle, John McDermid, Alan Stephenson, Mike Bardill and Stuart Hutchesson. A family-oriented software development process for engine controllers, *Proceedings of the 3rd International Conference on Product Focused Software Process Improvement*, 2001.

I declare that the work in this thesis is original work I undertook between the dates of registration for the Degree of Doctor of Philosophy at the University of York.

Chapter 1

Introduction

This chapter describes the motivation, context and intended contribution of the thesis. The pressures and constraints on safety–critical software developments are summarised, as are approaches that have been proposed to address these issues: formal methods, graphical specifications and automated testing. When used in isolation, the shortcomings of each of these techniques prevent their full potential from being realised within industrial software engineering processes. This chapter proposes that an integration of formal methods, graphical specifications and automated testing can overcome these problems and significantly increase the efficiency and effectiveness of high integrity verification and validation activities.

1.1 Safety–critical software development

Software is taking an increasingly important role in safety–critical applications, which range from aircraft "fly-by-wire" systems to chemical processing plants. The proliferation of software control is enabling more complex systems to be developed and manufacturers are continually extending the functionality and authority of the software within these systems. The failure of safety–critical software can endanger human lives and can also have severe financial and environmental consequences. As a result, there is a strong need to ensure that the software controlling these systems is of the highest possible integrity. However, increasing the complexity of the software also increases the risk that errors will be introduced during its construction and so, error detection practices and techniques must also be continually

improved.

The development activities typically associated with ensuring the correct and safe operation of the software are known collectively as Verification and Validation (V&V). During *verification*, the software is checked against some model of the system that represents the intentions of the designers and the wishes of the customer. This model is typically documented in a set of requirements specification documents. The requirements specify the functional and non-functional properties of the system under development. Functional requirements describe the actions of the system in terms of the operations that should take place in response to a series of external events. Non-functional requirements place constraints on this behaviour, such as memory limitations, real-time deadlines and safety properties. The system is also *validated* to ensure that any assumptions made during its development were correct and that the delivered system satisfies both the customer and safety certification authorities. By validating the requirements at an early stage in the development, much unnecessary and costly re-work can be avoided.

Figure 1.1 shows a lifecycle model [Pur89] that is often used to describe the flow of work in safety–critical software development projects. The left hand side of the 'V' illustrates the steps taken in constructing the software through the successive refinement of requirements into code. The right hand side of the model illustrates the activities carried out to verify, validate and certify the software. Figure 1.1 also shows example V&V relationships between different phases (as dotted lines) which are not typically shown in the standard model. These relationships form quality gates through which the product must pass before entering the next stage of development.

As well as the need for increased integrity, software developments also come under commercial pressures to shorten the product development time. Therefore in addition to increasing the effectiveness of the process at producing error-free software there is a competitive need to perform these activities more quickly and with less manpower.

1.2 Formal methods

Before a safety–critical system can be deployed, it must be shown to have been developed in accordance to the relevant certification standards or guidelines (e.g. DO-178B [RTC92] for commercial aircraft, Defence Standard (DS) 00-55 [Min97] for military equipment or

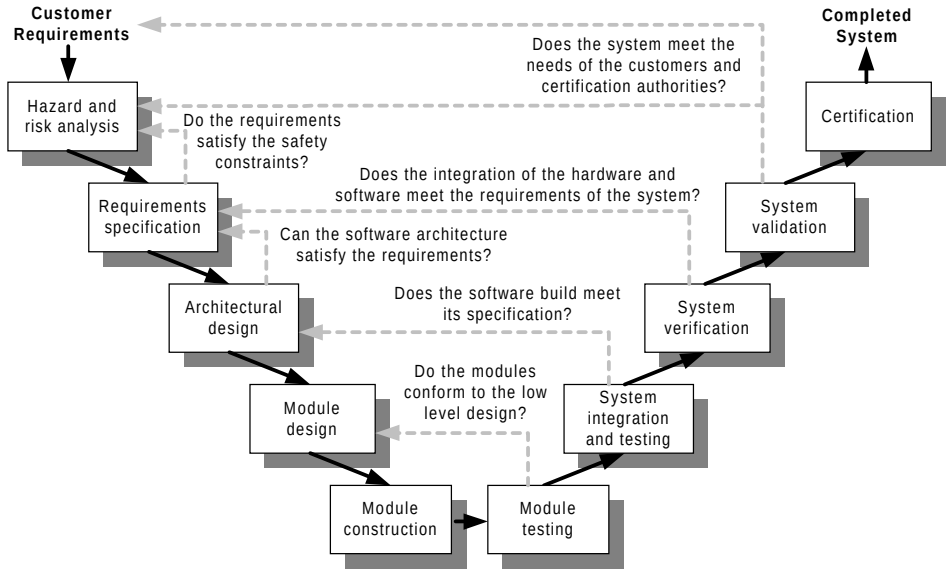


Figure 1.1: The V lifecycle model

IEC-61508 [IEC98], a generic standard for safety–related systems). DS 00-55, for example, calls for formalised development of software that includes animation of the requirements specification, static analysis and dynamic testing of code. Previous efforts to satisfy these standards have concentrated on the use of model-based formal specification notations such as Z [Int99] and VDM-SL [Int96]. Proof that the software conforms to its functional specification has also been performed using static analysis tools such as the SPARK Examiner [Bar97].

Fully formal development of code, where the specification is iteratively refined in provable steps until the software is produced, is typically only used on the most critical software projects or highly critical sub-components, e.g. reactor protection systems. Dynamic testing of the software is therefore required to verify those parts of the system that have not been formally proven. Indeed, even in the cases where the correctness of the software has been proven against its specification, testing is still required to validate that the mathematical model underpinning the development was appropriate, that assumptions made about the compilers and environment in which the software operates were accurate and that the system as a whole satisfies the customers' expectations.

The use of formal methods has long been advocated as a means of improving the devel-

opment of high integrity systems. Despite evidence to support this advocacy, e.g. [HB95, KHCP00], formal methods have still to gain widespread use in the software industry. This limited application of potentially useful verification techniques is due to the high cost of both constructing the specification and performing the proof work, as well as the need for engineers who are both experienced in the domain and have sufficient training in the formal methods. Industrial acceptance of formal methods will require the development of powerful tools to support the formal analysis and pragmatic approaches to using these tools within a software process [DR96, HB96].

1.3 Graphical specification notations

One of the greatest obstacles to the successful deployment of formal methods is the need to use a mathematical notation for the representation of the requirements specifications. Formal specifications can be extremely time consuming and costly to produce. Small changes in the requirements can result in much rework of the specification and accompanying analysis. The traditional approach to formal development therefore does not fit well with commercial environments where the requirements may change throughout the product's development.

Formal specifications must be validated against some higher–level requirements, typically written in natural language and some model of the environment in which the system operates. To perform this validation, expert knowledge of the application domain is required as well as the ability to review and understand the formal specification. If the formal specification is couched in a notation unfamiliar to the engineer performing the validation, or the specification is written in such a way that the review is unnecessarily complicated, then the effectiveness of the validation will be diminished. Leveson [Lev95] emphasises the importance of readability and reviewability of the specifications as follows:

"...the semantic distance between the model in the expert's mind and the specification model should be minimised. In addition, reading the specification or reviewing the results of an analysis should not require training in advanced mathematics."

Statecharts [Har87] is a notation that attempts to bridge the semantic gap between the engineer and the formal specification by allowing complex behaviour to be defined in concise diagrams. The graphical design and animation facilities of many of the tools (such

as Statemate [HLN+88] and MATLAB/Stateflow [TM01]) supporting the notation are a major contributing factor to its popularity. Other popular graphical notations that are also supported by commercial tools are SDL [Uni94] and UML [Cor99].

Statecharts go some way to satisfying Leveson's requirements on specification languages by hiding the formalism of the model but retaining a well defined set of semantics (although there are currently many different versions of the semantics in circulation [Bee94]). Statecharts provide a means of visualising changes to the system state and can be a more intuitive way of specifying functional behaviour than model–based notations such as Z or VDM-SL. However, the languages and supporting tools are often poorly integrated into the verification and validation parts of the process. The semantics of the notations tend to be complex and are often not obvious from inspection of the diagrams alone. This complicates the task of providing automated V&V support for these notations.

It is uncommon for only one graphical notation to be used to specify a large safety–critical system, or indeed that notations are used consistently between projects. Different graphical notations tend to be suited to different areas of functionality and the notations may also change as a result of commercial trends, new or outdated tool–support etc. It may not be cost effective to invest in automated V&V tool support particular to any one notation that may, in practice, only have a limited lifetime and audience. A generic approach to integrating graphical specification notations into the V&V process is therefore required.

1.4 Automated software testing

Despite the advantages of formal verification, testing remains the primary means of checking software against its specification. However, the rigorous testing needed to achieve the levels of confidence required in safety–critical software typically consumes more than 50% of total software development costs [Bei90]. The bulk of these costs involves the design, implementation and review of the tests rather than their physical execution. These tasks are extremely time-consuming and labour intensive. As a result they are often prone to human error. A means of automating these tasks would therefore greatly improve both the efficiency and integrity of the testing process.

Some test automation tools exist and are used in the industry already (e.g. AdaTEST [Inf97] and LDRA Testbed [Liv93]). These tools typically only provide mechanisms

for running the tests, evaluating outputs against some predetermined values and collecting structural code coverage metrics. In general, the tools are unable to generate test data from the requirements and therefore the test cases must be manually specified and validated.

Test case generation techniques developed in the research community have concentrated on deriving tests from formal functional specifications which, as described above, have not yet found widespread use in industry. The techniques also tend to be inflexible in terms of the criteria against which the tests are generated and their effectiveness at testing industrial applications has yet to be proven. Automated test case generation techniques that could be applied to the graphical specifications used in industry would therefore have a significant effect on the costs of V&V and also find a large market.

The criteria by which tests are selected are often ad-hoc or targeted towards achieving structural code coverage. The effectiveness of the tests at detecting certain classes of error is therefore uncertain and can lead to errors being missed and not detected until late in the development life-cycle or until the system is in operation. Any automated techniques developed as part of this thesis must therefore be explicit about the criteria used in the construction of the tests and allow these criteria to be extended and validated if necessary.

Testing is often made more difficult by decisions made during requirements specification and software design. A method of assessing the testability of requirements specifications and software architectures would therefore help to further reduce the cost of testing. The high cost of V&V is also compounded by the fact that it is often performed on many different iterations of the software. Iterations can be the result of faults in the software or faults in the requirements specification being detected late in the development lifecycle. Clearly, a better understanding of the requirements earlier in the project and a “right first time” approach to coding would dramatically decrease the recurring costs of V&V. This is obviously an ideal goal and would be difficult to foresee in practice. However, the use of tool supported graphical specification notations may allow the requirements to be validated early through peer review, animation, formal analysis and software prototyping. Automated test case generation would reduce V&V costs allowing more effort to be targeted at the requirements specification phase and at designing efficacious testing criteria.

1.5 An integrated approach

Formal methods have the potential for increasing the quality of high integrity software. In some situations the software can be proven to satisfy all of its requirements. Where this is not feasible, formal methods can be used to validate key properties of the requirements and provide a rich source of information for automated test case generation. Furthermore, by restricting the scope of the problem to particular application domains, e.g. Electronic Engine Controller (EEC) software for the aerospace domain, the set of mathematical problems facing the formal methods may be restricted to a set satisfiable by general purpose techniques or targeted heuristics. However, the formal notations themselves are currently a barrier to their use in industry.

Graphical specifications can also benefit V&V. They allow engineers to write and review specifications in a form that requires less training in formal mathematics and the resulting specifications can be animated and used to generate code to provide early validation of the specifications. In the cases where the graphical notations have a fixed semantics, formal analysis and test case generation may be applied to achieve some of the benefits of using formal specification techniques. Integrating these techniques into a practical engineering process requires a high level of automation. Furthermore, using the automation must not require significantly more skill in formal methods than is needed to construct the graphical specifications.

Based on the above observations, the primary contribution of the thesis can be summarised as satisfying the following hypothesis:

> **Graphical specification notations of proven practical use in the safety–critical domain can be formalised in a manner that permits the effective and efficient automation of V&V and in particular testing.**

To investigate this hypothesis, the thesis will develop and demonstrate a framework for applying automated analysis and test case generation to graphical specifications in a manner that is applicable to industrial high integrity software development projects. The framework will support a number of notations, as notations are rarely used consistently within and across projects. This will require the identification or development of a suitable intermediate formalism that is capable of capturing all the behaviour under test. Where possible, key tools within the framework will also remain generic (based on the intermediate formalism)

to reduce the effort required to integrate new notations, testing criteria and verification conditions. The thesis concentrates on the application of the framework to test case generation which requires the development of novel techniques in order to achieve the required level of integrity and automation.

1.6 Presentation of the thesis

The thesis will document the development of the framework and attempt to demonstrate the validity of the above hypothesis. In doing so, it will answer the following questions:

- **Is there a compelling need for the work?** The motivation behind the work can be found in preceding sections of this chapter.
- **What are the shortcomings of existing solutions?** The literature review (found in Chapter 2) will examine previous work in related areas and argue the absence of any existing solutions to the problem posed in the hypothesis.
- **How, technically, can the hypothesis be satisfied?** Chapter 3 presents a framework within which graphical notations can be integrated into a software engineering process and can be supported by formal approaches to analysis and testing. A translation of the two notations used throughout the thesis (Statecharts and PFS tables) into an intermediate formalism is presented in Chapter 4. These formal representations then form the basis of automated test case generation techniques that are guided by formalised testing heuristics. The problem of testing individual operations within the specification is first addressed in Chapter 5. Chapter 6 then presents techniques for testing sequences of operations, required when the complete system state is not directly observable and controllable or when specific scenarios are to be tested. Chapter 7 addresses the issue of test effectiveness and, supported by the formalisation of Chapter 5, examines the relative effectiveness of various testing criteria.
- **Is the approach theoretically sound?** The test cases will be formally derived from the specifications via a series of verifiable transformations. The resulting test cases will also be represented formally. Therefore, the integrity of the test generation techniques can be validated and under some conditions assumed to be correct through

construction. Chapters 5 and 6 which introduce the formal approach to generating tests also include arguments for the soundness of these techniques.

- **Can the techniques be applied to industrial scale examples?** The techniques will be demonstrated on case studies taken directly from an industrial application. Different aspects of the case studies are introduced in Chapters 4 to 7 to support the description and evaluation of various parts of the framework.

- **With what confidence can the evidence be used to support the hypothesis?** The degree to which the evidence submitted in the thesis can be used to validate the hypothesis will be assessed in Chapter 8. This chapter will examine the theoretical and practical aspects of the automated high integrity test generation methods. The results of the case studies and methods for assessing the results will also be evaluated with regards to how well they support the hypothesis or otherwise.

- **Has the hypothesis been adequately demonstrated?** Based on the evidence provided in the thesis, a decision will be made on whether or not the hypothesis has been successfully demonstrated. Factors for and against the hypothesis will be summarised and the method through which the argument was developed will be scrutinised. These final conclusions along with scope for future work will be documented in Chapter 9.

Chapter 2

Literature review

This chapter reviews a collection of techniques that form the background of the work described in the forthcoming chapters. Due to the integrative nature of the work, an exhaustive review of all related topics would prove too lengthy. Attention is therefore focused on the cornerstones of the framework; formalisation of graphical specifications, automated generation of tests from formal specifications and the use of automated formal methods in the analysis of graphical specifications and generation of test cases. Existing techniques are shown to be inadequate for the purpose described in the hypothesis and the originality of the work is therefore argued.

2.1 Introduction

This thesis investigates the integration of formal methods, graphical specifications and automated test generation techniques. In order to do so, the combination and improvement of a number of different techniques is required as well as the development of completely novel approaches to certain problems. This chapter reviews work that inspired the techniques described later in the thesis and highlights the shortcomings that led to the need for the integrated approach pursued here. By demonstrating the inadequacy of previous work to fulfil the requirements summarised in Chapter 1 (and discussed in more detail in Chapter 3), the originality of the approach described in the thesis is argued.

The review is focused on those areas of research considered core to the thesis: formal representation of graphical specifications, test case generation algorithms and automated

formal analysis techniques. Section 2.2 examines whether suitable formalisms exist that can be used as an intermediate formal representation of Statecharts (a graphical notation that will form the focus of many of the techniques described in this thesis). Section 2.3 reviews previous work on the generation of tests from formal specifications. Section 2.4 summarises automated formal analysis techniques that have been used in the validation of Statechart specifications and the generation of specification-based tests. The focus of the thesis (automated test generation) is reflected in the review and particular attention is paid to techniques for generating tests from formal specifications.

2.2 Formalisation of Statecharts

Statecharts [Har87] are a graphical notation for describing complex systems using concise diagrams. This is made possible through a combination of data and control constructs. The notation consists of extensions to traditional finite state machines (see Section 2.3.1) that include a complex transition label syntax, state encapsulation, orthogonality, a data store and an event broadcast mechanism. A fuller description of the subset of Statecharts studied in this thesis can be found in Chapter 4.

Many variants of Statecharts have been proposed. A good, though now outdated, summary of some of these was given by von der Beeck [Bee94]. The variants were typically designed to overcome problems in the semantics of the original notation and to optimise the notation towards particular application domains, e.g. [Mar91, JM94, LHHR94, BGK98, GCM98, MGB+98]. The variants differ in both the syntactic and semantic aspects of the notation. They can vary syntactically, for example, in the way that transitions can be triggered (by events only, or by certain values of variables, by timeouts, absence or combination of events etc.) or through the use or prohibition of inter-level transitions, state references and a state history mechanism. The notations also vary semantically, for example in the way the semantics are defined (operationally [HN96] or denotationally [HRdR92], using model-based specifications or algebraic specifications etc.) or through the restrictions that the semantics place on the behaviour of the system (e.g., determinism, duration of events etc.). The purpose of this thesis is to develop methods that can be integrated into industrial software development processes. Therefore the variant of Statecharts used here is that supported by the STATEMATE [HLN+88] tool from i-Logix.

Work has also been undertaken to formally specify the semantics of different Statecharts variants. Many different suggestions have been made using various notations for describing the semantics. Much of this work has been targeted at providing a deeper level of understanding of the behaviour implied by the diagrams. Other work has provided a translation from the diagrammatic form of Statecharts to a formalism that is more amenable to automated analysis and manipulation. For example, Mikk et al. [MLS97] suggested a semantically simpler graphical formalism as a means of clarifying Statechart behaviour. Behaviour that is otherwise obfuscated in the original notation, such as inter-level transitions, is explicitly specified in the new notation. The resulting hierarchical automata could then be used as an intermediate formalism to interface with tools such as the SPIN [Hol97] model checker. However, these tools would still need to interpret the semantics of hierarchical automata in order to process the specification and a further translation to an intermediate formalism (such as that used by the model checking tool) is required.

Other work by the same authors [MLPS97] defined the semantics of a subset of the Statechart notation supported by STATEMATE in the model-based formal specification language Z [Int99]. This provides a form which can be analysed and manipulated by standard type-checking and theorem proving tools and is therefore closer to satisfying the requirements of the hypothesis. However, for the purposes of test case generation, a formal definition of the semantics of the diagrams is of less interest than the formal specification of the transition structure of the particular Statechart under test. The conformance of this specification to the Statechart semantics could either be assumed or proven separately. The formalisation of the semantics as presented by Mikk et al. [MLPS97] may provide a useful basis for such a translation.

2.3 Test generation techniques

This section of the chapter examines test generation techniques based on formal specifications. As Statecharts are an extension of Finite State Machines (FSMs), techniques for generating tests from FSMs and Extended Finite State Machines (EFSMs) are an obvious place to start. However, the data component of the Statecharts is not covered well by these techniques and, inspired by the use of the notations in the formalisation of the Statechart semantics, techniques for generating tests from model-based specifications are examined.

This method of testing may also be applicable to reactive specifications as described by the PFS tabular notation (introduced in Chapter 4). The techniques are evaluated based on their applicability to the notations, potential for automation and ability to detect certain classes of fault.

2.3.1 Finite state machines

Finite state machines have been used extensively to specify telecommunication protocols, process control systems, lexical analysis, and many other applications. The problem of generating tests from FSMs is therefore well understood and much of the work has been driven by the need to demonstrate a telecommunication system's conformance to a standardised protocol specified using FSMs. A FSM is defined by a set of states. For each state, the system's behaviour is specified in terms of the operations that occur in response to each possible input and which state the system should move to, having consumed the input. A FSM is typically represented as a directed graph. The vertices of the graph correspond to states and the edges denote the transitions between the states. Some transitions may lead back to their originating state.

Often, the operations associated with the transitions are simplified to commands which output a single token in response to a single token input and are therefore trivial to check. A FSM is considered deterministic if, for each state, the reception of an input can lead to at most one transition being taken. A deterministic FSM M can be formally represented by the tuple $(S, s_0, X, Y, \delta, \lambda)$ where:

- S is the set of states, where s_0 is the initial state,
- X is the input language,
- Y is the output language,
- δ is the state transfer function, $X \times S \rightarrow S$,
- and λ is the output function, $X \times S \rightarrow Y$.

A fault model of finite state machines

Most specification-based test generation techniques are based on a fault-model that defines the class of potential errors that are detected by applying the tests against the implementation. In a seminal paper on the subject, Chow [Cho78] describes three ways in which an implementation can vary from its FSM specification. Assuming A' is an incorrect implementation of FSM A and both are minimal (there does not exist a machine with equivalent input/output behaviour with fewer states or transitions) they are:

- **Operation errors:** A' is not equivalent to A and can be made equivalent by changing *only* the output function λ of A'.
- **Transfer errors:** A' is not equivalent to A and can be made equivalent by changing *only* the state transfer function δ of A'.
- **Extra/missing states:** A' is not equivalent to A and can be made equivalent by reducing or increasing the number of states in A' and updating the δ and λ functions of A' accordingly.

FSM-based testing techniques can be evaluated and compared based on which of the above classifications of error can be detected using the technique.

Detecting operation errors

It is typically assumed that all operation errors can be detected by executing each transition in the FSM at least once. Several techniques have been developed to generate efficient transition covering test sequences. Initial work [NT81, SvB82, SvB84] led to the development of *transition tours*. However, these techniques resulted in test sequences that were not of optimal length. Later, an algorithm was presented by Uyar and Dahbura [UD94] that would find a minimal length transition tour based on the solution to the *Chinese postman problem*.

If each transition is given a positive weight, the *Chinese postman problem* [Kua62] is to find the minimum weight tour of a directed strongly connected graph. The algorithm given by [UD94] made use of the fact that a Chinese postman tour (CPT) exists for any strongly connected digraph and that an *Euler tour* of M is also a CPT of M. An Euler tour is a path through a graph G that contains each edge in G exactly once and starts and ends at the same

vertex. If the graph G representing M is symmetric then there exists an Euler tour and the CPT contains each transition exactly once. A directed graph is symmetric if each vertex has an equal number of incoming and outgoing transitions. If G is not symmetric then the CPT may contain some transitions more than once.

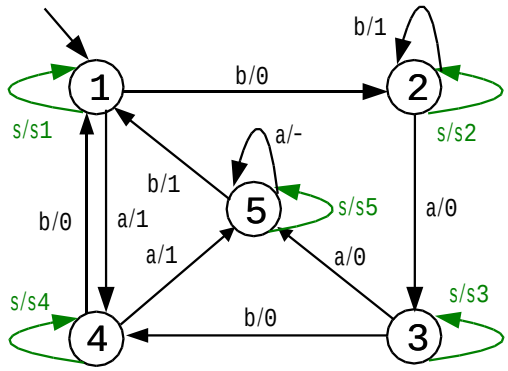


Figure 2.1: Example FSM with status transitions

Transition tour based techniques are only guaranteed to detect operation errors (based on the assumption that the error can be detected by a single execution of the transition). These techniques are not guaranteed to detect state transfer errors. A solution to this problem is often the inclusion of a *status message* and corresponding *status transitions*. Each state in the FSM is augmented with an additional reflexive transition. On receipt of a status message the status transition outputs a token unique to its state. The operation checking tour can then be extended with an additional status message/state identifier output pairing after each transition. Figure 2.1 (from [Ura92]) shows a FSM with added status transitions – those reflexive transitions that take input *s*.

Detecting transfer errors

The use of status transitions may not be possible in many implementations, for example, due to strict constraints on the input/output interfaces of the system. In these cases, the correctness of the transfer function must be inferred indirectly using state distinguishing sequences. Several techniques have been developed for generating such sequences.

The *W-method* [Cho78] generates test sequences that are the concatenation of elements from two sets P and Z. P is a set of sequences such that for each transition $s_i \rightarrow s_j$ in

a machine M with input x, there are input sequences p and px in P such that p forces the machine from its initial state s_0 to s_i. P can be constructed from the *reachability tree* of M. These sequences are used to test the output function of M. The reachability tree of Figure 2.1 (without the status transitions but including reset transitions) is given in Figure 2.2.

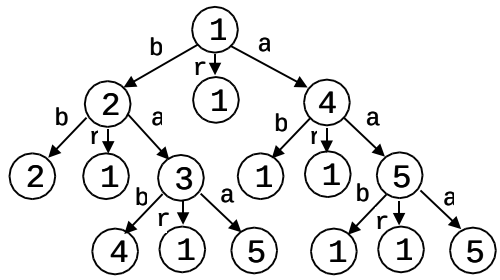


Figure 2.2: Reachability tree of the example FSM

The characterisation set W contains, for each state, a sequence that distinguishes that state from each other state in the machine M. If X is the input alphabet of M, the set Z is defined as $W \bigcup X \cdot W \cdots \bigcup X^{m-n} \cdot W$, where n is the number of states in the specification and m is the estimated number of states in the implementation. If the possibility of extra states is ignored then W can be used directly in place of Z. The sequences in P are used to move the implementation into the state s_i to be tested. The sequences in Z are then applied to verify the after state of the transition. The size of the test suite is proportional to the size of Z.

The *Partial W-Method* (Wp-Method) [FvBK+91] makes some improvements to the above algorithm and reduces the length of the test sequences. The implementation is assumed to have no missing or extra states and only a subset of W is used in certain cases.

The W-method and its variants determine exactly which state the implementation is in following a transition. Sabnani and Dahbura [SD88] observed that it was sufficient to only check whether the machine was in the expected state or not. The use of unique input/output (UIO) sequences was proposed to verify the transfer function. A UIO sequence for state s_i is an input/output trace unique for that state. UIO sequences can be generated for each state by checking increasingly longer sequences of input/output pairings until one is found that is unique for that state. Test sequences are then produced for each transition ($s_i \rightarrow s_j$) by concatenating an intialisation sequence from the initial state to s_i with the transition under

test and the UIO sequence for state *j*. Subsequences that are completely contained within other sequences can be removed and the remaining sequences can be connected using reset inputs.

The sequences generated using this approach are able to detect the same errors as the W-Method (assuming no extra or missing states) but can be much shorter. However it has been shown that it is PSPACE-complete to determine whether a state of a machine has a UIO and there are machines whose states have UIOs but only of exponential length [LY94].

Several improvements have been made to the UIO method to further optimise the length of the sequences. The Chinese postman tour and UIO method were combined [ADLU91] as a means of reducing the number of resets and subsequent intitialisation sequences required to fully test the machine using UIO sequences. The UIOs are used to construct a new graph with the same vertices as the original machine but whose transitions are augmented with transitions corresponding to the start and end points of the UIO sequences. A minimum cost tour is then calculated that traverses each UIO at least once. Further work by Shen et al. [SLD] and Ural [Ura92] describe methods by which UIO sequences can be chosen from a set of possibilities for each state such that the length of the resulting CPT is reduced. Other work [MP91, Hie97a] has exploited the fact that state checking sequences may overlap to produce shorter test sequences.

Evaluation of FSM-based testing techniques

The generation of test sequences from finite state machines is a well understood problem and a variety of techniques exist to generate sequences under various conditions and for various fault models. These techniques typically make assumptions that may or may not be practical in realistic testing conditions. For example, the reset feature assumed of the FSMs used in the W and UIO methods could result in errors going undetected which may otherwise have been found. Given that many errors which are due to additional states may only be reached after relatively long input sequences [FvBK+91], resetting the machine may prevent these problems from ever being encountered. The techniques also typically assume that there is never more than one point of failure in the implementation, or at least that state errors are not masked by subsequent errors in the state checking sequence.

Despite the similarities between FSMs and Statecharts, the techniques described above

are not directly applicable to Statecharts. Statecharts contain extensions to traditional finite state machines that greatly complicate the problem of test generation. Statecharts consist of a hierarchy of state machines that may include communicating orthogonal components. The interaction between these components must also be tested. The assumption that operation errors can be detected by a single execution of a transition may also not hold in Statechart specifications due to the complex data relations that may exist between the transitions. The following sections describe techniques that have been developed to test some of these extended features of finite state machines.

2.3.2 Extended finite state machines

The two most distinguishing features of Statecharts are the use of communicating concurrent components and the interaction between data (either in the form of input parameters or parts of the data store) and the control flow of the diagrams. These features complicate the task of specification-based testing. A summary of work addressing each of these issues separately is presented next.

Testing concurrency

The test generation techniques described so far apply only to systems consisting of a single FSM. Statecharts allow systems to be described using a set of communicating components. The single FSM-based techniques can not be applied to such systems because the triggering of a transition may be dependent on interactions with other components in the system. The problem of generating tests from concurrent systems of FSMs can be approached in either of two ways. The first approach is to combine the FSMs using reachability analysis to form a global FSM to which traditional FSM-based techniques can be applied. The second approach is to test each machine in isolation and then identify those parts of the system that interact. Tests are then generated to target these interactions.

Masiero et al. [MM94] described a reachability tree for Statechart specifications. Each node in the tree represents a Statechart *configuration*, a set of states that are active at any point in time. Beginning with the initial configuration (the initial states of each orthogonal component), the reachability tree is constructed in a breadth-first fashion by taking each possible combination of parallel transitions that can occur in the current configuration and

adding an edge from the current node. The edge represents the transitions taken and leads to a node representing the new configuration. The reachability analysis is stopped when a node is created that has already been encountered previously on the tree. However, the reachability graph produced does not accurately model the Statechart behaviour and is therefore insufficient for testing purposes. This is due to the definition of the stopping condition. A branch of the tree is terminated when a previously visited configuration is encountered. This assumes that when the system is in a configuration that has previously been reached, its future behaviour will already have been covered previously in the reachability analysis. This is not true as the values of variables in the Statechart may also affect the enabling of the transitions.

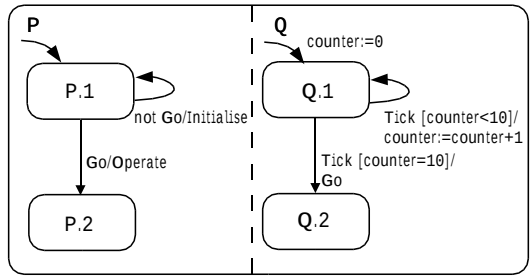


Figure 2.3: Simple Statechart with an internal variable

As an example, consider the simple Statechart of Figure 2.3. The initial configuration of the Statechart is $\{P.1, Q.1\}$. When the event *Tick* is received for the first time the value of the internal variable *counter* is less than 10 and the reflexive transitions will be taken, leaving the system in the configuration $\{P.1, Q.1\}$. At this point no other transitions can be taken as they both rely on the reflexive transition of state $Q.1$ executing a number of times more. The algorithm will terminate the reachability analysis as the configuration $\{P.1, Q.1\}$ has been visited previously in the analysis. This leaves two transitions and two states uncovered. Including the value of counter in the configuration would resolve this problem. However, in general, the inclusion of internal variables into the configuration for reachability analysis can cause the graph to rapidly increase in size as potentially an extremely large number of configurations may need to be covered. Even without the inclusion of internal variables in the configuration, the size of the reachability tree increases exponentially with respect to the number of orthogonal components.

Holzmann et al. [HGP92] observed that, for a section of a reachability graph that models transitions from concurrent machines performing independent actions, the outcome of all paths through that subgraph is the same. In this section of the graph, each machine performs a valid computation without violating any local assertions and all machines proceed together towards a composite final state. It follows from this observation that it is not necessary to systematically execute every combination of transitions in order to explore all possible states. Those sections of the reachability graph containing independent transitions are known as *partial orders*. Partial orders (also known as traces) are equivalence classes of sequences. Sequences belong to the same partial order if they can be obtained from each other by successively changing adjacent input symbols without changing the final state of the trace. Transitions are said to be independent if their triggering input affects only one machine and the behaviour of the transition does not affect any other machines in the system.

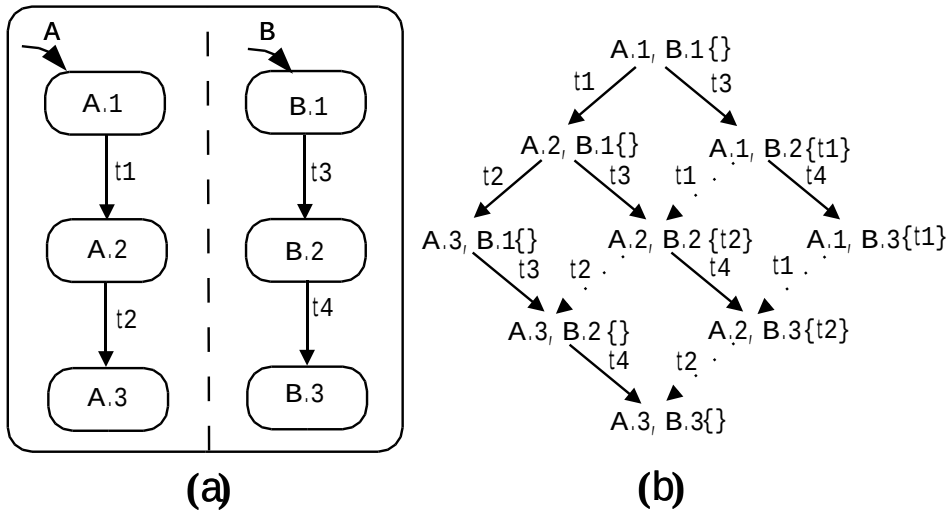


Figure 2.4: Sleep sets example

Sleep sets were first introduced by Godefroid [God90] as a means of computing the state reachability of concurrent machines using partial orders and therefore reducing the number of transition orderings that must be explored. Sleep sets are used to keep track of transitions that have been explored on an alternative path through the reachability graph and that are independent to the transition just taken. Transitions that are contained within the sleep set

are not explored as the reshuffling of the transitions would not result in a new partial order. An application of this algorithm is shown in Figure 2.4. Figure 2.4(a) shows a statechart containing only independent transitions and Figure 2.4(b) shows the corresponding reachability graph calculated using partial orders and sleep sets. Each node in the graph represents a configuration of states and the edges represent a transition (from either state component) that changes the configuration. The dotted transitions on the reachability graph, corresponding to elements in the sleep set, do not need to be executed to achieve full transition and configuration coverage. Each node is labelled with its corresponding sleep set.

There are several additional optimisations to reachability analysis, e.g. conflict sets [HGP92]. Optimisations on standard reachability analysis could potentially be used to construct the combined FSM from a system of concurrent machines that could be used in testing. Alternatively, the reachability analysis could be used to generate initialisation sequences to move the system into a state where a transition can be tested and where the initialisation sequence may need to traverse dependent transitions. Luo et al. [LvBP94], for example, produced a reachability tree of communicating Non-Deterministic Finite State Machines NDFSMs and applied the Wp-method (partial W method) to the reachability graph to generate the test sequences.

Hierons [Hie97b] describes a technique for testing Communicating Finite State Machines (CFSMs), where the transitions that can affect the behaviour of other machines are identified and tested separately. The system is defined as $M_1 \mid M_2 \mid \ldots \mid M_n$ where each M_i is described by the tuple $(S, s_0, X, Y, \delta, \lambda)$ with the same meaning as before. A transition t is defined as a communicating transition (CT) if its output is in the input alphabet of some other machine. Otherwise, it is a non-communicating transition (NCT). An example system of communicating FSMs is shown in Figure 2.5. The communicating transitions in Figure 2.5 are $t2, t5, t9, t13$ and $t16$.

Under certain conditions, it is possible to test a NCT t by moving the machine M_i into the initial state of t and executing t followed by an UIO sequence. A CT in M_i is said to feed back if its execution leads to a sequence of transitions including at least one more transition in M_i. Testing a CT that does not result in feedback is achieved by checking its output and final state and then verifying the final state of all machines affected by the CT (e.g. using UIO sequences). The test effort is minimised by finding a minimal cost sequence

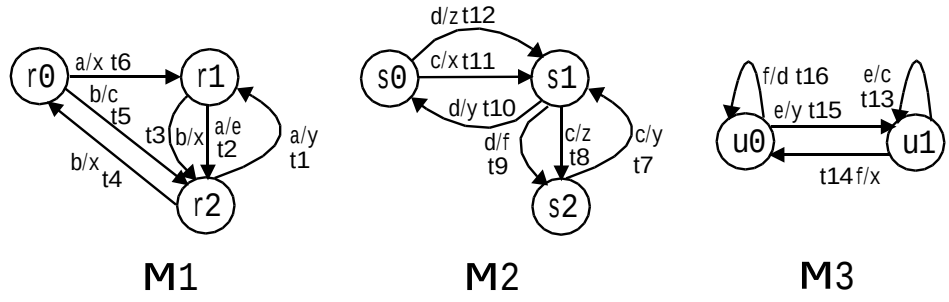


Figure 2.5: A system of communicating FSMs

that covers all NCTs and CTs.

The problem of testing semi-communicating finite state machines is complicated by interactions between parallel components that can mask faults in a transition under test. Consider a transition t from s_i to s_j in machine M_m that does not lead to feedback and produces output y that is in the input alphabet of another machine M_n. Fault masking can be caused by output errors where a transition in M_n consumes the incorrect output y', produces y and moves the system into the expected global state. If transition t can cause feedback, state transfer faults may also be masked. M_n may consume y and produce another output z which is in the input alphabet of M_i and that moves M_i from its incorrect state into the correct state s_j. To avoid these problems, a recent paper by Hierons [Hie01] proposes the use of test sequences that constrain the system to those configurations that do not lead to fault masking.

Combined control and data testing

The FSMs used in Statecharts as defined by Harel [Har87] are extended with data in the form of typed input and output parameters and internal variables. A transition may be triggered by a combination of events, input parameters and internal variables specified in the form of a *guard* predicate. The output function of the transition may transmit events (to other components in the Statechart or the environment) and update output parameters and internal variables. When testing Statecharts, it is not always sufficient to check the output function by only executing each transition once. Due to the interaction between the data and control flow it is also more complicated to move the Statechart into any particular state (e.g.

to test a particular transition). No work has been found to date that generates test sequences to simultaneously test both the transfer and output functions of Harel's Statecharts. Some work has studied the generation of test sequences from Extended Finite State Machines (EFSMs), FSMs that are extended with data in a similar way to Statecharts. This work is described next.

When testing an EFSM rather than a FSM, there are two main complications. Each transition is a relation over the internal variables, events, input parameters and output parameters and each state represents a set of values of the internal variables. Previous work in testing EFSMs can be categorised into three approaches: separate control and transition operation testing, combined control and data flow testing and fault-based testing.

The X-machine [IH96] approach to testing claims to be able to detect all faults based on an EFSM specification. X-machines are finite state machines extended with a *basic data set X* and a set of basic processing functions, Φ, that operate on X. Each transition in the machine is labelled with a function from Φ. The behaviour of the system is defined by sequences of applications of functions from Φ relating to sequences of transitions. *Stream* X-machines define the data set X as:

$X = \Gamma^* \times M \times \Sigma^*$

and each processing function as:

$\Phi : \Gamma^* \times M \times \Sigma^* \rightarrow \Gamma^* \times M \times \Sigma^*$

where Σ is the set of possible inputs, Γ is the set of possible outputs and M is the internal memory of the machine. Φ is therefore of the form whereby given a value of the memory and an input value, Φ can change the memory value and produce an output value. The input value is then discarded.

The stream X-machine testing method is as follows: the processing functions Φ are assumed correct (or independently testable in the implementation) and distinguishable by examining their outputs. A version of the X-machine is then produced that removes the details of the internal memory M and processing functions Φ leaving the inputs and outputs for each transition. The W-method is then applied to test the transfer function of the machine. The same set of assumptions was applied to the problem of generating tests from Statecharts [BHS98]. This paper also addressed the problem of testing concurrency in the Statecharts by computing the product machine of the orthogonal components. This would

lead to state-explosion in systems containing a large number of states and/or orthogonal components and does not address the issue of testing Statechart implementations that do not have independently testable transition functions.

Despite the claim to be able to find all faults, the assumptions made by the X-machine approach are impractical in many cases. In particular, the assumption that the data processing functions are independently testable requires that the refinement into the implementation preserves the structure of the specification. This is often not the case.

Several authors [MP92, CZ93] have proposed the application of data flow testing [RW85] to EFSMs in combination with control flow testing. Both these papers were based on the specification language *Estelle*, a formal description language based on the EFSM model. The papers use different algorithms for generating test sequences that cover all states and transitions and also all *definition-use (du)-paths* of variables in the system. A definition-use path is a sequence of transitions for a variable x such that x is assigned a value in the first transition and used either in a predicate or computation in the last transition of the sequence. Furthermore, the variable is not re-defined in any of the intermediate transitions. The transfer function of the machine is tested in the usual way using status transitions, the W-method, UIO sequences or one of their derivatives. The fault coverage of these methods is therefore equivalent to the combination of the state checking method used and all du-paths. In [CZ93] an optimisation of the UIO sequence approach is offered where checking sequences are only applied in converging states (states whose outward transitions lead to a common state). This is based on the intuition that any checking sequence for a state is also a checking sequence for all non-converging states on the sequence of transitions leading to the state that does not cover any converging states.

The test sequences described in [CZ93] are generated by first identifying all du-paths in the system and then generating sequences that traverse these paths from the initial state and lead back to the initial state. A state checking sequence is applied whenever a transition on these paths ends at a converging state. If the checking sequence defines a variable that is the basis for the current du-path, separate sequences must be used to check the data flow and state transfer behaviours of the transition. The authors of the paper suggested constraint satisfiability techniques [Mac77] to decide the executability of the sequences.

The same paper also proposed a solution to the problem of influencing self-loops. These

are reflexive transitions that define variables that have a use in another transition. An example of such a transition is given in Figure 2.3 (page 30). The transition from state $Q.1$ to itself defines the value of *counter* that is used in another transition ($Q.1 \rightarrow Q.2$). Standard techniques would result in a path that involves executing transition $Q.1 \rightarrow Q.1$ once followed by transition $Q.1 \rightarrow Q.2$ which is not executable. Symbolic execution was used to identify the recurrence relation of the looping transition and the exit condition of the state. The recurrence relation for the example is $counter := counter + 1$, which leads to the solution $counter_k = counter_0 + (1 * k)$, where k is the number of iterations of the transition. The exit condition for the state is $counter_k = 10$. The solution to these two equations results in $k = 10$ (where $counter_0 = 0$). Therefore, before the exit transition can be taken, transition $Q.1 \rightarrow Q.1$ must be executed 10 times.

An alternative to separate control and transition operation testing and combined control and data flow testing is the fault-based testing technique proposed by Wang [WL93]. In this technique, an explicit fault model of the system is constructed as a mutation of the original EFSM. Test sequences are then generated that are guaranteed to distinguish between the correct specification and the mutant. Although, leaving the fault-model to be specified by the tester provides a greater amount of flexibility in the faults that can be detected, it also involves much greater effort on the part of the tester. Each hypothesised fault must be modelled explicitly and a test sequence generated to detect this fault. This is a flexible but inefficient approach compared to other methods described in this chapter which are typically designed to detect a *class* of faults. Ideally, methods developed in this thesis will have the fault-model flexibility of Wang's techniques but have the efficiency of other techniques designed to detect general classes of errors rather than particular mutations.

Evaluation of extended finite state machine testing

This section has shown that techniques exist that can be applied to the various extensions of FSMs that form the Statechart notation. However, no techniques have been found as yet that target all these extensions together and could therefore be applied directly to Statecharts. A test generation technique for Statecharts would require a combination of the techniques described above. Furthermore, before test generation techniques can be developed, a fault-model of Statechart specifications must be defined against which the tests can be evaluated.

One of the key issues in generating test sequences from EFSMs is to avoid combinatorial explosion of the model used to derive the tests. The techniques described above attempted this in a number of ways. Partial sequences were developed to reduce the state-space of reachability analysis [God90] and the notion of semi-independent finite state machines [Hie97b] was suggested as a means of avoiding concurrent reachability analysis altogether. Similar intuition can be found in the combined control and data flow testing techniques [MP92, CZ93, WL93], where test sequences are targeted at detecting particular faults in the data operations rather than exhaustively testing all combinations of data values. The techniques did not address the issue of testing communicating EFSMs that share data interactions. This must be addressed when generating tests from Statechart specifications. Several of the techniques have a potential application to Statecharts. However before they can be applied, the techniques will need to be either extended or a suitable abstraction of Statecharts found so that they can be directly applied.

2.3.3 Model-based specifications

Section 2.2 described work that has used model-based notations [MLPS97] to formally describe the semantics of STATEMATE Statecharts and to represent particular Statechart specifications [Val98]. Model-based specifications facilitate proof-based analysis and can also provide an explicit definition of both the control and data parts of the specification for testing purposes. As will be demonstrated in the following chapters, model-based notations also have the potential for providing a common intermediate formalism for several types of graphical and tabular notations. It is therefore prudent to review work that has been carried out into automating the generation of test cases from model-based specifications.

Testing model-based specifications

Model-based specifications are based on predicate logic and set theory. The functionality of the system is expressed in terms of operations and functions that transform the state-space of the system. These operations and functions are expressed in terms of a standard collection of data types and type constructors. The state-space is restricted to valid states using state invariant predicates. Operations are expressed in terms of a pre-condition over the state and inputs that must hold before the operation can occur and a post-condition that

restricts the values of the state and outputs following the operation. Statechart transitions can be defined similarly where the guard and current state form the operation pre-condition and the action and after state form the post-condition. The state invariants are the set of constraints on the current set of active states corresponding to the structure of the Statechart and any type constraints placed on the variables referenced in the transition. Due to this analogy, techniques used to test model-based operations could potentially be applied to Statechart transitions.

Two model-based notations that have received a fair amount of research attention and tool support are Z [Int99] and VDM-SL [Int96]. Testing based on Z specifications was first proposed by Hayes [Hay86] in the context of testing abstract data types. Hayes described the testing problem as checking that the state invariants and pre-conditions are maintained in the implementation as well as the input-output relation. Hall [Hal88] first described the possibilities for automatically generating test domains for Z specifications based upon the partitioning of input sets, output sets and states. He suggested automatically parsing the specification while applying attributed grammar style operations to develop the partitions.

Dick and Faivre [DF93] suggested partitioning the specification into disjunctive normal form as a means of automatically extracting test cases from VDM-SL specifications. However, this leads to unstructured test sets with limited fault detection properties. The paper also described a method by which test cases could be sequenced, an important topic that will be revisited in Chapter 6. A more structured and flexible approach to test case derivation from model-based specifications is described next.

Category-partition testing

The category partition method [OB88] is based on partition testing and equivalence classes. The principle of partitioning the input domain with respect to properties of the specification was first introduced in 1975 in a paper by Goodenough and Gerhart [GG75]. Equivalence classes are formed by partitioning the input domain into sets of data that exhibit similar behaviour in the specification. It is then assumed that only one test point needs to be selected from each equivalence class to verify the implementation against the specification.

The first step of the category partition method consists of decomposing the specification into functional units that can be independently tested. The functional units are then analysed

to identify the *parameters* and *environment conditions* that affect the function's behaviour. Parameters are explicit inputs to operations and environment conditions are properties of the system state which hold when the function is executed. The next step is to choose *categories* of information that characterise major properties of the parameter or environment condition. Each category is then partitioned further into *choices* which represent sets of similar values (equivalence classes). The individual test cases can then be created by selecting values from choices for each category, taking into account the constraints on how the different choices interact.

Amla and Ammann [AA92] took Ostrand and Balcer's description [OB88] of the category partition method and used it as the basis for a method of deriving tests from Z specifications. The following mapping was given between properties of a Z specification and properties of the informal specifications described by Ostrand and Balcer:

- Each Z operation schema that changes or observes the state and any schema that produces an output is an independently testable functional unit.
- The input variables to the operation schema equate to parameters.
- State variables that are used in the pre-condition of an operation equate to environment variables which are used in specifying environment conditions.

There are two sources of category information in the Z specifications:

- The predicate part of the specifications (pre and post-conditions),
- The definitions of the parameter or environment variables, e.g. type information and state invariants.

This technique can be applied to the following Z operation that defines the integer square root $r!$ of the input $n?$, where $?$ and $!$ are Z convention to denote input and output parameters respectively. Where an exact square root is not available the greatest value integer is chosen that when multiplied by itself does not exceed the input.

SquareRoot

$n? : \mathbb{Z}$

$r! : \mathbb{Z}$

$n? \geq 0 \land$

$r! * r! \leq n? \land$

$\neg(\exists j : \mathbb{Z} \mid j > r! \bullet j * j \leq n?)$

The operation is itself an independently testable unit and has one input parameter ($n?$) and no environment variables. Category information can be derived from the type of the parameter and the pre-condition of the operation as follows:

- **Category 1:** $n? : \mathbb{Z}$
- **Category 2:** $n? \geq 0$

The following choices can be further identified from these categories:

- **Category 1 choices:** $n? < 0, n? = 0, n? > 0$
- **Category 2 choices:** $n? = 0, n? = 1, n? > 1$

Note that the first choice of category 1 is infeasible as it invalidates the pre-condition of the operation.

Stocks, Carrington et al. [CS94, MC98, MCM$^+$98, SC93, SC96] extended the partition-based approach applied to Z specifications by showing how the tests and the test suite structures themselves can be formally specified using the same notation as the specification (Z). However, the heuristics used to partition the specification were left informally specified and the scope for automating the partitioning process itself was left unexplored. The Classification Tree Method [GG93] is a similar technique for repeatedly applying test heuristics to a specification to derive a hierarchy of abstract test case specifications. Leaves in the tree are combined and instantiated to form the test data. Tool support [SCES97] has been developed for this method and allows a tester to structure and select test cases using an intuitive user interface. The testing heuristics used by the method were restricted to type analysis and reduction to disjunctive normal form.

The techniques described above have addressed the problem of constructing test cases but have not included the issues of selecting test data or checking the outputs of the implementation against the input/output relations given in the specification. The ability to select representative test data and to check the results of the tests against some correct implementation of the requirements is essential to be able to implement a complete testing strategy. Therefore it is important to ensure that test case generation techniques are compatible with appropriate test oracle and test data generation techniques.

Fault-based testing

An alternative and complementary approach to category partition testing is to target the generation of test data at revealing specific faults in the implementation that can be represented as mutations of the specification. This method of test data generation (hereafter referred to as *fault-based*) was inspired by mutation testing [Bud85, DLS78]. Mutation testing involves generating a number of copies of the program under test, each modified slightly to represent a fault typical of those that might occur during actual development. Test data that are generated to detect these faults (kill the mutants) are then assumed to have a good probability of detecting arbitrary faults in the actual program under test. Budd [BG81] extended this concept to the domain of specifications given in predicate calculus. Potential faults are represented as mutants of the specification. Data is then generated that can distinguish between the mutated and original specification. Although a number of potential mutations were given as examples, a technique for generating data to kill the mutants was not discussed.

A similar approach was developed by Kuhn [Kuh99]. Faults were represented as mutated formal specifications and the conditions for distinguishing them from the original were calculated. This information was then used to calculate a fault coverage hierarchy. Although Kuhn's approach was more formal and could therefore be used to analyse properties of the error detection condition, detection conditions were shown only for mutations involving Boolean logic operators. Automation of the method was not discussed.

Recent work [VB02] has used the Z notation to specify certain testing criteria (Modified Condition/Decision Coverage and Reinforced Condition/Decision Coverage). This allowed the authors to prove and compare the fault coverage properties of the criteria. However, the

use of the formalisation to automatically generate test data satisfying the criteria was not discussed.

Mutation testing has also been used to assess the quality of specification-based testing techniques and to guide the selection of test heuristics to generate more efficient tests which also achieved a high mutation score [AW96]. In this case the algebraic specification notation OBJ was used.

Evaluation of model-based specification testing

Model-based specifications have the potential to form the basis of high integrity automated test case generation activities. They allow for an explicit description of the behaviour under test and are amenable to formal analysis and automatic manipulation. Various techniques have been developed for extracting test cases from specifications written in Z and VDM-SL. These include simplistic but fully automated approaches such as reduction to disjunctive normal form and more structured approaches where the testing heuristics must be supplied by the user and much less automation is provided. A balance needs to be achieved that allows for a flexible approach to choosing testing strategies but automates the application of these generic strategies to the specification to produce the test cases.

Regardless of how the test cases are derived, the process of checking the test inputs and outputs against the specification requires a *test oracle*. If enough refinement information is known to transform the concrete inputs and outputs of the implementation into their equivalent at the specification level, the formal specification can be used to form a test oracle. Test oracles can be derived in conjunction with the test cases [RAO92, HP95] and therefore also have the potential for automation.

By using formal notations to represent the test case specifications themselves, automated test data generation becomes possible. Yang [Yan95] used genetic algorithms to generate data to populate the test cases. This data was designed to show an implementation's conformance to its specifications. Therefore it was only possible to draw conclusions about the correctness of the implementation with respect to the set of test data with which it was exercised. Similar techniques were used by Tracey [TCM98], this time using code-level requirements in terms of function pre and post-conditions and simulated annealing was used as the data searching technique. Tracey's techniques differed from Yang's how-

ever, by searching for data that would show the implementation's non-compliance from its specifications and therefore has more worth as a targeted fault-finding technique.

Despite the potential of testing based on model-based specifications, the reliance on formal notations such as Z and VDM-SL (as well as the under-developed automated support for the techniques) has hindered the popularity of the techniques. This is equally true for techniques based on algebraic specifications [MG83, BGM91]. Test generation based on algebraic specifications has not been studied in detail here as the approach is better suited for testing primitive data types rather than the larger specifications that would be produced for the systems of interest to this thesis.

2.4 Automated formal analysis

2.4.1 Automated formal analysis of Statechart specifications

Model checking techniques [CK96, CW96, Hol97] have recently been applied to the problem of analysing certain properties of Statecharts. Model checking is a highly automated technique that exhaustively explores the state space of a specification searching for counter-examples to particular user specified properties. One of the main tasks involved in applying model-checking techniques to Statecharts is limiting the size of the state-space that must be explored. Several features of Statecharts such as state hierarchy, orthogonal states and data variables can cause *state explosion* leading to an infeasibly large state-space for the model checkers to explore.

Alur and Yannakakis [AY98] explored the effect that state hierarchy has on model checking FSM specifications and proposed methods for avoiding the state explosion problem. This paper showed that properties of hierarchical FSMs could be checked without flattening the states and therefore avoiding one of the causes of state explosion. Model checking has also been applied directly [BBD+99] to Statechart diagrams as supported by the commercial STATEMATE [HLN+88] tool. In this approach, safety properties and counter-examples are respectively specified and reported graphically using timing diagrams. Without abstraction, model checking practical designs can be time consuming and in many cases infeasible. The authors presented a compositional approach to verification as a means of restricting the state-space explosion and described some primitive abstraction techniques

that made certain properties more tractable.

Model checking is a highly automated approach to verification but is restricted in the size and types of systems to which it can be applied due to the state explosion problem. Theorem proving is an alternative approach that can potentially be applied to a much wider range of systems than model checking (including those with infinite state-space). However, theorem proving in general has much less automated support and typically involves a great deal of highly skilled effort on the part of the user to prove even fairly simple assertions about a system. Little work has so far been carried out into the application of theorem proving techniques to Statechart specifications. Armstrong [AB96, Arm98] has shown how Statechart specifications can be represented formally and theorem proving techniques used to verify safety properties of the specification. The theorem prover Proofpower HOL was used to semi-automate the proof work. However, this required many trivial sub-goals to be discharged that may otherwise have been "assumed" in a manual approach. This greatly increased the amount of effort required to discharge the proofs. The authors suggested that many of the problems involved in applying automated theorem proving were due to the generic nature of the theorem provers. Specialised proving tools dedicated to the particular logics used to represent (for example) Statecharts would result in much more effective automation.

2.4.2 Model checking for test case generation

One of the first papers to describe the application of model checkers to the automated testing problem was by Callahan et al. [CSE96]. The authors used the SPIN model checker [Hol97] to demonstrate the conformance of a program trace (the values of selected variables measured at different points in the program) to a model of the specification. The authors also demonstrated how sequences could be generated that belong to a particular equivalence class defined by a specification of a sequence constraint.

Model checkers have also been used to generate test cases [ABM98] based on a fault-based approach similar to that of Budd [BG81]. The model checker specification is mutated and standard model checking is then used to generate scenarios that distinguish between the original specification and the mutation. This technique has the advantage of being automated and also has the ability to automatically detect equivalent mutants (using this

technique if a mutant is equivalent, then no counter-examples can be found). This work was based on specifications written using SCR [Hen80] tables. The same authors later extended the work [AB99] to evaluate the mutation coverage of particular test cases specified as properties to be proven in the model. If an inconsistency was found between the specification of the test case and the mutated model, the test case was said to kill the mutant.

Provided a tractable abstraction of Statecharts can be found, model checkers have the potential for automatically generating test sequences in a highly automated manner. Also, model checking can be an extremely flexible mechanism as the type of test sequences generated depends entirely on the property to be checked which is expressed in terms of temporal logic formulae. If a mapping between test criteria and these formulae can be found, an automated approach can be envisaged whereby the test criteria can be expressed in the same notation as the original specification and tests automatically generated according to these criteria.

2.5 Summary

This chapter examined three areas of work in an attempt to identify suitable techniques that could be used in the automatic and formal derivation of high integrity test sets from graphical notations. Section 2.2 showed that the semantics of Statechart specifications have been thoroughly examined and that many variations of the notation now exist. The semantics of Statecharts have been formalised many times using various formal notations. However, with a few exceptions (e.g. [MLS97]), most of this work has been targeted towards providing a formalism such that the semantics of Statecharts can be investigated in greater detail. As such, the formalisms tend to include more detail than is required for testing, where in most cases the conformance of the specification to the chosen semantics can be assumed or proven as a separate activity. A formalisation of Statecharts is therefore required that is amenable to automated and formal analysis but contains just that information required to verify the conformance of the implementation against the transition function of the specification.

Section 2.3 examined various techniques for generating tests from different formalisms. These formalisms can be used to represent different aspects of a Statechart specification such as finite-state machines, transition functions and concurrent processes. However, no

testing technique was found that covers all aspects of Statecharts. Therefore, in order to be able to exploit the large body of work in testing from formal specifications, either extensions to the techniques must be found or a suitable abstraction of Statecharts must be defined such that the techniques can be applied directly. A higher level of automation of the EFSM and model-based specification techniques is also required than has been presented so far in much of the work reviewed.

Section 2.4 examined work that applied automated or semi-automated formal analysis techniques to properties of Statecharts and test case generation in general. Theorem provers require very targeted problems in order not to require a highly skilled level of interaction from the user. Model checking is a fully automated technique, once the Statechart and properties have been couched in a suitable representation, and therefore shows more promise. However, model checking becomes intractable for larger specifications. In these cases a property preserving abstraction must be applied to reduce the state space explored by the model checkers. It has been shown that model checking can be applied to the problem of test case generation. If suitable abstractions can be found, model checking has a potential use in generating test cases from Statechart specifications.

In conclusion, no work has been found to date that completely addresses the high integrity generation of tests from graphical specifications. However, some promising techniques exist that have been applied to formalisms similar to individual aspects of Statecharts. Techniques must be developed that are able to automate the generation of suitable tests for all aspects of Statechart specifications (control flow, data flow and concurrency) while avoiding the state explosion problem and to do so in a manner in which the testing criteria are flexible and the fault coverage of the techniques can be assessed.

Chapter 3

An automated V&V framework

This chapter summarises the technical aspects of the thesis and describes the process context within which they were developed. Success criteria for the work are defined as are some constraints that will be used to limit the scope of the techniques and increase the tractability of the problems. The individual technologies required to perform the integration of graphical specifications, formal methods and automated testing are introduced.

3.1 Introduction

Chapter 1 introduced the constraints on high integrity software development processes and described how traditional approaches to applying formal methods and automated testing to these processes have not yet had a significant impact on the cost or quality of systems produced in industry. It was proposed that an integration of graphical specifications, formal methods and automated testing techniques was required. Chapter 2 reviewed work in applying formal methods to graphical specification notations and generating test cases from various formal notations. However, none of the published work has fully addressed the issues involved in automatically generating test sets from the types of graphical and tabular notations used in the safety-critical industry.

This chapter summarises the technical aspects of the thesis and places the work within the wider context of the software engineering process and the constraints placed on the process by the target domain. A framework is presented for the integration of graphical specifications, formal methods and automated V&V. The framework can be exploited for

both specification validation (by proving particular properties in the specification) and software verification (by generating test sets and program proof obligations). The majority of the thesis will focus on applying the framework to the automatic generation and evaluation of test sets. However, evidence that the framework is applicable to specification validation will also be presented. The remaining chapters of the thesis describe the techniques developed to implement the framework and applies them to industrially applicable examples.

Criteria for the success of the framework are presented in the following section. Section 3.3 presents the constraints that will be placed on the scope of the work in order to increase the tractability of the problems to be solved. The framework itself is then summarised in Section 3.4, followed by more detailed descriptions of the key technologies that must be developed in order to implement the framework.

3.2 Criteria for success

The objectives of the framework are derived from the current needs of the safety–critical software industry and the shortcomings of existing approaches to test case generation presented in Chapters 1 and 2 respectively. Based on this assessment, the following set of criteria have been identified that will need to be satisfied by the framework for it to be considered a success in terms of improving existing software engineering practices.

1. The framework should be based on specification notations that have been proven effective for modelling systems in the domain of interest (i.e. safety–critical embedded control systems) and do not require expertise in formal methods in order to create the specifications.

2. A sufficient level of automation should be achieved such that input from the user is in the form of the specification and a description of properties to be verified or the testing criteria to be met. The supporting toolset should interpret these properties and perform the necessary formal manipulation of the specifications to produce the desired result.

3. The framework should be capable of collecting the verification and testing evidence required to meet certification standards and guidelines including any formal specification and proof required.

4. The framework should be flexible enough that it can be applied to a number of specification notations, while re-using key tools and techniques.

5. The outputs of the framework should be in a form such that the integrity of the results can be reasoned about to a high level of confidence.

6. The techniques should lead to a more cost-effective development process than existing methods.

3.3 Framework constraints

The primary case studies for the thesis are based on Electronic Engine Controllers (EECs) for aircraft engines. These are real–time, safety–critical, fault-tolerant computer systems embedded in complex engineering products. In particular, the work focuses on the discrete aspects of the systems. Well-established mathematically based processes are already in place for modelling and validating the continuous aspects of the control software and are outside the scope of this thesis.

Typically a variety of engineering notations is used to specify such systems. However, due to the restricted domain, all specifications share common attributes and capture similar information which reduce the set of (mathematical) problems that need to be solved when applying formal analysis to models of the specifications. By focusing on this set of problems it is anticipated that a greater amount of automation may be achieved than if solutions were designed for a wider scope of applications. In analysing example system specifications several common characteristics were discovered:

- The software requirements did not involve the storage and maintenance of complex information structures, typically only integer or 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. 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, i.e. only one outcome is specified for each situation.

Notations such as Statecharts [Har87], the Software Cost Reduction (SCR) method [Hen80] and PFS [GCM98] have been designed with these restrictions in mind and are therefore less expressive than formal notations such as Z [Int99]. Therefore only a subset of the intermediate formal notation will be needed to provide the intermediate representation of the engineering notations.

In addition to the restrictions placed on the domain and specification notations, the framework will also be limited to the verification of certain types of properties. It is anticipated that different intermediate formalisms will perform best for different types of properties. For example, for analysing temporal properties of the specification, a formalism based on temporal logic would be most suitable. This thesis will focus on the functional properties of the system. The intermediate formalisms and techniques therefore only need to be able to represent the functional behaviour of the system. Other instantiations of the framework could replace the formalism with another more suitable to, for example, temporal properties.

It is also assumed that the means of recording the graphical and tabular specifications are already in place, e.g. through the use of commercial tools such as STATEMATE [HLN+88]. Furthermore, the process will be supported by a high level requirements structure from which the specifications are refined.

3.4 Overview of the technical approach

To achieve the objectives outlined above, a combination of technologies is required. Systems in the restricted domain of embedded safety-critical control systems are modelled using graphical or tabular specification notations. The specifications are then translated into a common intermediate formal notation which will form the basis of the automated V&V activities. This allows for a formal validation of key properties of the specification that may detect errors at this early stage of development and allows certain assumptions to be made to simplify testing. The formal representation also forms a solid basis for the test case generation activities as all behaviour in the specification is explicit and in a machine processable form.

These activities are designed to be performed within the wider context of a quality controlled software development process. Therefore the intermediate work products (such as the intermediate formalism, test case design criteria, specification validation proofs) should

be presented in a form that can be reviewed for correctness. Additionally, feedback from later in the process should be used to assess the effectiveness of the specification validation and test case generation activities. The techniques are therefore designed to be flexible such that the verification criteria can be updated as part of a self optimising process.

As the automated specification validation and test case generation activities will be performed in the formal methods domain, the results of these activities should be reported in such a manner that engineers need no more training to interpret the results than to write the original graphical specifications. This is required in order to satisfy the success criteria outlined in Section 3.2.

3.4.1 Formal representation of graphical specifications

Although graphical and tabular notations are popular with engineers, their semantics are often unclear. This can lead to the unintentional specification of invalid (with respect to the high level requirements) or hidden behaviour (the dynamic behaviour of the specification is not clear from inspection of the diagrams alone). This is a recognised disadvantage of using such notations. However, adhering to a well understood subset of the notation, following usage guidelines and applying formal analysis to verify certain properties can help.

The approach taken in this thesis is to translate the graphical requirements into a core formal notation. The behaviour implied by the specification is made explicit for the purposes of V&V which can be supported by formal reasoning. By explicitly specifying all 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. The thesis demonstrates this approach by formalising two notations used within the aerospace domain and applying a common set of validation and test case generation tools. A strict translation process allows a fixed structure to be enforced on the resulting formal specification. Knowledge of this structure is exploited in the development of automated heuristics, e.g. for generating test data and discharging proof obligations.

The translation requires a definition of the semantics of the graphical notations. Notations are therefore chosen for which a formally specified semantics already exists (in

particular, for Statecharts, the semantics that is chosen describes the animation capabilities of the STATEMATE tool). Chapter 4 describes the subset of Statecharts used in the case studies and a tabular notation based on PFS Tables [GCM98] for describing reactive behaviour. Based on a definition of their semantics, a translation to Z is described. The thesis then focuses on how automated test generation techniques can be applied to the formal representation of the specifications.

3.4.2 Specification validation

Many accidents in which software was involved can be traced to flaws in requirements specifications of the corresponding safety–critical systems [Lev95]. Therefore, before the graphical requirements are used as the basis of any coding or testing activities they should be validated against the high level requirements and other desirable properties. This is best achieved using a diverse range of techniques. In addition to traditional review methods, the use of modelling tools allows the specifications to be animated and prototype code to be generated. Producing a formal representation of the specifications also allows for a more rigorous approach to validation through proof. If desirable properties of the requirements such as completeness, determinism or safety constraints can be formally specified, the translation process can be used to produce the proof obligations for these properties. By using a generic template for each type of proof obligation, repeating patterns in the proofs can be exploited when automating the proof work.

Adherence to certain properties such as completeness and determinism also allows for some simplifying assumptions to be made when automatically generating specification covering test sets. In defining the translation to the formal representation, Chapter 4 will also show how completeness and determinism properties can be specified.

The specification validation should be integrated with the specification development environment and performed automatically. The results should then be reported to the developer in a format that does not rely on formal methods experience to interpret. This enables the validation to become an integral part of the specification development process, improving the quality of the specifications as they are passed on to the next development stage.

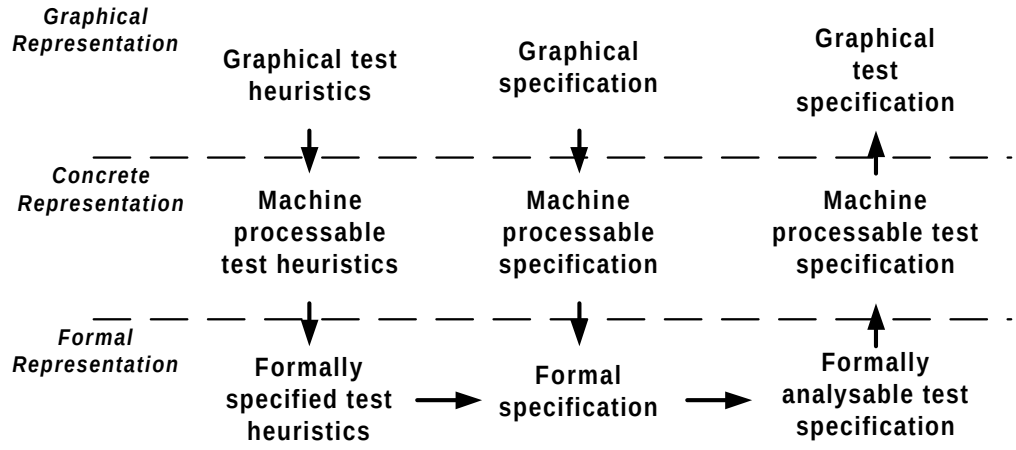


Figure 3.1: Generating test suites from graphical notations

3.4.3 Test case generation

Automated support for testing has the potential to introduce significant cost savings as well as introducing a greater level of rigour and accuracy into the testing process. Formal methods can be used to facilitate the automation. However, to reduce the requirements on formal methods skills, the automation needs to be sufficiently effective so that the engineer only has to supply guidance to the system in the form of a specification of the testing criteria to be met. The tool set will then generate test cases and data based on the criteria and formal specification. By formalising the testing criteria, comparisons can be made between them and optimal criteria can be developed for particular applications. Test cases can be automatically generated based on a formal specification of the behaviour under test and a formal definition of generic testing heuristics. Chapter 5 shows how this approach can be applied to testing individual operations in the specification. Chapter 6 extends the techniques to the problem of testing sequences of operations in order that assumptions on the testing environment can be relaxed. The process by which the tests will be generated is summarised in Figure 3.1. In particular the thesis will examine the problem of generating the formal specification (middle column) and automatically applying testing heuristics to this specification (bottom row). The concrete representation in the diagram refers to the interfaces used to extract the appropriate information from the graphical modeling tools. This could be in the form of propriety interfaces or standard interchange formats such as XML [Con01].

The automated testing covered in this thesis will be *black box* [Bei90]. No details are known about the implementation of the particular portions of the system under test and all test information is gained from the specification. This style of testing can be applied to several levels of system integration from the individual code module to software/hardware integration. In each case, a specification of the behaviour under test and the interfaces to this component are needed.

3.4.4 Evaluating the effectiveness of the framework

The ability to automatically apply specification validation and test case generation allows not only for rapid feedback of any errors found in the product, but also provides information for continual process improvement efforts. For example, Chapter 5 describes how test sets can be automatically generated based on a formal specification of the test criteria. These test sets can be automatically applied to the implementation and many mutations (deliberately incorrect versions of the implementation) [DLS78] in order to assess the tests' effectiveness at detecting particular faults. This information can be fed back into the process for deciding which test criteria to use. Mutation analysis has the advantage of being an automated approach to assessing a test set's ability to detect certain classes of error but whether the mutations are representative of errors that actually occur in practice is debatable. However, it is typically assumed that if the test data is able to kill a significant portion of the mutants, enough coverage of the software will be achieved to also detect a high proportion of the actual software errors. The technique suffers from the phenomenon of equivalent mutants, mutations of the program that do not lead to erroneous behaviour. The identification and removal of equivalent mutants must normally be performed manually which is a time consuming and difficult process.

A high level of automation in the test generation – test execution – test evaluation cycle would allow the most effective test criteria to be quickly selected for any particular application. A formal approach to comparing different test criteria is presented in Chapter 7.

3.5 Summary

This chapter presented an overview of the automated V&V framework that will be used to examine the hypothesis, discussed its role within a software engineering process and

presented a number of criteria for its success. The framework depends on the ability to automate the proof and test generation effort based on the restricted set of mathematical constraints that result from the choice of domain and translation into a formal representation. The following chapters show the feasibility of this approach by demonstrating the individual component technologies of the framework.

Chapter 4

Formalising graphical specifications

This chapter demonstrates how a formal representation of graphical specifications can be automatically generated for the purposes of specification validation and test case generation. Two notations are introduced that will be used throughout the rest of the thesis to demonstrate the techniques: Statecharts and PFS tables. In both cases, the semantics of the notations are informally described followed by a template for an explicit and formal specification of the behaviour described in the graphical notations. An example application of the formalisation is described by showing how properties of the specification can be formally and automatically verified.

4.1 Introduction

The automated high integrity V&V framework described in Chapter 3 proposed the use of intuitive graphical notations to bridge the "semantic gap" between the engineers and the formal specifications required in order to perform automated formal analysis and test case generation. By transforming various graphical notations into a common intermediate form, a re-usable tool set can be developed to perform the analysis and test case generation. This step can also be used to make explicit behaviour that may otherwise have been implicit in the source notation and therefore forms a stronger basis for automated test case generation. This chapter examines some of the issues involved in performing this transformation by showing how a popular graphical notation, Statecharts [Har87], and a locally developed tabular notation, PFS tables [GCM98], can be integrated into the framework. The chapter

demonstrates how a suitable formal representation can be found for each of these notations that makes explicit all functional behaviour for testing purposes and how common techniques can be used to validate the completeness and determinism of specifications written in both notations.

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 inspection of the diagrams alone. This poses a problem when designing automated specification validation and functional testing techniques based on the Statechart notation. An understanding of the complex Statechart semantics is required in order to interpret the behaviour implied by the diagrams and generate tests based on that behaviour.

Precisely defining the semantics of Statecharts has proven extremely challenging and a number of authors have proposed formalisations of the semantics using a variety of techniques and viewpoints [DJHP97, HN96, LvdBC99, HRdR92, US94, MLPS97]. The formalisation approach described in this chapter is less involved than the work of previous authors as the aim is to formally specify the behaviour of a particular Statechart rather than to generically specify the semantics of the Statecharts notation. To further simplify the translation, only a subset of the Statecharts notation will be used and certain assumptions will be made about the use of this subset. However, enough issues remain in the formalisation of the specifications for this chapter to serve its purpose in illustrating the difficulties involved in performing such translations and how they can be overcome.

The chapter is structured as follows. A subset of the Statechart notation is first proposed in Section 4.2 that will form the basis of the formalisation. In Section 4.3, a formalisation of the behaviour under test is described that will be used as the input to the automated test case generation activities. Section 4.4 will then demonstrate how completeness and determinism checks can be automated based on this formal definition. Section 4.5 shows how the same principles can be applied to the formalisation and validation of a second notation, PFS tables. The application of the formalisation and validation techniques to a case study is described in Section 4.6. Finally, Section 4.7 summarises the contribution of the chapter and assesses key properties of the formalisation.

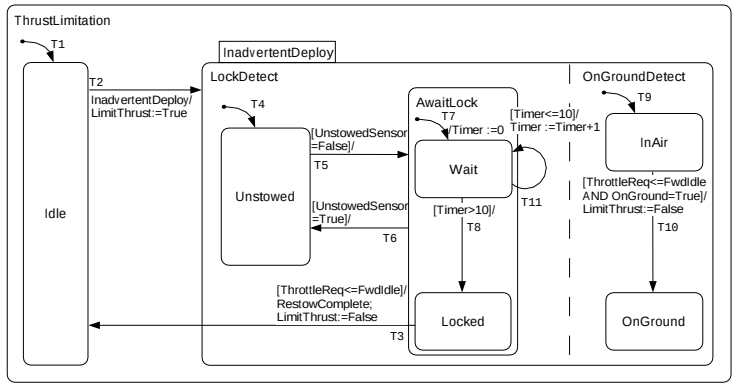


Figure 4.1: Example Statechart: Thrust Limitation

4.2 A subset of Statecharts

Rather than attempting to provide a formalism for the entire notation as defined by Harel [Har87] and implemented in tools such as STATEMATE [HLN+88], a subset of the notation is studied. This subset covers the most important aspects of Statecharts that distinguish them from traditional finite state machines: internal memory, orthogonal components, hierarchical states and typed inputs and outputs. The subset is also sufficient to model the applications of interest yet restrictive enough such that a straightforward formalisation is possible.

An example Statechart, given in figure 4.1, is typical of those specified in the target domain and describes a portion of a jet Engine Electronic Controller (EEC) specification for a commercial airline application. The Statechart describes the cancellation of a thrust limiting 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 into place (by a separately specified component) 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. This example will be used throughout this chapter to demonstrate the formalisation. The subset of the notation is now summarised in more detail.

4.2.1 Parameters and internal variables

Statecharts convert traces of input parameters into traces of output parameters. Parameters may be typed values (e.g. Booleans, enumerated types, integers or reals), or events (single valued elements which remain present in the system for one step of the execution only). The input parameters for Figure 4.1 are as follows: *InadvertentDeploy* (event), *UnstowedSensor* (Boolean), *OnGround* (Boolean) and *ThrottleReq* (integer). The output parameters are *RestowComplete* (event) and *LimitThrust* (Boolean). Statecharts can also include variables that are used for internal calculations only. These items are not observable or controllable by the environment but can be referenced and updated on the transition labels. Internal variables can be either events or typed values. Figure 4.1 includes one internal variable, *Timer*, of type integer.

4.2.2 States

Each Statechart has a state hierarchy. There is one state which is highest in the hierarchy (root state) and each state may contain a set of sub-states which are lower in the hierarchy than their surrounding super-state. States that contain no sub-states are basic states. Non-basic states can be OR-states or AND-states. When an OR-state is active, one and only one of its immediate sub-states is active. The system moves between these sub-states according to the order in which the transitions connecting them are triggered. If a transition terminates at a basic state, then that state and all of its ancestors in the hierarchy become 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 4.1. 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 active. *InadvertentDeploy* is the only AND-state in the example and its subcomponents, *LockDetect* and *OnGroundDetect* are separated by the dotted line.

The set of states in the Statechart that are active at any point in time is known as a configuration. The set of valid configurations VC of states S is restricted by the semantic definition of OR and AND-states. Example valid configurations of Figure 4.1 are $\{$*ThrustLimitation*, *Idle*$\}$ and $\{$*ThrustLimitation, InadvertentDeploy, LockDetect, OnGroundDetect, Unstowed, InAir*$\}$.

e	true if the event *e* is received
not e	true if the event *e* is not received
e_1 *and* e_2	conjunction of the expressions e_1 and e_2
e_1 *or* e_2	logical disjunction of the expressions e_1 and e_2

Table 4.1: Syntax of event expressions

true	true
false	false
di	value of Boolean typed data item *di*
not e	negation of the Boolean typed expression *e*
$e_1 = e_2$	equality of two equally typed expressions
e_1 != e_2	inequality of two equally typed expressions
e_1 *and* e_2	logical AND of two Boolean typed expressions
e_1 *or* e_2	logical OR of two Boolean typed expressions
$e_1 > e_2$	greater than comparison of two numerically typed expressions
$e_1 < e_2$	less than comparison of two numerically typed expressions
$e_1 >= e_2$	greater than or equal to comparison of two numerically typed expressions
$e_1 <= e_2$	less than or equal to comparison of two numerically typed expressions
$-e$	negation of the numerically typed expression *e*
$e_1 + e_2$	addition of two numerically typed expressions
$e_1 - e_2$	subtraction of one numerically typed expression from another
$e_1 * e_2$	multiplication of two numerically typed expressions
e_1/e_2	division of one numerically typed expression by another

Table 4.2: Syntax of guard expressions

4.2.3 Transitions

Control is passed between states via transitions. Each transition is labelled with an event expression, a guarding condition and an action, each of which is optional. Event expressions consist of predicates defining the presence or absence of events in the current step. Event expressions can be combined using standard Boolean logic operators. The set of event operators is shown in Table 4.1. Parentheses can also be used to override the precedence and associativity rules as defined in the STATEMATE semantics. Guarding conditions are formed by logical and relational expressions on input parameters and internal variables. The condition operators supported by the subset are shown in Table 4.2. Parentheses can again

be used to override precedence and associativity between sub-expressions. The evaluation of the guard expression must return a Boolean value. For the purposes of this subset, data items may take Boolean, integer, floating point or enumeration types. Data items and events may be input/output parameters or internal variables.

Actions consist of a combination of event broadcast expressions and assignments to data items. The subset of action operators is shown in Figure 4.3. Parentheses can be used to resolve any precedence and associativity issues. The syntax used for forming transition labels is as follows:

EventExpression [GuardCondition]/Actions

The semantics of a label can be described as follows:

$$EventExpr(I, V) \wedge GuardCond(I, V) \Rightarrow Action(I, V, V', O)$$

where $I, V, V'$and O represent possible combinations of the individual input parameters, internal variables, updated internal variables and output parameters respectively.[1] *EventExpr* returns *true* if the currently active inputs or internal events satisfy the event expressions, *GuardCond* returns *true* if the present values of the input parameters and internal variables satisfy the guard condition and *Action* defines the relation between the input parameters, internal variables, updated internal variables and the output parameters as defined in the transition actions. If any part of the label is missing, its corresponding predicate takes the value *true*. For example, if no event expression or guard is specified for a particular transition, the action is defined in each step in which its source state is active.

If the pre-condition (conjunction of the event and guard conditions) of more than one transition from any given OR-state is satisfied 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. For example in Figure 4.1, transitions *T3* and *T6* may become simultaneously enabled. The scope of *T3* (*ThrustLimitation*) is higher in the state hierarchy than the scope of *T6* (*LockDetect*) and therefore *T3* takes priority.

[1]For example, I is defined as $(I_1 \times I_2 \times \ldots \times I_n)$, where $I_1, \ldots I_n$ represent the types of each of the input parameters.

ev	broadcast event *ev*
$e_1;\ e_2$	execute action e_1 and e_2 (simultaneously)
$e_1 + e_2$	addition of two numerically typed expressions
$e_1 - e_2$	subtraction of one numerically typed expression from another
$e_1 * e_2$	multiplication of two numerically typed expressions
e_1/e_2	division of one numerically typed expression by another
$v := e$	assign variable *v* the value of equally typed expression *e*

Table 4.3: Syntax of action expressions

4.2.4 Statechart computations

The computation that is performed by the Statechart is represented by a series of *statuses* and a trace of input and output parameters. A status consists of a set of currently active states (i.e. a configuration), the set of internal events generated in the last step and the values of all internal variables.

The Statechart semantics used here (relating to those used in the STATEMATE tool from i-Logix [HLN+88, HN96]) contains two alternative models of time, each of which has an effect on the set of transitions that are triggered during each step. The synchronous time model was chosen for the subset and assumes that the system executes a single step every time unit, reacting to changes made in the status or to the parameters since the last step. The changes to the status caused by the transitions enabled by the current inputs are processed in the next step along with the values of any input parameters that may have been altered during the execution of the step. The synchronous time model was chosen because it is best suited to the specification of individual software components which are most often implemented using some synchronous triggering mechanism such as a cyclic executive scheduler. This thesis focuses on the testing of such software components. The asynchronous time model is better suited to specifying the behaviour of a system at a higher level of abstraction, e.g. the way in which different components in a distributed system interact. Future work should evaluate the application of the techniques described here to the asynchronous model in order that they might be used to test system behaviour which is expressed in this manner.

4.3 Modelling the behaviour to be tested

4.3.1 Making the behaviour under test explicit

The semantics of Statechart transitions is rather involved and can impose 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 notation on which the test generation techniques are based. Generic tools can then be used that do not require an understanding of the semantics of the notation from which the formal specification was generated. For the subset of Statecharts under consideration, this implicit behaviour occurs in the following four situations:

- **Transitions terminating at non-basic states.** 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.

- **Completeness assumption.** The 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.

- **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.

Specifying transitions terminating at non-basic states

Although in the graphical notation of Statecharts, transitions may originate from and terminate at non-basic states, the execution of a *step* always begins and ends in a set of basic states. This behaviour can be explicitly defined by placing restrictions on the source and target configurations of a transition.

The semantics of Statecharts describes the behaviour of transitions terminating at non-basic states as compound transitions, combining the original transition (terminating at the state boundary) with the default transition of the target state. The actions of the component transitions are performed simultaneously. It may be necessary to combine several default transitions depending on whether the default transition terminates at a non-basic state itself and whether an AND-state appears somewhere in the combination chain.

The target configuration of the transition is explicitly defined by including the original target state (and all its ancestors) and all the target states of the default transitions forming the compound transition. 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 no longer need to be considered.

Specifying transitions originating from non-basic states

Transitions originating from non-basic states 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 configuration set of each newly created transition). The original transition from the non-basic state then no longer needs to be considered. *T6* of Figure 4.1 is one such transition in the example that needs to be partitioned in this way. The transition is partitioned into transitions *T6.1* and *T6.2* which originate from basic states *Wait* and *Locked* respectively.

Based on this analysis, the configuration restrictions for the transitions in Figure 4.1 are given in Table 4.4. The basic states are given in bold text. The source and target configurations are fully specified with the exception of orthogonal state behaviour. The values of orthogonal states are not included at this stage. If there is the potential for interaction between parallel components as a result of the priority resolution mechanism, then the guard of the transition may be extended with a predicate over the source restriction sets of any conflicting transitions as explained below.

Specifying the completeness assumption

The paragraphs above described how to specify all possible means of exiting a basic state, but not what happens if none of the transitions is enabled. This is implied in the Statechart

Transition	**Source Configuration**	**Target Configuration**
T2	ThrustLimitation, **Idle**	ThrustLimitation, InadvertentDeploy, LockDetect, OnGroundDetect, **Unstowed**, **InAir**
T3	ThrustLimitation, InadvertentDeploy, LockDetect, OnGroundDetect, AwaitLock, **Locked**	ThrustLimitation, **Idle**
T5	ThrustLimitation, InadvertentDeploy, LockDetect, OnGroundDetect, **Unstowed**	ThrustLimitation, InadvertentDeploy, LockDetect, OnGroundDetect, AwaitLock, **Wait**
T6.1	ThrustLimitation, InadvertentDeploy, LockDetect, OnGroundDetect, AwaitLock, **Wait**	ThrustLimitation, InadvertentDeploy, LockDetect, OnGroundDetect, **Unstowed**
T6.2	ThrustLimitation, InadvertentDeploy, LockDetect, OnGroundDetect, AwaitLock, **Locked**	ThrustLimitation, InadvertentDeploy, LockDetect, OnGroundDetect, **Unstowed**
T8	ThrustLimitation, InadvertentDeploy, LockDetect, OnGroundDetect, AwaitLock, **Wait**	ThrustLimitation, InadvertentDeploy, LockDetect, OnGroundDetect, AwaitLock, **Locked**
T10	ThrustLimitation, InadvertentDeploy, LockDetect, OnGroundDetect, **InAir**	ThrustLimitation, InadvertentDeploy, LockDetect, OnGroundDetect, **OnGround**
T11	ThrustLimitation, InadvertentDeploy, LockDetect, OnGroundDetect, AwaitLock, **Wait**	ThrustLimitation, InadvertentDeploy, LockDetect, OnGroundDetect, AwaitLock, **Wait**

Table 4.4: Source and target configurations for ThrustLimitation

Transition	**Guard calculation**
CompleteIdle	$\neg$*Guard*$(T2)$
CompleteUnstowed	$\neg$*Guard*$(T5)$
CompleteLocked	$\neg$(*Guard*$(T3) \vee$ *Guard*$(T6.2)$)
CompleteInAir	$\neg$*Guard*$(T10)$
CompleteOnGround	*True*

Table 4.5: Completing transitions for ThrustLimitation

semantics by the *completeness assumption*. The completeness assumption can be modelled by adding a reflexive transition to each basic state. The guard of the new transition is the negation of the disjunction of the guards of all transitions originating at the state. The transition has no action.

The target configuration of the completing transition is the same as its source configuration. 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, so if all sub-states are fully specified, then the disjunction of all its sub-states must by construction also be fully specified. Furthermore, completing transitions only need to be added to states whose behaviour is not already fully defined. For a method of determining state completeness see Section 4.3.2.

The guards of the completing transitions for the running example are given in Table 4.5. Completing transitions must be added 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* only when transition *T3* (from the parallel component) is not enabled. The method for specifying this behaviour is described next.

Specifying priority resolution

Where a higher priority transition can pre-empt another, the guard on the lower priority transition must be strengthened with the negation of the guard of the higher priority one. Additionally, if the higher priority transition is from a parallel component the negated guard should include that transition's source configuration. This prevents the mechanism from being over-restrictive in cases where a conflict is not possible unless the parallel component is in a certain state configuration.

Transition	**Guard calculation**
T2	No change
T3	No change
T5	No change
T6.1	No change
T6.2	$\mathit{Guard}(T6.2) \wedge$ $\neg \mathit{Guard}(T3)$
T8	$\mathit{Guard}(T8) \wedge$ $\neg \mathit{Guard}(T6.1)$
T10	$\mathit{Guard}(T10) \wedge$ $\neg(\mathit{Guard}(T3) \wedge \mathit{SourceConfiguration}(T3))$
T11	$\mathit{Guard}(T11)$ $\wedge \neg \mathit{Guard}(T6.1)$
CompleteIdle	No change
CompleteUnstowed	No change
CompleteLocked	No change
CompleteInAir	$\mathit{Guard}(\mathit{CompleteInAir}) \wedge$ $\neg(\mathit{Guard}(T3) \wedge \mathit{SourceConfiguration}(T3))$
CompleteOnGround	$\mathit{Guard}(\mathit{CompleteOnGround}) \wedge$ $\neg(\mathit{Guard}(T3) \wedge \mathit{SourceConfiguration}(T3))$

Table 4.6: Priority resolution calculations for the ThrustLimitation

The strengthening of the guards in this way ensures that the enabling conditions of the transitions are always unambiguously and explicitly specified. Conflicting transitions can be identified by examining the source configurations and guards. The guard on the lower priority transition can then be updated accordingly. The final changes to the guards that must be made to specify the priority mechanism are summarised in Table 4.6.

To summarise, in order to extract the relevant information for the formalisation, the following steps need to be followed:

1. **Ensure all transitions terminate at a basic state.** For all transitions that do not terminate at a basic state, compose the transition with the default transition of its target state. If the default transition does not terminate at a basic state, continue the composition until a basic state is reached. If any transition in the composition

chain terminates at an AND-state, then the default transitions of all the AND-state's children must be included.

2. **Ensure all transitions originate from basic states.** For all transitions that do not originate at a basic state, partition the transition so that a new transition is created that originates from each basic state within the original source state.

3. **Check for completeness and add completing transitions where necessary.** For each basic state, check its completeness, i.e. whether a transition out of the state has been defined for each possible combination of inputs. If the state is incomplete, add a completing reflexive transition whose guard is the negation of the disjunction of all other transitions out of the state and whose action is null.

4. **Explicitly specify priority resolution conditions.** If any transition can be pre-empted by a transition of higher priority, update the guard of the lower priority transition with the negation of the guard of the higher priority one. If the higher priority transition originates from a state orthogonal to the source of the lower priority transition, then the source configuration of the higher priority transition must also be included in the negated condition that it is added to the guard.

The result of this process is a set of transitions that originate and terminate at basic states, completely cover the input space for each state and whose triggering conditions are unambiguously defined (assuming the Statechart is deterministic). These transitions can then be translated directly into the chosen formalism.

4.3.2 Formalisation of the Statecharts subset

Now that the diagrammatic ambiguities have been resolved, enough information is known about the specification to be able to construct a meaningful test suite. In order to be able to re-use test generation technologies across notations, the specification is translated into a common intermediate formal notation. Z [Int99] has been found to be a convenient language for the purpose of describing the functional transition behaviour of a Statechart as well as a good basis for automated testing. Theorem proving tools exist that can be exploited to generate the tests and verify properties of the Statecharts. Z has been used by other authors to complement Statechart specifications [BGK98, GHD98] and to specify the Statechart

semantics [MLPS97, Val98]. The work of Büssow, Grieskamp et al. [BGK98, GHD98] integrated Z into a FSM-based notation allowing the system specifier to use both notations to describe the system. The work presented here differs by allowing the specifier to describe the system using Statecharts alone by providing a description of the same behaviour in Z for the purposes of test case generation and specification validation. In translating the diagrams into the formal specifications several trade-offs must be made:

- The primary purpose of the formal specification is to form the basis for automated V&V activities. Therefore the formal specification should be in a form that can be efficiently manipulated. This will typically involve using features of the formal notation that are most amenable to automated formal analysis but which may lead to a slightly more verbose specification than could otherwise be specified using alternative language constructs.

- To demonstrate the integrity of the V&V process it may be necessary to review the intermediate formal specification for conformance with the original requirements. Tools used to perform the translation may not be of demonstrable high integrity and some review of their output may be required. The formal representation must be in a form that can be easily interpreted by a sufficiently trained engineer. To aid review, the formal specification should be annotated with traceability tags and comments derived from the original specifications. Some trade-off may be required between ease of review and amenability to automated analysis.

Formalising transitions

The updated Statechart transitions are modelled as Z operation schemas. Each operation is defined in terms of a pre-condition and a post-condition. The pre-condition is a constraint over the source configuration, events and data values that allow the transition to be triggered. The post-condition is a constraint over the target configuration, generated events and updated data values that form the actions of the transition. The transition functions of the Statechart are therefore modelled as the set of operation schemas defining the transitions in the Statechart.

The template for transition operation schemas is as follows.

StepTemplate

$\Delta Status$
Parameters

$SourceConfiguration \subseteq ActiveStates$
EventExpression
GuardingCondition
ActionExpression
$TargetConfiguration \subseteq ActiveStates'$

Status is a schema used to store the currently active states (state configuration), internal events generated in the last step and the values of internal variables. *Parameters* is a schema defining the input and output parameters (as events or data-items). *EventExpression*, *GuardingCondition* and *ActionExpression* are derived from the transition label elements defined in Section 4.2 where the guard has been updated appropriately to resolve transition priorities. $\Delta Status$ is used to specify that the contents of the status are updated by the operation and the " $'$ " decoration is used to refer to the updated values of the status.

The *Status* schema contains all the information relating to the global state of the system. This includes: the set of current states, internal events and the values of local data-items. The *Status* schema for the example Statechart is as follows:

Status

$ActiveStates : \mathbb{F}_1\ States$
$Timer : \mathbb{Z}$

$configuration(ActiveStates)$

where *configuration* is an optional state invariant that can be used to verify or enforce the static semantics of valid configurations. The full definition of *configuration* for the example Statechart is given in Appendix A.1.

The *Parameters* schema defines the input and output parameters of the system. The *Parameters* schema for the example Statechart is as follows:

Parameters

$InadvertentDeploy? : Event$
$UnstowedSensor? : Boolean$
$OnGround? : Boolean$
$ThrottleReq? : \mathbb{Z}$
$RestowComplete! : Event$
$LimitThrust! : Boolean$

UnstowedSensor, *OnGround*, *ThrottleReq* and *InadvertentDeploy* are defined as input parameters (hence the *?* decoration). Likewise *LimitThrust* and *RestowComplete* are defined as output parameters (hence the *!* decoration).

The translation from Statechart specification to Z was automated as a command line tool (StateZ) that implements the same process of behaviour elicitation as described above. The tool is based on the Statemate [HLN+88] Application Programming Interface (API). Details of the Statechart are read and the semantic issues resolved before exporting the resulting transitions as a Z specification. Any semantic resolution steps performed are described in automatically generated comments in the specification that also serve to provide traceability information with respect to the original Statechart diagram. This information was found to be extremely useful when reviewing the results of the translation and relating tests generated from the Z representation to behaviour specified in the original Statechart.

This section is completed by giving some example transition specifications to illustrate the main points discussed so far. The complete specification of the transition structure of Figure 4.1 is given in Appendix A.1. The transition *T3* is specified as follows:

$$
\begin{array}{|l}
\hline
T3 \\
\Delta Status \\
Parameters \\
\hline
\{ThrustLimitation, InadvertentDeploy, LockDetect, \\
\quad OnGroundDetect, AwaitLock, Locked\} \subseteq ActiveStates \\
ThrottleReq? \leq FwdIdle \\
RestowComplete! = Present \\
LimitThrust! = False \\
\{ThrustLimitation, Idle\} \subseteq ActiveStates' \\
\hline
\end{array}
$$

FwdIdle is a constant defined in the auxiliary definitions part of the specification (see Appendix A.1). The transition cannot be pre-empted by any others and therefore the guard condition is unchanged from that given on the diagram. Parts of *Status* that are undefined (e.g. *OnGroundDetect*'s active state) are not relevant to the operation of *T3*. The use of the $\subseteq$ operator to constrain the active states therefore allows the parallel component to be in any configuration that conforms to the invariant of *Status*, which defines the set of all valid configurations. This allows operation schemas to be conjoined so that concurrent behaviour can be modelled. For example $T8 \wedge T10$ specifies the behaviour of the system

when transitions $T8$ and $T10$ are both simultaneously enabled.

The transition *T6.2* (one of a pair of transitions generated from transition *T6* which originates at a non-basic state) is specified as follows:

T6.2

$\Delta Status$
Parameters

$\{ThrustLimitation, InadvertentDeploy, LockDetect,$
$\quad OnGroundDetect, AwaitLock, Locked\} \subseteq ActiveStates$
$UnstowedSensor? = True \land$
$\neg ThrottleReq? \leq FwdIdle$
$RestowComplete! = Undefined$
$LimitThrust! = Undefined$
$\{ThrustLimitation, InadvertentDeploy, LockDetect,$
$\quad OnGroundDetect, Unstowed\} \subseteq ActiveStates'$

The pre-condition of the operation contains the negated guard of the pre-condition for transition *T3*, a higher priority transition that can pre-empt *T6.2*. The output parameters are assigned the special value *Undefined* as otherwise loose specification could imply that they may take any value. If orthogonal components in the specification reference the same outputs then a mechanism must be provided to override the *Undefined* value if necessary. Such mechanisms are discussed in more detail in Chapter 6 when they are required to resolve potential multiple write conflicts when generating test sequences from orthogonal components.

The completing transition for the state *InAir* is given below. $\Xi Status$ indicates that the values of *Status* are unchanged by the operation. Again, this transition can be pre-empted by *T3* and therefore contains *T3*'s negated pre-condition (including its source restriction).

CompleteInAir

$\Xi Status$
Parameters

$\{ThrustLimitation, InadvertentDeploy, LockDetect$
$\quad OnGroundDetect, InAir\} \subseteq ActiveStates$
$\neg((ThrottleReq? \leq FwdIdle \land OnGround? = True) \land$
$\quad \{ThrustLimitation, InadvertentDeploy, LockDetect,$
$\quad AwaitLock, Locked, OnGroundDetect\} \subseteq ActiveStates \land$
$\quad ThrottleReq? \leq FwdIdle)$
$RestowComplete! = Undefined$
$LimitThrust! = Undefined$

4.4 Checking state completeness and determinism

Given the formal specification of the Statechart, it is possible to ascertain whether particular properties are satisfied by the specification. In doing so, faults in the specification can be eliminated before they can propagate into the implementation. As an example, Heimdahl and Leveson [HL96] 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 in any given state is always defined for each set of inputs). In the formalism used here, the behaviour of a state can be defined as the disjunction of the transitions leaving it, i.e.

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

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

$$\vdash ? \forall Status, Parameters \mid BasicState \in ActiveStates \bullet \\ preT_1 \lor preT_2 \lor .. \lor preT_n$$

where $BasicState \in ActiveStates$ constrains the analysis to a particular state in the Statechart. A proof of this conjecture could be used to determine whether or not completing transitions are required for a particular state. If a counter-example to the conjecture can be found, the designer of the system can be alerted in order to determine whether the behaviour was deliberately left unspecified (in which case completing transitions can be added) or whether the incompleteness represents a design error.

A similar conjecture can be defined to demonstrate determinism (disjointness of the transition pre-conditions). Although it is not always desirable that specifications are deterministic (for example, when describing high level requirements), determinism affects the way that test results are evaluated and therefore it is at least desirable to know where non-determinism exists in the specification. The non-determinism conjecture is defined as follows:

$$\vdash ? \forall Status, Parameters \mid BasicState \in ActiveStates \bullet \\ \forall i, j : 1..n \mid i \neq j \bullet \neg(preT_i \land preT_j)$$

The StateZ Statechart to Z translation tool was extended to generate the above proof obligations for each state. A generic proof tactic was then constructed for the theorem prover CADiℤ that attempts to solve the conjectures and generate counter-examples where appropriate. The proof tactic simplified the conjecture and applied constraint solving techniques to search for a counter-example to the proof. Where the input space was finite, model checking was also applied to prove the property. If model checking was not applicable, the tool could not prove the property with any certainty. However, confidence could be gained in the correctness of the property and therefore time could be invested in the proof with more assurance that the property could be proven. The Statecharts used in the case studies had typically finite input spaces (consisting mainly of conditions derived from validated sensor inputs etc.) and therefore this weakness in the technique did not prove to be a problem.

The proof tactic was invoked via an Application Programming Interface (API) to the theorem prover. A command line tool (Healthy) was written to allow domain engineers to apply the automated analysis without needing to manipulate the Z directly. The counter examples were then presented to the engineer in an equally intuitive (non Z) manner. The use of this tool on a medium scale study is described in [BCGM00] and Section 4.6. The proof conjectures for the example Statechart from Figure 4.1 are given in Appendix A.1.4.

By not taking the sequential aspects of the Statechart into account, the proofs can lead to false negatives. A state may be non-deterministic for a given set of values for the status and parameters, but the non-determinism may correspond only to unreachable combinations of the states and internal variables. The conjectures could be updated to take this into account, but the analysis required to prove the conjectures would become more involved and would reduce the potential for automation. Alternatively, model checking could be used to perform a reachability analysis on the resulting counter-example. Model checking as a form of reachability analysis is discussed in more detail in Chapter 6.

4.5 Formalisation of tabular requirements

A tabular notation is used to define non state-based components of the control systems used as case studies in this thesis. This particular variant of the notation was derived from the Practical Formal Specification (PFS) project [GCM98, MGB+98], which produced a suite of notations and methods designed for the specification and validation of high integrity

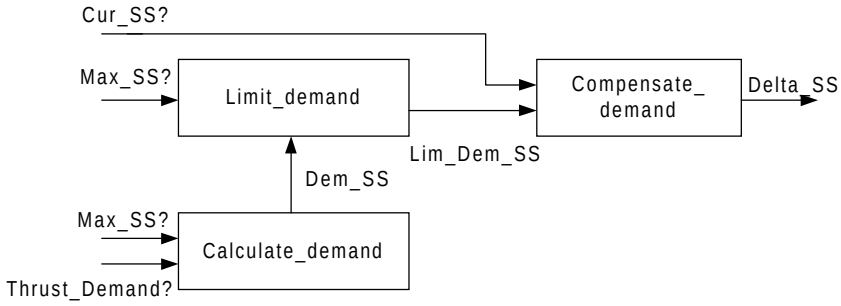


Figure 4.2: Adjust Thrust

aerospace control applications.

The schematic in Figure 4.2 describes a system specified in the reactive notation. The example specifies a portion of a flight control system for a jet engine propelled aircraft that adjusts thrust levels based on the current demand. As thrust is difficult to measure directly, the turbine spool speed is used to measure the current thrust and the desired change in spool speed is calculated as the required change in thrust.

The component has three inputs; the current spool speed *Cur_SS*?, the maximum allowed spool speed *Max_SS*? and the current thrust demand *Thrust_Demand*?. The component has one output, the required change to the current spool speed *Delta_SS*!. The component consists of three subcomponents, specified using the PFS tabular notation in Figures 4.3, 4.4 and 4.5 respectively. *Calculate_demand* calculates the demanded spool speed based on the demanded thrust. *Limit_demand* ensures that the demanded spool speed does not exceed the maximum allowed. *Compensate_demand* calculates the required change to the spool speed to realise the demanded thrust. The three components interact via the shared internal variables *Dem_SS* and *Lim_Dem_SS*. All capitalised identifiers are suitably valued constants defined elsewhere in the specification.

The parameters to the system are modelled as real numbers in the specification. However, fixed-point representation would typically be used in the implementation. The impact that this change of representation has on the system behaviour is an attribute that must be checked during testing. Information regarding the implementation types may also be used when designing testing heuristics. For example, a boundary value analysis (see Chapter 5)

could use the resolution of the fixed-point data type to define the smallest possible distance from a particular boundary.

Function:	Dem_SS
Assumption:	Max_SS? $<$ ABS_MAX_SS **AND**
	IDLE_THRUST_DEMAND $<$ Thrust_Demand? **AND**
	Thrust_Demand? $<$ MAX_THRUST_DEMAND **AND**
	Thrust_Demand? $\neq 0$
Guard 1:	True
Definition 1:	$((\text{Max\_SS?} * 0.05 + 30)/(100 * \text{Thrust\_Demand?})) + 30$

Figure 4.3: Calculate_demand

Function:	Lim_Dem_SS
Assumption:	None
Guard 1:	Dem_SS $\geq$ Max_SS?
Definition 1:	Max_SS?
Guard 2:	Dem_SS $<$ Max_SS?
Definition 2:	Dem_SS

Figure 4.4: Limit_demand

Function:	Delta_SS!
Assumption:	Cur_SS? $\geq$ ABS_MIN_SS **AND** Cur_SS? $\leq$ ABS_MAX_SS
Guard 1:	True
Definition 1:	Lim_Dem_SS $-$ Cur_SS?

Figure 4.5: Compensate_demand

Assumption fields are used to restrict the range of values over which the component is specified. This reduces the number of conditions that need to be considered in the specification but requires that the assumptions are discharged at some point with respect to the context in which the component is used. A typical use of the assumptions would be to define the safe usage conditions for a component. Only values within this range would then need to be considered. Some other logic would be required to validate that the input values remain within this range whenever the component is referenced by another component. The output value of the component is defined by one or more guard/definition pairs. The guards must be disjoint and complete and the definitions may consist of single values or arithmetic expressions referencing other components or variables within the system. The proof tactic used to check the determinism and completeness of Statechart states is also used to check

these properties. The defined value of the reactive components can be used as the inputs to other components. The semantics of these combined components is based on function composition. The template for the specification of the tables in Z is as follows:

PFSTable

$$
\begin{array}{l}
Inputs \\
DefinedValue \\
\hline
Assumption \Rightarrow \\
\quad ((Guard_1 \land Definition_1) \lor \\
\quad \ldots \lor \\
\quad (Guard_n \land Definition_n))
\end{array}
$$

If the assumption can be guaranteed within all contexts where the table is used, the implication can be replaced by conjunction. This aids analysis and reduces the number of test cases that must be derived for each table. The assumptions in the specification are propagated out towards the system interface at which point engineering intuition or other forms of evidence must be used to demonstrate that the assumptions are reasonable. In some cases, implementation decisions may be made to ensure that the assumptions are met (e.g. use of particular sensing devices that have guaranteed operating ranges). In later chapters, it is assumed that the assumptions have been proven before the tests are generated and therefore the following template is used.

PFSTable

$$
\begin{array}{l}
Inputs \\
DefinedValue \\
\hline
Assumption \land \\
\quad ((Guard_1 \land Definition_1) \lor \\
\quad \ldots \lor \\
\quad (Guard_n \land Definition_n))
\end{array}
$$

The Z specifications for each sub-component are given in Figures 4.6, 4.7 and 4.8. It is assumed that the assumptions are guaranteed within the context of use and therefore can be simply conjoined with the guard/definition predicates to restrict the valid input space of the specification. The complete *Adjust Thrust* component can be specified as the conjunction of its sub-components:

$$Adjust\_thrust == Calculate\_demand \land Limit\_demand \land Compensate\_demand$$

Calculate_demand

$$
\begin{array}{l}
Max\_SS? : \mathbb{R} \\
Thrust\_Demand? : \mathbb{R} \\
Dem\_SS : \mathbb{R} \\
\hline
Max\_SS? < ABS\_MAX\_SS \land \\
IDLE\_THRUST\_DEMAND < Thrust\_Demand? \land \\
Thrust\_Demand? < MAX\_THRUST\_DEMAND \land \\
Thrust\_Demand? \neq 0 \land \\
\quad Dem\_SS = \\
\quad (Max\_SS? * 0.05 + 30) \div (100 * Thrust\_Demand?) + 30
\end{array}
$$

Figure 4.6: Z operation for Calculate_demand

Limit_demand

$$
\begin{array}{l}
Max\_SS? : \mathbb{R} \\
Dem\_SS : \mathbb{R} \\
Lim\_Dem\_SS : \mathbb{R} \\
\hline
Dem\_SS \geq Max\_SS? \land Lim\_Dem\_SS = Max\_SS? \lor \\
Dem\_SS < Max\_SS? \land Lim\_Dem\_SS = Dem\_SS
\end{array}
$$

Figure 4.7: Z operation for Limit_demand

Compensate_demand

$$
\begin{array}{l}
Cur\_SS? : \mathbb{R} \\
Lim\_Dem\_SS : \mathbb{R} \\
Delta\_SS! : \mathbb{R} \\
\hline
Cur\_SS \geq ABS\_MIN\_SS \land Cur\_SS? \leq ABS\_MAX\_SS \land \\
\quad Delta\_SS! = Lim\_Dem\_SS - Cur\_SS?
\end{array}
$$

Figure 4.8: Z operation for Compensate_demand

4.6 Case studies

The techniques described in this thesis were applied to a large case study based on a commercial aerospace jet engine application. The formalisation, validation and testing work was performed within the context of a larger project investigating improved software engineering processes for the aerospace industry [ABB+01]. In particular, the project investigated model-based software development using graphical specifications and high level (i.e. requirements driven) re-use. The project provided a good context within which to evaluate the techniques and assess their impact on other software engineering activities.

A summary of the specification formalisation and validation performed for the thrust reverser case study is shown in Table 4.7. The numbers include only automatically generated proof obligations and the requirements errors found by discharging the proofs. The generation of the formal representation of the Statechart specification was achieved using the StateZ tool described previously. A tool was also developed to generate formal representations and the associated proof obligations from PFS reactive components. This tool was based on an XML [Con01] representation of the tables that could be automatically generated from a number of other sources including Microsoft Word documents. The proof obligations were discharged using the Healthy tool described in Section 4.4. StateZ, Healthy and the relevant extensions to the CADiℤ theorem prover were implemented by the author. The PFS translation tools were implemented by intern students working under the supervision of the author. The original specification document consisted of over 100 pages of graphical and tabular specifications with supporting informal text. The formalisation process resulted in approximately 400 pages of Z specification including healthiness proof obligations.

As a result of the high level of automation, the translation and validation took 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 with 128Mb of RAM running the Linux operating system, the largest of these proofs took no more than a second to discharge.

The results show that a significant number of requirements errors were detected for little additional effort. All the requirements errors were detected using the determinism and completeness checks. On analysis, the errors were found to be due to two distinct causes. The first type of error was the result of a mis-interpretation of some higher level

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

Table 4.7: Summary of results

requirements that resulted in an incorrect specification with respect to these requirements. These errors accounted for the greater proportion (around 70%) of total requirements errors found. The second type of error was found to be due to omission or ambiguity in the higher level requirements from which the graphical specification were produced. Although these errors were less frequent, their detection was deemed to be very important. This type of analysis was therefore found to be an extremely useful but efficient complement to traditional approaches such as peer review and animation. Furthermore, due to the high level of automation, the analysis could be performed by the developer of the specifications to increase their quality before passing the specifications on to more formal peer review stages of the process.

When the analysis revealed a fault, the time taken to review and rework the fault 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 faults 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 (another member of the research team) appeared to be consciously writing specifications to meet the healthiness conditions.

4.7 Summary

This chapter has demonstrated how graphical and tabular requirements can be represented as formal specifications for the purpose of specification validation and test case generation. Some graphical notations such as Statecharts leave some of the system behaviour implicit in the semantics of the diagrams. This behaviour must be made explicit in the translation to the formal specification to ensure that it is tested.

Only a subset of the Statecharts notation was covered here. Aspects such as static reactions and history connectors were not formalised. If the subset is extended (as the techniques are used on a larger number of industrial case studies), efforts will be made to formalise the behaviour using the same simple subset of the Z notation as was used in this chapter. This subset has been found to produce specifications that are amenable to automated analysis (for both theorem proving and test case design). Adhering to the Z subset when extending the notation, or adding a new notation to the framework will increase the potential for re-use of existing proof tactics and constraint solving techniques. So far, similar formalisations have been applied to the PFS variant of hierarchical state machines [GCM98] and a subset of the Stateflow notation [TM01].

Case study work has shown that, apart from the testing benefits, the automated translation to Z and validation can be used to improve and better understand both system models and the high level specifications from which they were derived. These techniques were found to have a significant impact on the quality of the work products at an early stage in development.

The formalisation presented in this chapter does not yet capture all of the sequential and parallel behaviour of the Statecharts. The formalism does capture enough information to generate tests where a high degree of controllability and observability can be assumed in the testing environment. The following chapter describes methods of automating test case design in these circumstances. However, where this level of controllability and observability is not practical, a fuller description of the sequential and parallel behaviour of the system is required. These extensions to the formalism are introduced in Chapter 6.

Chapter 5

Operation testing

Chapter 4 demonstrated how specifications written in graphical and tabular specification notations could be translated into a formal specification in Z. This chapter investigates how test cases can be automatically generated from Z specifications based on a formalisation of both partitioning and fault-based testing heuristics. This approach leads to tests that can be used to verify individual operations within the specification (e.g. Statechart transitions or reactive components). The following chapters examine how tests referencing persistent state can be sequenced to cover a wider range of testing scenarios and how the techniques can be used to investigate properties of the testing criteria and testability of the specifications.

5.1 Introduction

There is a strong need for automated test generation techniques to complement the increasing use of graphical and tabular notations for the specification of software systems. Translation of these notations into Z not only forms a strong basis for automated test design (all behaviour is made explicit and represented in a notation amenable to automated analysis) but enables generic testing tools to be used for a number of source notations. Automated testing techniques should allow a range of criteria to be applied as and when needed. The techniques should also produce test cases whose conformance to both the testing criteria and the original specification can be either guaranteed or easily demonstrated.

This chapter describes an automated and formal framework for applying partitioning and fault-based testing heuristics to Z specifications. The framework allows the tester to

reason about properties of the heuristics and can be used to ensure that certain properties of the test sets are true by construction. Fault-based heuristics can be defined in terms of the local effect of a fault on a particular expression or the effect of a fault on the complete input/output relation of the specification. Each style of definition has its own advantages and disadvantages. To distinguish these different approaches to fault-based testing, the terms weak and strong detectability are introduced.

This chapter is structured as follows. Section 5.2 introduces equivalence class testing concepts which are core to the techniques described here. Section 5.3 describes how testing heuristics can be formally specified and analysed and, within this context, introduces weak and strong conditions for detecting particular faults. Tool support that has been developed for this method of testing is described in Section 5.4 and the results of applying the method to some case studies are described in Section 5.5.

5.2 Equivalence class testing

Testing techniques for model-based notations such as Z (e.g. [AA92, DF93, SC93, Hie97a, SCES97]) are typically based on the principle of *equivalence classes* [GG75]. Equivalence classes are formed by partitioning the input domain into sets of data that, for testing purposes, exhibit similar behaviour. The *uniformity hypothesis* assumes that only one test point need be selected from each equivalence class to verify the implementation against the specification for all values from that class. In the methodology described here, equivalence classes are selected using *testing heuristics*. A partitioning heuristic partitions the valid input space of the specification into equivalence classes and can be based on the signature (variable type declarations) or predicate part of the specification. Selecting data from each equivalence class is then assumed to reveal a certain class of faults in the implementation. A fault-based heuristic identifies an equivalence class of the inputs which contains those values that reveal a particular hypothesised fault.

In the method presented here, the test cases themselves are represented as Z operations. This allows test data to be generated using conventional constraint solving techniques. Heuristics can also be applied to the test cases themselves allowing test data to be generated from a combination of heuristics. A test case is formally defined by conjoining *constraining predicates* to the predicate part of the schema operation to restrict the operation to the

subdomain considered interesting for the test.

As an example, a boundary value analysis partitioning heuristic[1] can be applied to the *Thrust_Demand*? $<$ *MAX_THRUST_DEMAND* predicate of the Z operation for the *Calculate_demand* component (Figure 4.6, page 79). This partitioning would result in the following subdomains:

- *Thrust_Demand*? $<$ *MAX_THRUST_DEMAND* $-$ *DELTA*

- *Thrust_Demand*? $<$ *MAX_THRUST_DEMAND* $\wedge$
  Thrust_Demand $\geq$ *MAX_THRUST_DEMAND* $-$ *DELTA*

where *DELTA* is a constant that defines a suitable distance from the boundary to be used for the analysis (e.g. based on the resolution of the data types in the implementation). The resulting test cases are shown below. The constraining predicate is highlighted in each case.

Partition$_1$

$$
\begin{array}{l}
\textit{Max\_SS}? : \mathbb{R} \\
\textit{Thrust\_Demand}? : \mathbb{R} \\
\textit{Dem\_SS} : \mathbb{R} \\
\hline
\textit{Max\_SS}? < \textit{ABS\_MAX\_SS} \wedge \\
\textit{IDLE\_THRUST\_DEMAND} < \textit{Thrust\_Demand}? \wedge \\
\textit{Thrust\_Demand}? < \textit{MAX\_THRUST\_DEMAND} \wedge \\
\boxed{\textit{Thrust\_Demand}? < \textit{MAX\_THRUST\_DEMAND} - \textit{DELTA}} \wedge \\
\textit{Thrust\_Demand}? \neq 0 \wedge \\
\quad \textit{Dem\_SS} = \\
\quad (\textit{Max\_SS}? * 0.05 + 30) \div (100 * \textit{Thrust\_Demand}?) + 30
\end{array}
$$

[1] Boundary value analysis consists of choosing data values on and around the boundary in order to verify that the boundary has been correctly placed and has the correct polarity.

Partition$_2$

$$
\begin{array}{l}
\mathit{Max\_SS}? : \mathbb{R} \\
\mathit{Thrust\_Demand}? : \mathbb{R} \\
\mathit{Dem\_SS} : \mathbb{R} \\
\hline
\mathit{Max\_SS}? < \mathit{ABS\_MAX\_SS} \land \\
\mathit{IDLE\_THRUST\_DEMAND} < \mathit{Thrust\_Demand}? \land \\
\mathit{Thrust\_Demand}? < \mathit{MAX\_THRUST\_DEMAND} \land \\
\boxed{\mathit{Thrust\_Demand}? < \mathit{MAX\_THRUST\_DEMAND} \land} \\
\boxed{\mathit{Thrust\_Demand}? \geq \mathit{MAX\_THRUST\_DEMAND} - \mathit{DELTA}} \land \\
\mathit{Thrust\_Demand}? \neq 0 \land \\
\quad \mathit{Dem\_SS} = \\
\quad (\mathit{Max\_SS}? * 0.05 + 30) \div (100 * \mathit{Thrust\_Demand}?) + 30
\end{array}
$$

Fault-based testing heuristics are less well documented in the research, in particular with respect to Z specifications. However, fault-based tests can be defined in a similar way to those derived using partitioning heuristics. A constraining predicate is introduced into the operation specification that restricts the valid input space to only those values that lead to the detection of a particular hypothesised fault. As an example, the following constraining predicate could be used to detect the mutant where $\mathit{Thrust\_Demand}? \neq 0$ is modified to $\mathit{Max\_SS}? \neq 0$.

DetectFault

$$
\begin{array}{l}
\mathit{Max\_SS}? : \mathbb{R} \\
\mathit{Thrust\_Demand}? : \mathbb{R} \\
\mathit{Dem\_SS} : \mathbb{R} \\
\hline
\boxed{\mathit{Max\_SS}? = 0 \land} \\
\mathit{Max\_SS}? < \mathit{ABS\_MAX\_SS} \land \\
\mathit{IDLE\_THRUST\_DEMAND} < \mathit{Thrust\_Demand}? \land \\
\mathit{Thrust\_Demand}? < \mathit{MAX\_THRUST\_DEMAND} \land \\
\mathit{Thrust\_Demand}? \neq 0 \land \\
\quad \mathit{Dem\_SS} = \\
\quad (\mathit{Max\_SS}? * 0.05 + 30) \div (100 * \mathit{Thrust\_Demand}?) + 30
\end{array}
$$

In the erroneous implementation, if *Max_SS*? takes the value 0, the variable *Dem_SS* will not be defined, leading to the error being detected. In this example, the constraining predicate $\mathit{Thrust\_Demand}? = 0$ could also have been used to detect the fault.

5.3 Formalising test generation heuristics

Automatically applying testing heuristics to the specification requires a method of describing the heuristics. In a high integrity process it should also be possible to demonstrate that the resulting test cases accurately represent the testing heuristic and are sound (reflect the behaviour of the specification). It may also be desirable to compare the various fault-finding abilities of different heuristics (e.g. partitioning vs. fault-based heuristics) in order to optimise the set of heuristics that must be applied to each specification for a particular fault coverage. Formally specifying the heuristics allows all of these issues to be addressed.

5.3.1 Partitioning heuristics

Partitioning heuristics can be represented as theorems in Z that describe the equivalence between the original specification and the resulting test cases. The template below is used for defining partitioning heuristics where the input space defined by P is partitioned into the subdomains $P_1 \ldots P_n$ and $Vars(P)$ is a function that returns declarations of all variables referenced in the predicate to be partitioned, P. Note that this template does not require the subdomains to be disjoint, only complete. A separate proof is required to demonstrate disjointness (see Section 5.2). The partitioning heuristic template is thus:

$$\vdash ? \; \forall Vars(P) \bullet P \Leftrightarrow P_1 \lor \ldots \lor P_n$$

Based on this template, the boundary value analysis partitioning heuristic that was used in section 5.2 can be formally defined as follows:

$$\begin{array}{l} \vdash ? \; \forall A, B : \mathbb{R} \bullet A < B \Leftrightarrow \\ (A < B - DELTA) \lor \\ (A < B \land A \geq B - DELTA) \end{array}$$

A number of other partitioning heuristics are defined in Table 5.1. Partitioning heuristics can be classified into three groups. The first defines how predicates whose operands are typed expressions can be partitioned. The boundary value analysis heuristic described above is a member of the first group as each of its operands is an expression of type $\mathbb{R}$. The second group defines how predicates whose operands are themselves predicates can be partitioned. Heuristics 5 and 6 are members of this group and were proposed by Dick and Faivre [DF93] as an extension of their DNF-based test selection approach. In these cases, the operands

Group 1: Typed expressions as operands to the partitioning heuristic	
1: Boundary value analysis applied to $<$	$\forall A, B : \mathbb{Z} \bullet A < B \Leftrightarrow$ $A = B - 1 \lor A < B - 1$
2: Boundary value analysis applied to $>$	$\forall A, B : \mathbb{Z} \bullet A > B \Leftrightarrow$ $A = B + 1 \lor A > B + 1$
3: Boundary value analysis applied to $\leq$	$\forall A, B : \mathbb{Z} \bullet A \leq B \Leftrightarrow$ $A = B \lor A = B - 1 \lor A < B - 1$
4: Boundary value analysis applied to $\geq$	$\forall A, B : \mathbb{Z} \bullet A \geq B \Leftrightarrow$ $A = B \lor A = B + 1 \lor A > B + 1$
Group 2: Predicates as operands to the partitioning heuristic	
5: Disjunction partitioning	$\forall A, B : \mathit{Boolean} \bullet A \lor B \Leftrightarrow$ $(A \land B) \lor (\neg A \land B) \lor (A \land \neg B)$
6: Implication partitioning	$\forall A, B : \mathit{Boolean} \bullet A \Rightarrow B \Leftrightarrow$ $\neg A \lor (A \land B)$
Group 3: Type based partitioning heuristics	
7: Integer type partitioning	$\forall X : \mathbb{Z} \bullet$ $X < 0 \lor X = 0 \lor X > 0$
8: Finite powerset type partitioning	$\forall X : \mathbb{F}\, T \bullet$ $\#X = 0 \lor \#X = 1 \lor \#X > 1$

Table 5.1: Examples of formalised partitioning heuristics

have a Boolean type and are therefore quantified over the type *Boolean*, defined to represent the values of predicates. Such a type does not exist in standard Z and therefore must be explicitly defined.

$$Boolean == \{[|\ true], [|\ false]\}$$

The third group includes heuristics based on type information of variables. Type-based partitioning was suggested by Stocks and Carrington as a heuristic to be used in the Test Template Framework. Heuristic 8 is derived from an informal definition given in their paper [SC96]. In heuristics 7 and 8, the variable X represents the variable whose type forms the basis of the partitioning. Heuristic 8 is defined generically in terms of the type T. Table 5.1 is not an exhaustive list, any number of partitioning heuristics can be defined based on the subset of the Z notation used in the specifications of interest. Some heuristics (such as the ones given in Table 5.1) are generic across different specifications and domains. It is likely, however, that for a given application domain, specific heuristics will also be defined based on experience of common programming faults. The method of automating the partitioning based on the formal specification of the heuristic (see Section 5.4) must be flexible enough to allow new heuristics to be added without changing the underlying means of automation.

To generate abstract test cases using the partitioning heuristics, the parameters of a heuristic are instantiated with the operands of the predicate to be partitioned. The resulting terms of the disjunct on the right hand side of the equivalence are then used as the constraining predicates. As an example, the expression $x < y$ would be used to instantiate the variables A and B from pattern 1 in the table with x and y respectively resulting in the expression $(x = y-1) \lor (x < y-1)$ from which the constraining predicates can be extracted. A more detailed discussion of how this process is implemented using a particular theorem prover is given in Section 5.4.

5.3.2 Proving properties of partitioning heuristics

Equivalence class testing assumes that a hypothesised fault lies in one of the partitioned subdomains and that the fault can be detected by selecting data from each subdomain (assuming these subdomains satisfy the uniformity hypothesis). For this type of testing to be successful it is important that the subdomains cover the entire input space of the predicate to avoid missing faults that may lie in an unspecified subdomain. Partitioning heuristics

such as those given in Table 5.1 should therefore result in subdomains that, in total, cover the valid input space of the predicate to be partitioned.

In most cases, the subdomains should also be disjoint, otherwise test data could be selected that test a common portion of the input space leaving other parts untested [SCES97]. By explicitly defining the partitioning heuristics, their completeness and disjointness can be formally verified. Any test cases generated using these heuristics can then be assumed to be also complete and disjoint.

A proof conjecture for the completeness of a partitioning heuristic can be formulated as follows[2]:

$$\vdash ?\ \forall\, Vars(P) \bullet P \Leftrightarrow P_1 \vee \ldots \vee P_n$$

A proof conjecture for the disjointness of the heuristic (using pairwise comparison of the partitions) can be formulated as follows:

$$\vdash ?\ \forall\, Vars(P) \bullet \forall\, i, j : 1 \,..\, n \mid i \neq j \bullet \neg(P_i \wedge P_j)$$

These checks are now demonstrated by proving the completeness and disjointness of the partitioning heuristic for implication (row 6 in Table 5.1). The proofs for these properties are trivial and are illustrated by the logic maps of Figure 5.1. Figure 5.1(a) shows the logic map for the implication operation. The subdomains are complete if the area they cover equals the area covered by the implication operation. The subdomains are disjoint if the areas of the map covered by each subdomain do not overlap. It can be seen from Figure 5.1(b) that both these conditions are met.

The equivalent proof obligations for completeness and disjointness respectively are as follows:

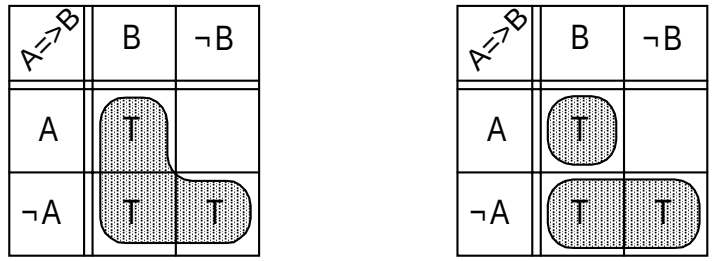
$$\vdash ?\ \forall\, A, B : Boolean \bullet A \Rightarrow B \Leftrightarrow \neg A \vee (A \wedge B)$$

$$\vdash ?\ \forall\, A, B : Boolean \bullet \neg(\neg A \wedge (A \wedge B))$$

5.3.3 Fault-based heuristics

Fault-based (or mutation) testing can be an effective means of assessing other testing strategies, such as partition testing. A test set can be evaluated against a number of hypothesised

[2]Note that this is equivalent to the template used to define the heuristic.



(a) Implication

(b) Partitions of implication

Figure 5.1: Illustration of partitioning heuristic proof

faults that are considered representative of a large proportion of the actual faults likely to appear in the implementation. The effectiveness of the target test set at detecting these faults is then used as a measure of test effectiveness or completeness. Performing this evaluation using mutations of the specification allows such an analysis to be performed much earlier in the life-cycle as it does not rely on the pre-existence of an implementation. Fault-based testing can also be an effective means of generating the test set itself, based on the intuition that if a test is adequate to detect a slight variation in the correct behaviour, that part of the system is being exercised adequately [ABM98]. If a separate test case is written to detect each mutant, due to the large number of potential mutations this approach may lead to an infeasibly large number of test cases. However, selective (program-based) mutation testing studies [ORZ93, OLR+96, MB99] suggest that a relatively small number of mutations can lead to the detection of a large class of faults. Automation may allow significant examination of similar results applied to specification-based mutants.

The condition under which a fault is detected can be formally specified and used as the constraining predicate for a test case. Fault detection conditions can take one of two forms. A *necessary condition* constrains the variables such that a mutated *expression* in the predicate part of the schema evaluates differently to the original. However, based on the evaluation of other parts of the schema predicate and the variables being observed during testing, the necessary condition may not always be sufficient to detect the fault (it may not manifest itself at the outputs or after state). A *sufficient condition* constrains the variables such that the *observed values* (i.e. outputs or after state) evaluate differently from the origi-

nal. The difference between necessary and sufficient fault detection conditions is analogous to weak and strong mutation testing as defined by Howden [How82] for program-based mutation testing.

Necessary conditions

If a fault in the implementation can reveal itself as a mutation of the expression E to E_m in the specification, then the *necessary condition* for detecting the fault can be generically described as follows (based on a definition by Kuhn [Kuh99]):

$$\vdash? \exists \mathit{Vars}(E);\ \mathit{Vars}(E_m) \bullet \neg(E = E_m)$$

Informally, there exists a set of values for the variables in E and E_m such that E and E_m evaluate differently. Selecting values from this set will ensure that the fault reveals itself in the evaluation of the sub-expression. By allowing the sets $\mathit{Vars}(E)$ and $\mathit{Vars}(E_m)$ to differ, the class of faults covered by variable replacement can also be modelled.

Suppose that in implementing the *Calculate_demand* specification from Section 4.5 the expression $100 * \mathit{Thrust\_Demand}?$ used in the calculation of *Dem_SS* is implemented as $100 + \mathit{Thrust\_Demand}?$, the necessary condition for detecting this fault is expressed as follows:

$$\vdash? \exists \mathit{Thrust\_Demand}? : \mathbb{R} \bullet \neg(100 * \mathit{Thrust\_Demand}? = 100 + \mathit{Thrust\_Demand}?)$$

Necessary detection conditions for this type of fault ('$*$' replacement by '$+$') can be generically specified as:

$$\vdash? \exists A, B : \mathbb{Z} \bullet \neg(A * B = A + B)$$

Any number of such generic fault-based heuristics can be defined, limited only by the subset of the notation used to specify the system under test. In general, the necessary conditions may not be satisfiable as the mutations may lead to equivalent behaviour (equivalent mutants). A hypothesised fault is *weakly detectable* if the formalisation of the necessary condition can be shown to be a valid theorem. That is, some values can be found that reduce the expression to true. The fault is said to be weakly detectable because there is no guarantee that other expressions in the system will not mask the fault at the outputs or prevent the input conditions for detecting the fault from occurring altogether. However, the probability of detecting the fault is strongly increased if the necessary condition is satisfied.

Despite this reduction in fault detection compared to strong detectability (described below), weak detectability can be advantageous as the conditions can be calculated in advance (in the form of generic fault-based heuristics) and many mutations may be considered simultaneously, reducing the number of test cases. The process of automating fault-based tests based on necessary conditions will therefore be more efficient than that for sufficient conditions (see below). A trade-off can be made between the cost of deriving and performing the tests and the probability of faults being masked. Automation of fault-based testing will allow for experimentation to provide better insight into these trade-offs.

The manner in which faults in the implementation reveal themselves as mutations of the specification will depend on the refinement between specification and code. An examination of this relationship may therefore point towards more testable implementations of the specifications. Fault-based heuristics can be used to generate the test cases in a similar fashion to partitioning heuristics. The parameters are instantiated with the operands of the expression containing the hypothesised fault. The resulting predicate then defines the equivalence class of the test.

Sufficient conditions

Suppose that in testing the *AdjustThrust* example (from Section 4.5), *Cur_SS*?, *Max_SS*? and *Thrust_Demand*? are the only controllable inputs and that *Delta_SS*! is the only observable output. The values of *Dem_SS* and *Lim_Dem_SS* would be hidden from the testing environment. The necessary conditions for detecting faults within each of the subcomponents may not be *sufficient* to detect the fault when the component is tested as a whole, because of fault-masking in any of the other subcomponents (or expressions within subcomponents).

In order to guarantee detection of the fault, the *sufficient condition* must be constructed. Whereas the necessary condition restricts the range of values of the variables to those that distinguish between the original and mutated expression, the sufficient condition must restrict the values of the inputs to those that can distinguish between the original and mutated relation between the controllable inputs and observable outputs. The sufficient condition is constructed using the following pattern:

$$\vdash ?\ \exists\, Inputs\_Outputs(P) \bullet \neg(P \Leftrightarrow P_m)$$

where P denotes the original predicate part of the operation and P_m denotes the predi-

cate part of the schema with the mutation applied and $\mathit{Inputs\_Outputs}(P)$ represents the controllable inputs and observable outputs of P. In order to apply this technique to the running example and the $+$ to $*$ fault described in Section 5.3.3, the original and mutated forms of *Adjust_Thrust* must first be expanded as follows. The mutated sub-expression in $\mathit{Adjust\_Thrust}_m$ is highlighted.

Adjust_Thrust

$$
\begin{array}{l}
\mathit{Cur\_SS}? : \mathbb{R} \\
\mathit{Delta\_SS}! : \mathbb{R} \\
\mathit{Dem\_SS} : \mathbb{R} \\
\mathit{Lim\_Dem\_SS} : \mathbb{R} \\
\mathit{Max\_SS}? : \mathbb{R} \\
\mathit{Thrust\_Demand}? : \mathbb{R} \\
\hline
\mathit{Max\_SS}? < \mathit{ABS\_MAX\_SS} \land \\
\mathit{IDLE\_THRUST\_DEMAND} < \mathit{Thrust\_Demand}? \land \\
\mathit{Thrust\_Demand}? < \mathit{MAX\_THRUST\_DEMAND} \land \\
\mathit{Thrust\_Demand}? \neq 0 \land \\
\quad \mathit{Dem\_SS} = \\
\quad (\mathit{Max\_SS}? * 0.05 + 30) \div (100 * \mathit{Thrust\_Demand}?) + 30 \\
\land \\
\mathit{Dem\_SS} \geq \mathit{Max\_SS}? \land \mathit{Lim\_Dem\_SS} = \mathit{Max\_SS}? \lor \\
\mathit{Dem\_SS} < \mathit{Max\_SS}? \land \mathit{Lim\_Dem\_SS} = \mathit{Dem\_SS} \\
\land \\
\mathit{Cur\_SS} \geq \mathit{ABS\_MIN\_SS} \land \mathit{Cur\_SS}? \leq \mathit{ABS\_MAX\_SS} \land \\
\quad \mathit{Delta\_SS}! = \mathit{Lim\_Dem\_SS} - \mathit{Cur\_SS}?
\end{array}
$$

Adjust_Thrust$_m$

$$
\begin{array}{l}
Cur\_SS? : \mathbb{R} \\
Delta\_SS! : \mathbb{R} \\
Dem\_SS : \mathbb{R} \\
Lim\_Dem\_SS : \mathbb{R} \\
Max\_SS? : \mathbb{R} \\
Thrust\_Demand? : \mathbb{R} \\
\hline
Max\_SS? < ABS\_MAX\_SS \land \\
IDLE\_THRUST\_DEMAND < Thrust\_Demand? \land \\
Thrust\_Demand? < MAX\_THRUST\_DEMAND \land \\
Thrust\_Demand? \neq 0 \land \\
\quad Dem\_SS = \\
\quad (Max\_SS? * 0.05 + 30) \div \boxed{(100 + Thrust\_Demand?)} + 30 \\
\land \\
Dem\_SS \geq Max\_SS? \land Lim\_Dem\_SS = Max\_SS? \lor \\
Dem\_SS < Max\_SS? \land Lim\_Dem\_SS = Dem\_SS \\
\land \\
Cur\_SS \geq ABS\_MIN\_SS \land Cur\_SS? \leq ABS\_MAX\_SS \land \\
\quad Delta\_SS! = Lim\_Dem\_SS - Cur\_SS?
\end{array}
$$

Simplifying and restricting the schemas to the input and output variables results in a definition of *Delta_SS*! in terms of the inputs only.

Adjust_Thrust

$$
\begin{array}{l}
Cur\_SS? : \mathbb{R} \\
Delta\_SS! : \mathbb{R} \\
Max\_SS? : \mathbb{R} \\
Thrust\_Demand? : \mathbb{R} \\
\hline
(Max\_SS? < ABS\_MAX\_SS \land \\
IDLE\_THRUST\_DEMAND < Thrust\_Demand? \land \\
Thrust\_Demand? < MAX\_THRUST\_DEMAND \land \\
Thrust\_Demand? \neq 0 \land \\
(Max\_SS? * 0.05 + 30) \div (100 * Thrust\_Demand?) + 30 \geq Max\_SS? \land \\
Cur\_SS \geq ABS\_MIN\_SS \land Cur\_SS? \leq ABS\_MAX\_SS \land \\
\quad Delta\_SS! = Max\_SS? - Cur\_SS?) \\
\lor \\
(Max\_SS? < ABS\_MAX\_SS \land \\
IDLE\_THRUST\_DEMAND < Thrust\_Demand? \land \\
Thrust\_Demand? < MAX\_THRUST\_DEMAND \land \\
Thrust\_Demand? \neq 0 \land \\
(Max\_SS? * 0.05 + 30) \div (100 * Thrust\_Demand?) + 30 < Max\_SS? \land \\
Cur\_SS \geq ABS\_MIN\_SS \land Cur\_SS? \leq ABS\_MAX\_SS \land \\
\quad Delta\_SS! = (Max\_SS? * 0.05 + 30) \div (100 * Thrust\_Demand?) + 30 - Cur\_SS?)
\end{array}
$$

Adjust_Thrust$_m$

$$
\begin{array}{l}
Cur\_SS? : \mathbb{R} \\
Delta\_SS! : \mathbb{R} \\
Max\_SS? : \mathbb{R} \\
Thrust\_Demand? : \mathbb{R} \\
\hline
(Max\_SS? < ABS\_MAX\_SS \land \\
IDLE\_THRUST\_DEMAND < Thrust\_Demand? \land \\
Thrust\_Demand? < MAX\_THRUST\_DEMAND \land \\
Thrust\_Demand? \neq 0 \land \\
(Max\_SS? * 0.05 + 30) \div (100 + Thrust\_Demand?) + 30 \geq Max\_SS? \land \\
Cur\_SS \geq ABS\_MIN\_SS \land Cur\_SS? \leq ABS\_MAX\_SS \land \\
\quad Delta\_SS! = Max\_SS? - Cur\_SS?) \\
\lor \\
(Max\_SS? < ABS\_MAX\_SS \land \\
IDLE\_THRUST\_DEMAND < Thrust\_Demand? \land \\
Thrust\_Demand? < MAX\_THRUST\_DEMAND \land \\
Thrust\_Demand? \neq 0 \land \\
(Max\_SS? * 0.05 + 30) \div (100 + Thrust\_Demand?) + 30 < Max\_SS? \land \\
Cur\_SS \geq ABS\_MIN\_SS \land Cur\_SS? \leq ABS\_MAX\_SS \land \\
\quad Delta\_SS! = (Max\_SS? * 0.05 + 30) \div (100 + Thrust\_Demand?) + 30 - Cur\_SS?)
\end{array}
$$

The test case specification of the sufficient condition to detect the hypothesised fault based on the template:

$$\vdash? \exists Inputs\_Outputs(P) \bullet \neg(P \Leftrightarrow P_m)$$

which can be represented using schema calculus as follows:

Detect_Fault

$$
\begin{array}{l}
Inputs\_Outputs(P) \\
\hline
\neg(P \Leftrightarrow P_m)
\end{array}
$$

is therefore:

Detect_Fault

$$
\begin{array}{l}
\mathit{Cur\_SS}? : \mathbb{R} \\
\mathit{Delta\_SS}! : \mathbb{R} \\
\mathit{Max\_SS}? : \mathbb{R} \\
\mathit{Thrust\_Demand}? : \mathbb{R} \\
\hline
\neg(((\mathit{Max\_SS}? < \mathit{ABS\_MAX\_SS} \land \\
\mathit{IDLE\_THRUST\_DEMAND} < \mathit{Thrust\_Demand}? \land \\
\mathit{Thrust\_Demand}? < \mathit{MAX\_THRUST\_DEMAND} \land \\
\mathit{Thrust\_Demand}? \neq 0 \land \\
(\mathit{Max\_SS}? * 0.05 + 30) \div (100 * \mathit{Thrust\_Demand}?) + 30 \geq \mathit{Max\_SS}? \land \\
\mathit{Cur\_SS} \geq \mathit{ABS\_MIN\_SS} \land \mathit{Cur\_SS}? \leq \mathit{ABS\_MAX\_SS} \land \\
\quad \mathit{Delta\_SS}! = \mathit{Max\_SS}? - \mathit{Cur\_SS}?) \\
\lor \\
(\mathit{Max\_SS}? < \mathit{ABS\_MAX\_SS} \land \\
\mathit{IDLE\_THRUST\_DEMAND} < \mathit{Thrust\_Demand}? \land \\
\mathit{Thrust\_Demand}? < \mathit{MAX\_THRUST\_DEMAND} \land \\
\mathit{Thrust\_Demand}? \neq 0 \land \\
(\mathit{Max\_SS}? * 0.05 + 30) \div (100 * \mathit{Thrust\_Demand}?) + 30 < \mathit{Max\_SS}? \land \\
\mathit{Cur\_SS} \geq \mathit{ABS\_MIN\_SS} \land \mathit{Cur\_SS}? \leq \mathit{ABS\_MAX\_SS} \land \\
\quad \mathit{Delta\_SS}! = \\
\quad (\mathit{Max\_SS}? * 0.05 + 30) \div (100 * \mathit{Thrust\_Demand}?) + 30 - \mathit{Cur\_SS}?)) \\
\Leftrightarrow \\
((\mathit{Max\_SS}? < \mathit{ABS\_MAX\_SS} \land \\
\mathit{IDLE\_THRUST\_DEMAND} < \mathit{Thrust\_Demand}? \land \\
\mathit{Thrust\_Demand}? < \mathit{MAX\_THRUST\_DEMAND} \land \\
\mathit{Thrust\_Demand}? \neq 0 \land \\
(\mathit{Max\_SS}? * 0.05 + 30) \div (100 + \mathit{Thrust\_Demand}?) + 30 \geq \mathit{Max\_SS}? \land \\
\mathit{Cur\_SS} \geq \mathit{ABS\_MIN\_SS} \land \mathit{Cur\_SS}? \leq \mathit{ABS\_MAX\_SS} \land \\
\quad \mathit{Delta\_SS}! = \mathit{Max\_SS}? - \mathit{Cur\_SS}?) \\
\lor \\
(\mathit{Max\_SS}? < \mathit{ABS\_MAX\_SS} \land \\
\mathit{IDLE\_THRUST\_DEMAND} < \mathit{Thrust\_Demand}? \land \\
\mathit{Thrust\_Demand}? < \mathit{MAX\_THRUST\_DEMAND} \land \\
\mathit{Thrust\_Demand}? \neq 0 \land \\
(\mathit{Max\_SS}? * 0.05 + 30) \div (100 + \mathit{Thrust\_Demand}?) + 30 < \mathit{Max\_SS}? \land \\
\mathit{Cur\_SS} \geq \mathit{ABS\_MIN\_SS} \land \mathit{Cur\_SS}? \leq \mathit{ABS\_MAX\_SS} \land \\
\quad \mathit{Delta\_SS}! = \\
\quad (\mathit{Max\_SS}? * 0.05 + 30) \div (100 + \mathit{Thrust\_Demand}?) + 30 - \mathit{Cur\_SS}?)))
\end{array}
$$

If the sufficient condition can be shown to be satisfiable, the fault is said to be *strongly detectable*. The above test case can be solved using the inputs: $\mathit{Max\_SS}? = 30.2434$, $\mathit{Thrust\_Demand}? = 30$ and $\mathit{Cur\_SS}? = 28.77594$ (generated using the non-linear constraint solver LINGO [Inc01]).

In general, it may not always be possible to construct a satisfiable sufficient condition for all possible mutations. In these cases the mutants can be said to be semantically *equivalent* to the original specification. As in partition-based testing, the effectiveness of the technique will depend on certain styles of specification. The number of possible solutions to the sufficient conditions can be used as a measure of how testable an implementation is against faults that reveal themselves as mutations in the specification. The larger the number of solutions to the sufficient conditions, the higher the probability that one of these solutions would be selected during standard (e.g. random or partition) testing procedures. In some cases it may not be efficient to generate and solve the sufficient conditions for all hypothesised faults. In these cases, the necessary condition tests can be generated which are not as likely to detect faults as their equivalent sufficient conditions but may be computationally less expensive to generate.

5.4 Tool support

This section describes how the formalised partitioning and fault-based testing heuristics described previously have been implemented using a general purpose theorem prover. CADiℤ [Toy96, Toy00] is a Z type checker and theorem prover that allows a user to interactively browse, type check and perform proofs upon a Z specification. CADiℤ also allows proof tactics to be written in a lazy functional notation [Toy98]. Tactics can be invoked from within a CADiℤ window and applied to any proof obligation on the screen. The tactic language includes parametrisation and pattern matching facilities to enable generic tactics to be written for implementing particular proof *patterns*. When such a tactic is applied, the parameters are instantiated with the currently selected parts of the specification (or typed input from the user) and any remaining looseness in the specification of the tactic is instantiated using pattern matching. Generic proof tactics were used as the basis for automating the generation of abstract test cases based on the formal specification of the testing heuristic.

Testing heuristics are specified as named "lemmas" and stored in a separate Z file that is included within the scope of any specifications being tested. The completeness, disjointness and satisfiability properties of these heuristics can be proven using the CADiℤ interactive proof mode. The corresponding proofs can be recorded as proof tactics to be replayed at a later date (for example, by a new user or a certification authority who is as yet unconvinced

of the heuristics' validity). As a result of the automated and generic nature of the test generation mechanism, a new heuristic can be added to the tool by simply adding it to the set of lemmas. This allows for great flexibility when designing tests, as optimal heuristics can be quickly identified through experimentation since the addition of a new heuristic does not require changes to either the source code or proof tactics. By ensuring that a heuristic is specified as a lemma (it can always be reduced to true), it can be added to the operation under test without changing the behaviour of the specification. The parameters of the heuristic can then be instantiated with the operands of the predicate under test and the result used to define the constraining predicate or predicates of the test. In this sense, the heuristics define generic *patterns* for the test cases.

The proof tactic used to apply partitioning heuristics to operation specifications is now described using an example application of a boundary value analysis heuristic. The parameters to the tactic are the predicate to be partitioned and the name of the partitioning heuristic to use. The description is supported by the partitioning example from Section 5.2.

1. Once the partitioning heuristic has been proven to be a tautology (by proving the completeness property described above), it can be added to the predicate part of the specification without changing the behaviour of the operation schema. The boundary value analysis heuristic (highlighted) is introduced to the *Calculate_demand* schema as follows:

Calculate_demand

$$
\begin{array}{l}
Max\_SS? : \mathbb{R} \\
Thrust\_Demand? : \mathbb{R} \\
Dem\_SS : \mathbb{R} \\
\hline
Max\_SS? < ABS\_MAX\_SS \land \\
IDLE\_THRUST\_DEMAND < Thrust\_Demand? \land \\
Thrust\_Demand? < MAX\_THRUST\_DEMAND \land \\
\boxed{(\forall A, B : \mathbb{R} \bullet A < B \Leftrightarrow (A < B - DELTA) \lor (A < B \land A \geq B - DELTA))} \land \\
Thrust\_Demand? \neq 0 \land \\
\quad Dem\_SS = \\
\quad (Max\_SS? * 0.05 + 30) \div (100 * Thrust\_Demand?) + 30
\end{array}
$$

2. Because the heuristic has been proven valid for *all* values of its parameters, it can be instantiated with the operands of the predicate to be partitioned and remain valid. In

the example, *A* and *B* are instantiated with *Thrust_Demand*? and *MAX_THRUST_DEMAND* respectively leaving the instantiated form of the heuristic conjoined with the generic form. The instantiation is performed using the pattern matching mechanism of the tactic language.

Calculate_demand

$$
\begin{array}{l}
Max\_SS? : \mathbb{R} \\
Thrust\_Demand? : \mathbb{R} \\
Dem\_SS : \mathbb{R} \\
\hline
Max\_SS? < ABS\_MAX\_SS \land \\
IDLE\_THRUST\_DEMAND < Thrust\_Demand? \land \\
Thrust\_Demand? < MAX\_THRUST\_DEMAND \land \\
(\forall A, B : \mathbb{R} \bullet A < B \Leftrightarrow A < B - DELTA \lor (A < B \land A \geq B - DELTA)) \land \\
(\boxed{Thrust\_Demand? < MAX\_THRUST\_DEMAND} \Leftrightarrow \\
(Thrust\_Demand? < MAX\_THRUST\_DEMAND - DELTA) \lor \\
\quad (Thrust\_Demand? < MAX\_THRUST\_DEMAND \land \\
\quad Thrust\_Demand? \geq MAX\_THRUST\_DEMAND - DELTA)) \land \\
Thrust\_Demand? \neq 0 \land \\
\quad Dem\_SS = \\
\quad (Max\_SS? * 0.05 + 30) \div (100 * Thrust\_Demand?) + 30
\end{array}
$$

After some simplification to remove the generic form of the heuristic and second *Thrust_Demand*? < *MAX_THRUST_DEMAND* predicate (highlighted above), this leaves:

Calculate_demand

$$
\begin{array}{l}
Max\_SS? : \mathbb{R} \\
Thrust\_Demand? : \mathbb{R} \\
Dem\_SS : \mathbb{R} \\
\hline
Max\_SS? < ABS\_MAX\_SS \land \\
IDLE\_THRUST\_DEMAND < Thrust\_Demand? \land \\
Thrust\_Demand? < MAX\_THRUST\_DEMAND \land \\
\boxed{(Thrust\_Demand? < MAX\_THRUST\_DEMAND - DELTA \lor} \\
\quad \boxed{Thrust\_Demand? < MAX\_THRUST\_DEMAND \land} \\
\quad \boxed{Thrust\_Demand? \geq MAX\_THRUST\_DEMAND - DELTA)} \land \\
Thrust\_Demand? \neq 0 \land \\
\quad Dem\_SS = \\
\quad (Max\_SS? * 0.05 + 30) \div (100 * Thrust\_Demand?) + 30
\end{array}
$$

3. The disjunction of the predicates defining the subdomains (highlighted above) is distributed to the outer-most predicate of the schema.

Calculate_demand

$$
\begin{array}{l}
\mathit{Max\_SS}? : \mathbb{R} \\
\mathit{Thrust\_Demand}? : \mathbb{R} \\
\mathit{Dem\_SS} : \mathbb{R} \\
\hline
(\mathit{Max\_SS}? < \mathit{ABS\_MAX\_SS} \land \\
\mathit{IDLE\_THRUST\_DEMAND} < \mathit{Thrust\_Demand}? \land \\
\mathit{Thrust\_Demand}? < \mathit{MAX\_THRUST\_DEMAND} \land \\
\mathit{Thrust\_Demand}? < \mathit{MAX\_THRUST\_DEMAND} - \mathit{DELTA} \land \\
\mathit{Thrust\_Demand}? \neq 0 \land \\
\quad \mathit{Dem\_SS} = \\
\quad (\mathit{Max\_SS}? * 0.05 + 30) \div (100 * \mathit{Thrust\_Demand}?) + 30) \\
\lor \\
(\mathit{Max\_SS}? < \mathit{ABS\_MAX\_SS} \land \\
\mathit{IDLE\_THRUST\_DEMAND} < \mathit{Thrust\_Demand}? \land \\
\mathit{Thrust\_Demand}? < \mathit{MAX\_THRUST\_DEMAND} \land \\
\mathit{Thrust\_Demand}? < \mathit{MAX\_THRUST\_DEMAND} \land \\
\mathit{Thrust\_Demand}? \geq \mathit{MAX\_THRUST\_DEMAND} - \mathit{DELTA} \land \\
\mathit{Thrust\_Demand}? \neq 0 \land \\
\quad \mathit{Dem\_SS} = \\
\quad (\mathit{Max\_SS}? * 0.05 + 30) \div (100 * \mathit{Thrust\_Demand}?) + 30)
\end{array}
$$

4. Schema calculus is then used to separate the operation into schemas for each test case. This results in the schemas $\mathit{Partition}_1$ and $\mathit{Partition}_2$ from section 5.2.

The proof steps described above made no assumptions about the predicate to be partitioned or the structure of the testing heuristic other than that it followed the template defined in Section 5.3.1. It should therefore be clear how the proof tactic is generalised to all partitioning strategies defined according to the template.

The tactic for applying fault-based heuristics is described next and demonstrated using the same schema as above and the following fault based heuristic:

$$\exists A, B : \mathbb{R} \bullet A * B \neq A + B$$

applied to the expression $(100 * \mathit{Thrust\_Demand}?)$ to detect a fault where the multiplication is mutated into an addition.

1. Once the fault-based heuristic has been shown to be satisfiable, it can be added to the predicate part of the specification without changing the behaviour of the operation schema. The fault-based heuristic (highlighted) is introduced to the *Calculate_ demand* schema as follows:

Calculate_demand

$$
\begin{array}{l}
Max\_SS? : \mathbb{R} \\
Thrust\_Demand? : \mathbb{R} \\
Dem\_SS : \mathbb{R} \\
\hline
Max\_SS? < ABS\_MAX\_SS \land \\
IDLE\_THRUST\_DEMAND < Thrust\_Demand? \land \\
Thrust\_Demand? < MAX\_THRUST\_DEMAND \land \\
\quad (Thrust\_Demand? < MAX\_THRUST\_DEMAND \land \\
\quad Thrust\_Demand? > MAX\_THRUST\_DEMAND - DELTA)) \land \\
Thrust\_Demand? \neq 0 \land \\
\boxed{(\exists A, B : \mathbb{R} \bullet A * B \neq A + B)} \land \\
\quad Dem\_SS = \\
\quad (Max\_SS? * 0.05 + 30) \div (100 * Thrust\_Demand?) + 30
\end{array}
$$

2. The heuristic can now be instantiated with the operands of the target expression (100 and *Thrust_Demand*) leaving the instantiated form of the heuristic disjoined with the generic form. The instantiation is performed using the pattern matching mechanism of the tactic language.

Calculate_demand

$$
\begin{array}{l}
Max\_SS? : \mathbb{R} \\
Thrust\_Demand? : \mathbb{R} \\
Dem\_SS : \mathbb{R} \\
\hline
Max\_SS? < ABS\_MAX\_SS \land \\
IDLE\_THRUST\_DEMAND < Thrust\_Demand? \land \\
Thrust\_Demand? < MAX\_THRUST\_DEMAND \land \\
\quad (Thrust\_Demand? < MAX\_THRUST\_DEMAND \land \\
\quad Thrust\_Demand? > MAX\_THRUST\_DEMAND - DELTA)) \land \\
Thrust\_Demand? \neq 0 \land \\
(\exists A, B : \mathbb{R} \bullet A * B \neq A + B \lor \\
100 * Thrust\_Demand? \neq 100 + Thrust\_Demand?) \land \\
\quad Dem\_SS = \\
\quad (Max\_SS? * 0.05 + 30) \div (100 * Thrust\_Demand?) + 30
\end{array}
$$

3. After some simplification to reduce the generic form of the heuristic to *true*, this leaves:

$$
\begin{array}{|l}
\hline
\textit{Calculate\_demand} \\
\textit{Max\_SS}? : \mathbb{R} \\
\textit{Thrust\_Demand}? : \mathbb{R} \\
\textit{Dem\_SS} : \mathbb{R} \\
\hline
\textit{Max\_SS}? < \textit{ABS\_MAX\_SS} \land \\
\textit{IDLE\_THRUST\_DEMAND} < \textit{Thrust\_Demand}? \land \\
\textit{Thrust\_Demand}? < \textit{MAX\_THRUST\_DEMAND} \land \\
\quad (\textit{Thrust\_Demand}? < \textit{MAX\_THRUST\_DEMAND} \land \\
\quad \textit{Thrust\_Demand}? > \textit{MAX\_THRUST\_DEMAND} - \textit{DELTA})) \land \\
\textit{Thrust\_Demand}? \neq 0 \land \\
\boxed{(\textit{true} \lor} \\
\boxed{100 * \textit{Thrust\_Demand}? \neq 100 + \textit{Thrust\_Demand}?)} \land \\
\quad \textit{Dem\_SS} = \\
\quad (\textit{Max\_SS}? * 0.05 + 30) \div (100 * \textit{Thrust\_Demand}?) + 30 \\
\hline
\end{array}
$$

4. The disjunction (highlighted above) is distributed to the outer-most predicate of the schema before the *true* predicate is simplified away. In this way, the original operation is also preserved in order that alternative fault-based heuristics may be applied to it. The distribution results in the following schema:

Calculate_demand

$$
\begin{array}{l}
\mathit{Max\_SS?} : \mathbb{R} \\
\mathit{Thrust\_Demand?} : \mathbb{R} \\
\mathit{Dem\_SS} : \mathbb{R} \\
\hline
(\mathit{Max\_SS?} < \mathit{ABS\_MAX\_SS} \land \\
\mathit{IDLE\_THRUST\_DEMAND} < \mathit{Thrust\_Demand?} \land \\
\mathit{Thrust\_Demand?} < \mathit{MAX\_THRUST\_DEMAND} \land \\
\quad (\mathit{Thrust\_Demand?} < \mathit{MAX\_THRUST\_DEMAND} \land \\
\quad \mathit{Thrust\_Demand?} > \mathit{MAX\_THRUST\_DEMAND} - \mathit{DELTA})) \land \\
\mathit{Thrust\_Demand?} \neq 0 \land \\
\quad \mathit{Dem\_SS} = \\
\quad (\mathit{Max\_SS?} * 0.05 + 30) \div (100 * \mathit{Thrust\_Demand?}) + 30) \\
\lor \\
(\mathit{Max\_SS?} < \mathit{ABS\_MAX\_SS} \land \\
\mathit{IDLE\_THRUST\_DEMAND} < \mathit{Thrust\_Demand?} \land \\
\mathit{Thrust\_Demand?} < \mathit{MAX\_THRUST\_DEMAND} \land \\
\quad (\mathit{Thrust\_Demand?} < \mathit{MAX\_THRUST\_DEMAND} \land \\
\quad \mathit{Thrust\_Demand?} > \mathit{MAX\_THRUST\_DEMAND} - \mathit{DELTA})) \land \\
\mathit{Thrust\_Demand?} \neq 0 \land \\
100 * \mathit{Thrust\_Demand?} \neq 100 + \mathit{Thrust\_Demand?} \land \\
\quad \mathit{Dem\_SS} = \\
\quad (\mathit{Max\_SS?} * 0.05 + 30) \div (100 * \mathit{Thrust\_Demand?}) + 30)
\end{array}
$$

5. Schema calculus is then used to separate the operation into two schemas that represent the original operation (to which alternative fault-based or partitioning strategies can be applied) and the operation restricted by the constraining predicate for (weakly) detecting the hypothesised fault.

By applying the heuristics as a series of Z proof steps and proving the equivalence or satisfiability represented by the formal definition of the heuristic, the test case derivation procedure results in test cases that are guaranteed to maintain conformance with the original specification and inherit the properties of the generic testing heuristic (e.g. completeness and disjointness).

Once generated, test specifications are instantiated with test data via a similar mechanism to the application of test heuristics. A proof tactic is applied that simplifies the constraint and applies either the SMV model checker [Ber00] or a simulated annealing constraint solver [CT97] to generate a set of data satisfying the test specification. In the few cases where these "built-in" constraint solvers were not applicable to the constraints, the

stand-alone constraint solver LINGO [Inc01] was used. Ideally, CADiℤ would be extended to include constraint solvers applicable to a wider range of constraints.

As part of future work, the API to CADiℤ could be used in order to provide an intuitive user interface environment for specifying and applying testing heuristics to Z specifications. This may be achieved by producing a custom made environment or integrating the tool via the API to existing test management tools. As part of this expansion, techniques could be implemented for extracting predicates from the Z specification to provide a choice of suitable candidates for partitioning to the user as well as a list of potential specification mutations for which data can be generated. CADiℤ currently only supports the application of necessary conditions for fault-based testing, but future work could also extend the tool to be able to apply sufficient conditions, which unlike partitioning heuristics and necessary conditions, can not be defined generically using the Z lemma mechanism described above because the structure of each sufficiency condition depends on the particular operation that is being analysed.

5.5 Case studies

The automated test case generation techniques have been applied to a number of specifications including the large case study described in Section 4.6. Table 5.2 summarises the results of applying disjunction and boundary value analysis to these specifications. The higher ratio of test cases to operations for the reactive components reflected the complexity of the predicates used in these parts of the specification. The Statechart transitions were typically triggered by simple combinations of conditions, whereas the reactive specifications were used to calculate the values of these conditions based on more complex relationships between sensor values and other inputs to the system.

The test cases were transformed into test scripts using an extension of the test data generation facility which exported the test data in the form of an AdaTEST [Inf97] test script. The resulting test script then needed to be modified to represent the variable names and data types used in the implementation. The source code was produced based on a fixed template for implementing Statecharts and reactive tables. This resulted in a relatively simple and predictable refinement between the specifications and the code. The modifications to the test scripts could therefore be automated using simple search and replace commands within

State-based components:		**Reactive components:**	
State-machines	9	Tables	34
States	48		
Transitions	112	Definition/Guard pairings	84
Z operations	112	Z operations	34
Number of test cases	262	Number of test cases	237
Test cases per operation	2.34	Test cases per operation	6.97

Table 5.2: Summary of results

a text editor. If the refinement to code had not been so regular more manual effort would have been required to generate the concrete test scripts. However, the use of graphical and tabular specifications encourages strict programming styles and implementation patterns and therefore this method of automated testing seems well suited to the notations. The informal text in the Z specification (automatically generated by StateZ to provide traceability information and a description of semantic resolution steps) was used to generate comments in the test script. These comments were found to be extremely useful for relating a failed test script to a fault corresponding to a particular part of the original specification.

Automated partition-based testing was also used on a second case study to examine the relationship between specification and code structural coverage criteria and the relative merits of specification-based or program-based test generation techniques. For this case study, the code and specifications were again taken from a commercial electronic engine controller project and consisted of several Statecharts and around 200 hundred lines of Ada code were selected to be typical for the application. Unsurprisingly, the structural code coverage achieved by running tests designed to achieve specification coverage was dependent on the refinement. As in the other case studies, this refinement was found to be fairly limited and a good level of coverage was achieved (100% Decision coverage, and 83.33% Boolean Operand Effectiveness[3]) using test cases automatically generated using the disjunction partitioning heuristic. Discrepancies between the code coverage and the specification coverage of the tests indicated either areas of refinement in the code or faults in the implementation. In either case, more intensive manual inspection could be prioritised towards these areas. The same study demonstrated how structural code coverage was not sufficient to detect all

[3]Similar to MC/DC coverage mandated for safety-critical aerospace applications

faults in the code. The use of fault-based heuristics was shown to be able to detect a number of these faults. Results regarding the use of fault-based heuristics will be discussed in more detail in Chapter 7.

The high level of automation led to the generation of a far greater number of test cases which were produced in much shorter time than the manual approaches typically used in industry. Unit testing is traditionally seen as a relatively low value V&V method mainly due to the time consuming approach to designing tests and generating test scripts and the comparatively few errors found with respect to other testing techniques. The high level of automation based on a strict specification of the expected behaviour allowed errors to be found in the code extremely quickly (the testing of a module would take 1 or 2 hours compared to 1 or 2 days using traditional processes). The use of such techniques may serve to increase the effectiveness of unit testing, which is still a required part of the testing process as it is currently the only practical means of obtaining mandated structural coverage metrics. The case studies also demonstrated that a combination of structural (e.g. disjunction analysis) and fault-based heuristics was required in order to achieve the greatest fault coverage. The selection of an efficient subset of fault-based heuristics as a means of reducing the size of the test set while maintaining its effectiveness is discussed in more detail in Chapter 7.

5.6 Summary

This chapter has presented a method for automatically applying partitioning and fault-based testing heuristics to Z operations as a means of generating test cases. The heuristics are applied in such a way that a high level of automation is achieved. The most significant advancement over previous work in the area is the formalisation of the testing heuristics and the exploitation of this formalisation to provide a provable and automated means of test case generation. Under the same framework, the chapter also introduced the concept of automated fault-based testing as a means of increasing the probability that the tests will detect particular classes of faults. Case study work has shown how the automated techniques can significantly improve the efficiency and effectiveness of unit testing processes.

The chapter described the method in terms of its applicability to general Z specifications and demonstrated the technique in terms of a reactive specification described using PFS-style tables. However, in order to be able to apply these techniques to the specifica-

tions generated from Statecharts, extensions to the techniques are required. Consider the following schema (from Section 4.3.2) which describes a transition (*T3*) from the example used in the same chapter.

$$
\begin{array}{|l}
\hline
T3 \\
\Delta Status \\
Parameters \\
\hline
\{ThrustLimitation, InadvertentDeploy, AwaitLock, \\
Locked, OnGroundDetect\} \subseteq ActiveStates \\
ThrottleReq? \leq FwdIdle \\
RestowComplete! = Present \\
LimitThrust! = False \\
\{ThrustLimitation, Idle\} \subseteq ActiveStates' \\
\hline
\end{array}
$$

Typically, when testing a Statechart specification, it may be that the only controllable and observable elements in the system are the input and output parameters. The status (state and internal variables) may not be directly testable as they may either be stored as internal variables to which the test environment does not have access or they may be refined as part of the implementation process and therefore may no longer map directly to the same elements in the specification. This gives rise to two problems. Firstly, the values of the status need to be set to values that satisfy the pre-condition of the operation and secondly the updated values of the status need to be verified following an execution of the transition. Suppose that the transition is implemented incorrectly and the implementation is not transferred into the *Idle* state. Because the values of the outputs are not dependent on the updated value of the status variable *ActiveState*, it is not possible to measure this mutation during a single step test. However, as the values of the status will affect subsequent operations, the fault may be observed in future transitions. The next chapter describes techniques for generating sequences of Z operations for setting the values of the status for a particular test and generating sequences of test cases that are capable of detecting faults in the status transfer function through indirect observation using the input and output parameters of the system.

Chapter 6

Test sequence generation

This chapter presents a method of generating sequences of test cases from Z specifications, and in particular Z specifications derived from Statechart diagrams. A combination of equivalence class testing, abstraction and FSM-based test sequence generation techniques is described. Optimisations to the generic method are then presented based on the fixed structure of the Z specifications generated from Statecharts. These optimisations facilitate better modelling of concurrency and non-determinism in the abstraction.

6.1 Introduction

Testing state-based systems is often hindered by the fact that the system state is not directly controllable or observable at the testing interface. Sequences of operations are therefore required to bring the system into a particular configuration for a test and to indirectly observe the effect of a test on the state of the system. The formulation of sufficient fault detection conditions for state-based systems requires an extension to the techniques described in the previous chapter as updated internal variables are typically not referenced until a later operation. Errors involving these state variables therefore might not be detected by a single invocation of an operation. Testing under these conditions involves finding *sequences* of operations that are sufficient to detect mutations of the system state. This is the implicit assumption behind many of the FSM-based test sequence generation techniques described in Chapter 2 (e.g. [Cho78, SD88, FvBK$^+$91]). However, these techniques are only applicable to state-based specifications with very simple transition relations. In general, the system

state may consist of a combination of internal state (in the classic FSM definition of the term) and variables which may also be referenced in the guards and actions of transitions.

This chapter describes how test sequences for detecting faults in the system state as well as those that satisfy other user defined properties (e.g. state initialisation) can be generated for systems specified using Statecharts and more generally Z that makes use of the schema Δ notation. In order to generate test sequences, a method of selecting feasible sequences of operations is required. Section 6.2 describes how a finite state machine can be derived from a set of Z operations for this purpose. This Abstract Finite State Machine (AFSM) describes possible sequences of the operations and is used in Section 6.3 in conjunction with traditional FSM-based approaches to generating state checking sequences. Section 6.4 describes how model checking can be used to generate more general sequences based on a specification of desired properties of the tests and to validate the feasibility of the sequences. Section 6.5 presents experience of using these techniques on a number of example specifications. Finally, the key contributions and results of the Chapter are summarised in Section 6.6.

6.2 Constructing the abstract finite state machine

Dick and Faivre [DF93] proposed the construction of an abstract finite state machine as a means of sequencing test cases derived from and specified in the VDM-SL notation. Murray et al. [MCM+98] have since shown how the technique can be applied to Z specifications in combination with the Category-partition method discussed in Section 2.3.3. An alternative description of the abstract finite state machine is presented here that also includes abstraction of the inputs and outputs. The work also varies from other authors in this area by considering the application of automated test sequence generation techniques to the resulting AFSM and by taking into account orthogonality in the specification in a manner that limits the effect of state explosion.

The purpose of constructing the abstract finite state machine is to calculate the set of possible sequences of operations from which the test sequences can be selected. Two operations can be sequenced if some values for the system state can be selected from the post-condition of the first that also satisfy the pre-condition of the second. To calculate all possible sequences of operations, the disjoint set of equivalence classes of the system state therefore need to be calculated. These equivalence classes, described as constraints over the

state space, form the abstract states of the machine. The method for identifying the abstract states is as follows:

1. Extract the set of predicates that define the projection of the pre-conditions of the operations onto the system state (*State*) i.e. hide the inputs, outputs and after state. This can be achieved using Z's schema projection operator as follows:

   $$BeforeState == Op \upharpoonright State$$

   Some simplification to remove the resulting existentially quantified hidden variables may be required. The new restricted pre-conditions may overlap.

2. Extract the predicates that define the projection of the post-condition onto the system state (hiding the inputs, outputs and before state). Again some simplification may be necessary and the resulting references to the system state should be renamed to remove the $'$ after state decoration.

   $$AfterState == Op \upharpoonright State'[/']$$

3. The set of predicates from steps 1 and 2 are combined and used to form a set of equivalence classes of the system state. The disjoint set of abstract states is calculated by partitioning each pair of overlapping predicates A and B into the following abstract states: $A \wedge B$, $A \wedge \neg B$ and $\neg A \wedge B$. Any states whose predicates can be simplified to false are discarded. Ensuring that the abstract states are disjoint will reduce (though not completely eradicate) non-determinism in the AFSM.

Figure 6.1 contains a Z specification that contains one variable (*Counter*), one input (*Reset*?) and one output (*Output*!). While *Reset*? is *False* the system repeatedly outputs *False* fives times followed by *True*. When *Reset*? is true, the output is *False* and the counter is set to 0, resetting the sequence. Table 6.1 summarises the calculation of the abstract states for the example. The first column lists the state pre-conditions, the second column lists the state post-conditions. The last column lists the disjoint set of abstract states calculated from these predicates with *false* predicates removed.

As an example use of the schema projection operator, the state pre-condition for *Op1* is calculated as follows:

$$Op1Pre == Op1 \upharpoonright State$$

$Boolean == \{True, False\}$

State

$Counter : \mathbb{N}$

Reset

$Reset? : Boolean$
$Output! : Boolean$
$\Delta State$

$Reset? = True \land$
$Counter' = 0 \land$
$Output! = False$

*Op*1

$Reset? : Boolean$
$Output! : Boolean$
$\Delta State$

$Reset? = False \land Counter < 5 \land$
$Counter' = Counter + 1 \land$
$Output! = False$

*Op*2

$Reset? : Boolean$
$Output! : Boolean$
$\Delta State$

$Reset? = False \land Counter \geq 5 \land$
$Counter' = 0 \land$
$Output! = True$

Figure 6.1: Simple Z specification

Pre-conditions	**Post-conditions**	**Disjoint abstract states**
$Counter < 5$	$Counter = 0$	1: $Counter = 0$
$Counter \geq 5$	$Counter > 0 \land Counter < 6$	2: $Counter > 0 \land Counter < 5$
true		3: $Counter = 5$
		4: $Counter > 5$

Table 6.1: Calculation of abstract states for simple Z example

This is equivalent to the following schema definition:

*Op*1*Pre*

State

$\exists\, \mathit{Reset}? : \mathit{Boolean};\ \mathit{Output}! : \mathit{Boolean};\ \mathit{Counter}' : \mathbb{N} \mid$
$\quad \mathit{Reset}? = \mathit{False} \land \mathit{Counter} < 5 \land$
$\quad \mathit{Counter}' = \mathit{Counter} + 1 \land$
$\quad \mathit{Output}! = \mathit{False} \bullet \mathit{true}$

Each operation is assumed to be satisfiable (this could be checked as part of the specification validation activities), therefore values can be found for *Reset?* and *Output!* that satisfy their defining predicates. These predicates can then be reduced to true and the existential quantifier removed. Note that the predicate $\mathit{Counter}' = \mathit{Counter} + 1$ cannot be removed so easily as its satisfiability depends on the value of *Counter*. However it can be proven that for all possible values of *Counter*, there exists a values of *Counter*$'$ such that $\mathit{Counter}' = \mathit{Counter} + 1$. Thus leaving the following definition for the pre-condition.

*Op*1*Pre*

State

$\mathit{Counter} < 5$

In general, the simplification to remove the hidden variables may be a non trivial step and for this reason, the generation of the abstract states has not yet been automated. To ensure that the steps in the automation can be formally validated (as in the approach to operation-based testing) the identification of the abstract states should be performed within an automated theorem prover. This can be achieved in two ways: targeted proof tactics can be written to simplify commonly occurring constructs in the specification or proof tools can be used (or developed) that are able to simplify a wide range of conjectures into the desired form. In the short term, the former option may be more realistic as automated theorem provers will need to significantly improve before such simplifications can be performed without user interaction. However, a large number of proof tactics may be required to cover different constructs in the specification.

Applying the abstraction technique to a set of Z operations describing test cases will result in the enumeration of all disjoint equivalence classes to be tested in the system state.

Abstract Inputs		**Abstract Outputs**	
I1	$Reset? = False$	O1	$Output! = False$
I2	$Reset? = True$	O2	$Output! = True$

Table 6.2: Abstract inputs and outputs for simple Z example

Therefore if all abstract states can be visited and checked using sequences derived from the AFSM, faults resulting in the mutation of the system state would be strongly detectable, assuming that the equivalence classes are sufficient to reveal the fault.

A similar abstraction is applied to the controllable and observable parts of the specification (typically the input and output parameters) by extracting the predicates referring to only these variables. The disjoint partitions of the resulting predicates are calculated as before and the resulting sets of predicates form the sets of abstract input and output events X and Y respectively. Applying abstraction to the inputs and outputs is not in itself necessary to be able to identify sequences of operations but is important in order to reduce the state space to be explored by the test sequence generation techniques discussed later in this chapter. The abstract inputs and outputs for the example are summarised in Table 6.2.

The AFSM can be specified in terms of abstract states, inputs and outputs using the following tuple and definitions. An AFSM M is represented by the tuple $(S, s_0, X, Y, \delta, \lambda)$ where:

- S is the set of abstract states, where s_0 is the initial state, derived from the post-condition of the initialisation operation (assuming that the initialisation operation is deterministic).
- X is the input language (disjoint equivalence classes of the input predicates),
- Y is the output language (disjoint equivalence classes of the output predicates),
- δ is the state transfer function, $X \times S \rightarrow S$,
- and λ is the output function, $X \times S \rightarrow Y$.

The elements of the δ function are calculated by checking, for each combination of states S_i and S_j (including $S_i = S_j$) and input X_k whether there exists an operation Op such that the following predicate is true:

$$S_i \wedge X_k \wedge S'_j \wedge Op$$

Similarly the elements of the λ function are calculated by checking, for each combination of state S_i, input X_k and output Y_j, whether there exists an operation Op such that the following predicate is true:

$$S_i \wedge X_k \wedge Y_j \wedge Op$$

The resulting AFSM is minimised and any unreachable states (such as state 4 in the above example) are removed. Figure 6.2 shows the completed AFSM for the simple Z example from Figure 6.1. The abstraction may introduce non-determinism into the AFSM even in cases where the original specification is deterministic. However, this will typically be caused by only a subset of the state variables, for example those derived from Statechart variables that are referenced in both the guard and action of reflexive transitions. In the worst case, all state variables can lead to non-determinism as a result of the abstraction negating any benefits gained by the abstraction. However, the case studies implied that this situation was infrequent. More details on the effects of non-determinism and various techniques to resolve it under certain conditions are given in Section 6.2.1.

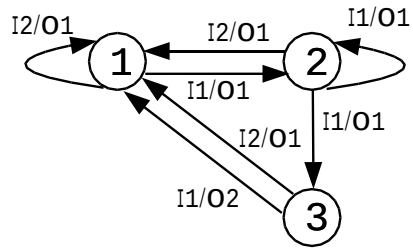


Figure 6.2: Abstract state machine for simple Z specification

6.2.1 Statechart-based optimisations

The abstraction can be applied to the Z formalisation of Statechart transitions described in Chapter 4. However, as the size of the specifications grow, factors leading to non-determinism and state explosion will have an increased impact on the efficiency of the test sequence generation techniques. Optimisations to the techniques can be used to limit these effects. However, the core techniques and tool sets should remain as generic as possible. Therefore, the optimisations presented here are either restricted to the formalisation phase

(translation to Z) or are generic to style of Z specifications used to model all source notations. The optimisations are targeted towards modelling concurrent aspects of the system and reducing the amount of unnecessary non-determinism introduced by the abstraction. Each of these problems is described in turn below followed by optimisations specific to the Z representation of Statecharts.

Over abstraction through loose specification

If an internal variable in a Statechart is not updated on a transition, the value of the variable following the transition is not explicitly specified. This is necessary in order to avoid contradictions when transitions from parallel components reference shared variables in action expressions. However, the semantics of Statecharts define that if an internal variable is not defined by *any* transitions in a step, then it retains its value from the previous step. The formalisation of Statechart transitions presented in Chapter 4 does not resolve this semantic issue and therefore the loose specification is propagated into the Z operations. This loose specification can lead to over abstraction in the AFSM resulting in test sequences being generated that may not be feasible in the original Statechart.

Over abstraction results when an internal variable is defined within an action, restricting it to a particular equivalence class, and is then referenced in the guard of a later transition. If transitions can occur between the defining and referencing transitions that do not reference the internal variable either in their guards or predicates, the defined equivalence class of the variable is lost as a result of the loose specification (the equivalence class of the variable following the transition is defined by the predicate *true*). Transitions could then be assigned to one of these abstract states that would otherwise have led to a contradiction between the guard and equivalence class predicate had the actual values of the internal variable been taken into account.

An example of this abstraction would have occurred had the *ThrustLimitation* Statechart (first presented on page 59) been modified so that transition *T5* contained the guard $Timer = 0$ and transition *T1* initialised the value of *Timer* to 0. In this version of the Statechart, the sequence of transitions: *T1, T2, T5, T11, T6.1, T5* would not be feasible as *T11* sets the value of *Timer* such that *T5* can no longer be triggered. However, an AFSM created using the abstraction method given above would allow this sequence as the equivalence class of

Timer is "forgotten" by the transition T6.1 which does not reference it in either its guard or action.

This over abstraction can be resolved in several ways. The formalisation of the Statecharts could be extended to ensure that the values of internal variables are always explicitly specified while avoiding conflicts when different orthogonal components reference the same variable. Alternatively, if references to the internal variables are restricted to a single writer/multiple reader relationship in terms of those orthogonal components that write to and read from the variable, a simpler extension to the formalism can be used. Each operation in the writing component that does not define the value of the variable can be extended to include the following relation in its post-condition:

$$InternalVariable' = InternalVariable$$

This would ensure that the next values of all internal variables are always defined. This latter solution was used here as the single writer/multiple reader restriction also has other significant advantages (see below). However as part of future work, a formalisation should be developed to allow for multiple writer/multiple reader relationships between orthogonal components referencing shared system state that does not result in over abstraction or state explosion.

State explosion through concurrency

Applying the abstraction described in Section 6.2 to Z specifications representing concurrent systems (i.e. the pre-conditions of more than one operation can be satisfied simultaneously) can lead to state explosion. Constructing the set of abstract states from the disjoint equivalence classes of the combined system state is equivalent to generating the product machine of the specification. The size of the resulting AFSM will therefore rise exponentially with the number of parallel components, states within these components and internal variables. The problem is further exacerbated if a large number of equivalence classes exist for the internal variables, e.g. due to the application of many testing heuristics. State explosion can drastically reduce the efficiency of test sequence generation algorithms, reducing the size of specification to which the techniques can be practically applied.

State explosion can be avoided under certain circumstances by modelling orthogonal components of the system as concurrent AFSMs and explicitly specifying the interactions

between them. The formulation of the AFSMs will depend on the type of interaction between the orthogonal components of the system. Components can interact in one of the following ways:

Full independence: The orthogonal components do not interact at all. The concurrent AFSMs can be represented as separate AFSMs that share the same set of input and output parameters but do not reference any shared state data on either the pre or post-conditions of any of their transitions.

Multiple read/single write: The orthogonal components interact via shared access to the common system state. The shared elements of the system state can only be written to by one component but can be read by any of the components in the system in the form of additional constraints on the pre-conditions of the operations. The abstract state machines must be extended to include these constraints on the pre-conditions of the transitions.

Multiple read/multiple write: The orthogonal components interact via shared access to a common system state. The shared elements of the system state can be written to and read by any of the components in the system. The abstract state machines must be extended to include a shared memory that can be updated on the post-conditions of the transitions and referenced in the pre-conditions. This memory need only contain elements of the system state shared between components on a multiple read/multiple write basis. All other parts of the system state can be abstracted using the techniques described above.

Efficiently modelling concurrency in Statechart specifications

If the interactions between the orthogonal components of the Statechart are limited to multiple reader/single writer relationships over elements of the status, the system can be represented as a set of concurrent AFSMs and the interactions between them as additional pre-condition constraints over the currently active (abstract) states of the other machines.

The orthogonal components in the specification must first be identified and the abstract states for these components calculated. The example Statechart from page 59 can be rewritten as the two Statecharts shown in Figures 6.3 and 6.4 respectively, where the complete-

ness, state hierarchy and priority resolution semantics have been made explicit on the transitions. In general, the number of orthogonal components will equal the number of children of the AND-states in the system which are either basic states or whose descendants are only OR and basic states.

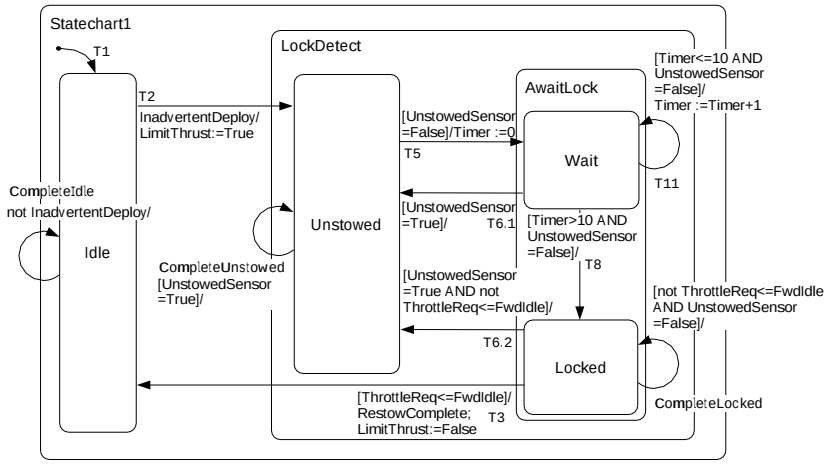


Figure 6.3: First orthogonal component

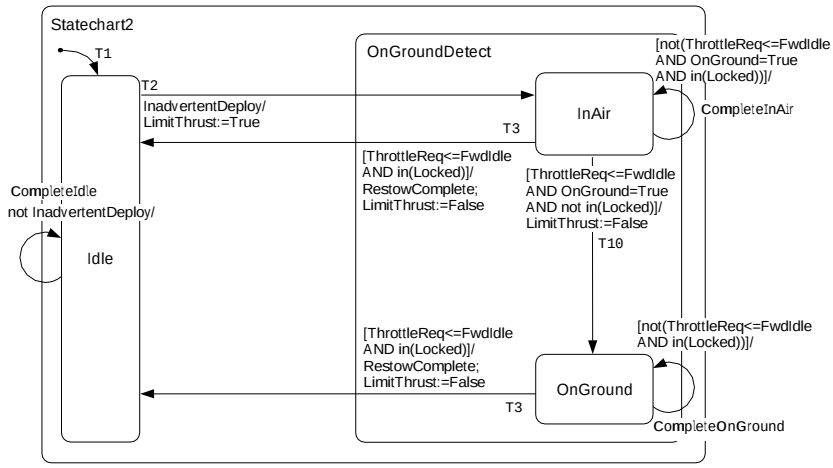


Figure 6.4: Second orthogonal component

The template for specifying the Statechart transitions in Z (first given in Section 4.3.2) is updated as follows. For operations in the first component:

*Component*1*Operation*

Δ*Parameters*
Δ*Status*1
*Status*2

SourceConfiguration $\subseteq$ *ActiveStates*1
EventExpression
GuardingCondition
*Constraints*2
ActionExpression
TargetConfiguration $\subseteq$ *ActiveStates*1$'$

where *Constraints*2 is a predicate over elements in *Status*2, whose value is not changed by the schema. A similar template is used for operations from the second component that reference elements in the status of the first. If internal variables are shared between components, they are declared in the *Status* corresponding to the component that writes to the variable. *Status*1 and *Status*2 are defined for this example as follows:

$$States1 == \{Statechart1, Idle, LockDetect, Unstowed, AwaitLock, Wait, Lock\}$$

$$States2 == \{Statechart2, Idle, OnGroundDetect, InAir, OnGround\}$$

*Status*1

*ActiveStates*1 : $\mathbb{F}_1$ *States*1
Timer : $\mathbb{Z}$

---

*configuration*1(*ActiveStates*1)

*Status*2

*ActiveStates*2 : $\mathbb{F}_1$ *States*2

---

*configuration*2(*ActiveStates*2)

Based on these templates, transition *T10* would be specified as follows, where the constraint over *Status1* is highlighted:

Abstract state	**Equivalence class of Status1**
1	$\{Statechart1, Idle\} \subseteq ActiveStates1$
2	$\{Statechart1, LockDetect, Unstowed\} \subseteq ActiveStates1$
3	$\{Statechart1, LockDetect, AwaitLock, Wait\} \subseteq ActiveStates1 \land Timer = 0$
4	$\{Statechart1, LockDetect, AwaitLock, Wait\} \subseteq ActiveStates1 \land Timer < 9 \land Timer \neq 0$
5	$\{Statechart1, LockDetect, AwaitLock, Wait\} \subseteq ActiveStates1 \land Timer = 9$
6	$\{Statechart1, LockDetect, AwaitLock, Wait\} \subseteq ActiveStates1 \land Timer = 10$
7	$\{Statechart1, LockDetect, AwaitLock, Wait\} \subseteq ActiveStates1 \land Timer = 11$
8	$\{Statechart1, LockDetect, AwaitLock, Wait\} \subseteq ActiveStates1 \land Timer > 11$
9	$\{Statechart1, LockDetect, AwaitLock, Locked\} \subseteq ActiveStates1$

Table 6.3: Summary of abstract states for first orthogonal component

$T10$

$\Delta Parameters$
$Status1$
$\Delta Status2$

$\{Statechart2, OnGroundDetect, InAir\} \subseteq ActiveStates2$
$ThrottleReq? \leq FwdIdle \land$
$OnGround? = True \land$
$\boxed{\neg(Locked \in ActiveStates1)}$
$LimitThrust! = False$
$RestowComplete! = Undefined$
$\{Statechart2, OnGroundDetect, OnGround\} \subseteq ActiveStates2$

The abstract states for the AFSMs are constructed from the pre and post-condition constraints of $Status1$ and $Status2$ using the same method as described in Section 6.2. The predicates for these equivalence classes are extracted from the pre and post-conditions of operations from both components to include references in the constraint predicates. Assuming that boundary value analysis has been applied to partition the Z specification (see Appendix A.2), the abstract states based on the minimal partition of the equivalence classes of *Status1* and *Status2* are summarised in Tables 6.3 and 6.4 respectively.

The AFSMs for the example components are shown in Figures 6.5 and 6.6 respectively. Abstract state 8 of *Statechart1* was found to be unreachable and therefore removed. The Z specification of the abstract states for *ThrustLimitation* is given in Appendix A.3.

Abstract state	**Equivalence class of Status2**
1	$\{Statechart2, Idle\} \subseteq ActiveStates2$
2	$\{Statechart2, OnGroundDetect, InAir\} \subseteq ActiveStates2$
3	$\{Statechart2, OnGroundDetect, OnGround\} \subseteq ActiveStates2$

Table 6.4: Summary of abstract states for second orthogonal component

Where a transition depends on a constraint over the current state of an orthogonal component it is labeled accordingly using the notation $(S = n)$ where S is the AFSM which the constraint references and n is the state of that AFSM which must or must not be active in order for the transition to be enabled. The definitions of δ and λ for each concurrent AFSM i are updated to incorporate the constraints as follows:

- $\delta_i = X \times S_i \times C_1 \times ... \times C_n \rightarrow S_i$
- $\lambda_i = X \times S_i \times C_1 \times ... \times C_n \rightarrow Y_i$

for n orthogonal components where C_j represents the set of abstract states for each AFSM j ($i \neq j$) that are referenced in the pre-condition of the transition. Each transition in the diagrams may be triggered by a number of abstract input events. Therefore the actual number of transitions in the AFSM, each triggered by a single abstract input, are far more than shown in the diagrams (819 in total for this example).

Non-determinism

The abstraction described at the start of this section (page 111) can lead to non-determinism in the abstract finite state machine, even in cases where the original specification is deterministic. This can cause test sequences to be generated that satisfy the abstract sequence constraints but are not valid sequences through the original specification. An example of such non-determinism is shown in Figure 6.5. Transition *T11* from state 4 can either lead to state 5 or back to state 4. There are several possible solutions to this problem each of which has its own advantages and disadvantages.

Generate and test: The non-determinism can be accepted and on generating each sequence, the original specification can be used as an oracle to verify the feasibility

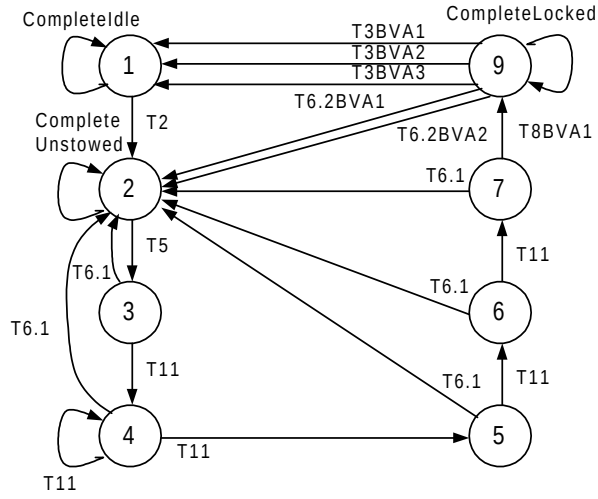


Figure 6.5: AFSM for Statechart1

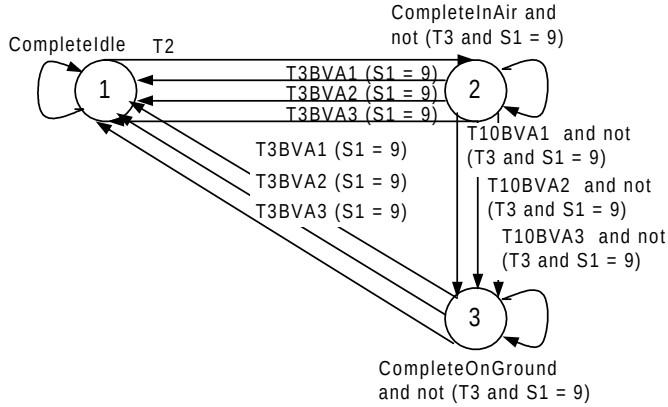


Figure 6.6: AFSM for Statechart2

of the sequence. A number of sequences are generated until a valid one is found. This method has the advantage of preserving the simple and generic method of generating the abstract finite state machine but is less efficient (a potentially large number of sequences must be generated before a valid sequence is found). The method also requires an automated oracle mechanism based on the original specification.

Resolve non-determinism in the AFSM: The non-determinism can be resolved in the abstract finite state machine before constraint solving is applied. This has the advantage that the constraint solvers will not produce infeasible sequences, but has the disadvantage of requiring specialised algorithms to resolve the non-determinism and may result in increasing the state-space of the AFSM, reducing the efficiency of the test sequence generation techniques.

Resolve non-determinism on the fly: The sequence generation algorithms (model checkers and specialised tools) could be modified to store the non-abstract (deterministic) form of the specification. Actual values of the state variables could be recorded as sequences are generated. If a non-deterministic choice occurs, the original specification is used as an oracle to resolve the non-determinism. The state space that is searched would remain that of the abstract specification, however extensive modifications to the tools would be required in order to store and reference the non-abstract specification.

Resolve non-determinism later: If the non-deterministic behaviour can be removed without impacting the feasibility of the test sequence generation algorithms, *partial sequences* can be generated which contain gaps where the non-deterministic behaviour would otherwise occur. A less abstract, deterministic representation of the system can then be used to fill in these gaps during constraint solving.

In practice, a combination of the above techniques could be used based on a trade-off between the efficiency of the constraint solving techniques and the need to develop specialised algorithms for resolving the non-determinism. It is always possible to resolve the non-determinism if the original specification is deterministic. However, in the worst case, resolving the non-determinism may lead to a specification of the same size and complexity as the original. The following section describes how “resolving the non-determinism in the

abstraction" and "resolving non-determinism later" techniques were applied to Statechart specifications.

Resolving non-determinism in Statechart abstractions

A common situation in which non-determinism is introduced into abstractions of Statechart specifications is in self-influencing loops. An example of this is caused by transition $T11$ in Statechart1 (Figure 6.3) and leads to the non-determinism on the transitions out of abstract state 4 in the AFSM for this component. The non-determinism in the abstraction results from the method of extracting the abstract states. The equivalence classes for the state component of operation $T11BVA1$ (where *Timer* is included as a state variable) are shown as abstract states 4 and 5 in Table 6.3. The current value of Timer is not always preserved, only its range. The non-determinism arises because operation $T11BVA1$ from abstract state 4 can lead to either state 4 or 5 depending on the current value of *Timer*. Because the value of *Timer* is not explicitly recorded in the AFSM, this leads to non-determinism. In the deterministic (non abstracted) version of the specification, the operation would be repeated a fixed number of times before exiting abstract state 4. This information can be used to resolve the non-determinism in the abstraction. Several approaches were applied to this problem and are described next.

Resolving non-determinism in the AFSM: Symbolic execution can be used to identify the recurrence relation and exit condition of the looping transition and therefore calculate the number of times it must be executed before leaving the state [CZ93]. This information can be used to re-introduce determinism in the AFSM. The state relation for operation $T11BVA$ is:

$$\{Statechart1, LockDetect, AwaitLock, Wait\} \subseteq ActiveStates1 \land$$
$$Timer' = Timer + 1 \land$$
$$\{Statechart1, LockDetect, AwaitLock, Wait\} \subseteq ActiveStates1'$$

which leads to the solution

$$\{Statechart1, LockDetect, AwaitLock, Wait\} \subseteq ActiveStates1_0 \land$$
$$Timer_k = Timer_0 + (1 * k) \land$$
$$\{Statechart1, LockDetect, AwaitLock, Wait\} \subseteq ActiveStates1_k$$

where k is the number of repetitions of the operation. The exit condition is

$$Timer_k = 9 \wedge$$
$$\{Statechart1, LockDetect, AwaitLock, Wait\} \subseteq ActiveStates1_k$$

(from the definition of abstract state 5). The value of $Timer_0$ is 1 (from the definition of abstract state 3.) Substituting these values into the solution for the recurrence relation gives (only values of *Timer* need be considered):

$9 = 1 + k$

$k = 8$

therefore the looping form of $T11BVA1$ must be repeated 8 times before abstract state 4 can be exited. This can be modelled in the AFSM by replacing abstract state 4 with 8 sub-states, each connected by the operation $T11BVA1$ with the final sub-state leading to abstract state 5 via the operation $T11BVA1$.

The practicality of this technique will be determined by the number of iterations of the looping operation and whether or not the recurrence relation can be solved statically. The time complexity of the test sequence generation algorithm discussed in Section 6.3 rises exponentially with the number of transitions in the AFSM. By introducing the additional states (and associated transitions), the time taken to generate the sequences was therefore greatly increased. For self-influencing loops that can be iterated more than just a very few times, this will lead to AFSMs for which the test sequences will take an infeasibly long time to calculate.

Resolving non-determinism later: Non-determinism can also be resolved once the test sequences have been generated. UIO sequences can be generated that do not fully execute the behaviour of the self-influencing transition. Model checking (see Section 6.4) is then used to fill in the gaps in the resulting sequences. This is achieved by removing the looping transition that causes the non-determinism in the AFSM. In the running example, the transition *T11* from abstract state 4 to itself would be removed. The transition is re-introduced to the model checking specification as well as any additional variables that are needed to resolve the non-determinism when generating sequences of concrete test data. This approach has the advantage that the efficiency of the test sequence generation techniques is not affected. Introducing additional variables to resolve the non–determinism in the model checking reduces the efficiency of the model checking. However, if the range of these variables can be kept suffi-

ciently small, the negative impact on the model checking appears to be less than if the determinism was resolved for the test sequence generation.

A further optimisation would be to omit the self-influencing transition altogether from the model checking specification and use the symbolic execution technique described above to calculate the missing parts of the sequences. This would ensure that additional variables did not need to be added to the model checking specification (thereby avoiding additional state explosion) but would require additional analysis of the sequences once generated. Further work is required to investigate this and other potential optimisations in further detail.

6.3 Generating state checking sequences

In order to verify that the system state is updated to the correct equivalence class following each operation, a checking sequence is required for each abstract state in the AFSMs. As the abstract states are constructed from the equivalence classes of the internal variables of the system, the ability of the abstract state checking sequences to verify the values of the internal variables will rely on whether the equivalence classes satisfy the uniformity hypothesis (see Chapter 5). The ability of the abstract state checking sequences to detect faults in the system state is therefore dependent on the fault detection capabilities of the testing heuristics used to select the equivalence classes.

Checking sequences can be generated using the Unique Input/Output (UIO) sequence derivation algorithm [SD88]. The algorithm described here differs slightly from that applied to traditional finite state machines in that it must be applied to systems of communicating FSMs (via references to each others' state). UIO sequences are generated for each AFSM separately. A sequence is said to be *output distinguishable* from another if the sequences produce different traces of outputs

The UIO algorithm is applied to each state s_i in turn and can be summarised as follows:

1. Set the length (l) of the current sequence to 1.

2. For all sequences of length l originating at state s_i check for output distinguishability against all sequences that can be triggered by the same trace of inputs and constraints

Parameter	Step 1	Step 2	Step 3	Step 4
InadvertentDeploy?	*Present*	*Present*	*Present*	*Present*
ThrottleReq?	$< FwdIdle - 1$	$< FwdIdle - 1$	$< FwdIdle - 1$	$< FwdIdle - 1$
UnstowedSensor?	*False*	*False*	*False*	*False*
OnGround?	*False*	*False*	*False*	*False*
LimitThrust!	Undefined	Undefined	Undefined	*False*
RestowComplete!	Undefined	Undefined	Undefined	*Present*

Table 6.5: UIO sequence for abstract state 5 from Statechart1

originating at all states s_j, where $s_j \neq s_i$. If the sequence is output distinguishable against all these sequences, it is the UIO sequence for state s_i.

3. If no output distinguishing sequence is found of length l, increase l by 1 and repeat steps 2-3.

The use of shared outputs (more than one component writes to the same output) complicates the calculation of output distinguishability as the output may have been produced by the transition under examination or an orthogonal component in the system. Such shared outputs could be removed from the calculation of output distinguishable transitions. However, this may result in no UIO sequences being generated for certain components that wrote to only shared outputs. Therefore, all outputs are used to generate the UIO sequences and a model checker used (see below) to generate a sequence of inputs that produces the UIO sequence while constraining all other machines to states and transitions that do not reference the shared output at the same point in the sequence.

The UIO sequence that would be generated based on this algorithm for abstract state 5 from Statechart1 (Figure 6.5) is shown in Table 6.5 and the UIO sequence for abstract state 2 from Statechart2 (Figure 6.6) is shown in Table 6.6. The UIO sequences for all states in the *ThrustLimitation* statechart are given in Appendix A.4.

The UIO sequences were generated using a tool called FSM which reads an XML [Con01] specification of the abstract finite state machine and uses a breadth first search algorithm to generate UIO sequences for each state. As the number of possible transitions in the AFSMs may be extremely large based on the minimal partition of the input equivalence classes, the minimal partitions of the transitions are also generated by the tool. The

Parameter/Constraint	Step 1
InadvertentDeploy?	*Present*
ThrottleReq?	$< FwdIdle - 1$
UnstowedSensor?	*True*
OnGround?	*False*
LimitThrust!	*False*
RestowComplete!	Undefined
*Status*1	$\neg(Locked \in ActiveStates1)$

Table 6.6: UIO sequence for abstract state 2 from Statechart2

input specification includes the minimal partition of the equivalence classes of each input (and output) and for each transition, and which of these equivalence classes are needed for the transition to fire. The tool then generates the partitioned transitions based on the possible configurations of the unspecified inputs that can cause the transition to fire.

UIO sequences can only guarantee the detection of errors in the δ and λ functions of the AFSM. Additional states may not necessarily be detected. Future work should therefore explore other forms of state checking sequences such as the W-method [Cho78] to detect additional abstract states. The choice of which method to apply will also depend on a better understanding of how faults in the implementation state reveal themselves as either mutations of the δ function or whether they also result in additional abstract states.

6.4 Property-based test sequence generation

Model checking is a technique for verifying properties of state-based systems. A computation tree of a model of the system is enumerated and checked against a specification of the properties to be verified. If a contradiction between the specification and the model is found, a counter-example showing a sequence that leads to a state that does not satisfy the specification is generated. The ability to generate counter-examples makes model-checking a convenient means of generating test sequences from finite state machines. The finite state machine is defined as the model and the negation of the path constraint for the desired test sequence is defined as a liveness property to be checked against the model [CSE96]. The model checker then attempts to find a counter-example to this property, which due to double negation results in a test sequence satisfying the property.

Model checking is a completely automated technique (once the input to the model checker has been produced) and is therefore a suitable technology for the automated testing framework that is the target of this thesis. Model checking also has the advantage that tests according to different criteria can be generated by simply varying the property to be checked against the model. This results in a similar level of flexibility as the pattern-based operation testing techniques described in Chapter 5.

The model checker SMV [Ber00] was used to explore the possiblities of model checking-based test sequence generation. The system of AFSMs is specified as a SMV module. A module consists of a number of variables which are used to represent the inputs, outputs and state variables of the AFSMs. The δ and λ functions are implemented as pairs of case statements over pairings of S and X. Each case statement defines the next value of S and Y respectively. Any outputs which are referenced by more than one component are defined for each component that uses them as separate variables in the model. Invariants over these outputs to ensure that they do not take different defined values are specified in the `INVAR` section. The SMV representation of the *ThrustLimitation* example is given in Appendix A.5. The template for SMV models corresponding to a system of concurrent AFSMs is as follows:

```
MODULE main
-- Declare the variables that make up the model
VAR
-- The system has m abstract inputs each with n equivalence classes
  X1:  {X1.1, ..  ,X1.n}
  ...
  Xm:  {Xm.1, ..  ,Xm.n}
-- The system has m abstract outputs each with n equivalence classes
  Y1:  {Y1.1, ..  ,Y1.n}
  ...
  Ym:  {Ym.1, ..  ,Ym.n}
-- The system has m AFSMs each with n abstract states,
-- one of which will be labeled as the initial state Si.0
  S1:  {S1.1, ..  ,S1.n}
  ...
  Sm:  {Sm.1, ..  ,Sm.n}
-- Initialise any relevant variables
ASSIGN
  init(S1) := S1.0;
  ...
  init(Sm) := Sm.0;
```

```
-- Define the delta and lambda functions
-- For each AFSM k
--enumerate the delta function for all(X1.i,..., Xm.i, S1.i,...,Sm.i, Sk.j)
  next(Sk) := case
    S1 = S1.i &...& Sm = Sm.i & X1 = Xm.i & ...  & Xm = Xm.i:  Sk.j;
    ...
  esac;
-- For each output Yk referenced by AFMS k
--enumerates the lambda function for all(X1.i,..., Xm.i, S1.i,...,Sm.i, Yk.i)
  next(Yk) := case
    S1 = S1.i &...& Sm = Sm.i & X1 = Xm.i & ...  & Xm = Xm.i:  Yk.i;
    ...
  esac;
-- Invariant section
-- used to ensure that shared outputs are written to consistently
INVAR
-- For each output variable set Yi..Yj corresponding to a shared output
((!(Yi = Undefined) & ..  & !(Yi = Undefined))-> (Yi = ...  = Yj))
```

The specification property can place constraints on the abstract states, inputs and outputs. Therefore, sequences can be generated that can be described as constraints on any of these elements. For example, to generate a sequence that terminates at some state *i* for initialisation purposes, the following template is used (stating that at some point in the future the state *S* can take the value S_i)[1]:

```
SPEC
  !EF(S = Si)
```

A similar template can be used to generate a sequence that leads to a particular output:

```
SPEC
  !EF(Y = Yi)
```

Sequences of abstract states, inputs and outputs can also be generated from a partial specification of the sequence. This is useful, for example, when generating test sequences to satisfy high level use cases defined in terms of configurations of the system that can be mapped onto equivalence classes in the formal representation. Such an ordering of the outputs would be defined using the following template that will generate a sequence producing outputs p, q and r in that order but possibly interleaved by other values:

[1]Note: the negation ! is required to ensure the sequence is generated as a counter-example to the given property.

```
SPEC
  !EF(Y = Yp & EF(Y = Yq & EF(Y = Yr)))
```

Model checking can be used to generate the initialisation sequences needed to bring the system into a particular state for testing a transition. After the transition has been executed, the checking sequence generated using the UIO method can be applied. For example in order to test transition $T10$ AFSM2 must be in state 2 and AFSM 1 must not be in state 9. The initialisation sequence can be generated by finding the counter-example to the following property.

```
SPEC
  !EF(Statechart2 = 2 and !( Statechart1 = 9))
```

The feasibility of the UIO sequences can also be checked by combining the specification for the initialisation sequence (as above), the transition to be tested and the UIO sequence for the after state of the transition based on the following pattern.

```
SPEC
  !EF( (Initial state of transition to be tested and any constraints ) &
  EX((Pre-condition of transition to be tested ) &
  EX(Specification of UIO sequence)))
```

If this pattern was repeated for each transition, the resulting sequences could be used to test the implementation for faults in the δ and λ operations based on the fault and uniformity hypotheses used in the original partitioning of the system.

The generation of the SMV input was automated using the same tool (FSM) that was used to generate the UIO sequences. However, for the SMV model, it was not necessary to perform the additional partitioning step based on the loose specification of the transition guard conditions. The SMV model for *ThrustLimitation* shown in Appendix A.5 was generated using this tool.

6.5 Case studies

The test sequence generation techniques were applied to a number of Statecharts taken from the Thrust Reverser demonstrator project (introduced in Chapter 4). The results of the UIO sequence generation are summarised in Table 6.7. For many states, no UIO sequences were

Statechart	**States (Basic)**	**Transitions**	**Shortest UIO**	**Longest UIO**	**No. of UIOs**
Original Statecharts					
ThrustLimitation	7	13	1	$> 6^*$	7
AbstractControls	9	30	-	-	0
EngineOutUnavailable	3	5	1	1	2
EngineOutLanding	4	7	-	-	0
InadvertentRestow	3	10	1	1	2
InadvertentDeploy	4	12	4	$> 2^*$	0
DetectJam	4	12	1	$> 3^*$	3
DetectThrustLimitation	4	12	1	$> 2^*$	4
MaintenanceMode	2	5	1	1	2
Testable Statecharts					
EngineOutUnavailable'	3	5	1	1	3
EngineOutLanding'	3	7	1	$> 5^*$	3
InadvertentRestow'	3	10	1	1	3
InadvertentDeploy	4	12	1	$> 1^*$	3
DetectJam	4	12	1	$> 3^*$	4
*The length of the final test sequence was dependent on the length of timers. Actual length = length given + timer value					

Table 6.7: Summary of UIO Sequence results

found (the last column indicates the number of *abstract* states for which an UIO sequence was found).

The testability (based on the ability to generate checking sequences) of the Statecharts was found to be highly dependent on the distinguishability of the transitions and the distribution of distinguishable transitions within the Statechart. Many of the Statecharts in the case studies contained only a small proportion of "distinguishable" transitions. These transitions could be identified as they often appeared as the last element of a UIO sequence. For example, in testing the AFSM for the first orthogonal component of the *ThrustLimitation* Statechart (see Figure 6.5), 7 of the 8 UIO sequences terminated in transition *T3*. The lack of distinguishability in Statechart transitions is a consequence of the ability to loosely specify transition guards. If no constraining predicate is given for an input parameter, any value may trigger the transition. For example, transitions *T5* and *CompleteIdle* from Figure 6.5 may have distinct transition labels but may be triggered by the same set of inputs, i.e any combination of the inputs in which $UnstowedSensor = False$ and the event *InadvertentDeploy* is not present. As neither transition defines an output, they cannot be distinguished from each other through inspection of the input and output parameters during a single step.

The bottom rows of Table 6.7 correspond to versions of the Statecharts that were rewritten to avoid these problems while minimising the overall changes to the functionality of the design. The original Statecharts were not designed with testability in mind, they were constructed as part of a modelling exercise by another member of the research group. The problems encountered when generating test sequences were therefore extremely useful in investigating properties that affect the testability of Statechart specifications.

The changes to the Statecharts to make them more testable can be classified as follows. Firstly, the distance between distinguishable transitions was reduced, thereby reducing the maximum length of the state checking sequences. This can result in increasing the length of the shortest sequences but due to the exponential growth in calculation time based on the length of the sequences, the net affect was positive. In order to retain a similar functionality, these changes were typically restricted to shifting the definition of outputs by only one step. For example, a defining output was set while entering a state instead of on leaving the state. The second type of change concerned reducing the domain-to-range ratio by adding

outputs or increasing the number of equivalence classes of the outputs. This approach, in the extreme case, would be equivalent of adding status messages to the outputs (see Section 2.3.1). However, more generally, instead of leaving outputs undefined in many transition actions, an additional value was added to the output type that could be used to specify that a particular set of conditions or state had been met, or to signify certain way-points in the Statechart (e.g. in the running example Statechart, the output *LimitThrust*! could be changed to produce the values *True*, *False* and *AwaitingLock*).

For some specifications, the generation of the UIO sequences was computationally expensive, despite the input space abstraction in the finite state machine. The need for determinism in the AFSM from which the UIO sequences were generated limited the amount of abstraction that could be applied to the system state. For example, a deterministic abstraction of the *Wait* state from Figure 6.3 would require an abstract state for each value of *Timer*. The time to compute UIO sequences rises exponentially with their length and therefore such behaviour drastically increased the amount of time and memory required for the sequence generation. The alternative, 'resolve non-determinism later' approach described in Section 6.2.1 was proven to be the most efficient approach in these situations. A breadth first search algorithm was used to explore the state space of the possible UIO sequences and was found to decrease the average time required for the computation compared to a depth first search. However, this approach was found to be extremely memory consuming as all potential UIO subsequences of length $l-1$ from a particular state need to be kept in memory when checking for sequences of length l.

Table 6.8 summarises the results of applying the UIO test sequences to the *ThrustLimitation* example compared against simple operation-based tests with no state checking and operation-based tests generated based on boundary value and disjunction analysis. The UIO sequences show some improvement over partitioned test cases run as a transition covering sequence but the results also imply that many state transfer errors were detected as a side effect of transition covering sequences and that the equivalent classes used to construct the AFSM were insufficient to detect all faults. The undetected faults were found to be due to the fact that the completing transitions were only tested superficially (i.e. no testing heuristics were applied to generate the test data). This demonstrates the need for more rigorous negative testing and an argument for this technique based on a formal analysis of different

Test set	Mutations killed (out of 36)	Mutation score
Unpartitioned (transition covering sequence)	25	69.44%
Partitioned (transition covering sequence)	31	86.11%
UIO sequences	32	88.88%

Table 6.8: Mutation testing comparison of UIO sequences against operation testing

fault detection conditions is presented in the next chapter.

6.6 Summary

This chapter addressed the problem of generating sequences of test cases from Z specifications. Test sequences are required when parts of the system state are not directly observable or controllable at the testing interface. A generic approach was presented based on Z specifications which involved generating an Abstract Finite State Machine (AFSM) where the abstract states correspond to disjoint equivalent classes of those variables in the system which are hidden from the testing interface. These equivalence classes can be extracted directly from the original specification or based on test cases generated using the techniques from Chapter 5 for a greater level of fault detection in the system state.

When applied to the Z specification of Statecharts, the generic approach to constructing the AFSM suffers from over abstraction, state explosion and non-determinism which reduces the efficiency of the test sequence generation techniques. Optimisations to reduce the impact of these problems were presented based on the fixed structure of Z specifications generated from Statecharts and the restriction on communication between parallel components in the Statecharts to a single writer/multiple reader relationship.

Two approaches to generating test sequences from the AFSMs were explored. The Unique Input/Output sequence algorithm was used to generate sequences to determine whether the system has reached a particular abstract state. Model checking was used to generate test sequences based on a specification of the certain sequential properties of the tests. This technique can be used to generate not only initialisation sequences but also sequences based on, for example, high level use cases that describe a (possibly incomplete) sequence of operations for which suitable test data must be found.

The difficulties in forming the abstraction and generating the test sequences (in par-

ticular state checking sequences) highlight a significant feature of Statecharts. That is, the ability to express complex behaviour in concise diagrams. This is often seen as an advantage of the notation but it can also lead to untestable specifications. In particular, interactions between orthogonal components and a high domain-to-range ratio of the input/output relation result in long test sequences or an input space that is so large that even short test sequences may take an impractical amount of time to generate. The case studies demonstrated these problems and in many cases no UIO sequences were found. For most of the Statecharts however, simple changes could be made which did not adversely affect the functionality but led to more testable Statecharts.

Chapter 7

Test adequacy

Chapter 5 described how testing heuristics based on partitioning and fault-based criteria could be formally specified and automatically applied to specifications. This chapter describes how testing heuristics can be evaluated for their fault-detection capabilities and how they can be designed to maximise the number of faults detected for the fewest possible test cases. A mutation-based analysis of test adequacy is described and discussed within the context of weak and strong fault detectability.

7.1 Introduction

The quality of a test set can be described in terms of its effectiveness and efficiency. The effectiveness of any one test is its ability to detect a number of faults. The efficiency of a test set is the number of tests required to obtain a pre-determined fault coverage. Maximising the effectiveness and efficiency of a test set will lead to reduced testing costs. This is true even in cases where the test cases are automatically generated, as each abstract test case generated from the specification will typically need to be refined into concrete test data so that it can be run against the implementation. Each test case will also be subject to configuration management and may consume costly resources on test rigs.

The formalised approach to automated test case generation presented in Chapter 5 encourages rigorous test case design and evaluation. By formalising both testing heuristics and fault-detection conditions, the relationship between them can be formally analysed. This chapter examines whether the formalisation of fault-detection conditions can be used

to design efficient and effective test heuristics.

A method for comparing the effectiveness of testing heuristics is described in Section 7.2. This then forms the basis for increasing the effectiveness and efficiency of test cases based on weak and strong mutation. These techniques are described in Sections 7.3 and 7.4 respectively.

7.2 Comparing testing heuristics

The fault-detection capability of a testing heuristic can be defined as the probability that data generated according to the heuristic will detect a particular hypothesised fault. To demonstrate that a partitioning heuristic $P \Leftrightarrow P_1 \lor ... \lor P_n$ can detect a hypothesised fault represented as the mutation P_m of P, the following conjecture must be proven:

$$\vdash ? \forall Vars \bullet (P_1 \Rightarrow \neg(P \Leftrightarrow P_m)) \lor ... \lor (P_n \Rightarrow \neg(P \Leftrightarrow P_m))$$

where *Vars* represents all variables referenced by P and P_m. If the above conjecture is proven, the probability of the fault being detected is 1. Otherwise, the probability of the fault being detected is calculated by the ratio of the size of input space that satisfies the conjecture against the total size of the input space of the specification. The weak or strong detectability of the fault is, as before, determined by the scope of P and P_m in relation to the testable inputs and outputs of the system.

A similar analysis can be used to demonstrate whether a fault-based heuristic designed according to a particular mutation (P_{m1}) is also sufficient to detect another mutation (P_{m2}).

$$\vdash ? \forall Vars \bullet \neg(P \Leftrightarrow P_{m1}) \Rightarrow \neg(P \Leftrightarrow P_{m2})$$

This analysis can be used to construct an ordering of the fault detection conditions for all possible mutations of a particular expression in the specification. In the above analysis, where the condition can not be proven either true or false, the simplification of the conjecture represents the set of additional constraints on the input space required to ensure that both faults are detected.

7.3 Efficient weak detection conditions

The above definitions can be used to determine the set of tests required to weakly detect all possible mutations of different expression types. Designing heuristics to detect more than

one fault will lead to more efficient and effective test sets. Due to the limited subset of Z used to represent the specifications, the number of expression types is limited, restricted to a subset of the logical, relational and arithmetic operators. Mutation analysis is now applied to an operator from each of these classes and testing heuristics are designed to detect all hypothesised mutations.

The specification mutations define abstractions of possible faults in the implementation. The class of mutations detected in the specification may not directly map to the same class of mutations of the program source code and therefore the mutation adequacy at the specification level may not correspond to the mutation adequacy at the code level. However, it is anticipated that a high (and potentially perfect) mutation score at the specification level would correspond to a high mutation score at the code level. The case study described in Section 7.3.4 examines this relationship in more detail.

7.3.1 Logical expressions

The logical operators used in the subset are as follows: $\neg$, $\wedge$ and $\vee$. In order to assess the relative detectability of the mutations, a mutation set and the corresponding fault detection conditions for each operator must first be identified. Only syntactically correct mutants and mutants that conform to the subset are considered. Mutants of the disjunction operator along with the corresponding fault detection condition and simplified fault detection condition are given in Table 7.1[1]. For each mutant, x, y and z may correspond to either Boolean typed variables or expressions.

Considering first the mutations that do not require the introduction of the third operand z, (variable negation fault, expression negation fault, operator reference fault, missing binary operations), it can be seen from Table 7.1 that the following two cases cover each of these faults:

- $x \wedge \neg y$
- $\neg x \wedge y$

This is demonstrated by using the definition from Section 7.2 for comparing two fault

[1]More mutations can be hypothesised than are given in the table. For the purposes of the study however, the mutations shown are assumed to lead to representative results.

Mutant of $x \vee y$	**Fault detection condition**	**Simplification**
Variable reference fault:		
DVR1: $x \vee z$	$\neg((x \vee y) \Leftrightarrow (x \vee z))$	$(\neg x \wedge y \wedge \neg z) \vee (\neg x \wedge \neg y \wedge z)$
DVR2: $z \vee y$	$\neg((x \vee y) \Leftrightarrow (z \vee y))$	$(x \wedge \neg y \wedge \neg z) \vee (\neg x \wedge \neg y \wedge z)$
Variable negation fault:		
DVN1: $\neg x \vee y$	$\neg((x \vee y) \Leftrightarrow (\neg x \vee y))$	$\neg y$
DVN2: $x \vee \neg y$	$\neg((x \vee y) \Leftrightarrow (x \vee \neg y))$	$\neg x$
Expression negation fault:		
DEN: $\neg(x \vee y)$	$\neg((x \vee y) \Leftrightarrow \neg(x \vee y))$	*true*
Operator reference fault:		
DOR: $x \wedge y$	$\neg((x \vee y) \Leftrightarrow (x \wedge y))$	$(x \wedge \neg y) \vee (\neg x \wedge y)$
Extra binary operators:		
DEB1: $(x \vee y) \vee z$	$\neg((x \vee y) \Leftrightarrow ((x \vee y) \vee z))$	$\neg x \wedge \neg y \wedge z$
DEB2: $(x \vee y) \wedge z$	$\neg((x \vee y) \Leftrightarrow ((x \vee y) \wedge z))$	$(x \wedge \neg z) \vee (y \wedge \neg z)$
Missing binary operators:		
DMB1: x	$\neg((x \vee y) \Leftrightarrow x)$	$\neg x \wedge y$
DMB2: y	$\neg((x \vee y) \Leftrightarrow y)$	$x \wedge \neg y$

Table 7.1: Necessary fault detection conditions for the disjunction operator

detection conditions. For example, the first missing binary operator fault detection condition (DMB1) will detect the variable negation fault (DVN2) if the following predicate reduces to true:

$$\begin{aligned}&\neg\big((x \vee y) \Leftrightarrow x\big) \Rightarrow \neg x\\ &\equiv (\neg x \wedge y) \Rightarrow \neg x\\ &\equiv \mathit{true}\end{aligned}$$

By modifying the conditions to include $\neg z$ as follows:

- $x \wedge \neg y \wedge \neg z$
- $\neg x \wedge y \wedge \neg z$

the variable reference faults and the second extra binary operator fault (DEB2) can also be detected. Therefore, with only 2 test cases, 9 out of the 10 mutants can be detected. The condition $\neg x \wedge \neg y \wedge z$ would ensure that the remaining fault (DEB1) was detected. This analysis demonstrates the need for negative testing. The third condition, $\neg x \wedge \neg y \wedge z$, requires that the original expression evaluates to false. Based on the methodology described in Chapter 5 and the suggested partitioning tactic for disjunctions that results in the test cases $x \wedge y$, $x \wedge \neg y$ and $\neg x \wedge y$ it can be seen that this case would not be generated. Testing based on a complete specification would allow this case to be generated. The conditions for negative tests are not necessarily covered by simply executing the negative parts of the specification (e.g. completing transitions). Therefore the cases may need to be explicitly generated based on additional testing heuristics. If the specification is not complete then the negative cases can not be adequately tested as the expected result of the test is undefined.

The analysis also shows that as long as the cases $x \wedge \neg y$ and $\neg x \wedge y$ are covered, the case $x \wedge y$ does not add to the fault detection properties for the mutants studied here. The disjunction partitioning heuristic can therefore be made more efficient by removing this case.

The efficient testing heuristics for the $\vee$ operator are as follows:

$$\mathit{EfficientDis}1 == \exists x, y, z : \mathit{Boolean} \bullet x \wedge \neg y \wedge \neg z$$

$$\mathit{EfficientDis}2 == \exists x, y, z : \mathit{Boolean} \bullet \neg x \wedge y \wedge \neg z$$

$$\mathit{EfficientDis}3\mathit{Neg} == \exists x, y, z : \mathit{Boolean} \bullet \neg x \wedge \neg y \wedge z$$

If the specification is complete, the third case can be covered by negative testing.

The feasibility of these three cases will depend on the different variables and expressions which z is able to represent. If z represents a Boolean variable and is the only other variable in the system (besides x and y), then these three cases are sufficient. Otherwise, cases will need to be constructed for each possible representation of z, given the other variables and all possible syntactically compatible expressions. The number of test cases required to cover faults requiring consideration of z will therefore rise exponentially with respect to the number of variables. Realistically, a heuristic could be applied to choose values for other variables in the system in order to increase the probability that z will take the correct value for a test case. Alternatively, if conditions over z were not included in the test case specifications and values were randomly chosen for other variables not referenced by x and y, the probability of the test case detecting the fault will depend on the probability of the arbitrarily chosen values causing z to evaluate to the correct value for all choices of z.

When deciding on an approach to handle mutants containing the expression or variable z, the probability of the faults that can be represented by the different choice of z should be taken into consideration. For example, based on the competent programmer hypothesis, only small deviations from a correct implementation need be considered. This may imply that it is sufficient to take into consideration cases where z is only ever a variable rather than an expression. This will significantly reduce the complexity of generating the test cases and will restrict the number of test cases generated. However, these assumptions may need to be reconsidered if the refinement to code becomes more complex. If the code is automatically generated it is feasible that the code generator could produce code that is significantly different, e.g. based on a mutation of the syntax tree corresponding to the code structure.

7.3.2 Relational expressions

The relational operators in the subset are $<$, $>$, $\leq$, $\geq$, $=$ and $\neq$. As before, a mutation analysis is performed based on syntically correct mutants that remain within the Z subset. Mutants of the $\leq$ operator along with the corresponding fault detection conditions and simplified fault detection conditions are shown in Table 7.2.

The operator reference faults can be detected by 2 conditions:

Mutant of $x \leq y$	**Fault detection condition**	**Simplification**
Variable reference fault:		
LTEVR1: $x \leq z$	$\neg(x \leq y \Leftrightarrow x \leq z)$	$(x \leq y \wedge x > z) \vee (x > y \wedge x \leq z)$
LTEVR2: $z \leq y$	$\neg(x \leq y \Leftrightarrow z \leq y)$	$(x \leq y \wedge z > y) \vee (x > y \wedge z \leq y)$
Operator reference fault:		
LTEOR1: $x < y$	$\neg(x \leq y \Leftrightarrow x < y)$	$x = y$
LTEOR2: $x > y$	$\neg(x \leq y \Leftrightarrow x > y)$	*true*
LTEOR3: $x \geq y$	$\neg(x \leq y \Leftrightarrow x \geq y)$	$x \neq y$
LTEOR4: $x = y$	$\neg(x \leq y \Leftrightarrow x = y)$	$x < y$
LTEOR5: $x \neq y$	$\neg(x \leq y \Leftrightarrow x \neq y)$	$x \geq y$

Table 7.2: Necessary fault detection conditions for the less than or equal to operator

- $x = y$
- $x < y$

The fault coverage of these conditions can be demonstrated by reducing the following predicates to true:

$$
\begin{aligned}
&(LTEOR1 \Rightarrow LTEOR2) \wedge (LTEOR1 \Rightarrow LTEOR5) \\
&\equiv (x = y \Rightarrow \mathit{true}) \wedge (x = y \Rightarrow x \geq y) \\
&\equiv \mathit{true} \wedge \mathit{true}
\end{aligned}
$$

$$
\begin{aligned}
&LTEOR4 \Rightarrow LTEOR3 \\
&\equiv x < y \Rightarrow x \neq y \\
&\equiv \mathit{true}
\end{aligned}
$$

That is, the simplified fault detection condition for LTEOR1 is sufficient to also detect LTEOR2 and LTEOR5 and the fault detection condition for LTEOR4 is also sufficient to detect LTEOR3.

The variable reference faults again require the intervention of the variable z. Testing heuristics for the variable reference fault LTEVR1 will now be considered by examining different values for z:

$z = y$: If z takes the same value as y the fault is undetectable. If z can only be a variable, this situation can be avoided by assigning values to y that are different to all other variables in the specification.

$z = y - k$**:** If k is a non-zero positive constant of any value, substituting $y - k$ for z into the simplified fault detection condition for LTEVR1 results in the following expression:

$$\begin{aligned}&(x \leq y \land x > y - k) \lor (x > y \land x \leq y - k)\\ &\equiv (x \leq y \land x > y - k) \lor \mathit{false}\\ &\equiv (x \leq y \land x > y - k)\end{aligned}$$

$x = y$ would satisfy this condition and therefore detect the fault in all cases where $z < y$.

$z = y + k$**:** If k is a non-zero positive constant, substituting $y + k$ for z in the simplified fault detection condition for LTEVR1 results in the following expression:

$$\begin{aligned}&(x \leq y \land x > y + k) \lor (x > y \land x \leq y + k)\\ &\equiv \mathit{false} \lor (x > y \land x \leq y + k)\\ &\equiv (x > y \land x \leq y + k)\end{aligned}$$

Any values for x and y such that $y < x \leq y + k$ are therefore sufficient to detect this fault. The value of k cannot be predicted but is at least the smallest increment in the type of x and y. Therefore, choosing the value $y + \delta$ for x where delta is the smallest element of the type will detect the fault for all values of k as $x \leq y + \delta \land x \leq y + k$ would be true for all possible values of k.

The fault detection conditions for LTEVR1 are as follows:

- $y \neq z$
- $x = y \land z < y$
- $x = y + \delta \land z > y$

A similar case by case analysis can be applied for the fault LTEVR2 and results in the same conditions.

The fault detection conditions can be covered by the following heuristics:

$$\mathit{EfficientLTE1} == \exists x, y, z : \mathbb{R} \bullet x = y \land z < y$$

$$\mathit{EfficientLTE2} == \exists x, y : \mathbb{R} \bullet x < y$$

$$\mathit{EfficientLTE3Neg} == \exists x, y, z : \mathbb{R} \bullet x = y + \delta \land z > y$$

Note that once again, negative testing is required to satisfy one of the conditions.

7.3.3 Arithmetic expressions

The arithmetic operators in the subset are $+$, $-$, $*$ and $\div$. Mutants for the $+$ operator are shown in table 7.3 along with the fault detection conditions and simplified fault detection conditions. Each of the fault detection conditions can be combined using conjunction to form the following efficient testing heuristic for the $+$ operator:

$$
\begin{array}{l}
\exists\, x, y, z : \mathbb{R} \bullet y \neq z \wedge \\
x \neq z \wedge \\
y \neq z \wedge \\
y \neq 0 \wedge \\
x \neq 0 \wedge \\
x \neq (x * y) - y \wedge \\
x \neq (x \div y) - y \wedge \\
z \neq 0 \wedge \\
z \neq 1 \wedge \\
z \neq (x + y) * (x + y) \wedge \\
z \neq y * y \wedge \\
z \neq x * x \wedge \\
x + y \neq 0 \wedge
\end{array}
$$

It can be shown that this heuristic implies each of the fault detection conditions given in the table and is satisfiable by at least some values of x, y and z. Efficient testing heuristics for each operator in the subset can be found in Appendix B.

7.3.4 Case study

The use of efficient fault-based testing heuristics was evaluated on the example reactive specification from page 76, the single-schema representation of which is shown in Figure 7.1. The efficient test heuristics described in the previous sections were applied to each sub-expression in the specification, where a sub-expression is identified by the use of an operator symbol. Test data for each of the resulting test cases was then generated using the LINGO [Inc01] non-linear constraint solver. These test cases were applied to 49 mutations of a correct implementation of the specification written in Ada. The mutations were checked for equivalence by hand and no equivalent mutants were found. Tests were generated where z values in the constraints were first ignored and then instantiated with the different variables, resulting in 46 and 109 test cases respectively. Values of z denoting expressions rather than simple variable references were not tested. The same number of tests were also run

Mutant of $x+y$	**Fault detection condition**	**Simplification**
Variable reference faults:		
ADDVR1: $x+z$	$x+y \neq x+z$	$y \neq z$
ADDVR2: $z+y$	$x+y \neq z+y$	$x \neq z$
Variable negation faults:		
ADDVN1: $x+(-y)$	$x+y \neq x+(-y)$	$y \neq 0$
ADDVN2: $-x+y$	$x+y \neq -x+y$	$x \neq 0$
Expression negation faults:		
ADDEN: $-(x+y)$	$x+y \neq -(x+y)$	$x+y \neq 0$
Operator reference faults:		
ADDOR1: $x-y$	$x+y \neq x-y$	$y \neq 0$
ADDOR2: $x*y$	$x+y \neq x*y$	$x \neq (x*y)-y$
ADDOR3: $x \div y$	$x+y \neq x \div y$	$x \neq (x \div y)-y$
Extra operator faults:		
ADDEO1: $x+y+z$	$x+y \neq x+y+z$	$z \neq 0$
ADDEO2: $x+y-z$	$x+y \neq x+y-z$	$z \neq 0$
ADDEO3: $(x+y)*z$	$x+y \neq (x+y)*z$	$z \neq 1 \wedge (x+y \neq 0)$
ADDEO4: $x+(y*z)$	$x+y \neq x+(y*z)$	$z \neq 1 \wedge y \neq 0$
ADDEO5: $(z*x)+y$	$x+y \neq (z*x)+y$	$z \neq 1 \wedge x \neq 0$
ADDEO6: $(x+y) \div z$	$x+y \neq (x+y) \div z$	$z \neq 1 \wedge (x+y \neq 0)$
ADDEO7: $z \div (x+y)$	$x+y \neq z \div (x+y)$	$z \neq (x+y)^2$
ADDEO8: $x+(y \div z)$	$x+y \neq x+(y \div z)$	$z \neq 1 \wedge y \neq 0$
ADDEO9: $x+(z \div y)$	$x+y \neq x+(z \div y)$	$z \neq y^2$
ADDEO10: $(x \div z)+y$	$x+y \neq (x \div z)+y$	$z \neq 1 \wedge x \neq 0$
ADDEO11: $(z \div x)+y$	$x+y \neq (z \div x)+y$	$z \neq x^2$
Missing operator faults:		
ADDMO1: x	$x+y \neq x$	$y \neq 0$
ADDMO2: y	$x+y \neq y$	$x \neq 0$

Table 7.3: Necessary fault detection conditions for the add operator

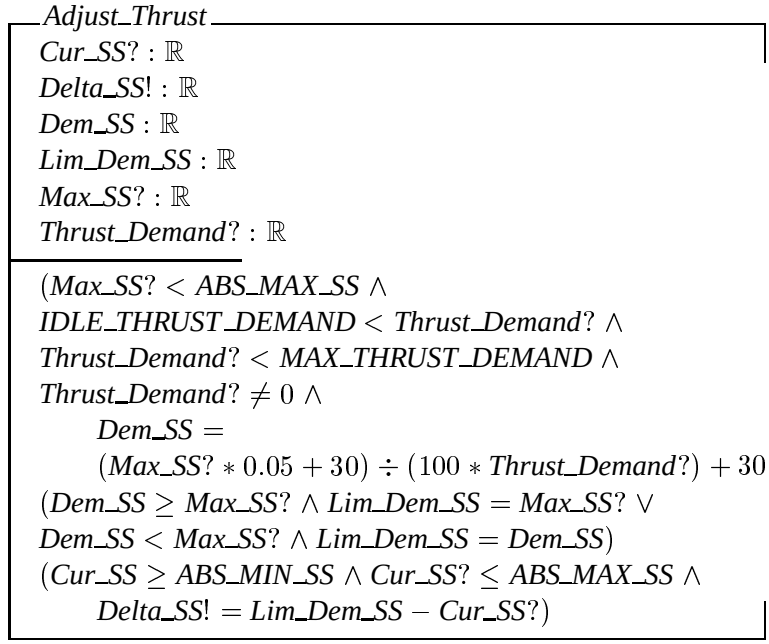


Figure 7.1: Z schema for Adjust_Thrust example

with randomly generated values and outputs calculated for these values using an oracle implementation. The results of the tests are summarised in Table 7.4.

Test set 1 (weak mutation ignoring *z* constraints) achieved a reasonably high mutation score. Undetected mutants fell into two categories, mutants that survived through fault masking in other areas of the code and mutants that survived through inadequate coverage of *z* values for constant replacement. The first category highlights a weakness of weak detectability. The empirical evidence shows that this method of testing is still sufficient to detect a large number of faults (which were equally distributed through the program and therefore equally likely to be masked). The second category again highlights a predicted weakness of the test set, as the tests were not targeted towards constant or variable replacement. However, these results also show that the tests were adequate at detecting a large proportion of faults of this type as an almost equal number of operand and operator replacement (22 and 27 respectively) mutations were applied. It can therefore be concluded, from the limited empirical evidence, that efficient weak mutation based on operator replacement fault hypotheses is sufficient to also detect a large proportion of other faults. Further work

Test set 1: Efficient weak mutation (ignoring z constraints)	
Number of tests	46
Number of mutants	49
Mutants killed	43
Total mutation score	87.75 %
Mutation score per test case	1.91 %
Mutants survived through fault masking	4
Mutants survived through constant replacement	2
Test set 2: Same number of random tests	
Mutants killed	32
Total mutation score	65.3 %
Mutation score per test case	1.42 %
Test set 3: Efficient weak mutation (with variable-based z constraints)	
Number of tests	109
Number of mutants	49
Mutants killed	44
Total mutation score	89.80 %
Mutation score per test case	0.82 %
Mutants survived through fault masking	4
Mutants survived through constant replacement	1
Test set 4: Same number of random tests	
Mutants killed	32
Total mutation score	65.3 %
Mutation score per test case	0.60 %

Table 7.4: Results of weak mutation-based tests

should include wider scale experimentation to confirm or deny these results. The results from test set 2 show that the first test set performs better than randomly chosen test data by a large margin (22.45 %) confirming the relationship between the specification-based fault hypotheses and actual test effectiveness.

Test set 3 (weak mutation with variable-based *z* constraints) achieved a slightly better mutation score than test set 1 (one extra mutant was detected). This confirms the impression that the first test set was adequate at indirectly detecting a large number of the variable replacement faults. Despite the slightly increased effectiveness of this test set over the first, its efficiency is far lower as more test cases were required. Interestingly, test set 4 detected exactly the same number of errors as test set 2. This implies that those errors undetected by random testing are those that require input data from a relatively small input domain that is not likely to be covered through random data selection. This effect is demonstrated by the constant replacement mutant that was undetected by all test sets. The mutant corresponded to the following expression:

$$\mathit{Thrust\_Demand} \neq 0$$

where the value 0 was replaced by another constant of arbitrary value. For any such value there are only two data points in the input domain that would lead to this mutation being weakly detected ($\mathit{Thrust\_Demand} = \mathit{Mutated\_Value}$ and $\mathit{Thrust\_Demand} = 0$). Random testing is therefore highly unlikely to select one of these values.

It was observed that the constraint solver often chose the same set of values for several test cases implying that an even greater level of efficiency can be achieved by combining test cases that can be satisfied by the same set of inputs. The repetition in the test data is most likely due to the search strategy used by the constraint solver. A randomised selection of data from the valid input domain for each test case may lead to a slightly higher test score due to the increased chance of errors being detected indirectly (i.e. errors that are not specifically targeted by the test case) due to a larger proportion of the input space being covered. However, the evidence from the random tests show that the remaining errors are unlikely to be detected by simply increasing the number of random tests.

The case study highlighted some interesting issues regarding testability. Some parts of the specification were found to be very testable, certain mutants were easily detected by random testing and weak mutation test cases that were not directly targeted towards their

detection. Other mutants were found to be extremely difficult to kill such as the $\neq$ example given above. The ease with which mutants can be detected is related to the proportion of the total input space that satisfies the fault detection condition for that mutant. The testability of a specification, based on arbitrary means of test data selection, can therefore be defined according to the size of the valid input space for the fault detection conditions for each possible mutation of the specification. However, as demonstrated by the example, some parts of the specification may be more testable than others. Examination of the testability of different expressions in the specification may allow manual review effort and other alternatives to testing to be more effectively focused.

7.4 Efficient strong detection conditions

The case study highlighted the shortcomings of weak detectability-based testing heuristics. For example, some mutants were not detected due to fault masking by other parts of the program. Testing based on sufficient fault detection conditions increases the probability that the test data will lead to a fault being observed at the outputs. This is made possible by including the complete specification in the fault-detection conditions and not just the sub-expression where the fault occurs. As a result, it is not possible to generically specify efficient testing heuristics to cover a number of faults, because the conditions will vary for each specification. A different approach to generating efficient and effective test data for strong detectability of faults is therefore required.

A method for generating sufficient test data for particular hypothesised faults was presented in Chapter 5 and can be summarised as follows. If a fault can be represented as the mutation P_m of specification P, the set of sufficient test data for detecting this fault is defined by the following expression:

$$\exists\, Inputs, Outputs \bullet \neg(P \Leftrightarrow P_m)$$

This expression can be extended to specify test data that is capable of strongly detecting several faults as follows:

$$\exists\, Inputs, Outputs \bullet \neg(P \Leftrightarrow P_{m1}) \wedge \ldots \wedge \neg(P \Leftrightarrow P_{m2})$$

A subsumption hierarchy may also exist for a certain set of faults, demonstrated by the satisfiability of the following expression:

$$\forall \mathit{Inputs}, \mathit{Outputs} \bullet \neg(P \Leftrightarrow P_{m1}) \Rightarrow \neg(P \Leftrightarrow P_{m2})$$

Unlike for necessary conditions, where the fault detection conditions can be generically specified, the subsumption hierarchy of sufficient conditions may differ for each specification under test and must therefore be calculated each time the tests are generated. The generation of efficient and effective sufficient test data can be couched as the following optimisation problem: Find the minimum number of test values that cover the sufficient fault detection conditions for all hypothesised faults. In all but the worst case, this will involve finding data that cover multiple sufficient conditions, implicitly exploiting the fault detection condition subsumption hierarchy for the specification under test. The predicate templates given above for subsumption and multiple condition satisfaction inspired the following two proposals for automatically generating efficient test data for sufficient fault detection conditions.

Statically determine a sufficient condition subsumption hierarchy: Initially, the sufficient conditions for all hypothesised faults are generated. Conditions are removed from this set if it can be shown that one condition is implied by another. This process is continued until no more conditions can be removed in this way. The final set of conditions form the roots of a set of fault detection subsumption hierarchies that cover all faults. In the worst case, no conditions will imply any other and therefore the number of test data generated will equal the number of hypothesised faults. In the best case, the reduction will result in a single condition that covers all faults. The ability to select the optimal set of conditions will depend on the strategy used to compare and remove the conditions based on the implication analysis. The success of this approach will depend on the availability of powerful general purpose theorem provers capable of carrying out the simplifications with the minimal amount of user input. The number of conditions that can be removed in this way will also depend greatly on the structure of the specification.

Optimal sufficient condition satisfaction: Alternatively, each sufficient condition could be defined as a separate optimisation goal and each goal evaluated in turn. For each iteration of the search for any particular goal, the satisfaction of all goals can be assessed. Any goals that are coincidentally satisfied by the current iteration can be

removed from the set of goals yet to be satisfied. Otherwise, the closest solution to each unsatisfied goal can be recorded and used as a starting point for the search for that goal. A similar method developed by Wegener et al. [WBS02] was shown to be successful in reducing the computation time required to satisfy multiple optimisation goals for the automated generation of test data to achieve structural program coverage.

These two approaches could be combined as follows. Implications where the subsumption expression resolves to true for *all* data could be identified and removed and an optimisation-based approach could then be used to satisfy the remaining implications that are true for *some* data only. These approaches to efficient strong mutation testing have not yet been implemented due to the requirement of specialised tool support, the development of which was considered outside the scope and resources of this thesis. An investigation into the feasibility of the approaches and a comparison between them should therefore be conducted as part of further work.

7.5 Summary

This chapter presented mechanisms for evaluating the relative potential fault coverage of different testing heuristics. These mechanisms were then used to define a set of efficient heuristics for a subset of Z based on analysis of the necessary fault detection conditions for a number of specification mutants. A case study demonstrated that these testing heuristics were effective at detecting not only the faults for which they were designed but also a large proportion of other faults. The case study showed that the efficient weak mutation test data also performed significantly better than the same number of random tests.

The case study highlighted the weakness of necessary fault-detection conditions in that some faults were not detected due to masking by other parts of the specification or implementation. The extension of the techniques to include sufficient fault detection conditions was discussed and several approaches to automating the generation of efficient test data were proposed. These approaches include the static determination of a fault subsumption hierarchy for each specification and the use of optimisation-based search techniques. The tool support required to evaluate these techniques was not available at the time of writing and therefore future work should include the development of prototype tool support for

these methods to enable a feasibility study to be performed.

Chapter 8

Evaluation

This chapter provides a critical evaluation of the work presented in the thesis. Each of the technical strands is evaluated with respect to its strengths and weaknesses and the supporting evidence for the assessments is given. The framework itself is then evaluated with respect to the success criteria outlined in Chapter 3.

8.1 Introduction

Chapters 3 to 7 presented a framework and associated techniques for the application of formal methods to automated verification and testing based on graphical specification notations. Each aspect of the framework has different strengths and weaknesses. In order to assess the overall success of the work and the satisfaction of the hypothesis presented in Chapter 1, a balanced evaluation of all parts of the framework needs to be made. The strength with which the case study evidence can support the evaluation also needs to be assessed. The evaluation described in this chapter provides the foundation for the final assessment of the satisfaction or otherwise of the hypothesis which is presented in Chapter 9.

This chapter is structured as follows: in Section 8.2, the technical aspects of the framework are evaluated for their strengths, weaknesses and advancement over previous work in the area. The techniques are evaluated in the same order as they were presented in the preceding chapters, i.e. formalisation of graphical and tabular notations, operation-based testing, test sequence generation and test adequacy evaluation. The strength of the evidence

provided by the case studies is assessed in Section 8.3 and the role of automated tool support is examined in Section 8.4. The overall success of the framework is then examined in Section 8.5. In particular, the success criteria presented in Chapter 3 are revisited and the strengths and weaknesses of the framework (i.e. the integration of the individual techniques) are assessed. The results of the evaluation are summarised in Section 8.6 and the work is assessed against an alternative set of success criteria.

8.2 Evaluation of techniques

8.2.1 Formalisation of graphical specifications

The representation of graphical and tabular specifications using a common intermediate formal notation is a key part of the framework. The formalisation provides the link between practical engineering notations and notation independent, automated V&V techniques. Without the formalisation, engineers would need to manually write the formal specifications – a practice that is typically considered too inefficient for commercial projects. Alternatively, graphical notations could be used, but either manual approaches to V&V would be required or the integrity of the automation could not be argued as rigorously.

Formalisations were described for two graphical notations, STATEMATE Statecharts and PFS tabular specifications. These formalisations were made possible by the existence of a semantic definition for each of the source notations. The applicability of the techniques is therefore restricted to graphical notations whose semantics are well defined. The strengths of the formalisation step can be summarised as follows:

Making all behaviour explicit: The formalisation of the graphical and tabular notations made behaviour explicit that may otherwise have been overlooked in less formal V&V activities. Examples of such behaviour in Statecharts are the completeness assumption and priority resolution of conflicting transitions. The formalisation of the transition guards enabled this behaviour to be identified and the necessary tests to be automatically generated.

Formal validation of specification properties: The formalisation also allowed proof conjectures to be constructed for particular properties of the specification. Based on the

common structure of these conjectures and the restricted subset of Z used to specify them, generic automated proof tactics were produced to discharge (and if necessary, generate counter-examples for) completeness and determinism properties of both Statecharts and PFS Tables. The results of these proofs allowed numerous errors to be found in the specification models used in the case studies as well as the requirements from which they were derived (see Section 4.6). This validation step allowed many errors to be detected at an early phase of development, saving much rework.

Tool re-use through a common intermediate formal representation: The use of Z as a common intermediate representation enabled the automated validation and testing techniques to be re-used for both Statechart and PFS tabular specifications. This was most evident in the case studies described in Sections 4.6 and 5.5 where the same tools were used to discharge a large number of proofs and generate a large number of test cases based on Z specifications generated from both notation types.

The weaknesses of the formalisation step can be summarised as follows:

Notation subsetting: Only a subset of the Statechart notation was formalised. Therefore there is no guarantee that the full notation could be formalised in Z and integrated into the framework. However, the chosen subset was found to be sufficient to model industrial case studies and therefore the use of a subset of the notation did not appear to compromise the goal of providing techniques applicable to the chosen domain of aerospace control software. A wider range of example specifications is required to confirm this point and some extension of the subset may inevitably be required. Certain parts of the notation will be simpler to formalise than others. For example, restrictions over currently active states of orthogonal components and timers could be formalised using a similar style of predicate as already used for configuration restrictions and events respectively.

Limitations of Z as an intermediate notation: Aspects of Statecharts such as the asynchronous time model, where an atomic operation may consist of a sequence of transitions are more difficult to formalise in Z. Representing such behaviour in Z results in an extremely verbose specification and a large increase in the amount of proof work required to validate particular properties. As an example, proving properties

of sequences using the CADiℤ proof tool requires a great deal of effort to discharge general properties of sequences, (e.g. the finiteness of any given sequence). However, this portion of the proof does not directly contribute to the verification of the property being investigated as it can be assumed that the translation will always result in well formed sequences. In such cases, an alternative formalism (e.g. process algebra) may have been better suited to modelling the behaviour.

Restricted domain: A number of assumptions were made about the specification to be modelled by deliberately restricting the scope of the work to the aerospace control domain. For example, the range of data types could be restricted to those used in the models (integers, reals, enumerated types and Booleans). Extending the framework to other domains may require a wider variety of types (e.g. records or lists) to be modelled. The application of automated proof techniques to such types may be less efficient and generic than demonstrated in this thesis.

Many researchers have studied the formalisation of graphical notations for various purposes ranging from an analysis of semantic properties of the notations to the use of model checking to check properties of the specifications. The thesis differs from other work in that the formalisation of graphical notations is performed as a means of extracting an explicit mathematical specification of the behaviour to be tested and as a means of providing a common intermediate format to allow the testing tool set to be applied to specifications written in a number of source notations.

Other specification notations other than Z may have been used for the formalisation. However, with hindsight, Z was a good choice of formalism. Despite some aspects of Statecharts being difficult to model in Z, such as the sequential properties, the notation was found to be a good method of specifying the transition functions. No problems were found when modelling PFS tables in Z and it is anticipated that Z would also be a convenient means of formalising other reactive specification notations such as MATLAB/Simulink [TM01]. The greatest benefit of using Z was the availability of the automated theorem prover CADiℤ with its ability to specify generic proof tactics which formed the core of the automation activities.

8.2.2 Operation-based testing

Test cases were automatically generated based on generic formal specifications of testing criteria that were applied to the specification using automated theorem provers. By formalising the testing heuristics, properties of the heuristics themselves could be analysed (such as completeness and satisfiability) and the relative fault detection capabilities of the heuristics could be compared. This led to the definition (in Chapter 7) of efficient test heuristics that could detect a large range of faults. Both partitioning and fault-based heuristics were specified under the same framework allowing tests to be generated based on common general purpose heuristics or targeted towards particular types of faults.

The strengths of the automated operation-based testing techniques can be summarised as follows:

Flexibility: The testing technique allows various testing heuristics to be specified and applied to a range of Z specifications. New heuristics can be added without changing the toolset providing they can be specified either as a partitioning expression or as a fault detection condition. The flexibility of the technique was demonstrated in the case studies where a large number of test cases were generated using various heuristics from Z specifications derived from both Statecharts and PFS tables.

High integrity: The conformance of the test cases to both the specification and testing heuristics could be demonstrated through proof and rigorous argument. This is significant as the results of the tests could be used to reason about the integrity of safety–critical software. Each step in the derivation of the test cases and test data was performed as an automated proof step. Furthermore, the steps used to generate the tests could be recorded in the tool for future analysis, should the integrity of the tests be questioned.

Structural coverage: By selecting certain partitioning heurisitcs (e.g. disjunction analysis) and exploiting completeness in the formal specifications, the test method can be used to achieve pre-determined levels of structural *specification* coverage. By analysing the corresponding structural *code* coverage achieved by running these tests, more intense manual effort can be prioritised into areas of the code representing either refinement of the specification or potential faults.

The weaknesses of the operation-based testing techniques can be summarised as follows:

Negative testing: Test cases were specified by conjoining constraining predicates that represent the subdomain of the input space to be tested with the predicate part of the specification. In order to be able to solve the resulting constraint (to generate the test data), the constraining predicate must be consistent with the operation under test. As a result, negative tests can not be generated from an operation. If the entire specification can be shown to be complete, negative testing can be performed by generating tests from operations whose guards are the logical negation of the operation under test. An alternative approach would be to conjoin the constraining predicates with the disjunction of all operations. This would have allowed negative tests to be identified from operation specifications while resulting in a constraint that is satisfiable. However, the resulting constraint would be much larger than if test data was generated from individual operation specifications, and in many cases complete specifications may not exist. For example, in the PFS approach, the behaviour where an assumption is not satisfied is commonly not specified.

Specifying testing heuristics: The testing heuristics are presently specified using the Z notation. However, the specifications to be tested are written in graphical or tabular notations such as Statechart and PFS tables. There is therefore a semantic gap between the specifications to be tested and the representation of the testing heuristics. Future work should examine ways in which the heuristics could be defined using similar notations to those used to write the specifications.

Limited exploration of applicability: The testing techniques and methods were only verified for a very limited subset of Z. Although by design, the techniques should be generally applicable, the evidence provided in the thesis is inconclusive.

The automated testing method described in Chapter 5 is an improvement over previous work in the area in both the flexibility of the testing heuristics and the level of automation achieved. Previous work such as that by Dick and Faivre [DF93] has been restricted to simple heuristics such as reduction to disjunctive normal form, and the testing heuristics used by Stocks et al. [SC93] were not formalised and therefore the application of the heuristics

could not be automated. The formalisation of the testing heuristics also allowed for automated test case generation and test adequacy evaluation (see Chapter 7) to be performed within the same framework. This is a significant advance as the evaluation of test adequacy strongly affects the design of the testing heuristics.

8.2.3 Test sequence generation

Sequences of test cases were generated for situations where all parts of the system state were not directly observable and/or controllable at the testing interface. The generic approach to sequencing Z specified test cases involved generating an abstract finite state machine from the testing equivalence classes of the system state, inputs and outputs. Although this allows sequences to be generated that have the same fault detection power as the operation testing approach, a number of difficulties were encountered.

The strengths of the test sequence generation techniques can be summarised as follows:

Combined control and data testing: The testing technique described in Chapter 6 makes no distinctions about the control and data parts of the specification. The abstract states are calculated based on those parts of the system that are not directly observable or controllable at the testing interface. This may include a combination of data and state variables. For the example in Chapter 6, the abstract state consisted of the Statechart configurations and an internal timer variable. This flexibility allows tests to be generated to suit the amount of controllability and observability available at the particular testing interface.

Flexibility of testing criteria: Test sequences were generated from Z operation schemas. These operation schemas may represent test cases generated using various testing heuristics as described in Chapter 5 defining the equivalence classes of the system state to be tested. The state checking sequences verify the system state according to these equivalence classes. The system state can therefore be tested according to arbitrary testing hypotheses depending on the testing heuristics used to generate the test cases. The use of a model checker to generate test sequences also provided a great deal of flexibility in the properties of the test sequences, limited only by the expressiveness of the model checking property language.

The weaknesses of the test sequence generation techniques can be summarised as follows:

Identification of the abstract states: The process of identifying the abstract states from the Z operations has not yet been automated, though a procedure for doing so is defined in Section 6.2. The reason for this is that the simplification required to project the operation predicates on to the state variables, inputs and outputs respectively may require a significant amount of proof work. The automation of this simplification is currently outside the capabilities of the proof tools used. There are two approaches that could be used to automate this step. In the first, simplification tactics could be derived for commonly occurring structures in the specification. Such tactics could be constructed using the existing proof tool, CADiℤ, however a large number of such tactics may be required and would need to be continually updated as new constructs are used in the specifications. Alternatively, more powerful theorem provers could be produced that are capable of performing the simplification in a fully automatic manner.

Validity of testing assumptions: The fault detection capability of the state checking sequences depended on several assumptions made about the faults in the system. In particular, faults are not masked by a coupling effect and the implementation has no additional states. These assumptions were no more restrictive than those used in the established FSM-based techniques from which the techniques were derived, but more work is required to better understand the effect that the abstraction has on these assumptions, e.g to determine what faults lead to additional *abstract* states in the AFSM.

State explosion: Several Statechart-based optimisations were proposed and were successful in reducing the state space of the test generation algorithms. However, these optimisations placed more demands on the toolset and eroded some of the toolset's generality. The difficulties in testing Statecharts are not seen to be a problem with the techniques used to generate the test cases, but inherent in the Statecharts themselves. Therefore, the most effective approach to generating test sequences from Statecharts would be to design the system in such a way that untestable properties of Statecharts

are avoided and that a suitable decomposition of the system could be performed for testing (e.g. concurrent components could be tested in isolation).

A number of advancements were made over previous work in this area. With respect to testing Statecharts, Bogdanov et al. [BHS98] assume that the transition operation can be tested separately to the control flow where as Hierons et al. [HSS01] apply a similar method to that described here but restricted to non-concurrent state machines whose transitions are specified as Z operations. The work described in this thesis differs from Hierons et al. in that the sequences are generated from (a subset of) Statecharts as defined by Harel and supported by the STATEMATE modelling environment. A greater level of flexibility in the testing heuristics exists due to the automated operation testing techniques from Chapter 5 and sequences can also be generated for Statecharts with limited concurrency.

8.2.4 Test adequacy evaluation

The formalisation of testing heuristics made possible a systematic approach to predicting the efficiency and effectiveness of the resulting test sets. The fault coverage of different heuristics can be compared and heuristics constructed to explicitly cover a wider variety of faults. Two alternative approaches to constructing efficient and effective test sets were described. The first, based on the computation of efficient necessary fault-detection conditions (weak mutation) was applied to a case study and was shown to detect a large number of program mutations. However, weak mutation cannot guarantee the detection of faults as expressions elsewhere in the specification or implementation may mask the faults, preventing them from being observed at the testing interface. A method of selecting efficient and effective test sets based on sufficient fault detection conditions (strong mutation) was therefore presented. The generation of tests of this type was presented as an optimisation problem. However, the tool support required to evaluate the method was outside the scope and resources of the thesis and is therefore to be developed as part of future work.

The strengths of the test adequacy evaluation techniques can be summarised as follows:

Systematic testing: The ability to evaluate the fault coverage of a test set facilitates a systematic approach to test set construction. The completeness of the test set can be determined as the proportion of some predetermined set of fault detection conditions

that are covered by the test set. The efficiency of the test set can be determined by the number of test cases required to cover the fault detection conditions. The use of such measures for test evaluation and control can increase the confidence in the integrity of the delivered software by providing quantifiable measures with which to evaluate the quality of the test set.

Reduced number of test cases: Efficient test heuristics result in smaller test sets which reduce the amount of resources required to refine the tests into concrete test data, run the tests and evaluate the results. Reducing the number of test cases generated from the specification also has a positive effect on the test sequence generation techniques, as the amount of time and memory required to generate the tests increases rapidly with the number of test cases.

Analysis of testability: The number of potential solutions to fault detectability conditions can be used as a measure of testability. If it can be shown that some potential faults are hard to detect through testing, manual review or static analysis can be targeted towards these parts of the system.

The weaknesses of the test adequacy evaluation techniques can be summarised as follows:

Relevance of fault-detection conditions: The fault hypotheses used to generate the test cases refer to mutations of the specification. It is assumed that faults in the implementation reveal themselves as mutations of the specification. The positive results of the case studies implied that these assumptions are reasonable. However, the relationship might not be uniform across implementations and faults. More experimentation is required to investigate in more detail the way in which faults in the implementation reveal themselves as mutations of the specification.

Requirement on negative testing: Some faults require negative tests to be generated. The proposed method of automated operation-based testing described in Chapter 5 does not allow for the generation of negative tests. The detection of certain faults might therefore be dependent on the completeness of the specification (see Section 8.2.2).

The techniques described in Chapter 7 cover a wider range of faults than similar work in this area by Kuhn [Kuh99] whose analysis was restricted to Boolean expressions and operators. Furthermore, the techniques are fully integrated into the automated testing framework allowing the results of the analysis to be easily evaluated. This allows optimised heuristics to be developed for particular problem domains and source notations.

8.3 Evaluation of the case study evidence

Empirical evidence was gathered in order to demonstrate the validity of the techniques as well as their applicability to realistic problem domains. The empirical evidence was collected as part of controlled experiments as well as within a software engineering process demonstration project. The use of the techniques within the demonstration process has been documented in several papers [MGB+98, BCGM00, ABB+01].

All case study material was either taken directly, or derived, from specifications developed for aerospace engine control projects and therefore could be seen to accurately represent the set of problems encountered in the target domain. None of the specifications was originally conceived by the author. The effect of this independence was highlighted in Chapter 6 where many of the Statecharts were found to be untestable. By constructing functionally similar Statecharts from which tests could be generated, much insight was gained into design attributes of the specifications that contribute towards testability.

Several approaches were taken to assess the effectiveness of the generated test sets. Different programs were tested and different versions of the same program were tested that were constructed using different methods, i.e. written by hand and automatically generated using different levels of optimisation. This allowed the quality of the test sets to be assessed across different programming styles and mitigated against the possibility that the results could be artificially positive due to a particularly testable implementation. For each program, mutation analysis was applied to generate a number of variants of the program, each of which varied from a correct implementation in a slightly different way. The number of mutants that were detected by the test set could then be used to compare the relative effectiveness of different test sets.

A number of conclusions were drawn from the experiments. First, the techniques were applicable to specifications typical of the target domain. Second, each of the techniques

contributed to improving the final quality of the system and did so in a far more efficient manner than existing manual processes. The experiments also demonstrated the limitations of the techniques. For example, generating sequences of test cases from specifications with a large number of input equivalence classes would cause the prototype tools to take an infeasible amount of memory and time resources to generate sequences or to conclude that no sequences existed. This led to the definition of design criteria for such systems which could avoid these problems.

8.4 Evaluation of tool support

A number of prototype tools were developed to investigate the potential for automating the techniques described in the thesis. Tools were developed to generate Z representations of the graphical and tabular specifications, including the proof obligations required to prove completeness and determinism of the specifications. An automated theorem prover formed the basis of a number of tools that discharged the proof obligations, generated test cases and generated the test data. Another tool generated model checker specifications and Unique Input/Output sequences based on an XML definition of abstract finite state machines. These tools were successful both in demonstrating the potential for automation and facilitating the collection of the empirical evidence.

Plans are currently being made to integrate the tool set into the sponsoring company's software engineering processes possibly through partnership with commercial modelling tool suppliers. However, before the tools and techniques can be deployed, several improvements must be made. The specification validation techniques would benefit greatly from a better integration into the modelling tools used to generate and animate the specifications. An interface should be constructed to allow the generic tool set to be integrated into the most suitable modelling tools for the particular project (e.g. to allow counter-example traces to be animated). The range of validation activities could be greatly increased if desirable properties in the specification could be couched using the same graphical notation and translated into, for example, the invariant part of a model checker specification.

The test case generation toolset described in Chapter 5 is based on the automated theorem prover CADiℤ. Z specifications are browsed and expressions within operations selected for the application of a testing heuristic which is selected by typing in the name of the cor-

responding lemma. This level of flexibility was found to be extremely useful in conducting the experiments. However, it is unreasonable to expect an engineer not trained in Z to generate tests in this way. Based on the API to CADiℤ, an intuitive user interface could be envisioned whereby the user selects a set of testing heuristics which would then be applied at each applicable point in the specification. This would remove the need to work directly with the Z specification and allow the tester to concentrate on the selection of an effective set of test criteria based on cost/fault detection capability trade-offs. The Classification Tree Editor tool, developed by DaimlerChrysler [SCES97] would be a suitable candidate for such an interface. The test suite structure is represented in a graphical form, where nodes on a tree represent individual operations or test cases. Applying a testing heuristic to a node generates a new set of child nodes which can either be refined into test data or further partitioned.

Presently, the XML specification of the abstract finite state machines used for the test sequence generation techniques is constructed by hand using a standard XML editor. The automatic generation of the XML from a Z specification of the test cases is considered feasible but was outside the resources of this thesis. XML was used in other parts of the toolset (tabular specifications to Z generation) as an intermediate form and may make a convenient data interchange format to integrate the different tools. As an example, an XML interchange format for UML could allow both Z specifications to be easily generated from UML modelling tools and the generated test sequences to be represented as message sequence charts.

8.5 Evaluation of the framework

The automated V&V framework presented in Chapter 3 integrated the techniques described above into a process compatible with industrial software development activities. The use of the term framework describes both the integrative and methodological nature of the work. Each of the techniques could be used in isolation, e.g. to generate test cases from existing Z specifications. However, the techniques have added value when integrated. For example, without the formalisation of graphical notations the testing techniques would be restricted to software projects whose engineers are experienced in specifying systems in Z. The validation activities ensure that the tests are based on a solid understanding of the behaviour of

the system. Likewise, generating test sequences from test cases generated according to well defined and validated test hypotheses increases their fault detection capability.

The framework also represented a methodology, where graphical specifications are formalised to form a stronger basis of the V&V activities. This methodology was successfully applied to two notations and feasibility studies have been performed for several other variants of Statecharts. The methodology could be extended to facilitate other V&V activities such as program proof. Given some definition of the refinement relation, the formal specifications could be refined into program-level assertions that could be proven using a combination of static analysis tools such as SPARK Examiner [Bar97] and theorem provers such as CADiℤ [Toy96]. The same program level assertions could also form the basis of white-box falsification testing as developed by Tracey [TCM98].

Despite the generic design of the framework, some optimisations specific to Statecharts were proposed. These were required in order to make the automated test generation techniques more efficient. However, most of these optimisations were restricted to the formalisation phase which is particular to the source notation and therefore does not affect the generic parts of the framework. Therefore these optimisations were not seen to significantly affect the generality of the framework.

Based on the evidence presented in this chapter, the set of success criteria presented in Chapter 3 can now be assessed.

1. **Objective:** The framework should be based on specification notations that have been proven effective for modelling the domain of interest (e.g. safety–critical embedded control systems) and do not require expertise in formal methods in order to create the specifications.

   Evaluation: Graphical and tabular notations were successfully integrated into the framework. These notations (Statechart and PFS) are currently in use within the aerospace industry and have proven to be popular with the domain engineers.

2. **Objective:** A sufficient level of automation should be achieved such that input from the user is in the form of the specification and a description of properties to be verified or the testing criteria to be met. The supporting toolset should interpret these properties and perform the necessary formal manipulation of the specifications to produce the desired result.

Evaluation: A significant level of automation was achieved, however the user of the toolset must still work on the level of formal specifications in places. The Z specifications are browsed and testing heuristics are applied to selected expressions. The toolset could be extended so that the input from the user is in the form of a selection of heuristics which would then be applied wherever possible in the Z specification or to operations corresponding to selected parts of the graphical or tabular specifications.

3. **Objective:** The framework should be capable of collecting the verification and testing evidence required to meet certification standards and guidelines including any formal specification and proof required.

   Evaluation: Chapter 5 demonstrated how certain heuristics can be used to achieve structural coverage of the specification that when applied to the implementation also achieve a high level of structural code coverage (depending on the amount of refinement between the specification and code) as mandated by certification standards and guidelines.

   For high integrity systems requiring formal specification and proof, the framework provides a cost effective solution to the problem of the semantic gap between the engineers and customers perception of the requirements and the formal specifications. The specifications can be specified by the engineers with the domain knowledge using toolsets that increase productivity and allow the specifications to be validated using animation techniques. The formal representations of these specifications, although automatically generated, are documented with traceability information and the rationale for any semantic resolution steps. These formal specifications can then be used as the basis for refinement and proof activities if required (e.g. for highly critical portions of the system).

4. **Objective:** The framework should be flexible enough that it can be applied to a number of specification notations, while re-using key tools and techniques.

   Evaluation: The thesis described the application of the framework to two notations, Statecharts and PFS tables. A feasibility study into the application of the framework to other variants of Statecharts has been carried out as part of industrial collaboration work [Bur00]. As discussed in Section 8.5, the framework techniques could also be

extended to cover additional V&V activities such as program-proof.

5. **Objective:** The outputs of the framework should be in a form such that the integrity of the results can be reasoned about to a high level of confidence.

   Evaluation: The method of automation allows for a rigorous review of correctness and therefore facilitates the collection of evidence to demonstrate that the process has been performed correctly. The automatic translation into the formal specification also includes the documentation of traceability information and the rationale behind any semantic resolution steps that were taken. This information can be compared against both the original specification and formal representation as part of the argument for the correctness of the translation.

   The process of generating the test cases, although automated, is based on a series of simple proof steps that are recorded as a proof tactic. Either the proof tactic itself or each individual application of the proof tactic can be scrutinised for correctness. The tools can be configured such that the individual transformations performed when generating the tests are recorded.

6. **Objective:** The techniques should lead to a more cost-effective development process than existing methods.

   Evaluation: The empirical evidence captured as part of the case studies suggests that not only are the techniques cost effective in terms of reducing the cost of the testing phase but they are also capable of detecting a wider range of errors than manual techniques. The process of formalising and validating the specifications also led to an improved understanding of the semantics of the specification notations which is expected to increase the quality of the specifications. This should reduce rework later in the development due to poorly constructed or misinterpreted specifications. However, the techniques will need to be applied to live commercial projects for their cost benefits to be accurately assessed. This will require many other issues to be addressed than were within the scope of this thesis, such as staff training and the quality and reliability of the tool set.

8.6 Summary

The assessment described in this chapter has confirmed the success of the work. The individual components of the framework were successfully implemented and although individually they can be seen as improvements over existing methods, they have the greatest benefit when combined. On the whole, the success criteria first presented in Chapter 3 can be said to be met, though some issues require some further work.

An implicit theme in this thesis is the development of techniques to make formal methods more practical for commercial development processes. In [KDGN97], Knight presented some criteria for industrial acceptance of formal methods. Based on the evidence from the case study and experience of working with industrial partners, the industrial suitability of the work can be assessed according to the same criteria. The criteria are as follows:

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.

The success 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 author's opinion that the work described in this thesis has gone some way to satisfying these criteria, although admittedly for a particular domain and subset of V&V activities. This was accomplished by basing the formal 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. They can therefore 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 CADiℤ 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.

Chapter 9

Conclusions and future work

This chapter completes the thesis by revisiting the hypothesis presented in Chapter 1 and summarising the evidence for its satisfaction. Additional conclusions that can be drawn from the work are presented and areas of future work are discussed.

9.1 Introduction

The previous chapters have presented both the technical development and a detailed evaluation of an automated V&V framework designed to demonstrate the hypothesis presented in Chapter 1. In Section 9.2, the hypothesis is revisited and an argument presented for its satisfaction. Section 9.3 presents a number of conclusions that were drawn from the work and which are not documented elsewhere in the thesis. Finally Section 9.4 discusses potential for future work, in particular addressing open research questions, improved tool support and the wider applicablity of the work.

9.2 Revisiting the hypothesis

The hypothesis of the thesis presented in Chapter 1 stated that:

"Graphical notations of proven practical use in the safety-critical domain can be formalised in a manner that permits the effective and efficient automation of V&V and in particular testing."

This hypothesis was investigated by first presenting a framework for the integration of graphical and tabular notations with a formalised approach to specification validation and

testing. A translation of two notations in use in the safety-critical domain (Statecharts and PFS Tables) into the model-based notation Z was presented in Chapter 4. This formalisation made the behaviour to be tested explicit and also facilitated the use of automated proof techniques to validate certain properties of the specification. The first part of the hypothesis - *"Graphical notations of proven practical use in the safety-critical domain can be formalised"* was therefore satisfied. However only a subset of the Statechart notation was formalised and it was noted that some parts of the notation would either have led to a complex and verbose formal specification or would have required the use of an alternative intermediate formalism.

The formalisation and validation techniques themselves can contribute greatly to the quality of software (as demonstrated in the case studies). These validation techniques were found to complement well other standard techniques that would usually be applied at this phase of development such as peer review and animation based on pre-determined behaviour scenarios.

Chapter 5 presented a method of testing individual operations in the specification. Testing heuristics based on partitioning and fault-based strategies were formalised and automatically applied to the specification using an automated theorem prover. This allowed for a high level of flexibility in the testing criteria and also allowed important properties of the heuristics to be proven which the test cases would then inherit through construction. Chapter 7 described a method for designing efficient and effective testing heuristics based on an analysis of potential faults in the implementation that reveal themselves as mutations of the specification. This method of test case generation was found to be more efficient than existing approaches allowing mandated structural coverage to be achieved while ensuring the detection of a large number of faults. The automated testing method was not only efficient and effective but flexible in terms of the cost/confidence trade-offs that often need to be made in commercial projects. If the testing is to be performed within a limited budget, then a small number of heuristics can be chosen that have been shown to detect a large number (though possibly not all) faults.

Chapter 6 addressed the difficult problem of test case sequencing. This is neccessary as, in many situations, the test harness may not have access to internal information about the component under test (e.g. state and local variables). Test case sequencing was achieved by

generating an abstract finite state machine representation of the test cases where the abstract states, inputs and outputs represented equivalence classes in the specification to be tested. Those parts of the system not directly observable (encapsulated in the abstract state) could then be verified using state checking sequences using traditional FSM-based techniques. The abstract finite state machine representation was also found to be a convenient form for model checking, allowing test sequences to be generated whose properties could be specified using temporal logic formulae. Despite these successes, the feasibility and efficiency of the test sequence generation techniques were found to be highly dependent on particular specification styles.

The above paragraphs summarise the arguments for the satisfaction of the second half of the hypothesis: *"...in a manner that permits the effective and efficient automation of V&V and in particular testing."*. In particular, validation of targeted properties in the specification and operation-based testing were extremely successful and benefited greatly from formalisation of the specification, validation properties and test heuristics. The sequencing of test cases generated from Statechart transitions had less immediate success. However, the problems encountered in test sequence generation – long sequences and a large search space – can be seen to be more indicative of the potential of Statecharts to specify complex behaviour in concise diagrams than limitations of the techniques themselves.

9.3 Conclusions

This section summarises a number of conclusions that can be drawn from the work presented in this thesis.

The importance of formalisation: The use of an intermediate formal notation to integrate the formal validation and testing techniques with practical graphical and tabular notations was an integral part of the framework. One of the main objectives of this approach was to gain some of the benefits of formal methods while remaining within a software engineering process that was accessible to industry. The thesis demonstrated many advantages of this approach including the ability to make explicit, implicit behaviour in the original specifications, formally and automatically prove desirable properties in the specification and the rigorous yet flexible approach to automated test

generation. The advantage of this approach over other automated techniques based on tools that are targeted towards particular V&V problems is the transparency that working in the formal notation allows. The calculation performed by the tools can be examined by experts for correctness. By validating the outputs of the tools, much effort that may otherwise have been expended in certifying the toolset may be saved. The success of this approach in industry, however, will depend on the ability to construct toolsets that hide the intermediate formalisms from engineers, while storing the intermediate calculations of the tools in a manner that can be formally proven.

Testability of Statecharts: The case studies demonstrated that Statecharts can easily become untestable, leading to state checking sequences that require an impractical amount of computing resources to compute. This is due to the fact that a Statechart may contain subtle and complex interactions between its states, transitions and orthogonal components. Additionally, the ability to loosely specify transition guards, i.e. no constraints are given for certain input parameters, and the use of null actions can lead to many transitions being indistinguishable from one another. This phenomenon was found to have the greatest impact on the length of the checking sequences.

The value of fault-based testing: It is accepted that testing can never demonstrate the absence of all errors. However, the fault-based testing techniques discussed in this thesis were shown to be effective at detecting certain hypothesised errors and furthermore the same tests were also found to be effective at detecting many others not hypothesised during test design. The fault-based analysis allows areas of the specification (and therefore also code) that are unlikely to reveal errors to be identified. Manual review of the code and static analysis can then be targeted at these “high risk” areas. Due to the highly automated method of testing, there is a good argument for integrating the techniques described in this process with other high integrity verification methods such as program proof. Automated testing would be performed to detect as many errors as possible in a short period of time. Information from the test design process and structural coverage achieved by running the tests can then be used to guide targeted program proof, with the increased confidence of the proof resulting in success.

9.4 Future work

A number of potential areas for future work and open questions were identified and can be classified as follows:

- Open research questions
- Further development of the toolset
- Application of the techniques to a wider range of problems

9.4.1 Open research questions

Chapter 7 described a method by which efficient testing heuristics could be constructed based on an analysis of weak detectability conditions. This chapter also proposed that the problem of constructing efficient testing heuristics could be formulated as an optimisation problem allowing the process to be automated. In particular such automation, coupled with powerful constraint solvers, could be used to generate efficient test data based on a large number of sufficient fault detection conditions. Further work in this area should be combined with research into the way faults in the implementation reveal themselves as mutations of the specification as this is the underlying hypothesis of specification-based mutation testing. The relationship between actual faults and their abstract representation may be affected by a number of factors such as the amount of refinement between specification and code, code optimisation techniques and the method of code construction (e.g. hand-coded or automatically generated).

The test sequence generation techniques make several assumptions about the nature of faults in the implementation. For example, that a fault is not masked by another in the same sequence and that the implementation has no extra states. Future work should investigate how realistic these assumptions are, especially given the additional abstraction in the finite state machine (where additional states may be introduced by changing the boundaries of certain equivalence classes). Future work should also examine whether the assumptions can be weakened by using more sophisticated test sequence generation techniques (such as the W-method to detect additional states). Finally the formalisation of Statecharts in Z should be improved in order to allow the single writer/multiple reader restrictions on the

communications between orthogonal components to be lifted without resulting in a significant decrease in the performance of the test sequence generation algorithms.

9.4.2 Improved tool support

Some tool support was developed as part of the thesis work. These tools were designed to prototype the feasibility of the automated techniques as well as facilitating the collection of evidence for the case studies. However, the tools could be improved in a number of ways in order for more advanced experiments to be performed and for the techniques to be practical for real commercial projects.

Chapter 8 discussed ways in which an application programming interface to the automated theorem prover could be used to provide a more "user friendly" environment in which test heuristics could be selected and applied to the formal specification without the engineers having to manipulate the Z directly. Although CADiℤ includes a number of built-in constraint solvers, these were not applicable to all specifications such as those including non-linear constraints. Therefore the integration of a stronger set of constraint solvers into the theorem proving environment would allow for a greater amount of automation (by removing the need to translate constraints between constraint solver input formats) and may also allow the testing techniques to be applied to a wider subset of the Z notation (e.g. to include sequences, functions etc.).

The CADiℤ theorem proving environment is also part of a plan to automate the generation of efficient test data based on combined sufficient conditions for strongly detecting faults. A Z specification would be read into the tool and stored as an abstract syntax tree. Mutation operators could then be applied to this tree and the resulting Z constraint used to construct the fault-detectability condition for the particular mutant. Constraint solvers could then be used to verify the satisfiability of the resulting condition. An additional optimisation algorithm could be used to generate the minimal set of such conditions that satisfy a large number of mutants. By allowing the mutation operators to be specified by the user, test sets could be generated weighted against different sets of common faults that may vary across applications and implementation methods. For example, it is expected that the faults introduced into auto-generated code would vary from the typically hypothesised mutations based on the competent programmer hypothesis.

9.4.3 Wider applicability of the work

The formalisation presented in Chapters 4 and 6 was based on the synchronous time model of Statecharts. This time model specifies that inputs are processed at discrete moments in time and any changes to the state or internal variables of the Statechart resulting from these inputs do not cause further transitions to be taken until the next set of inputs are processed at the next discrete time step. This semantics closely models the way in which cyclically scheduled software processes behave and is therefore popular in specifying software functionality. The asynchronous model allows transitions to fire immediately their guarding conditions are satisfied. This may result in a chain of transitions being fired as the result of a single input. This style of specification is better suited to larger scale and distributed system behaviour. Further work should investigate how asynchronous behaviour may be integrated into the framework. In particular a suitable formalisation of the behaviour will be required, and in doing so an alternative or complementary notation to Z may be required. The test sequence generation techniques will need to be extended accordingly, in particular in relation to the communication between components. For example, a single interaction may lead to a cascade of transitions in parallel that could eventually lead to feedback – i.e. the triggering of further transitions in the component that accepted the first input.

A related piece of work may be the inclusion of full multiple reader/multiple writer interactions between parallel components. At present only multiple reader/single writer relationships are allowed and are modelled as constraints over parallel states in the guarding transitions on the abstract finite state machines. Multiple reader/multiple writer relationships would require the addition of post-condition constraints over parallel states which would subsequently need to be taken into account in the test sequence generation algorithms.

All too often, the non-functional properties of systems are not fully verified until the complete system has been constructed, at which stage it is too late to significantly alter the design and extensive compromises and workarounds are required. The application of a similar framework to non-functional properties such as timing would therefore present an extremely useful contribution to current software engineering practices as non-functional properties could be analysed at a very early stage in the product development and specification testability analysis could be used to influence the design of the system to ensure that it

meets its non-functional requirements.

The techniques in this thesis have so far only been demonstrated for a very restricted domain (aerospace embedded control systems) and a limited subset of Z. However, the techniques should be applicable to a wider number of systems. Assuming that testing heuristics can be designed and suitable constraint solvers are available, the generic approach to automated test case generation should be applicable to a wider subset of Z than described here. Extending the subset of Z may allow a wider variety of behaviour to be modelled in the original specifications.

To be able to apply the framework to other problem domains such as telecommunication systems, specification notations popular within these domains, such as SDL (Specification and Description Language) [Uni94] will need to be formalised. In order to use the same tool set and techniques described in this thesis, the individual operations of the specification will need to be modelled in Z and the control structure of the test cases will need to be modelled as abstract finite state machines. Future work should investigate whether other notations (and therefore domains) can be included into the framework by replacing the formalisation phase and making minimal changes to the toolset.

Appendix A

ThrustLimitation

A.1 Z specification

A.1.1 Type and constant definitions

$$FwdIdle == 5$$

$$Boolean ::= True \mid False \mid BUndefined$$

$$Event ::= Absent \mid Present \mid EUndefined$$

$$States ::== ThrustLimitation \mid Idle \mid InadvertentDeploy \mid LockDetect \mid OnGroundDetect \mid Unstowed \mid AwaitLock \mid Wait \mid Locked \mid InAir \mid OnGround$$

A.1.2 System state and parameters

$$relation(configuration\ \_)$$

$$
\begin{array}{l}
configuration\_ == \\
\{ActiveStates : \mathbb{F}_1\ States \mid \\
\quad (ThrustLimitation \in ActiveStates \Rightarrow \\
\quad\quad ((Idle \in ActiveStates \land \neg InadvertentDeploy \in ActiveStates) \lor \\
\quad\quad (InadvertentDeploy \in ActiveStates \land \neg Idle \in ActiveStates))) \land \\
\quad (InadvertentDeploy \in ActiveStates \Rightarrow \\
\quad\quad (LockDetect \in ActiveStates \land OnGroundDetect \in ActiveStates) \land \\
\quad (LockDetect \in ActiveStates \Rightarrow \\
\quad\quad ((Unstowed \in ActiveStates \land \neg AwaitLock \in ActiveStates) \lor \\
\quad\quad (AwaitLock \in ActiveStates \land \neg Unstowed \in ActiveStates))) \land \\
\quad (AwaitLock \in ActiveStates \Rightarrow \\
\quad\quad ((Wait \in ActiveStates \land \neg Locked \in ActiveStates) \lor \\
\quad\quad (Locked \in ActiveStates \land \neg Wait \in ActiveStates))) \land \\
\quad (OnGroundDetect \in ActiveStates \Rightarrow \\
\quad\quad ((InAir \in ActiveStates \land \neg OnGround \in ActiveStates) \lor \\
\quad\quad (OnGround \in ActiveStates \land \neg InAir \in ActiveStates))) \\
\quad (Idle \in ActiveStates \Rightarrow ThrustLimitation \in ActiveStates) \land \\
\quad (InadvertentDeploy \in ActiveStates \Rightarrow ThrustLimitation \in ActiveStates) \land \\
\quad (LockDetect \in ActiveStates \Rightarrow InadvertentDeploy \in ActiveStates) \land \\
\quad (OnGroundDetect \in ActiveStates \Rightarrow InadvertentDeploy \in ActiveStates) \land \\
\quad (Unstowed \in ActiveStates \Rightarrow LockDetect \in ActiveStates) \land \\
\quad (AwaitLock \in ActiveStates \Rightarrow LockDetect \in ActiveStates) \land \\
\quad (Wait \in ActiveStates \Rightarrow AwaitLock \in ActiveStates) \land \\
\quad (Locked \in ActiveStates \Rightarrow AwaitLock \in ActiveStates) \land \\
\quad (InAir \in ActiveStates \Rightarrow OnGroundDetect \in ActiveStates) \land \\
\quad (OnGround \in ActiveStates \Rightarrow OnGroundDetect \in ActiveStates)\}
\end{array}
$$

Status

$$
\begin{array}{l}
ActiveStates : \mathbb{F}_1\ States \\
Timer : \mathbb{N} \\
\hline
configuration(ActiveStates)
\end{array}
$$

Parameters

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

A.1.3 Transition operations

Transition $T2$. This transition does not terminate at a basic state and has therefore been composed with transitions $T4$ and $T9$.

$T2$

$\Delta Status$
Parameters

$\{ThrustLimitation, Idle\} \subseteq ActiveStates$
$InadvertentDeploy? = Present$
$LimitThrust! = True$
$RestowComplete! = EUndefined$
$\{ThrustLimitation, InadvertentDeploy, LockDetect,$
$Unstowed, OnGroundDetect, InAir\} \subseteq ActiveStates'$

Transition $T3$.

$T3$

$\Delta Status$
Parameters

$\{ThrustLimitation, InadvertentDeploy, LockDetect,$
$AwaitLock, Locked, OnGroundDetect\} \subseteq ActiveStates$
$ThrottleReq? \leq FwdIdle$
$RestowComplete! = Present$
$LimitThrust! = False$
$\{ThrustLimitation, Idle\} \subseteq ActiveStates'$

Transition $T5$. This transition does not terminate at a basic state and has therefore been composed with transition $T7$.

$T5$

$\Delta Status$
Parameters

$\{ThrustLimitation, InadvertentDeploy, LockDetect,$
$OnGroundDetect, Unstowed\} \subseteq ActiveStates$
$UnstowedSensor? = False$
$Timer' = 0$
$LimitThrust! = BUndefined$
$RestowComplete! = EUndefined$
$\{ThrustLimitation, InadvertentDeploy, LockDetect,$
$OnGroundDetect, AwaitLock, Wait\} \subseteq ActiveStates'$

Transition $T6$. This transition does not originate at a basic state and has therefore been partitioned into transitions originating at the *Wait* and *Locked* state respectively.

$T6.1$

$\Delta Status$

Parameters

$\{ThrustLimitation, InadvertentDeploy, LockDetect,$
$OnGroundDetect, AwaitLock, Wait\} \subseteq ActiveStates$
$UnstowedSensor? = True$
$LimitThrust! = BUndefined$
$RestowComplete! = EUndefined$
$\{ThrustLimitation, InadvertentDeploy, LockDetect,$
$OnGroundDetect, Unstowed\} \subseteq ActiveStates'$

Transition $T6.2$ can be pre-empted by transition $T3$ and is therefore strengthened with the negation of $T3$'s guard.

$T6.2$

$\Delta Status$

Parameters

$\{ThrustLimitation, InadvertentDeploy, LockDetect,$
$OnGroundDetect, AwaitLock, Locked\} \subseteq ActiveStates$
$UnstowedSensor? = True$
$\neg ThrottleReq? \leq FwdIdle$
$LimitThrust! = BUndefined$
$RestowComplete! = EUndefined$
$\{ThrustLimitation, InadvertentDeploy, LockDetect,$
$OnGroundDetect, Unstowed\} \subseteq ActiveStates'$

Transition $T8$. This transition can be pre-empted by transition $T6.1$ and is therefore strengthened with the negation of $T6.1$'s guard.

$T8$

$\Delta Status$
$Parameters$

$\{ThrustLimitation, InadvertentDeploy, LockDetect,$
$OnGroundDetect, AwaitLock, Wait\} \subseteq ActiveStates$
$Timer > 10$
$\neg UnstowedSensor? = True$
$LimitThrust! = BUndefined$
$RestowComplete! = EUndefined$
$\{ThrustLimitation, InadvertentDeploy, LockDetect,$
$OnGroundDetect, AwaitLock, Locked\} \subseteq ActiveStates'$

Transition $T10$. This transition can be pre-empted by transition $T3$, and is therefore strengthened with the negation of $T3$'s guard and configuration.

$T10$

$\Delta Status$
$Parameters$

$\{ThrustLimitation, InadvertentDeploy, LockDetect,$
$OnGroundDetect, InAir\} \subseteq ActiveStates$
$ThrottleReq? \leq FwdIdle$
$OnGround? = True$
$\neg (Locked \in ActiveStates \land$
$ThrottleReq? \leq FwdIdle)$
$LimitThrust! = False$
$RestowComplete! = EUndefined$
$\{ThrustLimitation, InadvertentDeploy, LockDetect,$
$OnGroundDetect, OnGround\} \subseteq ActiveStates'$

Transition $T11$. This transition can be pre-empted by transition $T6.1$ and is therefore strengthened with the negation of $T6.1$'s guard.

*T*11

$\Delta Status$
Parameters

$\{ThrustLimitation, InadvertentDeploy, LockDetect,$
$OnGroundDetect, AwaitLock, Wait\} \subseteq ActiveStates$
$Timer \leq 10$
$UnstowedSensor? = False$
$Timer' = Timer + 1$
$LimitThrust! = BUndefined$
$RestowComplete! = EUndefined$
$\{ThrustLimitation, InadvertentDeploy, LockDetect,$
$OnGroundDetect, AwaitLock, Wait\} \subseteq ActiveStates'$

Completing transition for the state *Idle*.

CompleteIdle

$\Xi Status$
Parameters

$\{ThrustLimitation, Idle\} \subseteq ActiveStates$
$\neg InadvertentDeploy? = Present$
$LimitThrust! = BUndefined$
$RestowComplete! = EUndefined$

Completing transition for the state *Unstowed*.

CompleteUnstowed

$\Xi Status$
Parameters

$\{ThrustLimitation, InadvertentDeploy, LockDetect,$
$OnGroundDetect, Unstowed\} \subseteq ActiveStates$
$UnstowedSensor? = True$
$LimitThrust! = BUndefined$
$RestowComplete! = EUndefined$

Completing transition for the state *Locked*.

CompleteLocked

$\Xi Status$

Parameters

$\{ThrustLimitation, InadvertentDeploy, LockDetect,$
$OnGroundDetect, AwaitLock, Locked\} \subseteq ActiveStates$
$\neg(ThrottleReq \leq FwdIdle \lor$
$UnstowedSensor? = True)$
$LimitThrust! = BUndefined$
$RestowComplete! = EUndefined$

Completing operation for the state *InAir*. This transition can be pre-empted by transition $T3$, if the Statechart is in a particular configuration, and therefore has been strengthened with the negation of $T3$'s guard.

CompleteInAir

$\Xi Status$

Parameters

$\{ThrustLimitation, InadvertentDeploy, LockDetect,$
$OnGroundDetect, InAir\} \subseteq ActiveStates$
$\neg((ThrottleReq? \leq FwdIdle \land OnGround? = True) \lor$
$(Locked \in ActiveStates \land ThrottleReq? \leq FwdIdle))$
$LimitThrust! = BUndefined$
$RestowComplete! = EUndefined$

Completing operation for the state *OnGround*. This transition can be pre-empted by transition $T3$, if the Statechart is in a particular configuration, and therefore has been strengthened with the negation of $T3$'s guard and configuration.

CompleteOnGround

$\Xi Status$

Parameters

$\{ThrustLimitation, InadvertentDeploy, LockDetect,$
$OnGroundDetect, OnGround\} \subseteq ActiveStates$
$\neg(ThrottleReq? \leq FwdIdle \land$
$Locked \in ActiveStates)$
$LimitThrust! = BUndefined$
$RestowComplete! = EUndefined$

A.1.4 Proof conjectures

The completeness and determinism conjectures for each state (before the completing transitions were added) are as follows:

Conjecture for the completeness of state *Idle*:

$$\vdash? \forall \mathit{Status};\ \mathit{Parameters} \mid \mathit{Idle} \in \mathit{ActiveStates} \bullet \\ \mathit{InadvertentDeploy}? = \mathit{Present}$$

This proof fails. A completing transition is therefore required for this state. There is only one transition out of *Idle*. It is therefore deterministic by default and no determinism proof is required, assuming the transition itself defines deterministic behaviour.

Conjecture for the completeness of state *Unstowed*:

$$\vdash? \forall \mathit{Status};\ \mathit{Parameters} \mid \mathit{Unstowed} \in \mathit{ActiveStates} \bullet \\ \mathit{UnstowedSensor}? = \mathit{False}$$

This proof fails. A completing transition is therefore required for this state. There is only one transition out of *Unstowed*. It is therefore deterministic by default and no determinism proof is required, assuming the transition itself defines deterministic behaviour.

Conjecture for the completeness of state *Wait*:

$$\vdash? \forall \mathit{Status};\ \mathit{Parameters} \mid \mathit{Wait} \in \mathit{ActiveStates} \bullet \\ (\mathit{UnstowedSensor}? = \mathit{True}) \lor \\ (\mathit{UnstowedSensor}? = \mathit{False} \land \mathit{Timer} \leq 10) \lor \\ (\mathit{UnstowedSensor}? = \mathit{False} \land \mathit{Timer} > 10)$$

The conjecture reduces to true. The state is therefore complete.

Conjecture for the determinism of state *Wait*:

$$\vdash? \forall \mathit{Status};\ \mathit{Parameters} \mid \mathit{Wait} \in \mathit{ActiveStates} \bullet \\ \neg((\mathit{UnstowedSensor}? = \mathit{True}) \land (\mathit{UnstowedSensor}? = \mathit{False} \land \mathit{Timer} \leq 10)) \land \\ \neg((\mathit{UnstowedSensor}? = \mathit{True}) \land (\mathit{UnstowedSensor}? = \mathit{False} \land \mathit{Timer} > 10)) \land \\ \neg((\mathit{UnstowedSensor}? = \mathit{False} \land \mathit{Timer} \leq 10) \land \\ \quad (\mathit{UnstowedSensor}? = \mathit{False} \land \mathit{Timer} > 10)$$

The conjecture reduces to true. The state is therefore deterministic.

Conjecture for the completeness of state *Locked*:

$$\vdash? \forall \mathit{Status};\ \mathit{Parameters} \mid \mathit{Locked} \in \mathit{ActiveStates} \bullet \\ (\mathit{ThrottleReq}? \leq \mathit{FwdIdle}) \lor \\ (\mathit{UnstowedSensor}? = \mathit{True} \land \mathit{ThrottleReq}? > \mathit{FwdIdle})$$

This proof fails, a completing transition is therefore required.

Conjecture for the determinism of state *Locked*:

$$\vdash ? \; \forall \mathit{Status}; \; \mathit{Parameters} \mid \mathit{Locked} \in \mathit{ActiveStates} \bullet$$
$$\neg (\mathit{ThrottleReq}? \leq \mathit{FwdIdle} \land$$
$$(\mathit{UnstowedSensor}? = \mathit{True} \land \mathit{ThrottleReq}? > \mathit{FwdIdle}))$$

This conjecture reduces to true. The state is therefore deterministic.

Conjecture for the completeness of state *InAir*:

$$\vdash ? \; \forall \mathit{Status}; \; \mathit{Parameters} \mid \mathit{InAir} \in \mathit{ActiveStates} \bullet$$
$$\mathit{ThrottleReq}? \leq \mathit{FwdIdle} \land \mathit{OnGround}? = \mathit{True} \land$$
$$\neg (\mathit{Locked} \in \mathit{ActiveStates} \land \mathit{ThrottleReq}? \leq \mathit{FwdIdle})$$

This proof fails. A completing transition is therefore required for this state. There is only one transition out of *InAir*. It is therefore deterministic by default and no determinism proof is required, assuming the transition itself defines deterministic behaviour.

There are no transitions out of state *OnGround* and therefore no completeness or determinism proofs are needed.

A.2 Test cases

The following test cases were generated based on boundary value analysis and disjunction analysis applied to the core behaviour (i.e. non-completing transitions).

Boundary value analysis applied to $\mathit{ThrottleReq}? \leq \mathit{FwdIdle}$ in operation $T3$.

T3BVA1

$\Delta \mathit{Status}$
$\mathit{Parameters}$

$\{\mathit{ThrustLimitation}, \mathit{InadvertentDeploy}, \mathit{LockDetect},$
$\mathit{AwaitLock}, \mathit{Locked}, \mathit{OnGroundDetect}\} \subseteq \mathit{ActiveStates}$
$\boxed{\mathit{ThrottleReq}? < \mathit{FwdIdle} - 1}$
$\mathit{ThrottleReq}? \leq \mathit{FwdIdle}$
$\mathit{RestowComplete}! = \mathit{Present}$
$\mathit{LimitThrust}! = \mathit{False}$
$\{\mathit{ThrustLimitation}, \mathit{Idle}\} \subseteq \mathit{ActiveStates}'$

*T*3*BVA*2

$\Delta Status$
Parameters

$\{ThrustLimitation, InadvertentDeploy, LockDetect,$
$AwaitLock, Locked, OnGroundDetect\} \subseteq ActiveStates$
$\boxed{ThrottleReq? = FwdIdle - 1}$
$ThrottleReq? \leq FwdIdle$
$RestowComplete! = Present$
$LimitThrust! = False$
$\{ThrustLimitation, Idle\} \subseteq ActiveStates'$

*T*3*BVA*3

$\Delta Status$
Parameters

$\{ThrustLimitation, InadvertentDeploy, LockDetect,$
$AwaitLock, Locked, OnGroundDetect\} \subseteq ActiveStates$
$\boxed{ThrottleReq? = FwdIdle}$
$ThrottleReq? \leq FwdIdle$
$RestowComplete! = Present$
$LimitThrust! = False$
$\{ThrustLimitation, Idle\} \subseteq ActiveStates'$

Boundary value analysis applied to $\neg ThrottleReq? \leq FwdIdle$ in operation $T6.2$.

*T*6.2*BVA*1

$\Delta Status$
Parameters

$\{ThrustLimitation, InadvertentDeploy, LockDetect,$
$OnGroundDetect, AwaitLock, Locked\} \subseteq ActiveStates$
$UnstowedSensor? = True$
$\boxed{ThrottleReq? = FwdIdle + 1}$
$\neg ThrottleReq? \leq FwdIdle$
$LimitThrust! = BUndefined$
$RestowComplete! = EUndefined$
$\{ThrustLimitation, InadvertentDeploy, LockDetect,$
$OnGroundDetect, Unstowed\} \subseteq ActiveStates'$

*T*6.2*BVA*2

$\Delta Status$
Parameters

$\{ThrustLimitation, InadvertentDeploy, LockDetect,$
$OnGroundDetect, AwaitLock, Locked\} \subseteq ActiveStates$
$UnstowedSensor? = True$
$\boxed{ThrottleReq? > FwdIdle + 1}$
$\neg ThrottleReq? \leq FwdIdle$
$LimitThrust! = BUndefined$
$RestowComplete! = EUndefined$
$\{ThrustLimitation, InadvertentDeploy, LockDetect,$
$OnGroundDetect, Unstowed\} \subseteq ActiveStates'$

Boundary value analysis applied to $Timer > 10$ in operation $T8$.

*T*8*BVA*1

$\Delta Status$
Parameters

$\{ThrustLimitation, InadvertentDeploy, LockDetect,$
$OnGroundDetect, AwaitLock, Wait\} \subseteq ActiveStates$
$\boxed{Timer = 11}$
$Timer > 10$
$UnstowedSensor? = False$
$LimitThrust! = BUndefined$
$RestowComplete! = EUndefined$
$\{ThrustLimitation, InadvertentDeploy, LockDetect,$
$OnGroundDetect, AwaitLock, Locked\} \subseteq ActiveStates'$

*T*8*BVA*2

$\Delta Status$
Parameters

$\{ThrustLimitation, InadvertentDeploy, LockDetect,$
$OnGroundDetect, AwaitLock, Wait\} \subseteq ActiveStates$
$\boxed{Timer > 11}$
$Timer > 10$
$UnstowedSensor? = False$
$LimitThrust! = BUndefined$
$RestowComplete! = EUndefined$
$\{ThrustLimitation, InadvertentDeploy, LockDetect,$
$OnGroundDetect, AwaitLock, Locked\} \subseteq ActiveStates'$

Disjunction analysis applied to $\neg(Locked \in ActiveStates \land ThrottleReq? \leq FwdIdle)$ in operation $T10$.

$T10DIS1$

$\Delta Status$

Parameters

$\{ThrustLimitation, InadvertentDeploy, LockDetect,$
$OnGroundDetect, InAir\} \subseteq ActiveStates$
$ThrottleReq? \leq FwdIdle$
$OnGround? = True$
$\boxed{\neg Locked \in ActiveStates \land \neg ThrottleReq? \leq FwdIdle}$
$\neg(Locked \in ActiveStates \land ThrottleReq? \leq FwdIdle)$
$LimitThrust! = False$
$RestowComplete! = EUndefined$
$\{ThrustLimitation, InadvertentDeploy, LockDetect,$
$OnGroundDetect, OnGround\} \subseteq ActiveStates'$

This test case is unsatisfiable and is therefore removed.

$T10DIS2$

$\Delta Status$

Parameters

$\{ThrustLimitation, InadvertentDeploy, LockDetect,$
$OnGroundDetect, InAir\} \subseteq ActiveStates$
$ThrottleReq? \leq FwdIdle$
$OnGround? = True$
$\boxed{Locked \in ActiveStates \land \neg ThrottleReq? \leq FwdIdle}$
$\neg(Locked \in ActiveStates \land ThrottleReq? \leq FwdIdle)$
$LimitThrust! = False$
$RestowComplete! = EUndefined$
$\{ThrustLimitation, InadvertentDeploy, LockDetect,$
$OnGroundDetect, OnGround\} \subseteq ActiveStates'$

This test case is unsatisfiable and is therefore removed.

$T10DIS3$

$\Delta Status$

Parameters

$\{ThrustLimitation, InadvertentDeploy, LockDetect,$
$OnGroundDetect, InAir\} \subseteq ActiveStates$
$ThrottleReq? \leq FwdIdle$
$OnGround? = True$
$\boxed{\neg Locked \in ActiveStates \land ThrottleReq? \leq FwdIdle}$
$\neg(Locked \in ActiveStates \land ThrottleReq? \leq FwdIdle)$
$LimitThrust! = False$
$RestowComplete! = EUndefined$
$\{ThrustLimitation, InadvertentDeploy, LockDetect,$
$OnGroundDetect, OnGround\} \subseteq ActiveStates'$

Boundary value analysis applied to $ThrottleReq? \leq FwdIdle$ in test case $T10DIS3$.

$T10BVA1$

$\Delta Status$

Parameters

$\{ThrustLimitation, InadvertentDeploy, LockDetect,$
$OnGroundDetect, InAir\} \subseteq ActiveStates$
$\boxed{ThrottleReq? < FwdIdle - 1}$
$ThrottleReq? \leq FwdIdle$
$OnGround? = True$
$\neg Locked \in ActiveStates \land ThrottleReq? \leq FwdIdle$
$\neg(Locked \in ActiveStates \land ThrottleReq? \leq FwdIdle)$
$LimitThrust! = False$
$RestowComplete! = EUndefined$
$\{ThrustLimitation, InadvertentDeploy, LockDetect,$
$OnGroundDetect, OnGround\} \subseteq ActiveStates'$

$T10BVA2$

$\Delta Status$

Parameters

$\{ThrustLimitation, InadvertentDeploy, LockDetect,$
$OnGroundDetect, InAir\} \subseteq ActiveStates$

$\boxed{ThrottleReq? = FwdIdle - 1}$

$ThrottleReq? \leq FwdIdle$

$OnGround? = True$

$\neg Locked \in ActiveStates \land ThrottleReq? \leq FwdIdle$

$\neg(Locked \in ActiveStates \land ThrottleReq? \leq FwdIdle)$

$LimitThrust! = False$

$RestowComplete! = EUndefined$

$\{ThrustLimitation, InadvertentDeploy, LockDetect,$
$OnGroundDetect, OnGround\} \subseteq ActiveStates'$

$T10BVA3$

$\Delta Status$

Parameters

$\{ThrustLimitation, InadvertentDeploy, LockDetect,$
$OnGroundDetect, InAir\} \subseteq ActiveStates$

$\boxed{ThrottleReq? = FwdIdle}$

$ThrottleReq? \leq FwdIdle$

$OnGround? = True$

$\neg Locked \in ActiveStates \land ThrottleReq? \leq FwdIdle$

$\neg(Locked \in ActiveStates \land ThrottleReq? \leq FwdIdle)$

$LimitThrust! = False$

$RestowComplete! = EUndefined$

$\{ThrustLimitation, InadvertentDeploy, LockDetect,$
$OnGroundDetect, OnGround\} \subseteq ActiveStates'$

Boundary value analysis applied to $Timer \leq 10$ in operation $T11$.

*T*11*BVA*1

$\Delta Status$
Parameters

$\{ThrustLimitation, InadvertentDeploy, LockDetect,$
$OnGroundDetect, AwaitLock, Wait\} \subseteq ActiveStates$
$\boxed{Timer < 9}$
$Timer \leq 10$
$UnstowedSensor? = False$
$Timer' = Timer + 1$
$LimitThrust! = BUndefined$
$RestowComplete! = EUndefined$
$\{ThrustLimitation, InadvertentDeploy, LockDetect,$
$OnGroundDetect, AwaitLock, Wait\} \subseteq ActiveStates'$

*T*11*BVA*2

$\Delta Status$
Parameters

$\{ThrustLimitation, InadvertentDeploy, LockDetect,$
$OnGroundDetect, AwaitLock, Wait\} \subseteq ActiveStates$
$\boxed{Timer = 9}$
$Timer \leq 10$
$UnstowedSensor? = False$
$Timer' = Timer + 1$
$LimitThrust! = BUndefined$
$RestowComplete! = EUndefined$
$\{ThrustLimitation, InadvertentDeploy, LockDetect,$
$OnGroundDetect, AwaitLock, Wait\} \subseteq ActiveStates'$

*T*11*BVA*3

$\Delta Status$
Parameters

$\{ThrustLimitation, InadvertentDeploy, LockDetect,$
$OnGroundDetect, AwaitLock, Wait\} \subseteq ActiveStates$
$\boxed{Timer = 10}$
$Timer \leq 10$
$UnstowedSensor? = False$
$Timer' = Timer + 1$
$LimitThrust! = BUndefined$
$RestowComplete! = EUndefined$
$\{ThrustLimitation, InadvertentDeploy, LockDetect,$
$OnGroundDetect, AwaitLock, Wait\} \subseteq ActiveStates'$

A.3 Abstract states

The abstract states, calculated based on schema projection onto *Status* and *Status*$'$ of all test cases and operations are as follows. The states are calculated for each orthogonal component in the Statechart which have the following *Status* declarations:

$$States1 ::= Statechart1 \mid Idle \mid LockDetect \mid Unstowed \mid AwaitLock \mid Wait \mid Lock$$

$$States2 ::== Statechart2 \mid Idle \mid OnGroundDetect \mid InAir \mid OnGround$$

$$relation(configuration1\ \_)$$

$$\begin{aligned}
&configuration1\ _ == \\
&\{ActiveStates : \mathbb{F}_1\ States1 \mid \\
&\quad (Statechart1 \in ActiveStates \Rightarrow \\
&\qquad ((Idle \in ActiveStates \land \neg LockDetect \in ActiveStates) \lor \\
&\qquad (LockDetect \in ActiveStates \land \neg Idle \in ActiveStates))) \land \\
&\quad (LockDetect \in ActiveStates \Rightarrow \\
&\qquad ((Unstowed \in ActiveStates \land \neg AwaitLock \in ActiveStates) \lor \\
&\qquad (AwaitLock \in ActiveStates \land \neg Unstowed \in ActiveStates))) \land \\
&\quad (AwaitLock \in ActiveStates \Rightarrow \\
&\qquad ((Wait \in ActiveStates \land \neg Locked \in ActiveStates) \lor \\
&\qquad (Locked \in ActiveStates \land \neg Wait \in ActiveStates))) \land \\
&\quad (Idle \in ActiveStates \Rightarrow Statechart1 \in ActiveStates) \land \\
&\quad (LockDetect \in ActiveStates \Rightarrow Statechart1 \in ActiveStates) \land \\
&\quad (Unstowed \in ActiveStates \Rightarrow LockDetect \in ActiveStates) \land \\
&\quad (AwaitLock \in ActiveStates \Rightarrow LockDetect \in ActiveStates) \land \\
&\quad (Wait \in ActiveStates \Rightarrow AwaitLock \in ActiveStates) \land \\
&\quad (Locked \in ActiveStates \Rightarrow AwaitLock \in ActiveStates)\}
\end{aligned}$$

$$relation(configuration2\ \_)$$

$$\begin{aligned}
&configuration2\ _ == \\
&\{ActiveStates : \mathbb{F}_1\ States \mid \\
&\quad (Statechart2 \in ActiveStates \Rightarrow \\
&\qquad ((Idle \in ActiveStates \land \neg OnGroundDetect \in ActiveStates) \lor \\
&\qquad (OnGroundDetect \in ActiveStates \land \neg Idle \in ActiveStates))) \land \\
&\quad (OnGroundDetect \in ActiveStates \Rightarrow \\
&\qquad ((InAir \in ActiveStates \land \neg OnGround \in ActiveStates) \lor \\
&\qquad (OnGround \in ActiveStates \land \neg InAir \in ActiveStates))) \land \\
&\quad (Idle \in ActiveStates \Rightarrow Statechart2 \in ActiveStates) \land \\
&\quad (OnGroundDetect \in ActiveStates \Rightarrow Statechart2 \in ActiveStates) \land \\
&\quad (InAir \in ActiveStates \Rightarrow OnGroundDetect \in ActiveStates) \land \\
&\quad (OnGround \in ActiveStates \Rightarrow OnGroundDetect \in ActiveStates)\}
\end{aligned}$$

*Status*1

$ActiveStates1 : \mathbb{F}_1\ States1$
$Timer : \mathbb{Z}$

$configuration1(Statechart1, ActiveStates1, States1)$

*Status*2

$ActiveStates2 : \mathbb{F}_1\ States2$

$configuration2(Statechart2, ActiveStates2, States2)$

A.3.1 Statechart component 1

*State*1_1

*Status*1

$\{Statechart1, Idle\} \subseteq ActiveStates1$

*State*1_2

*Status*1

$\{Statechart1, LockDetect, Unstowed\} \subseteq ActiveStates1$

*State*1_3

*Status*1

$\{Statechart1, LockDetect, AwaitLock, Wait\} \subseteq ActiveStates1$
$Timer = 0$

*State*1_4

*Status*1

$\{Statechart1, LockDetect, AwaitLock, Wait\} \subseteq ActiveStates1$
$Timer \neq 0 \land Timer < 9$

*State*1_5

*Status*1

$\{Statechart1, LockDetect, AwaitLock, Wait\} \subseteq ActiveStates1$
$Timer = 9$

*State*1_6

*Status*1

$\{Statechart1, LockDetect, AwaitLock, Wait\} \subseteq ActiveStates1$
$Timer = 10$

*State*1_7

*Status*1

$\{Statechart1, LockDetect, AwaitLock, Wait\} \subseteq ActiveStates1$
$Timer = 11$

*State*1_8

*Status*1

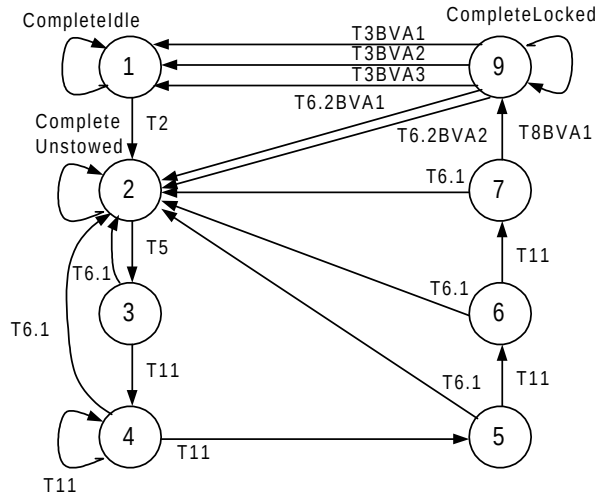
$\{Statechart1, LockDetect, AwaitLock, Wait\} \subseteq ActiveStates1$
$Timer > 11$

*State*1_9

*Status*1

$\{Statechart1, LockDetect, AwaitLock, Locked\} \subseteq ActiveStates1$

The minimal abstract finite state machine for component 1 is as follows:



A.3.2 Statechart component 2

*State*2_1

*Status*2

$\{Statechart2, Idle\} \subseteq ActiveStates2$

*State*2_2

*Status*2

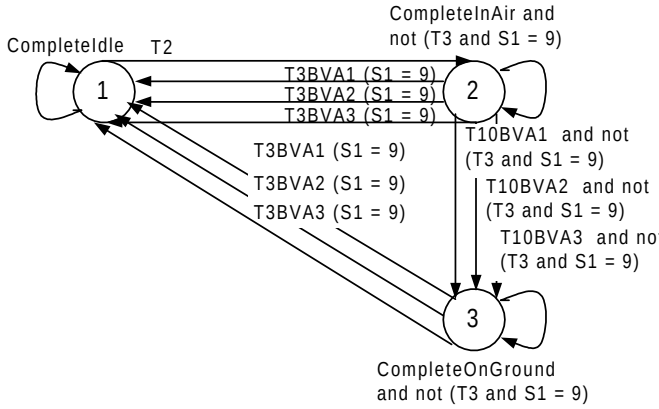
$\{Statechart2, OnGroundDetect, InAir\} \subseteq ActiveStates2$

*State*2_3

*Status*2

$\{Statechart2, OnGroundDetect, OnGround\} \subseteq ActiveStates2$

The minimal abstract finite state machine for component 2 is as follows:



A.4 UIO Sequences

Output of the FSM test sequence generating tool applied to *ThrustLimitation*:

```
   =============
Generating UIO sequences for machine:  Statechart2
The maximum length of the UIO sequences is 18 (2*n*2).
But using bound of 7.
-------------
Generating UIO sequence for state:  3 Found UIO Sequence
CompleteOnGroundDis2BVA1:
From 3 to 3
InadvertentDeploy:Present
UnstowedSensor:False
ThrottleReq:eq_FwdIdle
OnGround:True
when Statechart1 != 9
RestowComplete:EUndefined
LimitThrust:BUndefined
Freeing up Q...

-------------
Generating UIO sequence for state:  2
.
Found UIO Sequence
T10BVA1:
From 2 to 3
InadvertentDeploy:Present
ThrottleReq:eq_FwdIdle
OnGround:True
UnstowedSensor:True
```

```
when Statechart1 != 9
LimitThrust:False
RestowComplete:EUndefined
Freeing up Q...

-------------
Generating UIO sequence for state:  1
.
Found UIO Sequence
T2:
From 1 to 2
OnGround:False
ThrottleReq:lt_FwdIdle_Min_1
InadvertentDeploy:Present
UnstowedSensor:True
RestowComplete:EUndefined
LimitThrust:True
Freeing up Q...

==============
Generating UIO sequences for machine:  Statechart1
The maximum length of the UIO sequences is 128 (2*n*2).
But using bound of 7.
-------------
Generating UIO sequence for state:  9

Found UIO Sequence
T3BVA3:
From 9 to 1
InadvertentDeploy:Present
UnstowedSensor:False
ThrottleReq:eq_FwdIdle
OnGround:True
LimitThrust:False
RestowComplete:Present
Freeing up Q...

-------------
Generating UIO sequence for state:  7
.
Found UIO Sequence
T8BVA1
From 7 to 9
InadvertentDeploy:Present ThrottleReq:lt_FwdIdle_Min_1
UnstowedSensor:False
OnGround:True
RestowComplete:EUndefined
```

```
LimitThrust:BUndefined
T3BVA3:
From 9 to 1
InadvertentDeploy:Present
UnstowedSensor:False
ThrottleReq:eq_FwdIdle
OnGround:True
LimitThrust:False
RestowComplete:Present
Freeing up Q...

-------------
Generating UIO sequence for state:  6
..
Found UIO Sequence
T11
From 6 to 7
InadvertentDeploy:Present
ThrottleReq:lt_FwdIdle_Min_1
UnstowedSensor:False
OnGround:True
LimitThrust:BUndefined
RestowComplete:EUndefined
T8BVA1
From 7 to 9
InadvertentDeploy:Present
ThrottleReq:lt_FwdIdle_Min_1
UnstowedSensor:False
OnGround:True
RestowComplete:EUndefined
LimitThrust:BUndefined
T3BVA3:
From 9 to 1
InadvertentDeploy:Present
UnstowedSensor:False
ThrottleReq:eq_FwdIdle
OnGround:True
LimitThrust:False
RestowComplete:Present
Freeing up Q...

-------------
Generating UIO sequence for state:  5
...
Found UIO Sequence
T11
From 5 to 6
```

```
InadvertentDeploy:Present
ThrottleReq:lt_FwdIdle_Min_1
UnstowedSensor:False
OnGround:True
RestowComplete:EUndefined
LimitThrust:BUndefined
T11
From 6 to 7
InadvertentDeploy:Present
ThrottleReq:lt_FwdIdle_Min_1
UnstowedSensor:False
OnGround:True
LimitThrust:BUndefined
RestowComplete:EUndefined
T8BVA1
From 7 to 9
InadvertentDeploy:Present
ThrottleReq:lt_FwdIdle_Min_1
UnstowedSensor:False
OnGround:True
RestowComplete:EUndefined
LimitThrust:BUndefined
T3BVA3:
From 9 to 1
InadvertentDeploy:Present
UnstowedSensor:False
ThrottleReq:eq_FwdIdle
OnGround:True
LimitThrust:False RestowComplete:Present
Freeing up Q...

-------------
Generating UIO sequence for state:  4
....
Found UIO Sequence
T11
From 4 to 5
InadvertentDeploy:Present
ThrottleReq:lt_FwdIdle_Min_1
UnstowedSensor:False
OnGround:True
RestowComplete:EUndefined
LimitThrust:BUndefined
T11
From 5 to 6
InadvertentDeploy:Present
ThrottleReq:lt_FwdIdle_Min_1
```

```
UnstowedSensor:False
OnGround:True
RestowComplete:EUndefined
LimitThrust:BUndefined
T11
From 6 to 7
InadvertentDeploy:Present
ThrottleReq:lt_FwdIdle_Min_1
UnstowedSensor:False
OnGround:True
LimitThrust:BUndefined
RestowComplete:EUndefined
T8BVA1
From 7 to 9
InadvertentDeploy:Present
ThrottleReq:lt_FwdIdle_Min_1
UnstowedSensor:False
OnGround:True
RestowComplete:EUndefined
LimitThrust:BUndefined
T3BVA3:
From 9 to 1
InadvertentDeploy:Present
UnstowedSensor:False
ThrottleReq:eq_FwdIdle
OnGround:True
LimitThrust:False
RestowComplete:Present Freeing up Q...

-------------
Generating UIO sequence for state:  3
.....
Found UIO Sequence
T11
From 3 to 4
InadvertentDeploy:Present
ThrottleReq:lt_FwdIdle_Min_1
UnstowedSensor:False
OnGround:True
RestowComplete:EUndefined
LimitThrust:BUndefined
T11
From 4 to 5
InadvertentDeploy:Present
ThrottleReq:lt_FwdIdle_Min_1
UnstowedSensor:False
OnGround:True
```

```
RestowComplete:EUndefined
LimitThrust:BUndefined
T11
From 5 to 6
InadvertentDeploy:Present
ThrottleReq:lt_FwdIdle_Min_1
UnstowedSensor:False
OnGround:True
RestowComplete:EUndefined
LimitThrust:BUndefined
T11
From 6 to 7
InadvertentDeploy:Present
ThrottleReq:lt_FwdIdle_Min_1
UnstowedSensor:False
OnGround:True
LimitThrust:BUndefined
RestowComplete:EUndefined
T8BVA1
From 7 to 9
InadvertentDeploy:Present
ThrottleReq:lt_FwdIdle_Min_1
UnstowedSensor:False
OnGround:True
RestowComplete:EUndefined
LimitThrust:BUndefined
T3BVA3:
From 9 to 1
InadvertentDeploy:Present
UnstowedSensor:False
ThrottleReq:eq_FwdIdle
OnGround:True
LimitThrust:False
RestowComplete:Present
Freeing up Q...

-------------
Generating UIO sequence for state:  2
.....
Found UIO Sequence
T5
From 2 to 3
InadvertentDeploy:Absent
ThrottleReq:gt_FwdIdle_Plus_1
UnstowedSensor:False
OnGround:True
RestowComplete:EUndefined
```

```
LimitThrust:BUndefined
T11
From 3 to 4
InadvertentDeploy:Present
ThrottleReq:lt_FwdIdle_Min_1
UnstowedSensor:False
OnGround:True
RestowComplete:EUndefined
LimitThrust:BUndefined
T11
From 4 to 5
InadvertentDeploy:Present
ThrottleReq:lt_FwdIdle_Min_1
UnstowedSensor:False
OnGround:True
RestowComplete:EUndefined
LimitThrust:BUndefined
T11
From 5 to 6
InadvertentDeploy:Present
ThrottleReq:lt_FwdIdle_Min_1
UnstowedSensor:False
OnGround:True
RestowComplete:EUndefined
LimitThrust:BUndefined
T11
From 6 to 7
InadvertentDeploy:Present
ThrottleReq:lt_FwdIdle_Min_1
UnstowedSensor:False
OnGround:True
LimitThrust:BUndefined
RestowComplete:EUndefined
T8BVA1
From 7 to 9
InadvertentDeploy:Present
ThrottleReq:lt_FwdIdle_Min_1
UnstowedSensor:False
OnGround:True
RestowComplete:EUndefined
LimitThrust:BUndefined
T3BVA3:
From 9 to 1
InadvertentDeploy:Present
UnstowedSensor:False
ThrottleReq:eq_FwdIdle
OnGround:True
```

```
LimitThrust:False
RestowComplete:Present
Freeing up Q...

-------------
Generating UIO sequence for state:  1
.
Found UIO Sequence
T2:
From 1 to 2
OnGround:False
ThrottleReq:lt_FwdIdle_Min_1
InadvertentDeploy:Present
UnstowedSensor:True
RestowComplete:EUndefined
LimitThrust:True
Freeing up Q...
```

A.5 SMV specification

```
-- Generated by FSM
MODULE main

VAR

Statechart2:  {3,2,1};
Statechart1:  {9,7,6,5,4,3,2,1};
ThrottleReq:  {gt_FwdIdle_Plus_1,eq_FwdIdle_Plus_1,eq_FwdIdle,eq_FwdIdle_Min_1,
lt_FwdIdle_Min_1};
UnstowedSensor:  {True,False};
OnGround:  {True,False};
InadvertentDeploy:  {Absent,Present};
LimitThrust1:  {BUndefined,True,False};
LimitThrust2:  {BUndefined,True,False};
RestowComplete:  {EUndefined,Absent,Present};

ASSIGN
init(Statechart2) := 1;
init(Statechart1) := 1;

next( Statechart2):= case
--CompleteOnGroundDis2BVA1
Statechart2 = 3 & (ThrottleReq = eq_FwdIdle) & !(Statechart1 = 9):  3;
--CompleteOnGroundDis2BVA2
Statechart2 = 3 & (ThrottleReq = eq_FwdIdle_Min_1) & !(Statechart1 = 9):  3;
--CompleteOnGroundDis2BVA1
Statechart2 = 3 & (ThrottleReq = lt_FwdIdle_Min_1) & !(Statechart1 = 9):  3;
--CompleteOnGroundDis1BVA2
Statechart2 = 3 & (ThrottleReq = gt_FwdIdle_Plus_1):  3;
--CompleteOnGroundDis1BVA1
Statechart2 = 3 & (ThrottleReq = eq_FwdIdle_Plus_1):  3;
--CompleteInAirDis2BVA3
Statechart2 = 2 & (OnGround = False) & (ThrottleReq = eq_FwdIdle)
& !(Statechart1 = 9):  2;
--CompleteInAirDis2BVA2
Statechart2 = 2 & (OnGround = False) & (ThrottleReq = eq_FwdIdle_Min_1)
& !(Statechart1 = 9):  2;
--CompleteInAirDis2BVA1
Statechart2 = 2 & (OnGround = False) & (ThrottleReq = lt_FwdIdle_Min_1)
& !(Statechart1 = 9):  2;
--CompleteInAirDis1BVA2
Statechart2 = 2 & (ThrottleReq = gt_FwdIdle_Plus_1):  2;
--CompleteInAirDis1BVA1
Statechart2 = 2 & (ThrottleReq = eq_FwdIdle_Plus_1):  2;
--T3BVA3
```

```
Statechart2 = 3 & (ThrottleReq = eq_FwdIdle) & Statechart1 = 9:  1;
--T3BVA2
Statechart2 = 3 & (ThrottleReq = eq_FwdIdle_Min_1) & Statechart1 = 9:  1;
--T3BVA1
Statechart2 = 3 & (ThrottleReq = lt_FwdIdle_Min_1) & Statechart1 = 9:  1;
--T3BVA3
Statechart2 = 2 & (ThrottleReq = eq_FwdIdle) & Statechart1 = 9:  1;
--T3BVA2
Statechart2 = 2 & (ThrottleReq = eq_FwdIdle_Min_1) & Statechart1 = 9:  1;
--T3BVA1
Statechart2 = 2 & (ThrottleReq = lt_FwdIdle_Min_1) & Statechart1 = 9:  1;
--T10BVA1
Statechart2 = 2 & (OnGround = True) & (ThrottleReq = eq_FwdIdle)
& !(Statechart1 = 9):  3;
--T10BVA2
Statechart2 = 2 & (OnGround = True) & (ThrottleReq = eq_FwdIdle_Min_1)
& !(Statechart1 = 9):  3;
--T10BVA1
Statechart2 = 2 & (OnGround = True) & (ThrottleReq = lt_FwdIdle_Min_1)
& !(Statechart1 = 9):  3;
--CompleteIdle
Statechart2 = 1 & (InadvertentDeploy = Absent):  1;
--T2
Statechart2 = 1 & (InadvertentDeploy = Present):  2;
esac;

next( Statechart1):= case
--CompleteLockedBVA2
Statechart1 = 9 & (ThrottleReq = gt_FwdIdle_Plus_1)
& (UnstowedSensor = False):  9;
--CompleteLockedBVA1
Statechart1 = 9 & (ThrottleReq = eq_FwdIdle_Plus_1)
& (UnstowedSensor = False):  9;
--CompleteUnstowed
Statechart1 = 2 & (UnstowedSensor = True):  2;
--CompleteIdle
Statechart1 = 1 & (InadvertentDeploy = Absent):  1;
--T2
Statechart1 = 1 & (InadvertentDeploy = Present):  2;
--T3BVA3
Statechart1 = 9 & (ThrottleReq = eq_FwdIdle):  1;
--T3BVA2
Statechart1 = 9 & (ThrottleReq = eq_FwdIdle_Min_1):  1;
--T3BVA1
Statechart1 = 9 & (ThrottleReq = lt_FwdIdle_Min_1):  1;
--T5
Statechart1 = 2 & (UnstowedSensor = False):  3;
```

```
--T6.1
Statechart1 = 7 & (UnstowedSensor = True):  2;
--T6.1
Statechart1 = 6 & (UnstowedSensor = True):  2;
--T6.1
Statechart1 = 5 & (UnstowedSensor = True):  2;
--T6.1
Statechart1 = 4 & (UnstowedSensor = True):  2;
--T6.1
Statechart1 = 3 & (UnstowedSensor = True):  2;
--T6.2BVA2
Statechart1 = 9 & (ThrottleReq = gt_FwdIdle_Plus_1)
& (UnstowedSensor = True):  2;
--T6.2BVA1
Statechart1 = 9 & (ThrottleReq = eq_FwdIdle_Plus_1)
& (UnstowedSensor = True):  2;
--T11
Statechart1 = 6 & (UnstowedSensor = False):  7;
--T11
Statechart1 = 5 & (UnstowedSensor = False):  6;
--T11
Statechart1 = 4 & (UnstowedSensor = False):  5;
--T11
Statechart1 = 3 & (UnstowedSensor = False):  4;
--T8BVA1
Statechart1 = 7 & (UnstowedSensor = False):  9;
esac;

next( LimitThrust1):= case
--CompleteLockedBVA2
Statechart1 = 9 & (ThrottleReq = gt_FwdIdle_Plus_1)
& (UnstowedSensor = False):  BUndefined;
--CompleteLockedBVA1
Statechart1 = 9 & (ThrottleReq = eq_FwdIdle_Plus_1)
& (UnstowedSensor = False):  BUndefined;
--CompleteUnstowed
Statechart1 = 2 & (UnstowedSensor = True):  BUndefined;
--CompleteIdle
Statechart1 = 1 & (InadvertentDeploy = Absent):  BUndefined;
--T2
Statechart1 = 1 & (InadvertentDeploy = Present):  True;
--T3BVA3
Statechart1 = 9 & (ThrottleReq = eq_FwdIdle):  False;
--T3BVA2
Statechart1 = 9 & (ThrottleReq = eq_FwdIdle_Min_1):  False;
--T3BVA1
Statechart1 = 9 & (ThrottleReq = lt_FwdIdle_Min_1):  False;
```

```
--T5
Statechart1 = 2 & (UnstowedSensor = False):  BUndefined;
--T6.1
Statechart1 = 7 & (UnstowedSensor = True):  BUndefined;
--T6.1
Statechart1 = 6 & (UnstowedSensor = True):  BUndefined;
--T6.1
Statechart1 = 5 & (UnstowedSensor = True):  BUndefined;
--T6.1
Statechart1 = 4 & (UnstowedSensor = True):  BUndefined;
--T6.1
Statechart1 = 3 & (UnstowedSensor = True):  BUndefined;
--T6.2BVA2
Statechart1 = 9 & (ThrottleReq = gt_FwdIdle_Plus_1)
& (UnstowedSensor = True):  BUndefined;
--T6.2BVA1
Statechart1 = 9 & (ThrottleReq = eq_FwdIdle_Plus_1)
& (UnstowedSensor = True):  BUndefined;
--T11
Statechart1 = 6 & (UnstowedSensor = False):  BUndefined;
--T11
Statechart1 = 5 & (UnstowedSensor = False):  BUndefined;
--T11
Statechart1 = 4 & (UnstowedSensor = False):  BUndefined;
--T11
Statechart1 = 3 & (UnstowedSensor = False):  BUndefined;
--T8BVA1
Statechart1 = 7 & (UnstowedSensor = False):  BUndefined;
esac;

next( LimitThrust2):= case
--CompleteOnGroundDis2BVA1
Statechart2 = 3 & (ThrottleReq = eq_FwdIdle)
& !(Statechart1 = 9):  BUndefined;
--CompleteOnGroundDis2BVA2
Statechart2 = 3 & (ThrottleReq = eq_FwdIdle_Min_1)
& !(Statechart1 = 9):  BUndefined;
--CompleteOnGroundDis2BVA1
Statechart2 = 3 & (ThrottleReq = lt_FwdIdle_Min_1)
& !(Statechart1 = 9):  BUndefined;
--CompleteOnGroundDis1BVA2
Statechart2 = 3 & (ThrottleReq = gt_FwdIdle_Plus_1):  BUndefined;
--CompleteOnGroundDis1BVA1
Statechart2 = 3 & (ThrottleReq = eq_FwdIdle_Plus_1):  BUndefined;
--CompleteInAirDis2BVA3
Statechart2 = 2 & (OnGround = False) & (ThrottleReq = eq_FwdIdle)
& !(Statechart1 = 9):  BUndefined;
```

```
--CompleteInAirDis2BVA2
Statechart2 = 2 & (OnGround = False) & (ThrottleReq = eq_FwdIdle_Min_1)
& !(Statechart1 = 9):  BUndefined;
--CompleteInAirDis2BVA1
Statechart2 = 2 & (OnGround = False) & (ThrottleReq = lt_FwdIdle_Min_1)
& !(Statechart1 = 9):  BUndefined;
--CompleteInAirDis1BVA2
Statechart2 = 2 & (ThrottleReq = gt_FwdIdle_Plus_1):  BUndefined;
--CompleteInAirDis1BVA1
Statechart2 = 2 & (ThrottleReq = eq_FwdIdle_Plus_1):  BUndefined;
--T3BVA3
Statechart2 = 3 & (ThrottleReq = eq_FwdIdle) & Statechart1 = 9:  False;
--T3BVA2
Statechart2 = 3 & (ThrottleReq = eq_FwdIdle_Min_1) & Statechart1 = 9:  False;
--T3BVA1
Statechart2 = 3 & (ThrottleReq = lt_FwdIdle_Min_1) & Statechart1 = 9:  False;
--T3BVA3
Statechart2 = 2 & (ThrottleReq = eq_FwdIdle) & Statechart1 = 9:  False;
--T3BVA2
Statechart2 = 2 & (ThrottleReq = eq_FwdIdle_Min_1) & Statechart1 = 9:  False;
--T3BVA1
Statechart2 = 2 & (ThrottleReq = lt_FwdIdle_Min_1) & Statechart1 = 9:  False;
--T10BVA1
Statechart2 = 2 & (OnGround = True) & (ThrottleReq = eq_FwdIdle)
& !(Statechart1 = 9):  False;
--T10BVA2
Statechart2 = 2 & (OnGround = True) & (ThrottleReq = eq_FwdIdle_Min_1)
& !(Statechart1 = 9):  False;
--T10BVA1
Statechart2 = 2 & (OnGround = True) & (ThrottleReq = lt_FwdIdle_Min_1)
& !(Statechart1 = 9):  False; --CompleteIdle
Statechart2 = 1 & (InadvertentDeploy = Absent):  BUndefined;
--T2
Statechart2 = 1 & (InadvertentDeploy = Present):  True;
esac;

next( RestowComplete):= case
--CompleteOnGroundDis2BVA1
Statechart2 = 3 & (ThrottleReq = eq_FwdIdle) & !(Statechart1 = 9):  EUndefined;
--CompleteOnGroundDis2BVA2
Statechart2 = 3 & (ThrottleReq = eq_FwdIdle_Min_1)
& !(Statechart1 = 9):  EUndefined;
--CompleteOnGroundDis2BVA1
Statechart2 = 3 & (ThrottleReq = lt_FwdIdle_Min_1)
& !(Statechart1 = 9):  EUndefined;
--CompleteOnGroundDis1BVA2
Statechart2 = 3 & (ThrottleReq = gt_FwdIdle_Plus_1):  EUndefined;
```

```
--CompleteOnGroundDis1BVA1
Statechart2 = 3 & (ThrottleReq = eq_FwdIdle_Plus_1):  EUndefined;
--CompleteInAirDis2BVA3
Statechart2 = 2 & (OnGround = False) & (ThrottleReq = eq_FwdIdle)
& !(Statechart1 = 9):  EUndefined;
--CompleteInAirDis2BVA2
Statechart2 = 2 & (OnGround = False) & (ThrottleReq = eq_FwdIdle_Min_1)
& !(Statechart1 = 9):  EUndefined;
--CompleteInAirDis2BVA1
Statechart2 = 2 & (OnGround = False) & (ThrottleReq = lt_FwdIdle_Min_1)
& !(Statechart1 = 9):  EUndefined;
--CompleteInAirDis1BVA2
Statechart2 = 2 & (ThrottleReq = gt_FwdIdle_Plus_1):  EUndefined;
--CompleteInAirDis1BVA1
Statechart2 = 2 & (ThrottleReq = eq_FwdIdle_Plus_1):  EUndefined;
--T3BVA3
Statechart2 = 3 & (ThrottleReq = eq_FwdIdle) & Statechart1 = 9:  Present;
--T3BVA2
Statechart2 = 3 & (ThrottleReq = eq_FwdIdle_Min_1) & Statechart1 = 9:  Present;
--T3BVA1
Statechart2 = 3 & (ThrottleReq = lt_FwdIdle_Min_1) & Statechart1 = 9:  Present;
--T3BVA3
Statechart2 = 2 & (ThrottleReq = eq_FwdIdle) & Statechart1 = 9:  Present;
--T3BVA2
Statechart2 = 2 & (ThrottleReq = eq_FwdIdle_Min_1) & Statechart1 = 9:  Present;
--T3BVA1
Statechart2 = 2 & (ThrottleReq = lt_FwdIdle_Min_1) & Statechart1 = 9:  Present;
--T10BVA1
Statechart2 = 2 & (OnGround = True) & (ThrottleReq = eq_FwdIdle)
& !(Statechart1 = 9):  EUndefined;
--T10BVA2
Statechart2 = 2 & (OnGround = True) & (ThrottleReq = eq_FwdIdle_Min_1)
& !(Statechart1 = 9):  EUndefined;
--T10BVA1
Statechart2 = 2 & (OnGround = True) & (ThrottleReq = lt_FwdIdle_Min_1)
& !(Statechart1 = 9):  EUndefined;
--CompleteIdle
Statechart2 = 1 & (InadvertentDeploy = Absent):  EUndefined;
--T2
Statechart2 = 1 & (InadvertentDeploy = Present):  EUndefined;
--CompleteLockedBVA2
Statechart1 = 9 & (ThrottleReq = gt_FwdIdle_Plus_1)
& (UnstowedSensor = False):  EUndefined;
--CompleteLockedBVA1
Statechart1 = 9 & (ThrottleReq = eq_FwdIdle_Plus_1)
& (UnstowedSensor = False):  EUndefined;
--CompleteUnstowed
```

```
Statechart1 = 2 & (UnstowedSensor = True):  EUndefined;
--CompleteIdle
Statechart1 = 1 & (InadvertentDeploy = Absent):  EUndefined;
--T2
Statechart1 = 1 & (InadvertentDeploy = Present):  EUndefined;
--T3BVA3
Statechart1 = 9 & (ThrottleReq = eq_FwdIdle):  Present;
--T3BVA2
Statechart1 = 9 & (ThrottleReq = eq_FwdIdle_Min_1):  Present;
--T3BVA1
Statechart1 = 9 & (ThrottleReq = lt_FwdIdle_Min_1):  Present;
--T5
Statechart1 = 2 & (UnstowedSensor = False):  EUndefined;
--T6.1
Statechart1 = 7 & (UnstowedSensor = True):  EUndefined;
--T6.1
Statechart1 = 6 & (UnstowedSensor = True):  EUndefined;
--T6.1
Statechart1 = 5 & (UnstowedSensor = True):  EUndefined;
--T6.1
Statechart1 = 4 & (UnstowedSensor = True):  EUndefined;
--T6.1
Statechart1 = 3 & (UnstowedSensor = True):  EUndefined;
--T6.2BVA2
Statechart1 = 9 & (ThrottleReq = gt_FwdIdle_Plus_1) & (UnstowedSensor = True):
EUndefined;
--T6.2BVA1
Statechart1 = 9 & (ThrottleReq = eq_FwdIdle_Plus_1) & (UnstowedSensor = True):
EUndefined;
--T11
Statechart1 = 6 & (UnstowedSensor = False):  EUndefined;
--T11
Statechart1 = 5 & (UnstowedSensor = False):  EUndefined;
--T11
Statechart1 = 4 & (UnstowedSensor = False):  EUndefined;
--T11
Statechart1 = 3 & (UnstowedSensor = False):  EUndefined;
--T8BVA1
Statechart1 = 7 & (UnstowedSensor = False):  EUndefined;
esac;

INVAR
((!(LimitThrust1 = BUndefined) & !(LimitThrust2 = BUndefined)) ->
(LimitThrust1 = LimitThrust2))


SPEC
```

```
--!EF( !(LimitThrust1 = LimitThrust2) & !(LimitThrust1 = BUndefined)
& !(LimitThrust2 = BUndefined))
!EF( Statechart1 = 9)
```

Appendix B

Efficient testing heuristics

Efficient fault-based testing heuristic for $\neg x$:

$$EfficientNeg1 == \exists\, x, z : Boolean \bullet \neg x \land \neg z$$

$$EfficientNeg2 == \exists\, x, z : Boolean \bullet \neg x \land z$$

Negative testing should ensure the condition $x \land z$ is covered.

Efficient fault-based testing heuristic for $x \land y$:

$$EfficientCon == \exists\, x, y, z : Boolean \bullet x \land y \land \neg z$$

Negative testing should ensure the conditions $\neg x$ and $\neg y$ are covered.

Efficient fault-based testing heuristic for $x \lor y$:

$$EfficientDis1 == \exists\, x, y, z : Boolean \bullet x \land \neg y \land \neg z$$

$$EfficientDis2 == \exists\, x, y, z : Boolean \bullet \neg x \land y \land \neg z$$

Negative testing should ensure the condition $\neg x \land \neg y \land z$ is covered.

Efficient fault-based testing heuristic for $x = y$:

$$EfficientEQ == \exists\, x, y, z : \mathbb{R} \bullet x \neq z \land x = y$$

Negative testing should ensure the conditions $x < y$ and $x > y$ are covered.

Efficient fault-based testing heuristic for $x \neq y$:

$$EfficientNEQ1 == \exists x, y, z : \mathbb{R} \bullet x \neq z \land y \neq z \land x > y$$

$$EfficientNEQ2 == \exists x, y : \mathbb{R} \bullet x < y$$

Negative testing should ensure the condition $x = y$ is covered.

Efficient fault-based testing heuristics for $x < y$:

$$EfficientLT1 == \exists x, y, z : \mathbb{R} \bullet x \neq z \land x = y - \delta \land z < y$$

Negative testing should ensure the conditions $x = y \land z > y$ and $x > y$ are covered.

Efficient fault-based testing heuristics for $x > y$:

$$EfficientGT1 == \exists x, y, z : \mathbb{R} \bullet x \neq z \land x = y + \delta \land z > y$$

Negative testing should ensure the conditions $x = y \land z < y$ and $x < y$ are covered.

Efficient fault-based testing heuristic for $x \geq y$:

$$EfficientGTE1 == \exists x, y, z : \mathbb{R} \bullet x = y \land z > y$$

$$EfficientGTE2 == \exists x, y : \mathbb{R} \bullet x > y$$

Negative testing should ensure the condition $x = y - \delta \land z < y$ is covered.

Efficient fault-based testing heuristic for $x \leq y$:

$$EfficientLTE1 == \exists x, y, z : \mathbb{R} \bullet x = y \land z < y$$

$$EfficientLTE2 == \exists x, y : \mathbb{R} \bullet x < y$$

Negative testing should ensure the condition $x = y + \delta \land z > y$ is covered.

Efficient fault-based testing heuristic for $x + y$:

$$\begin{array}{l} \exists x, y, z : \mathbb{R} \bullet y \neq z \land \\ x \neq z \land \\ y \neq z \land \\ y \neq 0 \land \\ x \neq 0 \land \\ x \neq (x * y) - y \land \\ x \neq (x \div y) - y \land \\ z \neq 0 \land \\ z \neq 1 \land \\ z \neq (x + y) * (x + y) \land \\ z \neq y * y \land \\ z \neq x * x \land \\ x + y \neq 0 \end{array}$$

Efficient fault-based testing heuristic for $x - y$:

$$
\begin{array}{l}
\exists x, y, z : \mathbb{R} \bullet y \neq z \land \\
x \neq z \land \\
y \neq z \land \\
y \neq 0 \land \\
x \neq 0 \land \\
x \neq y \land \\
x - y \neq 0 \land \\
x \neq (x * y) + y \land \\
x \neq (x \div y) + y \land \\
z \neq 0 \land \\
z \neq 1 \land \\
z \neq (x - y) * (x - y) \land \\
z \neq y * y \land \\
z \neq x * x \land \\
x \neq 2 * y
\end{array}
$$

Efficient fault-based testing heuristic for $x * y$:

$$
\begin{array}{l}
\exists x, y, z : \mathbb{R} \bullet y \neq z \land \\
x \neq z \land \\
y \neq z \land \\
x \neq 0 \land \\
y \neq 0 \land \\
z \neq 0 \land \\
x \neq x * y * y \land \\
y \neq x * x * y \land \\
x \neq (x * y) - y \land \\
x \neq (x * y) + y \land \\
x \neq 1 \land \\
y \neq 1 \land \\
z \neq 1 \land \\
z \neq 2 * x * y \land \\
z \neq 2 * y \land \\
z \neq 2 * x \land \\
z \neq (x * y) * (x * y)
\end{array}
$$

Efficient fault-based testing heuristic for $x \div y$:

$$
\begin{array}{l}
\exists\, x, y, z : \mathbb{R} \bullet y \neq z \land \\
x \neq z \land \\
y \neq z \land \\
x \neq 0 \land \\
y \neq 0 \land \\
z \neq 0 \land \\
x \neq y * y \land \\
z \neq 2 * x \div y \land \\
z \neq 2 * y \land \\
z \neq 2 * x \land \\
y \neq 1 \land \\
z \neq 1
\end{array}
$$

Bibliography

[AA92] Nina Amla and Paul Ammann. Using Z specifications in category partition testing. *Proceeding of COMPASS 1992, Seventh Annual Conference On Computer Assurance*, pages 3–10, 1992.

[AB96] Jim Armstrong and Leonor Barroca. Specification and verification of reactive system behaviour: The railroad crossing example. *Real-Time Systems*, 10:143–178, 1996.

[AB99] P. Ammann and P. Black. A specification-based coverage metric to evaluate test sets. In *Proceedings of Fourth IEEE International High-Assurance Systems Engineering Symposium (HASE 99)*, November 1999.

[ABB+01] Karen Allenby, Simon Burton, Darren Buttle, John McDermid, Alan Stephenson, Mike Bardill, and Stuart Hutchesson. A family-oriented software development process for engine controllers. In *Proceedings of the 3rd International Conference on Product Focused Software Process Improvement*, September 2001.

[ABM98] P. Ammann, P. Black, and W. Majurski. Using model checking to generate tests from specications. In *In Proceedings of the 2nd IEEE International Conference on Formal Engineering Methods (ICFEM'98)*, December 1998.

[ADLU91] Alfred Aho, Anton Dahbura, David Lee, and Umit Uyar. An optimization technique for protocol conformance test generation based on UIO sequences and rural chinese postman tours. *IEEE Transactions On Communications*, 39(11):1604–1615, November 1991.

[Arm98] Jim Armstrong. Industrial integration of graphical and formal specifications. *Systems Software*, 40:211–225, 1998.

[AW96] S.P. Allen and M.R. Woodward. Assessing the quality of specification-based testing. In Sandro Bologna and Giacomo Bucci, editors, *Proceedings of the Third International Conference on Achieving Quality in Software*, pages 341–354. Chapman and Hall, 1996.

[AY98] R. Alur and M. Yannakakis. Model checking of hierarchical state machines. In *Proceedings of the 6th ACM Symposium on Foundations of Software engineering*, 1998.

[Bar97] John Barnes. *High Integrity Ada - The SPARK Approach*. Addison-Wesley, 1997.

[BBD+99] T. Bienmueller, U. Brockmeyer, W. Damm, G. Doehmen, C. Essman, H-J. Holberg, H. Hungar, B. Josko, R. Schloer, G. Wittich, H. Wittke, G. Clements, J. Rowlands, and E. Sefton. Formal verification of an avionics application using abstraction and symbolic model checking. In *Towards System Safety – Proceedings of the Seventh Safety-critical Systems Symposium*, 1999.

[BCGM00] Simon Burton, John Clark, Andy Galloway, and John McDermid. Automated V&V for high integrity systems, a targeted formal methods approach. In *Proceedings of the 5th NASA Langley Formal Methods Workshop*, June 2000.

[Bee94] Michael von der Beeck. A comparision of Statechart variants. In de Roever Langmaack and Vytopil, editors, *Formal Techniques in Real Time and Fault-Tolerant Systems, Lecture Notes in Computer Science, vol. 863*, pages 128–148. Springer, 1994.

[Bei90] Boris Beizer. *Software Testing Techniques*. Thomson Computer Press, 1990.

[Ber00] Sergey Berezin. The SMV web site. http://www.cs.cmu.edu/~modelcheck/smv.html/, 2000. The latest version of SMV and its documentation may be downloaded from this site.

[BG81] Timothy A. Budd and Ajei S. Gopal. Program testing by specification mutation. *Computer Program Testing*, pages 129–148, 1981.

[BGK98] Robert Buessow, Robert Geisler, and Marcus 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.

[BGM91] G. Bernot, M. Gaudel, and B. Marre. Software testing based on formal specifications: a theory and a tool. *Software Engineering Journal*, 6(6):387–405, 1991.

[BHS98] K. Bogdanov, M. Holcombe, and H. Singh. Automated test set generation for statecharts. In *Proceedings International Workshop on Current Trends in Applied Formal Methods*, 1998.

[Bud85] Timothy A. Budd. Mutation analysis: Ideas, examples, problems and prospects. *Computer Languages Program Testing*, 10(1):63–73, 1985.

[Bur00] Simon Burton. Feasibility study on the use of matlab/stateflow for developing state-based specifications. Technical report, Rolls-Royce UTC, Department of Computer Science, University of York, 2000.

[Cho78] Tsun S. Chow. Testing software design modeled by finite-state machines. *IEEE Transactions On Software Engineering*, SE-4(3):178–187, May 1978.

[CK96] E. Clarke and R. Kurshan. Computer-aided verification. *IEEE Spectrum*, 33(6):61–67, 1996.

[Con01] World Wide Web Consortium. eXtensible Markup Language. http://w3c.org/XML/, 2001.

[Cor99] Rational Software Corporation. Unified modeling language. Unified Modelling Language. Available at http://www.rational.com, 1999.

[CS94] David Carrington and Phil Stocks. A tale of two paradigms: Formal methods and software testing. Technical report 94-4, Department of Computer Science, University of Queensland, Febuary 1994.

[CSE96] J. Callahan, F. Schneider, and S. Easterbrook. Automated software testing using model checking. In *Proceedings 1996 SPIN Workshop*, August 1996. Also WVU Technical Report NASA-IVV-96-022.

[CT97] John Clark and Nigel Tracey. Solving constraints in LAW. LAW/D5.1.1(E), European Commission - DG III Industry, 1997. Legacy Assessment Work-Bench Feasibility Assessment.

[CW96] E.M. Clarke and J.M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 1996.

[CZ93] Samuel T. Chanson and Jinsong Zhu. A unified approach to protocol test sequence generation. In *IEEE INFOCOM'93*, pages 106–114, 1993.

[DF93] J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specifications. pages 268–284. Springer Verlag, April 1993.

[DJHP97] W. Damm, B. Josko, H. Hungar, and A. Pnueli. A compositional real-time semantics of STATEMATE designs. In *Compositionality: The Significant Difference, volume 1536 of LNCS*. Springer Verlag, 1997.

[DLS78] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, pages 34–41, April 1978.

[DR96] David Dill and John Rushby. Acceptance of formal methods: Lessons from hardware design. *IEEE Computer*, 29(4):23–24, April 1996.

[FvBK+91] Susumu Fujiwara, Gregor von Bochmann, Ferhat Khendek, Mokhtar Amalou, and Abderrazak Ghedamsi. Test selection based on finite-state machines. *IEEE Transactions On Software Engineering*, 17(6):591–603, June 1991.

[GCM98] 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.

[GG75] 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.

[GG93] Matthias Grochtmann and Klaus Grimm. Classification trees for partition testing. *Software Testing, Verification and Reliability*, 3:63–82, 1993.

[GHD98] Wolfgang Grieskamp, Maritta Heisel, and Heiko 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.

[God90] Patrice Godefroid. Using partial orders to improve automatic verification. *Proceedings of the Second Workshop On Computer-Aided Verification*, pages 176–185, 1990.

[Hal88] P. A. V. Hall. Towards testing with respect to formal specifications. *Second IEE/BCS Conference On Software Engineering*, pages 159–163, 1988.

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[Hay86] Ian Hayes. Specification directed module testing. *IEEE Transactions On Software Engineering*, 12(1):124–133, January 1986.

[HB95] M.G. Hinchey and J. P. Bowen. *Applications of formal methods*. Prentice-Hall, 1995.

[HB96] C. Michael Holloway and Ricky W. Butler. Impediments to industrial use of formal methods. *IEEE Computer*, 29(4):25–26, April 1996.

[Hen80] Kathryn L. Heninger. Specifying software requirements for complex systems: New techniques and their applications. *IEEE Transactions on Software Engineering*, 6(1), 1980.

[HGP92] Gerard Holzmann, Patrice Godefroid, and Didier Pirottin. Coverage preserving reduction strategies for reachability analysis. *IFIP Transactions C-Communication Systems*, 8:349–363, 1992.

[Hie97a] R. M. Hierons. Testing from a finite state machine: Extending invertibility to sequences. *The Computer Journal*, 40(4):220–230, 1997.

[Hie97b] R. M. Hierons. Testing from semi-independent communicating finite state machines. *IEE Proceedings In Software Engineering*, 144(5-6):291–295, 1997.

[Hie01] R. M. Hierons. Checking states and transitions of a set of communicating finite state machines. *Microprocessors and Microsystems, Special Issue on Testing and testing techniques for real-time embedded software systems*, 24(9):443–452, 2001.

[HL96] Mats Heimdahl and Nancy Leveson. Completeness and consistency in hierarchical state-based requirements. *IEEE Transactions On Software Engineering*, 22(6):363–377, June 1996.

[HLN+88] 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.

[HN96] David Harel and Amnon Naamad. The Statemate semantics of statecharts. *IEEE Transactions On Software Engineering And Methodology*, 5(4):293–33, Oct 1996.

[Hol97] G. J. Holzmann. The model checker spin. *IEEE Transactions On Software Engineering, Special Issue on Formal Methods in Software Practice*, 23(5):279–295, May 1997.

[How82] W. E. Howden. Weak mutation testing and completeness of test sets. *IEEE Transactions On Software Engineering*, (8):371–379, 1982.

[HP95] Hans-Martin Hoercher and Jan Peleska. Using formal specifications to support software testing. *Software Quality Journal*, (4):309–327, 1995.

[HRdR92] J.J.M. Hooman, S. Ramesh, and W.P. de Roever. A compositional axiomatization of statecharts. *Theoretical Computer Science*, (101):289–335, 1992.

[HSS01] R. M. Hierons, S. Sadeghipour, and H. Singh. Testing a system specified using Statecharts and Z. *Information and Software Technology*, 43(3):137–149, 2001.

[IEC98] International Electrotechnical Commission. *IEC 61508, parts 1 to 7, Functional safety of electrical/electronic/programmable electronic safety-related systems*, 1998.

[IH96] F. Ipate and M. Holcombe. An integration testing method that is proved to find all faults. *International Journal of Computer Mathematics*, 63:159–178, 1996.

[Inc01] LINDO Systems Inc. Lingo. http://www.lindo.com, 2001.

[Inf97] Information Processing Ltd., Bath, UK. *AdaTEST 95 User Manual*, June 1997.

[Int96] The International Organization for Standardization. *Information technology – Programming languages, their environments and system software interfaces – Vienna Development Method – Specification Language – Part 1: Base language. International Standard ISO/IEC 13817-1*, December 1996.

[Int99] The International Organization for Standardization. *Z Notation, Final Committee Draft, CD 13568.2*, August 1999.

[JM94] F Jahanian and A Mok. Modecharts: A specification language for real-time systems. *IEEE Transactions On Software Engineering*, 20(12):933–947, December 1994.

[KDGN97] 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.

[KHCP00] Steve King, Jonathon Hammond, Rod Chapman, and Andy Pryor. Is proof more cost-effective than testing? *IEEE Transactions on Software Engineering*, 26(8):675–686, August 2000.

[Kua62] M. K. Kuan. Graphic programming using odd or even points. *Chinese Math*, 1:273–277, 1962.

[Kuh99] D. Richard Kuhn. Fault classes and error detection capability of specification-based testing. *ACM Transactions on Software Engineering and Methodology*, 8(4):411–424, October 1999.

[Lev95] Nancy Leveson. *Safeware: System Safety And Computers*. Addison Wesley, 1995.

[LHHR94] N. G. Leveson, M. P. E. Heimdahl, H. Hildreth, and J. D. Reese. Requirements specification for process control systems. *IEEE Transactions On Software Engineering*, 20(9):684–707, September 1994.

[Liv93] Liverpool Data Research Associates Ltd.,Liverpool, UK. *LDRA Testbed Technical Description*, 1993.

[LvBP94] Gang Luo, Gregor von Bochman, and Alexandre Petrenko. Test selection based on communicating non-deterministic finite state machines using a generalized wp-method. *IEEE Transactions On Software Engineering*, 20(2):149–161, February 1994.

[LvdBC99] Gerald Luettgen, Michael von der Beeck, and Rance Cleaveland. Statecharts via process algebra. In J.C.M. Baeten and S. Mauw, editors, *Proc. 10th International Conference on Concurrency Theory (CONCUR'99) Eindhoven, The Netherlands, Lecture Notes in Computer Science, vol. 1664*, pages 399–414, August 1999.

[LY94] David Lee and Mihalis Yannakakis. Testing finite-state machines: State identification and verification. *IEEE Transactions on Computers*, 43(3):306–320, March 1994.

[Mac77] A. K. Mackworth. Consistency in networks of relations. *Artifical Intellegence*, 8:99–118, 1977.

[Mar91] F. Maraninchi. The Argos language, graphical representation of automata and description of reactive systems. In *IEEE Workshop on Visual Languages*, October 1991.

[MB99] E.S. Mresa and L. Bottaci. Efficiency of mutation operators and selective mutation strategies: an empirical study. *The Journal of Software Testing, Verification and Reliability*, 9(4):205–232, December 1999.

[MC98] Ian MacColl and David Carrington. Extending the Test Template Framework. In *Proceedings of the third northern formal methods workshop, Ilkley, UK*. British Computing Society, September 1998.

[MCM+98] L. Murray, D. Carrington, I. MacColl, J. McDonald, and P. Strooper. Formal derivation of finite state machines for class testing. Technical Report 98-03, Software Verification Research Centre, School of Information Technology, The University of Queensland, Brisbane 4072, Australia, February 1998.

[MG83] Paul McMullin and John Gannon. Combining testing with formal specifications. *IEEE Transactions of sofware engineering*, 9(3):328–335, May 1983.

[MGB+98] 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.

[Min97] Ministry of Defence, UK. *Requirements For the Procurement Of Safety Critical Software In Defense Equipment (00-55/Issue 2)*, August 1997.

[MLPS97] Erich Mikk, Yassine Lakhnech, Carsta Petersohn, and Michael Siegel. On the formal semantics of Statecharts as supported by Statemate. In *Proceedings Of The Second Northern Formal Methods Workshop*. British Computing Society, July 1997.

[MLS97] Erich Mikk, Yassine Lakhnech, and Michael 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.

[MM94] P. C. Masiero and J. C. Maldonado. A reachability tree for statecharts and analysis of some properties. *Information And Software Technology*, 36(10):615–624, 1994.

[MP91] Raymond E. Miller and Sanjoy Paul. Generating minimal length test sequences for conformance testing of communication protocols. *Proceedings of INFOCOM*, pages 970–979, 1991.

[MP92] Raymond E. Miller and Sanjoy Paul. Generating conformance test sequences for combined control and data flow of communication protocols. In *Protocol Specification, Testing and Verification, XII*, pages 13–27. Elsevier Science Publishers B.V. (North Holland), 1992.

[NT81] S. Naito and M. Tsunoyama. Fault detection for sequential machines by transition tours. *Proceedings of the 11th IEEE Fault Tolerant Computing Symposium*, pages 238–243, 1981.

[OB88] 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.

[OLR+96] J. Offut, A. Lee, G. Rothermel, R.H. Untch, and C. Zapf. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering Methodology*, 5(2):99–118, April 1996.

[ORZ93] J. Offut, G. Rothermel, and C. Zapf. An experimental evaluation of selective mutation. *Proceedings of the 15th International Conference on Software Engineering*, pages 100–107, May 1993.

[Pur89] STARTS Purchasers'Group. The STARTS purchasers' handbook: Software tools for application to large real time systems, 1989.

[RAO92] Debra Richardson, Stephanie Leif Aha, and Owen O'Malley. Specification based test oracles for reactive systems. *14th International Conference On Software Engineering*, pages 105–118, 1992.

[RTC92] RTCA. *RTCA DO-178B, Software Considerations in Airborne Systems and Equipment Certification*, 1992.

[RW85] S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions On Software Engineering*, April 1985.

[SC93] 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.

[SC96] Phil Stocks and David Carrington. A framework for specification-based testing. *IEEE Transactions On Software Engineering*, 22(11):777–793, November 1996.

[SCES97] Harbhajan Singh, Mirko Conrad, Gottfried Egger, and Sadegh Sadeghipour. Test case design based on Z and the classification-tree method. *First IEEE International Conference on Formal Engineering Methods*, November 1997.

[SD88] Krishan Sabnani and Anton Dahbura. A protocol test generation procedure. *Computer Networks And ISDN Systems*, 15:285–297, 1988.

[SLD] Y. N. Shen, F. Lombardi, and A. T. Dahbura. Protocol conformance testing using multiple UIO sequences. In *Protocol Specification, Testing And Verification 9*.

[SvB82] Bechet Sarikaya and Gregor von Bochmann. Some experience with test sequence generation for protocols. *Second International Workshop On Protocol Specification, Testing And Verification*, pages 555–567, 1982.

[SvB84] Bechet Sarikaya and Gregor von Bochmann. Synchronisation and specification issues in protocol testing. *IEEE Transactions on Communications*, 32:389–395, 1984.

[TCM98] Nigel Tracey, John Clark, and Keith Mander. Automated program flaw finding using simulated annealing. In *Software Engineering Notes, Proceedings Of The International Symposium On Software Testing And Analysis*, volume 23, pages 73–81. ACM/SIGSOFT, March 1998.

[TM01] The MathWorks. MATLAB Simulink/Stateflow. http://www.mathworks.com/, 2001.

[Toy96] Ian Toyn. Formal reasoning in the Z notation using CADiℤ. *2nd International Workshop on User Interface Design for Theorem Proving Systems*, July 1996.

[Toy98] Ian Toyn. A tactic language for reasoning about Z specifications. In *Proceedings of the Third Northern Formal Methods Workshop, Ilkley, UK*. British Computing Society, September 1998.

[Toy00] Ian Toyn. The CADiℤ web site. http://www.cs.york.ac.uk/~ian/cadiz/, 2000. The latest version of CADiℤ and its documentation may be downloaded from this site.

[UD94] M. Umit Uyar and Anton Dahbura. Optimal test sequence generation for protocols: The chinese postman algorithm applied to Q.931. *Conformance Testing Methodologies and Architectures for OSI Protocols*, pages 347–351, 1994.

[Uni94] International Specifications Union. ITU-T recommendations Z.100, Specification and Description Language, 1994.

[Ura92] Hasan Ural. Formal methods for test sequence generation. *Computer Communications*, 15(5):311–325, June 1992.

[US94] Andrew Uselton and Scott Smolka. A process algebraic semantics for Statecharts via state refinement. *Programming Concepts, Methods and Calculi*, (A-56):267–286, 1994.

[Val98] Sam Valentine. *Modeling Statemate Statecharts In Z (DCSC/TR/98/15)*. Dependable Computer Systems Centre (DCSC), University of York, October 1998.

[VB02] Sergiy A. Vilkomir and Jonathon P. Bowen. Reinforced Condition/Decision Coverage (RC/DC): A new criterion for software testing. In Didier Bert, Jonathon P. Bowen, Martin C. Henson, and Ken Robinson, editors, *ZB2002 Formal Specification and Development in Z and B, Lecture Notes in Computer Science, vol. 2272*, pages 291–308. Springer, 2002.

[WBS02] Joachim Wegener, Andre Baresel, and Harmen Sthamer. Evolutionary test environment for automated structural testing. *To appear: Information and Software Technology, special issue on the application of meta heuristic algorithms to problems in software engineering*, 2002.

[WL93] Chang-Jia Wang and Ming T. Liu. Generating test cases for EFSMs with given fault models. In *IEEE INFOCOM'93*, pages 774–781, 1993.

[Yan95] Xile Yang. The automatic generation of software test data from Z specifications. Technical report, Department Computer Science University of Glamorgan, February 1995.