Testing a System specified using Statecharts and Z

Rob Hierons[a], Sadegh Sadeghipour[b], and Harbhajan Singh[c]
[a] Department of Information Systems and Computing, Brunel University, Uxbridge, Middlesex, UB8 3PH, UK.
[b] IT Power Consultants, Jasmunder Str. 9,13355 Berlin, Germany.
[c] DaimlerChrysler AG, Systems Technology Research (F3S/K), Alt-Moabit 96a, D-10559 Berlin, Germany.

Abstract A hybrid specification language µSZ, in which the dynamic behaviour of a system is described using Statecharts and the data and the data transformations are described using Z, has been developed for the specification of embedded systems. This paper describes an approach to testing from a deterministic sequential specification written in µSZ. By considering the Z specifications of the operations, the extended finite state machine (EFSM) defined by the Statechart can be rewritten to produce an EFSM that has a number of properties that simplify test generation. Test generation algorithms are introduced and applied to an example. While this paper considers µSZ specifications, the approaches described might be applied whenever the specification is an EFSM whose states and transitions are specified using a language similar to Z.
keywords: Testing, Extended Finite State Machine, Statechart, Z, data abstraction.

1. Introduction

There has been much interest in the use of formal specification languages in order to improve software quality. However, even if a proof of the correctness, of the source code relative to the specification, is produced it is important to test the implementation against the specification. The existence of a formal specification introduces the possibility of automating much of the test generation process and thus of increasing the effectiveness and reducing the cost of testing ([4], [8], [9], [10], [12], [19]).

Where there are sequencing issues, languages such as LOTOS, SDL and Statecharts have been applied. Due to their diagrammatic nature and the presence of tool support, Statecharts have been found to be relatively easy to use. In µSZ, Statecharts are used to define sequencing and Z is used to define the data and the operations ([2]). The language µSZ was developed, for specifying embedded systems, within the ESPRESS project: a co-operation between a number of industrial and research institutions that was funded by the German Ministry BMBF.

The paper initially describes a technique, called *data abstraction*, that uses information derived from the Z specifications to refine the extended finite state machine (EFSM) that is defined by the Statechart. Data abstraction produces an EFSM with properties that can be utilised during test generation. These properties help solve the problems of setting up the initial state and checking the final state of a test. The properties thus assist test automation.

The paper then describes methods, inspired by work on testing from finite state machines and from X-machines, for testing against the refined EFSM. These methods check both the dynamic behaviour, specified in the Statechart, and the individual operations. The methods crucially depend upon properties of the EFSM introduced by data abstraction.

Section 2 provides a basic overview of some related concepts. The language µSZ is introduced in Section 3 and data abstraction is described in Section 4. Section 5 formalises a number of notions that assist in testing. The test methods are then given in Section 6 and extended, to the case where the EFSM formed by data abstraction is nondeterministic, in Section 7. Finally, in Section 8, conclusions are drawn.

2. Preliminaries

2.1. Extended Finite State Machines

A (deterministic) *Finite Automaton (FA) M* is defined by a tuple (S,s_1, δ,X) in which S is the (finite) set of states, $s_1 \in S$ is the initial state, δ is the state transition function, and X is the (finite) alphabet. If input $x \in X$ is applied to M, while in state s, a transition t is executed and M moves to state $s'= \delta(s,x)$. The transition t is defined by the tuple (s,s',x). The function δ can be extended, to take input sequences, giving δ^*.

A *Finite State Machine (FSM)* is a FA in which each transition has an associated output value. An *Extended Finite State Machine M* is a FA which in the system has an internal store and each transition has an associated function that is triggered by input and may alter the internal store as well as produce output and change the state. Given a state s and operation op, $\delta(s,op)=s'$ if the execution of op in state s moves M to s'.

An EFSM M has *reset capacity* if there exists some operation a such that for every state $s \in S$, $\delta(s,a)=s_1$. Further, the *implementation under test (IUT)* has *reliable reset capacity* if these transitions are known to be have been correctly implemented. An EFSM M is *strongly connected* if for every ordered pair of states (s,s') there is some sequence x of operations such that $\delta^*(s,x)=s'$ and *initially connected* if given $s \in S$ there is some x such that $s= \delta^*(s_1,x)$. If M is initially connected and has reset capacity then M is strongly connected.

Two states s and s' of an FSM are *distinguishable* if there is some input sequence x such that executing x from s and s' produces different output. Two states s and s' are *equivalent* if they are not distinguishable and two FSMs are *equivalent* if their initial states are equivalent. An FSM M is *minimal* if there is no equivalent FSM with fewer states. Then FSM M is minimal if M is initially connected and no two states of M are equivalent. There are standard algorithms that take an FSM M and generate an equivalent minimal FSM ([16]).

A *directed graph (digraph) G* is defined by a set of vertices V and a set of edges E. Each edge is defined by its initial vertex, its final vertex, and possibly a label. An EFSM may thus be modelled by a digraph in which the states are represented by vertices and the transitions are represented by edges. For more on digraphs see, for example, [6].

2.2. State checking when testing from FSMs

Suppose the problem is to test a transition t_i, that has input/output pair a/b, from FSM M. In order to test t_i it is not sufficient to just execute t_i and check the output produced, since the final state might be incorrect. Thus it is necessary to check the final state of t_i and this requires the input of further values.

There are a number of approaches to checking the state of an FSM, including the use of a distinguishing sequence, unique input/output sequences and a characterizing set ([18]). Given FSM M, an input sequence D is a *distinguishing sequence* if it distinguishes between any two distinct states s and s' of M. A *unique input/output sequence (UIO) u* for state s of M is an input/output sequence x/y with the property that if x is executed from any state $s' \neq s$ of M it produces an output not equal to y. Thus, u is capable of verifying s but not necessarily any other state. Naturally a distinguishing sequence is a UIO for every state. Where each state of

M has a known UIO, in order to test a transition t_i from *M* it is sufficient to move to the initial state of t_i, execute t_i, and follow t_i by the UIO for the expected final state of t_i.

An FSM need not have a distinguishing sequence or UIOs for each state. A *characterizing set W* is a set, of input sequences, with the property that for any *(s,s')*, *s¹s'*, there is some input sequence *xÎ W* that distinguishes between *s* and *s'*. If *M* has characterizing set *W*, in order to verify the final state of a transition t_i of *M* it is sufficient to execute t_i $|W|$ times, each time followed by a different element of *W*. Every minimal FSM has a chacterizing set.

3. Statecharts and Z

3.1. Overview

A plain sequential Statechart is a graphical representation of an EFSM in which each transition has an *operation* and a *guard*. The guard gives the preconditions of the transition. Each state is represented by a box and the initial state has an arrow, with no source state, entering it. A transition with guard *g* and operation *op* can be represented by an arrow with label *g/op*.

A Statechart is *deterministic* if each operation is deterministic and, for each pair of transitions leaving a state, the guards are mutually exclusive. A transition *t* is defined by the tuple $(s_i,s_j,g/op)$ in which s_i is the initial state, s_j is the final state, *g* is the guard, and *op* is the operation.

Because of the guards, a sequence of transitions may be infeasible or may only be feasible from particular values of the internal store. The problem of feasibility is, in general, undecidable. In Section 4 a method, that can eliminate the problem of feasibility, will be introduced.

A µSZ specification is a Statechart in which the internal store, the guards and operations are defined in Z. The internal store is represented by a set of variables and operations may refer to and alter the values of these values. A Statechart state has an associated set of constraints, on the internal store, that are defined by a Z schema. Naturally, a state also represents a condition upon the set of sequences of operations that are possible and this can be seen as a further, implicit, constraint. This implicit condition might be implemented through adding further variables to the store. The following notation will be used throughout the paper.

Definition
Given a state *s* of a Statechart, the set of values for the internal store corresponding to *s* shall be denoted *int(s)*. Given a transition $t=(s_i,s_j,g/op)$: $head(t)=s_i$; $tail(t)=s_j$; $guard(t)=g$; $fn(t)=op$.

The operation *fn* can be extended, to take a sequence of transitions and return a sequence of operations, giving fn^*.

Given a state *s*, for each input *x* and internal store *vÎ int(s)* there is at least one action: if none is specified the store does not change and there is no output. It will be assumed that any µSZ specification considered is deterministic and that each operation's Z specification is defined for all input. Thus, the operations have no preconditions: instead the guards give the preconditions of the transitions.

3.2. Example

ACC is part of an adaptive cruise control system. The whole cruise control system supports the driver of the controlled car by maintaining the desired speed and a safe distance between the controlled and any preceding car.

The cruise control switch, which is located near the steering wheel, controls *ACC*. The driver can define the current speed to be the desired speed through the *increase* and *decrease* commands. With each subsequent use of *increase* or *decrease*, the desired speed is increased or reduced, respectively, by a fixed amount. The command *off* switches the system off and *resume* retrieves the last desired speed. After each use of the lever, it returns to its middle position.

ACC receives the current speed of the car, the position of the control lever and information concerning the brake. It calculates the speed requested by the driver and transmits it to the speed and distance control component. Here only the specification of *ACC* is considered. A Statechart representing *ACC* is given in Figure 1. Throughout this paper the EFSM defined by this Statechart shall be denoted M_e.

Figure 1: Statechart representing *ACC*

The Statechart in Figure 1 has two states: *passive* and *active*. The state *active* represents the condition under which the cruise control system is acting and otherwise the system is in state *passive*. There are three transitions from state *passive*: *Define*, *Resume* and *NoAction*. Operation *Define* represents the cruise control system being activated through the use of the *increase* or *decrease* commands, the requested speed taking on the current speed. In contrast, *Resume* represents the situation in which the cruise control system has previously been in use, with a value for the requested speed, and the system simply restarts with the same requested speed. The operation *NoAction*, from state *passive*, completes the operations from *passive*: it provides a loop with null output for each input/internal store value for which no behaviour is specified.

Once in state *active*, the user may alter the requested speed, return to state *passive* or leave the system unaltered. The operation with guard *[Deactivating]* represents situations in which the cruise control is deactivated (through, for example, applying the brake) and thus moves to state *passive*. The operations *Increase* and *Decrease* alter the requested speed. There is an operation, with guard *[LoopActive]* that completes the operations from state *active*.

Figures 2 gives the Z specifications of the store, input and output while Figure 3 gives the guards and operations of *ACC*. Here, *opt* takes a type *T* and returns the type *opt(T) ={{x}|xÎ T}È{Æ }*. Essentially, *opt* is used to model the situation in which a value may not have been defined. Where a variable *v* of type *opt(T)* has the value *Æ* , *v* has yet to be defined. Where *v* has the value *{x}*, *x* represents the value that models the property of interest. The operation *def* takes *xÎ opt(T)* and returns *false* if and only if *x=Æ* : if the value of *x* has yet to be defined. The operation *val* takes an element *{x}* (of type *opt(T)* for some type *T*) and returns *x* (of type *T*).

Figure 2: The State, Input and Output of *ACC*

Figure 3: The Operations of *ACC*

4. Testing and Data Abstraction

4.1. Testing

When testing an operation defined in Z it is normal to partition the input domain into subdomains upon which the behaviour is believed to be uniform. It is then assumed that all tests within a subdomain are equivalent: this is an instance of the *uniformity hypothesis*. The uniformity hypothesis is an example of a test hypothesis that allows the generation of a finite test set that is guaranteed to determine correctness as long as the hypotheses hold ([5]). Tests are often taken around the boundaries of the subdomains.

There are a number of approaches for automating the partitioning of the input domain based on a Z specification ([17], [4], [10], [12]). Possibly the best known approach is the DNF method in which a specification is rewritten to canonical disjunctive normal form and the preconditions of the conjuncts are used to form the partition ([4]). Alternatively, the partition might be based on the tester's experience ([19]). In the example of Section 3, the operation *Increase* might be given two subdomains corresponding to:

- *val requestedSpeed +stepSpeed < max(allowed)*
- *val requestedSpeed +stepSpeed ³ max(allowed)*.

The behaviour, for each subdomain, is uniform: for the first the output is *val requestedSpeed +stepSpeed* and for the second the output is *max(allowed)*.

In some cases the tester believes that certain values or representatives of certain sets of values should be used during testing. Subdomains representing these values might be used to refine the partition. Alternatively, when the generation of the partition is automated, the Z specification can be rewritten to force such tests to be generated ([20]). In the example, the tester might decide that the operation *Resume* should be described in terms of three cases: the speed is the minimum allowed, the speed is the maximum allowed, and the speed lies between these values.

It has been noted that a partition of the internal store may form the basis for the generation of an FSM and this FSM may be used during testing ([7], [4], [10], [12]). The approach outlined in this section may be seen to be an extension to this in which, rather than generate an EFSM from a Z specification, the Z specifications of the operations are used to refine the EFSM defined by the Statechart.

When testing from a specification it is usual to utilise some expected relationship between the structures of the implementation and the specification. If for each operation in the specification there is a corresponding operation in the implementation, and these operations can be tested separately, an approach such as that outlined in [13] can be applied. At the other extreme, if there is no relationship between the structures of the specification and the implementation, the specification can act as little more than an oracle.

While the structures of the implementation and the specification may be different, the structure of the specification is often reflected in the implementation. A state of the specification usually represents some set of real conditions of the system and thus has some meaning in the implementation. Throughout this paper, it will be assumed that for any specification considered, the states of the specification represent conditions of the system.

Where the structure of the specification is reflected in the implementation it is important, during testing, to cover the specification. Approaches that cover a µSZ specification may test both the individual behaviour of the operations and the dynamic behaviour of the system. Different forms of coverage, for a μSZ specification, will be described in Section 6.

4.2. Data Abstraction

In order to reduce the problem of feasibility, the EFSM can be refined through a process that will be called *data abstraction*. Let D denote the domain constructed from the variables that define the internal store. Initially a partition of *D* is produced for each operation according to its Z specification. This might be achieved using an automated technique, such as the DNF method, or through the tester using their expert knowledge.

Suppose an operation *op* partitions *D* into subdomains $d_1,...,d_k$. Given a transition *t* from *M* with operation *op* and guard *g*, a *subtransition* t_i is produced for d_i if d_i is consistent with *g*. Using an abuse of notation, in which d_i is also used to represent a predicate that evaluates to *true* if and only if the internal store is in d_i, this subtransition t_i has precondition d_iÙg. Then, potentially, there is a subtransition corresponding to *t* for each subdomain generated for *op*.

The subtransitions corresponding to a transition *t* represent the tests for *t*: by the uniformity hypothesis, it is sufficient to test each subtransition once in order to test *t*. Naturally, the tester may choose to test some subtransitions more than once.

Consider a state *s* from *M*. Let $i_1,...i_{in}$ and $o_1,...,o_{out}$ denote the subtransitions, entering and leaving *s* respectively. The conditions on the internal store, defined by the preconditions of the o_k and the postconditions of the i_k, are generated. These conditions cover *int(s)* since the specification is complete. The conditions need not, however, be pairwise disjoint. The set of conditions is thus rewritten to canonical disjunctive normal form and the resultant conditions partition *int(s)*. These partitions define states in the new EFSM: for each subdomain *d* of the partition of *int(s)* there is a corresponding state that represents being in state *s* and having an internal store from *d*. This process, applied to each state, provides the states of the refined EFSM.

Suppose a subtransition t_i has been derived from a transition *t*, of *M*, that leaves a state *s*. Suppose also that *s* is partitioned into $s^1,...,s^k$. In the new EFSM, there is a transition representing t_i leaving each state s^j that is consistent with the precondition of t_i. This process, applied to each subtransition, defines the transitions of the refined EFSM.

The EFSM generated by data abstraction shall be denoted M_A. As noted, a subtransition may be represented by a number of transitions from M_A. Separate subdomains in the partitions of the operations may represent boundary tests: this leads to further subtransitions and thus tests.

If some states are not reachable, these states and the transitions leaving them are removed. By the construction of the partition, if t_i leaves state s^j then, for each internal store value *v* in s^j, there is some input value x_v such that (x_v,v) satisfies the precondition of t_i. Thus t_i is feasible from every internal store value consistent with s^j. Thus, if M_A is deterministic, all sequences generated from M_A are feasible. This greatly simplifies test generation.

It will be assumed that the problem is to produce tests from an EFSM M_A that has been produced from a sequential μSZ specification, with EFSM *M*, using data abstraction. It will be assumed that M_A is deterministic and strongly connected and is represented by the digraph *G*. The condition that M_A is deterministic will be weakened in Section 7. Data abstraction shall now be applied to *ACC*.

4.3. Example

Let the functions *low*, *mid*, and *high* be defined by: *low x* ⇔ *x=min(allowed); mid x* ⇔ *min(allowed)<x<max(allowed); high x* ⇔ *x=max(allowed)*. By considering the Z schemas and the types, the three operations *Resume*, *Increase*, and *Decrease* can provide the following conditions.

Resume:	*low(val requestedSpeed)*
	mid(val requestedSpeed)
	high(val requestedSpeed)
Increase:	*val requestedSpeed+stepSpeed ³ max(allowed)*
	val requestedSpeed+stepSpeed < max(allowed)
Decrease:	*val requestedSpeed-stepSpeed £ min(allowed)*
	val requestedSpeed-stepSpeed > min(allowed)

Considering the guard for *Resume* generates the conditions

def requestedSpeed
Ø *def requestedSpeed*

Consider the state *passive*. The only operation, which partitions the internal store, that leaves *passive* is *Resume*. There are two conditions given by the guard and three conditions given by the operation. However, one of the conditions given by the guard (*def requestedSpeed*) is partitioned by the three conditions given by the operation and thus there are the following four conditions and four corresponding states in the refined EFSM.

Ø *def requestedSpeed*
low(val requestedSpeed)
mid(val requestedSpeed)
high(val requestedSpeed)

Consider now the state *active*. The postconditions of *Resume* and *Define* are both *def val requestedSpeed*. Thus, since *active* is not the initial state and no operation from *active* can violate this condition, this condition must hold whenever the state is *active*. Since *Resume* does not alter the value of *requestedSpeed*, its postconditions generation the conditions *low(val requestedSpeed)*, *mid(val requestedSpeed)*, and *high(val requestedSpeed* for *active*. It is then necessary to consider the refinement of these conditions generated by *Increase* and *Decrease*. The conditions for *Increase* and *Decrease* partition the condition *mid(val requestedSpeed)*. With some rewriting, this leads to the following conditions.

low(val requestedSpeed)
min(allowed) < val requestedSpeed £ min(allowed)+stepSpeed
min(allowed)+stepSpeed < val requestedSpeed < max(allowed)-stepSpeed
max(allowed)-stepSpeed £ val requestedSpeed < max(allowed)
high(val requestedSpeed)

It is also possible to confirm that when the state is *active*, *allowed(val requestedSpeed)* must hold. The states, $passive_1,..,passive_4,active_1,...,active_5$, generated from these conditions are given in Figure 4. Some of the state definitions can be simplified.

Figure 4: The States

Consider now the process of producing subtransitions. As stated earlier, the partitioning of a transition generates a number of subtransitions. Suppose the tester has decided to test the operation *Deactivating* from each condition (*low*, *mid*, and *high*) of *val requestedSpeed* with each combination of input conditions, *lever?=off* and *brake?=activated*, that produces *true*. This leads to 9 subtransitions defined by conjoining each condition for *val requestedSpeed* with the three conditions $lever?=off \wedge \neg brake?=activated$, $\neg lever?=off \wedge brake?=activated$, and $lever?=off \wedge brake?=activated$. Similarly, *Define* has the three conditions for *currentSpeed?* and two conditions for *lever?* (*increase* and *decrease*) and thus leads to 6 subtransitions. The transitions *Resume*, *Increase* and *Decrease* have subtransitions corresponding to the conditions outline above while *LoopPassive* and *LoopActive* are not partitioned. Names for the subtransitions are given in Figure 5.

Figure 5: The subtransitions

The EFSM derived from M_e, by data abstraction, is given in two parts in Figure 6: the transition set is the sum of those given in the two parts. This EFSM will be denoted M_{Ae} throughout the paper.

Figure 6: The EFSM M_{Ae}

5. Distinguishing transitions and states

5.1. Distinguishing transitions and operations

When testing from a deterministic EFSM *M*, rather than an FSM, there are two main complications: a transition has a function rather than a single pair of input/output values and each state of *M* represents a set of values for the internal store of the system rather than a single value. These reduce the confidence provided by testing, as a transition may be correct for the input and internal store used in testing but not for others. The second problem can lead to difficulties in setting up the internal store for tests and reduces the confidence provided by state checking. It is still, however, possible to apply techniques similar to those used for FSMs. The tests produced may help check both the dynamic behaviour of the implementation (the sequences of operations allowed) and the functionality of the operations.

When testing a transition *t* from state s_j the correct test output might be produced by the execution of the wrong operation. This is an instance of *coincidental correctness*. The testing of *t* is strengthened if some test, that reduces this possibility, is used. Thus, ideally, when testing *t* from internal store $a Î int(s_j)$, some input *x* is chosen such that *(a,x)* satisfies *guard(t)* and the expected output *fn(t)(a,x)* is not one that might have been produced by any other operation from the specification: the operation within the transition is distinguished from every other operation. Testing is simplified if, for each *t* and *a*, there is such an *x*. The following definitions, in which *Op* denotes the set of operations in the Statechart, help formalise this notion. These definitions are inspired by the notion of output distinguishability used when testing from X-machines ([13])

Definition
A transition $t_i=(s_j,s_k,g/op)$ is *distinguishable from* an operation $op'\in Op$ if given $a\in int(s_j)$ there is some input x such that $g(a,x)$ and $op'(a,x)\neq op(a,x)$.

The notion of a transition being distinguishable from an operation is defined because, when testing a transition, it is sufficient for the operation in this transition to be distinguishable from every other operation on the subdomain defined by the guard of the transition. Later, the notion of two operations being distinguishable will be defined.

Definition
A transition $t_i=(s_j,s_k,g/op)$ is *distinguishable* if given $a\in int(s_j)$ there exists x such that $g(a,x)$ and for all $op'\in Op$, $op'\neq op\Rightarrow op'(a,x)\neq op(a,x)$.

Definition
A subtransition t_i, of a transition t from M, is *distinguishable* if all of the corresponding transitions from M_A are distinguishable.

Definition
An operation op is *distinguishable from* an operation $op'\in Op$ if all transitions containing op are distinguishable from op'.

Definition
An operation op is *distinguishable* if all transitions containing op are distinguishable.

Definition
If all the transitions of an EFSM are distinguishable then the EFSM is *distinguishable*.

From the definitions, M is distinguishable if and only if all of its subtransitions are distinguishable and thus M is distinguishable if and only if M_A is distinguishable. As there may be several copies of a subtransition in M_A, it will often be sufficient for there to be a distinguishable copy of each subtransition. Thus it is not always necessary for M to be distinguishable. The above definitions will, however, be used in the following sections. Sufficient conditions, for the application of the test techniques, will be given in Section 6.

If a transition t is distinguishable then any test for t, using an appropriate input, has an expected output that would not have been produced by any operation different from $fn(t)$. This avoids the form of coincidental correctness described earlier and helps check that the correct operation has been applied and the behaviour of the operation is correct for this input/internal store. Of course, the possibility of coincidental correctness cannot be completely eliminated. Weaker forms of distinguishability are considered in Section 5.4.

Throughout this paper it will be assumed that, given a transition t from M_A and an operation $op'\in Op$ with $op'\neq fn(t)$, there is some store/input pair (a,x) satisfying $guard(t)$ such that $fn(t)(a,x)\neq op'(a,x)$. If this is not the case for some t and op' then they are equivalent on $guard(t)$ so there is no need to distinguish them here. The techniques in this paper can easily be adapted where this property does not hold.

In the example, *Define* and *Resume* produce different output values if the input value c*urrentSpeed?* is not equal to *val requestedSpeed*. Thus, if the internal store is known the operations may be distinguished. The operations *Increase* and *Resume* are not distinguishable if *requestedSpeed=max(allowed)* as both then output the value *requestedSpeed*. Where the current speed is less than the maximum, these operations are distinguishable.

5.2. Distinguishing states using distinguishable transitions

A test might execute the correct operation, produce the expected output, and yet be erroneous because it has moved the system to the wrong state of the Statechart. Such errors might not be detected if the final state of each test is not checked.

When testing a transition *t* from M_A it is thus important to follow *t* by tests that distinguish *tail(t)* from other states. Given a state s_i from M_A, let *abs*(s_i) denote the corresponding state from *M* and let *L*(s_i) denote the set of sequences of subtransitions leaving s_i. In the example from Section 3, *abs*($passive_4$)=*passive* and c_3bf_1*Î L*($passive_4$) (since c_3 can change the state from $passive_4$ to $active_4$, *b* then leads to no change in state and finally f_1 can change the state to $active_5$).

In order to distinguish a state, which is expected to be s_i, from a state s_j it is sufficient to demonstrate that some sequence of operations, that can be executed from s_i but not s_j, can indeed be executed. Where *M* is distinguishable it is sufficient to use any element of *L*(s_i)-*L*(s_j) with appropriate input values. It should be noted that, since the final state of *M* is being checked, it is only necessary to distinguish s_i from s_j in M_A if *abs*(s_i)*¹ abs*(s_j).

Even if *M* is not distinguishable, it may be possible to use transitions from M_A in order to distinguish the states of M_A. Let $\sim_d$ denote the relation between transitions and operations such that $t\sim_d$ *op* if and only if *t* is not distinguishable from *op*. Given a sequence *x*, *#x* denotes the length of *x* and x_i denotes the ith element of *x*. A sequence of subtransitions *x* cannot, in general, be distinguished from any sequence of operations from

fuzz(x)={x'Î Op**|#x'=#x Ù(" i,1£i£#x ·*$x_i\sim_d x'_i$*)}.*

If *M* is distinguishable then *fuzz(x)={fn***(x)}*. In order to consider state verification it is useful to define the notions of distinguishing states and of M_A being state distinguishable. These are given by the following definitions.

Definition
A sequence *x* of subtransitions *distinguishes* s_i *from* s_j if *xÎ L*(s_i) and there is no *yÎ L*(s_j) such that *fn***(y)Î fuzz(x)*.

Definition
A state s_i from M_A is *distinguishable* if for each state s_j from M_A, with *abs*(s_j)*¹ abs*(s_i), there is some sequence *x* that distinguishes s_i from s_j.

Definition
M_A is *state distinguishable* if each state of M_A is distinguishable.

Ideally one sequence *x* distinguishes s_i from every s_j with *abs*(s_i)*¹ abs*(s_j). This sequence, which shall be called a *state identification sequence*, is like a UIO. Alternatively a set of state verification sequences, similar to a characterizing set, suffices.

The following conditions are (between them) sufficient, but not necessary, for M_A to be state distinguishable:

1. *M* is distinguishable.
2. For each ordered pair of states (s_i,s_j) from M_A, with $abs(s_i)$¹$abs(s_j)$, $L(s_i)$-$L(s_j)$¹Æ .

The second condition is particularly important as it is difficult to distinguish a state s_i from a state s_j if $L(s_i)$Í$L(s_j)$. These conditions, or alternative sufficient conditions, might be seen to be design for test properties. Throughout this paper it will be assumed that the second condition holds.

In the example, each $passive_i$ is distinguishable from each $active_j$ by using the operation *Define* with an appropriate value of *currentSpeed?*. There is no single operation that distinguishes each $active_i$ from all of the $passive_j$ but *Increase* and *Decrease* suffice between them. If, for example, *val requestedSpeed+stepSpeed<max(allowed)* (and thus the state is one of $active_1,\dots,active_4$) then *Increase* can be used.

The process of finding state distinguishing sequences is similar to the process of finding state verification sequences for an FSM. These sequences can be found using a breadth-first search.

5.3. Distinguishing states without using distinguishable transitions

Suppose $abs(s_i)$¹$abs(s_j)$ and $L(s_i)$-$L(s_j)$¹Æ but for all xÎ$L(s_i)$-$L(s_j)$, there is some yÎ$L(s_j)$ such that $fn^*(y)$Î$fuzz(x)$. Then there are sequences of operations that can be executed from s_i and not s_j but each has some element of $L(s_j)$ from which it is not, in general, distinguishable.

If xÎ$L(s_i)$-$L(s_j)$ then even though x does not, in general, distinguish s_i from s_j, x may distinguish s_i from s_j for some internal stores from $int(s_i)$. The process of verifying the final state of a transition t from M_A that ends in s_i, using x, then includes the problem of executing t in a manner that leads to an expected final store that allows the use of x to distinguish s_i from s_j. Naturally, this might not be feasible. When it is feasible, it may be difficult to find an input sequence that moves to an appropriate value for the internal store. There are thus sequencing and feasibility issues. The feasibility problem is reduced if, for each aÎ$int(s_i)$, there is some known sequence that distinguishes a from s_j.

It may be possible to engineer a system in order to make it easier to distinguish states or to use testing tools that help this process. A testing tool might add code, that outputs special values, or provide a function that produces a memory dump. The former would help distinguish operations while the latter would help distinguish states.

5.4. Weaker forms of distinguishability

Alternative definitions will allow the weakening of distinguishability, for a transition t, in two ways:
1. An input value x may only distinguish between $fn(t)$ and one other operation.
2. Particular values from $int(head(t))$ may be required for distinguishing t.

State distinguishability follows from transition distinguishability as before. As the definition of distinguishability becomes weaker, the set of systems that have this distinguishability increases but testing becomes more difficult.

Definition
The alternative definitions, for a transition $t_i=(s_j,s_k,g/op)$ being distinguishable, are:

1. given $a \in int(s_j)$ and $op' \in Op$ with $op' \neq op$ there is some input x such that $g(a,x)$ and $op'(a,x) \neq op(a,x)$.
2. there is some input x and $a \in int(s_j)$ such that $g(a,x)$ and for all $op' \in Op$ with $op' \neq op$, $op'(a,x) \neq op(a,x)$.
3. given $op' \in Op$ with $op' \neq op$ there is some input x and $a \in int(s_j)$ such that $g(a,x)$ and $op'(a,x) \neq op(a,x)$.

In the first case it is necessary to use multiple tests in order to distinguish *op* from all other operations: potentially one test for each $op' \neq op$. In the second case there is the problem of executing t from the correct internal store value from $int(head(t))$. The first case leads to a larger test while the second can make test generation more difficult. In the last case there are both problems.

Sometimes, due to the effort involved, it is impractical to use distinguishability in testing. Instead tests can be produced using input values that are not guaranteed to be distinguishable. Testing provides less confidence but should still be useful. Where possible, if $abs(s_i) \neq abs(s_j)$ and a transition t from M_A is to be tested and has $tail(t)=s_i$, during testing t should be followed by a sequence from $L(s_i)$-$L(s_j)$ at least once. Ideally, state distinguishing sequences are used.

6. Test Generation

6.1. Introduction

In this section a number of test criteria will be considered, each criterion insisting that every subtransition is tested in some way. Each criterion leads to a problem of the form: produce a test that satisfies the criterion. Some subtransitions may be tested more than once, possibly relating the number of tests to the criticality of and the confidence in a subtransition.

Given a subtransition t_i, that leads to more than one transition in M_A, it is necessary to choose one or more corresponding transitions from M_A to test. Each of these transitions chosen can be considered to be a copy of t_i. If possible, each copy chosen is distinguishable and, if state checking is used, has a distinguishable final state. Otherwise the choice made depends upon the test method applied. The number of times a transition t_{ij}, which is a copy of a subtransition t_i, must be tested will be denoted n_{ij}. The value of n_{ij} may be 0.

During testing it is possible to utilise distinguishability if, for each subtransition t_i, there is a corresponding transition t_{ij} from M_A such that:
1. The transition t_{ij} is distinguishable.
2. The state $tail(t_{ij})$, of M_A, is distinguishable.

If M_A does not have these properties, weaker forms of distinguishability may be used.

This section shall consider test generation when M_A is deterministic. Since M_{Ae} is nondeterministic, tests will not be generated from it until Section 7, in which nondeterminism is discussed. In the following, a number of test sequence generation methods, based on FSM test techniques, are described. The problem of generating a test input sequence, to trigger a sequence of transitions, will then be discussed.

6.2. Transitions Tours

A *transition tour* is a sequence that includes each transition at least once and starts and ends at the initial state. There are standard algorithms for producing a minimal length transition tour from a strongly connected FSM ([18]). What is required here, however, is the shortest tour that contains, for each t_{ij}, at least n_{ij} instances of t_{ij}. Again, n_{ij} may be 0.

This problem can be solved by representing M_A by a digraph $G=(V,E)$ and adding, for each transition t_{ij}, n_{ij} edges from the vertex that represents $head(t_{ij})$ to the vertex that represents $tail(t_{ij})$. Let E_C denote the set of extra edges and $G_C=(V,E \cup E_C)$. The problem can be represented by: find the shortest tour of G_C that contains every edge from E_C. This is an instance of the *Rural Postman Problem (RPP)*. While the RPP is NP-complete ([14]), there is a low order polynomial algorithm that will produce a tour that, under certain conditions, is guaranteed to be minimal ([1]). Where the tour is not minimal, a set of tours is produced and these can be connected ([9], [21]).

6.3. Transition Tours with State Checking

In this subsection it will be assumed that each state s_i of M_A has a single sequence u_i that distinguishes s_i from every state s_j, of M_A, with $abs(s_i) \neq abs(s_j)$. The approach outlined can be extended to the use of multiple sequences.

The transition tour method produces a test that is expected to cover every subtransition. It does not, however, explicitly verify the final state of any subtransition. Instead it is possible to include, for each transition t_{ij} with final state s_k, a subsequence of the form $t_{ij}u_k$ within the test. What is required is the shortest tour that contains, for each transition t_{ij}, n_{ij} copies of $t_{ij}u_k$.

This problem can also be represented in terms of the RPP. Again M_A is represented by a digraph $G=(V,E)$. For each transition test $t_{ij}u_k$, n_{ij} edges are added from the vertex that represents $head(t_{ij})$ to the vertex that represents the final state of u_k. Let E_C denote this set of edges. The problem is: find the shortest tour of $(V,E \cup E_C)$ that contains each edge from E_C at least once.

The RPP approach connects the individual transition tests. There might, however, be overlap between these transition tests. This overlap can be utilised to further reduce the test length ([8], [9], [22]).

It is possible, when minimising the cost, to weight the subtransitions. The weighting may depend upon the criticality of the subtransition or the confidence the tester has in a subtransition. A high criticality, or low confidence, is represented by a low weighting. If this is done, the test is likely to contain more copies of subtransitions with a low weighting.

6.4. Transition Trees

The transition tree method ([3]) involves generating a tree V, called a *reachability tree*, that is rooted at s_1 and contains each state of M_A. To execute a transition t_{ij} from M_A it is sufficient to execute the sequence $v(head(t_{ij})) \in V$, that moves to $head(t_{ij})$, followed by input for t_{ij} and then reset the system. If, for each subtransition t_i, this is done for one corresponding transition t_{ij} from M_A, a test that includes every subtransition is produced. This can be extended by, for each transition t_{ij} used, following $v(head(t_{ij}))t_{ij}$ by the state verification sequence(s) for $tail(t_{ij})$.

A transition tree that minimises the test effort can be produced using a breadth-first search starting at s_1. The breadth-first search continues until, for each subtransition t_i, the initial state of one copy t_{ij} of t_i has been met. It thus need not produce a full reachability tree for M_A: it is sufficient to reach the initial state of each transition to be used in testing. The transition tree approach requires the existence of a reliable reset operation.

6.5. Heuristics

In some cases the tester may believe that certain sequences are important to test, possibly because they are expected to be good at detecting faults or at checking critical aspects of the system, or because they are believed to represent a common use of the system. When a transition tour or a transition tour with state verification is to be produced, an extra test can be represented by a single edge that must be covered.

In general, heuristics can represent the tester's knowledge of a system and thus vary between systems and testers. Tests developed to satisfy some heuristic can be added to those developed from one of the standard coverage based techniques.

6.6 Generating Test Input

Once a test sequence has been generated from M_A it is necessary to produce a corresponding input sequence. Even though the test sequence is guaranteed to be feasible, only certain input sequences will trigger it. The input required to trigger a particular transition depends upon the internal store and thus depends upon the previous input values used. In order to see this, consider a system with one variable *x* that defines the internal store and one input value *in?*. Then a transition *t* might have precondition *in?=x*, in which case it is always feasible and does not lead to any partitioning of the internal store. However, in order to choose input to trigger *t* it is necessary to know the expected value of the internal store *x*.

The state and store, before the test begins, are known. It is thus sufficient, for each transition, to generate a test input that will trigger the transition and to determine the expected final state of the transition. If this can be done in advance a pre-set input sequence may be produced.

If it is not feasible to produce a pre-set input sequence, the input sequence may be developed during testing. It is possible to use the output produced, as well as the initial state and input, to determine the expected final store of a transition. If this is done, during testing the expected initial internal store for a transition t_{ij} is known when the transition is to be executed, as it can be derived from the input, output and initial internal store of the previous transition. The input for t_{ij} may thus be generated at this point.

7. Nondeterministic EFSMs

Even if *M* is deterministic, M_A may be nondeterministic. There will, however, be only one allowed next state when the internal store *a*Î *int(s)* and input *x* are known. Given state *s*, *a*Î *int(s)* and input *x* there will also be only one transition for *s* whose guard allows *(a,x)*. This underlying determinism can be utilised during testing.

Interestingly, some standard approaches, for testing from a nondeterministic FSM, are not directly applicable in this situation. This is because these techniques are based on the assumption, called the *complete-testing assumption*, that there is some *m* such that if an input sequence is repeated *m* times every possible resultant sequence will be executed ([15]). Here, however, an input sequence will always generate the same behaviour. In order to utilise a similar property it is necessary for each repetition to use different input values. Some behaviour may, however, only be exhibited if a state is reached via some alternative route.

Suppose M is distinguishable, $abs(s_i)$¹$abs(s_j)$, and there is some xÎ$L(s_i)-L(s_j)$. Then x distinguishes s_i from s_j whenever M_A has store aÎ$int(s_i)$ such that the sequence x is feasible from a. If x is not feasible from a, some other sequence is required. Nondeterminism may thus lead to a different state distinguishing sequences, or sets of sequences, for different internal stores within a state. The sequence, or sequences, applied depends upon the internal store.

When testing a transition t_{ij} it is necessary to move to $head(t_{ij})$. When M_A is nondeterministic it may be necessary to find an input sequence that triggers a particular sequence of transitions, and nondeterministic choices for these transitions, in order to reach $head(t_{ij})$. Testing is simplified if the following condition holds.

Definition
M_A is *initially d-connected* if, for each state s_i, there is some sequence of deterministic transitions that takes M_A from s_1 to s_i

If M_A has reliable reset capacity and is initially d-connected, a reachability tree can be developed and the transition tree method applied. If there is also a state verification sequence for each state s from M_A and internal store aÎ$int(s)$, the transition tree method with state verification may be applied.

It is possible to verify that, in the example, M_{Ae} is initially d-connected. In M_{Ae} it is sufficient, for state distinguishing, to use any c_j from $passive_i$ and from $active_i$ either f_j (i¹5) or g_j (i¹1). If the transition tree method is used the test shown in Figure 7 may be produced. For the test of the subtransitions f_1 and g_1, corresponding distinguishable transitions from M_{Ae} have been chosen. Removing any sequence contained within the beginning of another sequence further reduces the test.

Figure 7: A test for *ACC*

Suppose a subtransition t_i has two or more corresponding transitions in M_A that leave the same state but have alternative final states with different state identification sequences (e.g. f_2 and g_2). If the internal store for this execution has yet to be determined, several tests are required for t_i: one for each state identification sequence. Once input values are chosen and the internal store, before this subtransition is executed, is known only one of these tests is required.

If the deterministic transitions form a strongly connected subautomaton of M_A and there is a deterministic instance of each subtransition then the transition tour method can be applied. If the final state, of each transition t_{ij} from M_A being tested, also has a deterministic state identification sequence then it is possible to produce a transition tour with state checking.

If a transition tour with state checking is produced from M_{Ae} there will be much overlap between the transition tests, as the state verification sequences have length 1. This overlap can be utilised in order to reduce the test sequence length ([8], [9], [22]).

The condition that M_A is initially d-connected can be relaxed to the following.

Definition

M_A is *initially weakly d-connected* if, for each state s_i, there is some sequence of operations $l=op_1,...,op_k$ such that when this is executed from s_1 the final state is s_i.

The transitions within such a sequence may be nondeterministic: it is only necessary that all choices lead to s_i. A simple example, in which op_1op_2 moves M_A from s_1 to s_4, is shown in Figure 8.

Figure 8: A deterministic sequence containing nondeterminism

If M_A has reliable reset capacity but is not initially weakly d-connected it may be possible to apply the transition tree method. If there is no deterministic route from s_1 to a state s_i then it is necessary to find an input sequence that moves M_A to s_i. It may be possible to generate a route to s_i analytically, by considering the operations and the conditions under which the different options are chosen. Where this is not feasible it may be possible to find a route by applying some heuristic such as Tabu Search or Genetic Algorithms or by using adaptive testing.

In some cases it is possible to divide the states of M_A, producing a deterministic EFSM. This process of state splitting is not, however, guaranteed to terminate. If state splitting does terminate the approaches described in Section 6 can be applied. In M_{Ae} state splitting will terminate but is not required.

8. Conclusions

The use of Z to specify operations within a Statechart increases the degree of formalism and allows the automatic analysis of the specification. Given such a Statechart, data abstraction may be used to refine the corresponding EFSM in order to simplify testing. If data abstraction is applied and the EFSM M_A produced is deterministic then all sequences derived from M_A are feasible.

If a transition t with guard g and operation op is distinguishable, t can be checked by executing op with values that satisfy g and distinguish between op and all other operations from M. Such a test checks the behaviour of the operation and checks that the correct operation has been applied. It thus eliminates one possible form of coincidental correctness.

A transition may produce the expected output and yet be erroneous because it leads to the wrong final state. In testing it is possible to check the final state of a transition by using state distinguishing sequences. Once state distinguishing sequences are known it is possible to automate or semi-automate the test generation process. These tests check both the dynamic behaviour of the system and the behaviour of each individual operation.

A number of different notions of distinguishability have been discussed. These vary in the number of tests required and the ease with which tests can be generated from the sequences chosen. Where it is not feasible to use distinguishability, the same test generation algorithms may be applied. The tests should still cover the structure of the specification and provide some confidence in the final states of transitions being correct.

Sometimes, when M_A is nondeterministic, it is possible to divide the states of M_A to get a deterministic EFSM. The normal test techniques can then be applied. If a deterministic EFSM cannot be produced by splitting the states of M_A, the underlying determinism of the system can be utilised. If M_A has reliable reset capacity and is initially weakly d-connected, a reachability tree can be developed and the transition tree method applied. When M_A is not initially weakly d-connected, for a state s_i of M_A it is necessary to develop an input sequence x that reaches s_i.

The Z specification plays two roles in the test process: it defines subdomains upon which an operation is uniform and it defines the behaviour on these subdomains. It should be possible to adapt the techniques described in this paper to the combination of an EFSM with any other formalism that provides these two things.

The approach outlined can only be applied to sequential specifications. Where there is parallelism the specification can, however, be modelled by a set of Communicating EFSMs. It may be possible to tackle such cases by adapting techniques that are designed for testing from Communicating FSMs ([11], [15]).

References

1. Aho A.V., Dahbura A.T., Lee D., and Uyar, M.U., 1988, An Optimization Technique for Protocol Conformance Test Generation Based on UIO Sequences and Rural Chinese Postman Tours, *Protocol Specification, Testing, and Verification VIII*, 1988, Atlantic City, pp. 75-86, Elsevier (North-Holland).
2. Büssow R., Geisler R., Grieskamp W., and Klar M., 1997, The µSZ Notation Version 1.0, *TU Berlin, Report No. 97-26.*
3. Chow T.S., 1978, Testing Software Design Modelled by Finite State Machines, *IEEE Transactions on Software Engineering*, **4**, pp. 178-187
4. Dick J. and Faive A., 1993, Automating the generation and sequencing of test cases from model-based specifications, *FME ´93, First International Symposium on Formal Methods in Europe*, Odense, Denmark, 19-23 April, pp. 268-284.
5. Gaudel M.-C., 1995, Testing Can be Formal Too, *TAPSOFT '95: 6th International Joint Conference on Theory and Practice of Software Development, volume 915 of Lecture Notes in Computer Science*, pp. 82-96.
6. Gibbons A., 1985, *Algorithmic Graph Theory*, Cambridge University Press.
7. Hall P.A.V. and Hierons R.M., 1991, Formal Methods and Testing, *The Open University Computing Department, Tech Report No. 91/16.*
8. Hierons R.M., 1996, Extending Test Sequence Overlap by Invertibility, *The Computer Journal* **39** 4, pp. 325-330.
9. Hierons R.M., 1997, Testing From a Finite State Machine: Extending Invertibility to Sequences, *The Computer Journal*, **40** 4, pp. 220-230.
10. Hierons R.M., 1997, Testing From a Z Specification, *Journal of Software Testing, Verification and Reliability*, **7** 1, pp. 19-33.
11. Hierons R.M., 1997, Testing from semi-independent communicating finite state machines with a slow environment, *IEE Proceedings on Software Engineering*, **144** 5-6, pp. 291-295.
12. Hörcher, H.-M. and Peleska J., 1995, Using formal specifications to support software testing, *Software Quality Journal*, **4** 4, pp. 309-327.
13. Ipate F. and Holcombe M., 1997, Tests which are proved to find all faults, *International Journal of Computer Mathematics*, **63**, pp. 159-178.
14. Lenstra J.L. and Rinnoy Kan A.H.G., 1976, On General Routing Problems, *Networks*, **6**, pp. 273-280.
15. Luo G., von Bochmann G. and Petrenko A., 1994, Test Selection Based on Communicating Nondeterministic Finite-State Machines Using a Generalized Wp-Method, *IEEE Transactions on Software Engineering*, **20** 2, pp. 149-162.

16. Moore E.P., 1956, Gedanken-Experiments, In Shannon C. and McCarthy J.M. (eds), *Automata Studies*, pp. 129-153, Princeton University Press.
17. North, N. D., 1990, Automatic Test Generation for the Triangle Problem, *National Physical Laboratory Report DITC 161/90*.
18. Sidhu D.P. and Leung T.K., 1989, Formal Methods in protocol testing: a detailed study, *IEEE Transactions in Software Engineering*, **15** 4, pp. 413-426.
19. Singh H., Conrad M., and Sadeghipour S., 1997, Test Case Design Based on Z and the Classification-Tree Method, *First IEEE International Conference on Formal Engineering Methods*, Hiroshima, Japan, pp. 81-90.
20. Stocks P. and Carrington D., 1993, Test Template Framework: A specification-based testing case study, *SIGSOFT Software Engineering Notes*, **18** 3, pp. 11-18 July 1993.
21. Ural H., Wu X., and Zhang F., 1997, On Minimizing the Lengths of Checking Sequences, *IEEE Transactions on Computers*, **46** 1, pp. 93-99.
22. Yang B. and Ural H., 1990, Protocol Conformance Test Generation Using Multiple UIO Sequences with Overlapping, *ACM SIGCOMM 90: Communications, Architectures, and Protocols*, Sept 24-27 1990, Twente, The Netherlands, pp. 118-125.

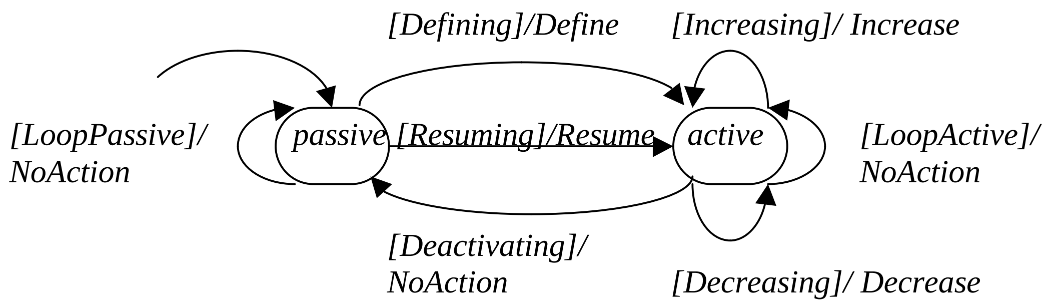


Figure 1: Statechart representing *ACC*

LeverPosition::=increase | decrease | resume | off | middle
PedalPosition::=activated | notActivated

allowed: P SPEED
stepSpeed: SPEED

$\forall x:SPEED \bullet x \in allowed \Leftrightarrow 40 \leq x \leq 160$
stepSpeed=5

DATA DataAcc
requestedSpeed: opt SPEED

def requestedSpeed $\Rightarrow$ val requestedSpeed $\in$ allowed

PORT Input_Ports
lever?: LeverPosition
brake?: PedalPosition
currentSpeed?: SPEED

PORT Output_Ports
requestedSpeed!: opt SPEED

Figure 2: The Store, Input and Output of *ACC*

INIT ACC

DataAcc'; Output_Ports

$\neg$def requestedSpeed' $\wedge$ $\neg$def requestedSpeed!

GUARD Defining

Input_Ports

lever? = increase $\vee$ lever? = decrease
currentSpeed? $\in$ allowed $\wedge$ brake? = notActivated

OP Define

ΔDataAcc
Input_Ports; Output_Ports

def requestedSpeed' $\wedge$ val requestedSpeed' = currentSpeed?
requestedSpeed! = requestedSpeed'

GUARD Resuming

DataACC; Input_Ports

def requestedSpeed $\wedge$ lever? = resume $\wedge$ brake? = notActivated

OP Resume

ΔDataACC
Output_Ports

requestedSpeed' = requestedSpeed
requestedSpeed! = requestedSpeed'

GUARD Increasing

Input_Ports

lever? = increase $\wedge$ brake? = notActivated

OP Increase

ΔDataACC
Ouput_Ports

def requestedSpeed'
val requestedSpeed' = min{val requestedSpeed+stepSpeed, max(allowed)}
requestedSpeed! = requestedSpeed'

GUARD Decreasing

Input_Ports

lever? = decrease ∧ brake? = notActivated

OP Decrease

ΔDataACC
Ouput_Ports

def requestedSpeed'
val requestedSpeed' = max{val requestedSpeed-stepSpeed, min(allowed)}
requestedSpeed! = requestedSpeed'

GUARD Deactivating

Input_Ports

lever? = off ∨ brake? = activated

GUARD LoopPassive

Input_Ports

lever? = off ∨ lever? = middle ∨ brake? = activated ∨
(¬ currentSpeed? ∈ allowed ∧ lever? ≠ resume) ∨
(lever? = resume ∧ ¬ def requestedSpeed)

GUARD LoopActive

Input_Ports

lever? = resume ∨ lever? = middle
brake? = notActivated

OP NoAction

ΔDataAcc
Output_Ports

¬def requestedSpeed!
requestedSpeed' = requestedSpeed

Figure 3: The Operations of *ACC*

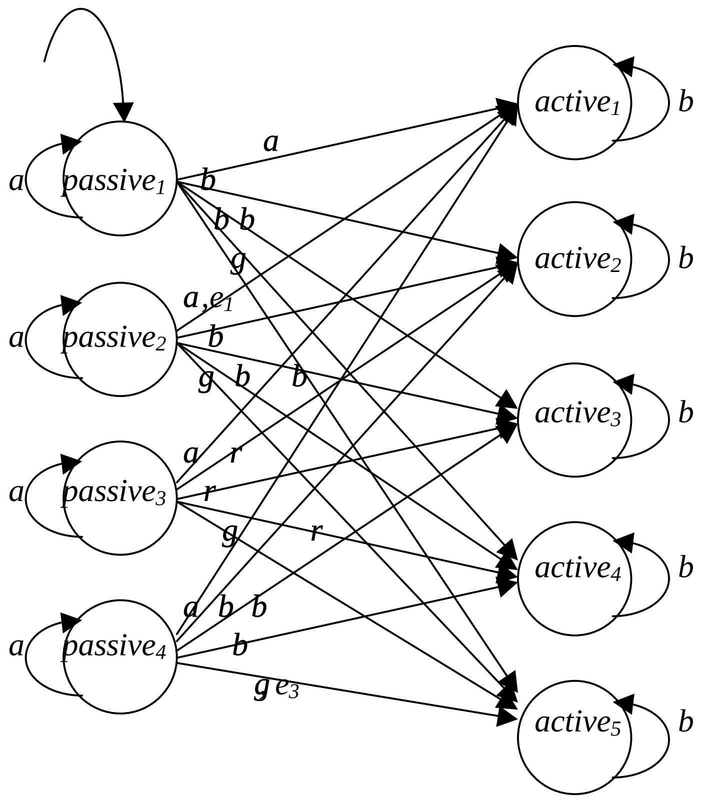
passive$_1$ [requestedSpeed: opt SPEED | Ø def requestedSpeed]
passive$_2$ [requestedSpeed: opt SPEED | def requestedSpeed Ù val requestedSpeed Î allowed Ù low val requestedSpeed]
passive$_3$ [requestedSpeed: opt SPEED | def requestedSpeed Ù val requestedSpeed Î allowed Ù mid val requestedSpeed]
passive$_4$ [requestedSpeed: opt SPEED | def requestedSpeed Ù val requestedSpeed Î allowed Ù high val requestedSpeed]

active$_1$ [requestedSpeed: opt SPEED | def requestedSpeed Ù val requestedSpeed Î allowed Ù low val requestedSpeed]
active$_2$ [requestedSpeed: opt SPEED | def requestedSpeed Ù val requestedSpeed Î allowed Ù min(allowed) < val requestedSpeed £ min(allowed)+ stepSpeed]
active$_3$ [requestedSpeed: opt SPEED | def requestedSpeed Ù val requestedSpeed Î allowed Ù min(allowed)+ stepSpeed < val requestedSpeed < max(allowed)- stepSpeed]
active$_4$ [requestedSpeed: opt SPEED | def requestedSpeed Ù val requestedSpeed Î allowed Ù max(allowed)- stepSpeed £ val requestedSpeed < max(allowed)]
active$_5$ [requestedSpeed: opt SPEED | def requestedSpeed Ù val requestedSpeed Î allowed Ù high val requestedSpeed]

Figure 4: The States

Transition	Tests
[LoopPassive]/ NoAction	*a*
[LoopActive]/ NoAction	*b*
[Defining]/ Define	$c_1,\ldots, c_6$
[Deactivating]/ NoAction	$d_1,\ldots, d_9$
[Resuming]/ Resume	e_1, e_2, e_3
[Increasing]/ Increase	f_1, f_2
[Decreasing]/ Decrease	g_1, g_2

Figure 5: The subtransitions



$a = c_1, c_2$; $b = c_3, c_4$; $g = c_5, c_6$; $r = c_3, c_4, e_2$

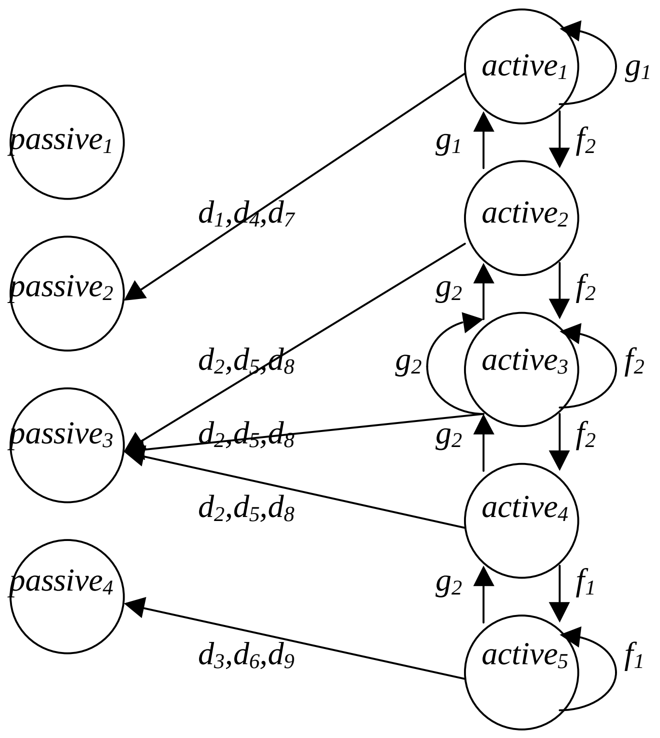


Figure 6: The EFSM M_{Ae}

Subtransition	Test
a	ac_1
b	c_1bf_2
c_1	c_1f_2
c_2	c_2f_2
c_3	c_3f_2
c_4	c_4f_2
c_5	c_5g_2
c_6	c_6g_2
d_1	$c_1d_1c_1$
d_4	$c_1d_4c_1$
d_7	$c_1d_7c_1$
d_2	$c_1f_2d_2c_3$
d_5	$c_1f_2d_5c_3$
d_8	$c_1f_2d_8c_3$
d_3	$c_5d_3c_1$
d_6	$c_5d_6c_1$
d_9	$c_5d_9c_1$
g_1	$c_1f_2g_1f_2$
g_2	$c_5g_2g_2$
f_1	$c_5g_2f_1g_2$
f_2	$c_1f_2f_2$
e_1	$c_1d_1e_1f_2$
e_2	$c_1f_2d_2e_2g_2$
e_3	$c_5d_3e_3g_2$

Figure 7: A test set for *ACC*

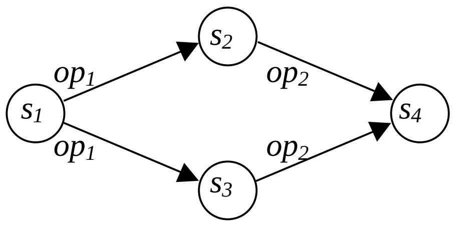


Figure 8: A deterministic sequence containing nondeterminism