Statechart testing method for aircraft control systems

K. Bogdanov*† and M. Holcombe

Department of Computer Science, The University of Sheffield, Regent Court, 211 Portobello Street, Sheffield S1 4DP, U.K.

SUMMARY
A number of current control systems for aircraft have been specified with statecharts. The risk of failures requires the use of a formal testing approach to ensure that all possible faults are considered. However, testing the compliance of an implementation of a system to its specification is dependent on the specification method and little work has been reported relating to the use of statechart-specific methods. This paper describes a modification of a formal testing method for extended finite-state machines to handle the above problem. The method allows one to demonstrate correct behaviour of an implementation of some system, with respect to its specification, provided certain specific requirements for both of them are satisfied. The case study illustrates these and shows the applicability of the method. By considering the process used to develop the system it is possible to reduce the size of the test set dramatically; the method to be described is easy to automate. Copyright © 2001 John Wiley & Sons, Ltd.

KEY WORDS: specification-based testing; formal methods; software testing; testing; aircraft systems software

1. INTRODUCTION
Many tools available today help accelerate software development. At the same time, verification of the compliance of an implementation to a specification is not significantly simplified by them. While automatic code generation eliminates the phase of translation of a system specification into code, it does not say anything about the behaviour of that program in the target controller. Verification by means of proofs of a compiler or of object code, while important, may not always provide the necessary assurance. At the same time, testing can be used as a form of a proof.

Statecharts [1,2] is a specification language derived from finite-state machines. It is rather rich in features including state hierarchy. Transitions can perform non-trivial computations unlike finite-state...
machines where they contain at most input/output pairs. Due to the graphical nature and a variety of constructs, statecharts have been widely used in projects; for instance, the TCAS 2 system [3,4] was specified with a very similar notation. The testing method to be described here is particularly suited for testing an implementation against a detailed specification or a design. Concurrency and static reactions as well as history and diagram connectors are not covered by the paper; refer to [5,6] for details.

The paper begins with an outline of the statechart notation using a part of an autopilot system\(^2\). In Section 3 the testing method is introduced for non-hierarchical statecharts and in Section 4 test case generation for hierarchical statecharts is given. Section 5 describes specific problems with the testing of the autopilot; concluding remarks can be found in Section 6.

2. STATECHARTS

The part of the autopilot considered supports the following commands:

- **climb** Makes an airplane climb (increase the height of flight). Execution of this command uses an elevator to make the plane face slightly upwards, continue this way until it reaches the desired height and then levels it (changes its position into the horizontal again).

- **descent** Is similar to **climb** but causes a descent.

- **flaps down** Moves flaps towards the tail of an airplane and pivots them downwards in order to increase the lifting and the dragging forces acting on the plane.

- **flaps up** Turns flaps upwards and retracts them so as to cancel the effect of the **flaps down** command.

- **terminate** Stops the command in progress. For the **climb** and **descent** commands it brings an elevator into the neutral position. This causes a plane to continue climbing or descending until an explicit command from a pilot. In the case when one of the flaps-related commands is active, flaps will be moved up and retracted.

- **level** Stops the climb or descent commands by levelling a plane at the current height.

The operation currently being executed by the autopilot is indicated on the cockpit instrument panel. The specification is given in Figure 1. The OFF state represents an idle autopilot; in the OPERATING state it performs a given task. The two different ways to cancel an execution of a command are provided by transitions **level** and **terminate**, with the latter having precedence over the former. The control of the plane over time is split into multiple phases, represented by the **do operating**, **do levelling** and **do terminating** transitions. Phases are changed by the remaining transitions.

In the following, the **underline** font is used to denote input and output variables and events (variables with a special property, explained in Subsection 2.3). Transition labels are given in **italics** and state names are CAPITALIZED. OFF and NORMAL are the initial states, indicated by transitions from

\(^2\)The case study being described is rather confidential and is based on examples provided by the DaimlerChrysler Research Laboratory (FT3/SM), the sponsor of the research. For this reason its presentation in this paper was modified so as to remove sensitive details of the original, at the same time preserving all problems and leading to the same conclusions.
The autopilot communicates with a pilot and on-board mechanisms. It serves as a controller which interprets button presses and sends appropriate commands to aircraft systems. Input events are climb, descent, flaps down, flaps up, terminate, and level. Output variables are operation and those controlling surfaces of the aircraft. The operation output can have one of the following values: climb, descent, flaps down, flaps up.

2.1. Transitions

It is possible to specify the operation of transitions of the autopilot as:

\[
\text{command} : \begin{cases} \text{df} \text{ climb} \lor \text{df} \text{ descent} \lor \text{df} \text{ flaps down} \\
\text{df} \text{ flaps up} / \text{time}' = 0 \land \\
(\text{df} \text{ climb} \implies \text{operation}' = \text{climb}) \\
\land \cdots \land \\
(\text{df} \text{ flaps up} \implies \text{operation}' = \text{flaps up}) \end{cases}
\]

\[\text{Only command, do operating and command complete labels are provided here.}\]
do-operating: time < duration(operation) \land \text{time}' = \text{time} + 1 \land 
perform-operation(operation, time) 

command-complete: time \geq duration(operation) \land \text{operation}' = \text{none} 

The part of the transition label before the ‘/’ sign represents the trigger, i.e. the precondition which is required to become true for a transition to occur. When a transition executes, the operation carried out is called an action. It is specified after the ‘/’ sign. For the command transition above, the action sets operation to the kind of command requested. The duration function determines the duration of an operation while the perform-operation generates signals to control surfaces, depending on the command and time. Primed variables describe new values of variables; unprimed ones describe current ones. df is a function returning true if an event it is supplied with was generated [7]. Transition labels where a precondition is satisfied are further referred to as triggered. A transition with a triggered label may only occur when a statechart is in its source state; such transitions are referred to as enabled. For example, if the statechart is in the OPERATING state with a very large value of time, then both command-complete and levelling-complete transitions will be triggered, but only command-complete will be enabled.

Triggers of transitions are evaluated regularly with a constant interval between evaluations. This causes do-operating to execute duration(operation) and perform-operation(operation, time) a number of times, followed by command-complete. Transitions are assumed to complete executing within the interval between evaluations.

### 2.2. State hierarchy

The statechart in the state NORMAL describes the behaviour of the autopilot when it is in that state. When NORMAL is entered via the terminating-complete transition, the transition terminates at the border of NORMAL and does not lead to any of the states inside. The blob (called a default connector) indicates the beginning of a default transition which is taken in this case. Usually it just points at some state to be entered. When the autopilot is initialized, the transition from the uppermost blob is taken to the NORMAL state, immediately followed by the one to the OFF state. If a state, such as OPERATING, is entered, all its higher-level states such as NORMAL are entered too.

A state containing a statechart is referred to as an OR state while that without any is referred to as a basic one. AND states contain concurrently executing statecharts; since concurrency is outside the scope of this paper, these states are not considered further. A statechart within a state (further called a substate statechart) is left when a transition from a state it is in is taken. For example, if the terminate transition is taken, the NORMAL state is left regardless of the state, OFF, OPERATING or LEVELLING, the autopilot was in. The equivalent statechart to that in Figure 1 is shown in Figure 2 where the hierarchy and connectors of Figure 1 are removed. To do the flattening, the state NORMAL has to be replaced by its contents and the outgoing transition terminating-complete replaced by the three corresponding transitions. Hierarchy of states imposes priorities on transitions; to retain these priorities, labels of transitions between OFF, OPERATING and LEVELLING have been appropriately modified in Figure 2. Furthermore, no connectors apart from a single default connector are present in the statechart.

Transitions in Figure 2 represent those which are taken in the original statechart in Figure 1. Such transitions are called full compound (abbreviated FCT) and consist of a transition from a state followed
by a number of default transitions. In the case where a statechart has no connectors, like the one in Figure 2, all its transitions are full compound. At any moment, it is possible to take only one full compound transition. Sets of states which are left and entered by full compound transitions are called configurations. For example, the initial configuration of the statechart in Figure 1 is \([\text{NORMAL}, \text{OFF}]\). Sequences of transitions (not necessarily those which could be taken) are called paths.

Interlevel transitions are the transitions which cross state boundaries. For instance, if \(\text{termination\_complete}\) was entering the OPERATING state, it would be considered interlevel.

2.3. Step semantics

In this section the Statemate step semantics given in [8] is described. Statecharts can follow one of the two types of semantics: synchronous or asynchronous.

Event generation or variable modification may enable transitions. For example, a pilot could press the \(\text{climb}\) button while the controller is in the OFF state, leading to the \(\text{command}\) transition becoming enabled. The system then takes any enabled full compound transition; if more than one is enabled, this is the case of non-determinism, prohibited during testing. Such an execution of an enabled transition is called a step. This transition may in turn generate events and make changes to variables. In the synchronous step semantics, during a step all changes, including those by the environment, are collected and applied after the step has ended; all events active in the step which were not generated again, such as \(\text{climb}\), are discarded. A possible loss of value by event variables is what makes them different from ordinary variables. Consider the \(\text{do\_operating}\) label to increase time which triggers \(\text{command\_complete}\). According to the step semantics, \(\text{command\_complete}\) will become enabled in the
next step, upon the next time triggers are evaluated. Additionally, since the climb event was discarded, transition command will not be triggered in the next step.

Asynchronous step semantics allows a statechart to perform more than one step in response to actions of an environment. This is accomplished by taking steps until no transition is enabled. Since this behaviour severely limits the observability and controllability of a statechart under test, it is prohibited during testing.

In this paper default transitions are required to have empty triggers; the general case is described in [5]. Applicability of the testing method also requires that only default transitions have empty triggers. This is necessary since otherwise chains of transitions may occur. For the benefit of controllability, the behaviour of a specification and implementation has to be deterministic; transitions taken in a step (as a part of a full compound transition) must not modify the same variable.

3. TEST GENERATION FOR SIMPLE STATECHARTS

In this section the testing method and its application to simple statecharts which do not contain state hierarchy are described. The statechart in the NORMAL state is simple, but the whole statechart is not. For this reason, the section only focuses on testing of the contents of NORMAL.

Simple statecharts are behaviourally-equivalent to X-machines. For this reason, the testing method for X-machines [9, 10] can be used for simple statecharts almost without changes. It has Chow’s W method [11] as a foundation and is based on a separation of function and transition diagram testing. The method concentrates on testing of the transition diagram; behaviour of labels of transitions is assumed to be tested in advance, which can be done, for example, using the disjunctive normal form (DNF) approach [12, 13]. As a result of the testing not revealing faults, the implementation is proven to be behaviourally equivalent to the specification. The requirements for the testing method are provided in Subsections 2.3, 3.2 and 3.3.

3.1. Introduction

According to the method, every transition is tested by visiting every state and generating events to trigger it from that state. The state that that transition has led the system to is then checked against the specified one. Unfortunately, it is not always possible to trigger a desirable transition. For example, consider the label command complete which occurs only when an externally inaccessible variable is set to some value. Consequently, it is necessary to augment it artificially by adding an extra input:

\[
\text{df trigger } \lor \text{ time } \geq \text{ duration(operation) } / \text{ operation'} = \text{ none.}
\]

Augmentation means a change to the system which does not affect the behaviour restricted to the original input and output variables. In the paper it is assumed that the time variable can be manipulated by a tester and thus augmentation of input is not necessary. For example, in order to trigger any of the time-dependent transitions, such as command, it is necessary to set time to a value larger than any used by these transitions in their duration statements. Such a value can be denoted by \(\infty\). For command, an extra output has also to be added because the pair of the triggering input time = \(\infty\) and output of operation = none does not uniquely identify it. Indeed, this trigger works for both levelling complete and termination complete, which produce the same output. An extra output
event \textit{testoutput} can be added to all of the completion transitions such that \textit{command\_complete} becomes
\[
\text{time} \geq \text{duration(operation)}/\text{operation}' = \text{none} \wedge \text{df testoutput} \wedge \\
\forall \text{vl testoutput} = \text{command\_complete}
\]

Here \text{vl} is used to define a value of a generated event. A further problem with observability is that if a tester attempts to invoke two labels \text{command} consecutively, the second transition, if it fires in a faulty implementation, will not modify an output, as if no transition was taken. For this reason, the \textit{testoutput} event has to be added to all transition labels of the autopilot.

To test the \textit{level} transition from the state \textit{OPERATING} to \textit{LEVELLING}, one should take the following steps.

- Starting from the initial state \textit{OFF}, enter \textit{OPERATING} by generating \textit{climb}. The \textit{operation} should change to \textit{climb} and \textit{testoutput} be generated with \text{vl testoutput} = \text{operation}.
- Trigger \textit{level} by generating \textit{level} and observe the \textit{operation} changing to \textit{none} as well as \textit{testoutput} changing to \textit{level}.
- Test that the \textit{LEVELLING} state was entered. This can be done by setting \textit{time} to a large value to trigger \textit{levelling\_complete} as transition \textit{levelling\_complete} exists only from the \textit{LEVELLING} state. The generation of the \textit{testoutput} event with the \textit{levelling\_complete} value has to be observed after triggering it.

Note that for every executed transition one needs to observe certain output changes to provide confidence that the right transition was executed. For example, if the implementation is faulty as shown in Figure 3, the \textit{level} transition does not change a state from \textit{OPERATING} to \textit{LEVELLING} and \textit{ANOTHER\_OFF} state lacks the \textit{command} transition. An attempt to trigger \textit{levelling\_complete} after invoking \textit{level} causes the \textit{command\_complete} transition to be executed. It will produce an output in \textit{testoutput} which is different from that of \textit{levelling\_complete}, allowing a tester to detect the first of the two faults.

In addition to testing \textit{level} between those two states, one needs to make sure that it does not emanate from any other state. This test is needed because in a faulty implementation transition \textit{level} could exist.
from some state other than OPERATING. In order to perform the test it is necessary to visit states OFF and LEVELLING and generate the \texttt{level} event. This should invoke no transition in the above states and thus no \texttt{testoutput} event would be generated.

### 3.2. Test case generation

During test case generation statecharts are treated as finite-state acceptors with inputs being transition labels. The result of the test case generation is thus a set of sequences of transition labels such as \texttt{command level}. In order to construct it, one has to build auxiliary sets. There are three of them:

**Set of transitions** (denoted by $\Phi$) is the set of transition labels of a statechart. It has to be constructed because the test method essentially tests an equivalence between the transition structures of a specification and an implementation.

For the NORMAL state of the autopilot,

$$\Phi = \{\text{command, command\_complete, do\_operating, level, levelling\_complete, do\_levelling}\}.$$  

**State cover** (denoted by $C$). To perform testing, all states have to be visited. A state cover $C$ is a set of sequences of transition labels, such that one can find an element from this set to reach any desired state starting from the initial one,

$$C = \{I, \text{command, command\_level}\}.$$  

Here $I$ is used to denote an empty sequence of transitions. In the following table the list of states is shown together with the corresponding elements of $C$.

<table>
<thead>
<tr>
<th>State</th>
<th>Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>OFF—the initial</td>
<td>$I$</td>
</tr>
<tr>
<td>OPERATING</td>
<td>$\text{command}$</td>
</tr>
<tr>
<td>LEVELLING</td>
<td>$\text{command_level}$</td>
</tr>
</tbody>
</table>

**Characterization set** (denoted by $W$). A characterization set allows a tester to check the state arrived at when triggering some transition, i.e. if it is the one expected. Above, in Subsection 3.1, the LEVELLING state was checked by trying to follow a path which exists from LEVELLING and not from any other state. For every pair of states, one can construct a path which exists from one of them and not from the other. For example, for states OFF and OPERATING it is possible to select $\text{command\_complete}$. A characterization set consists of such sequences for every pair of states,

$$W = \{\text{command\_complete, levelling\_complete}\}.$$  

Each element of this $W$ is a sequence consisting of a single transition.

For testing it is further assumed that specification of a system does not contain redundant states (having the same behaviour as some others), or those not entered by any transitions. This property is called \textit{minimality}. Let $n$ be the number of states in a specification (3 in the NORMAL state) and $m$ the maximal number of states in a minimal implementation. The actual implementation is not required to
possess the property of minimality. It is still possible to estimate the number of states in a behaviourally-
equivalent one which does satisfy it.

For testing a faulty implementation which may have one or more missing states, it is possible to
assume that the maximal number of states in such an implementation is at most \( n \). With \( m = n \), the set
of test cases \( T = (C \cup C \ast \Phi) \ast W \) (for some sets of sequences \( A, B \), \( A \ast B \) is the set multiplication
operation such that \( A \ast B = \{a b \mid a \in A, b \in B\} \), where \( a b \) is a concatenation of sequences \( a \) and \( b \)).
This test set deals with visiting every state (by applying \( C \) and verifying it (\( W \)). Every transition from
it is then tried (\( C \ast \Phi \) and the expected state checked (\( W \)).

Consider a faulty statechart with an extra state ANOTHER\_OFF reachable only from the
OPERATING one (Figure 3). When testing transitions of this statechart using \((C \cup C \ast \Phi) \ast W\),
all of them are triggered from all states the system can reach using the state cover. Consequently, only
transitions of \( W \) are tried from ANOTHER\_OFF rather than all transitions of \( \Phi \). This prevents such a
set of test cases from detecting the fault because \( W \) cannot distinguish ANOTHER\_OFF from OFF. In
order to cope with extra states, sequences of more than one transition, such as all pairs of transitions,
have to be tried from every state. The part of the set of test cases where all pairs of transitions are tried
can be expressed as \( C \ast \Phi \ast \Phi \ast W \). In the case where more than one extra state can be assumed, these
sequences of transitions have to be made longer. For the possibility of \( m - n \) extra states the set of test
cases is

\[
T = C \ast (\{I\} \cup \Phi \cup \Phi^2 \cup \cdots \cup \Phi^{m-n+1}) \ast W
\]

The size of the set of test cases can be computed [10] as follows:

\[
\text{Size} \, T \approx n^2 \ast |\Phi|^{m-n+1}
\]

The result follows from the size of \( C \) being \( n \) and the maximal size of \( W \) being \( n - 1 \) under assumption
that the number of labels \( |\Phi| \) is much greater than 1. For \( m = n \), \( \text{Size} \, T \approx n^2 \ast |\Phi| \).

3.3. Test data generation

For the statechart testing method to be applicable, all transitions have to be implemented correctly. This
means that a specification may be implemented with a different number of states, transitions could
traverse them in possibly random ways, but triggers and actions (Subsection 2.1) of transitions are
correctly implemented. Moreover, for any transition one should be able to make changes to externally
accessible variables of the system such that the desired transition will be triggered and all other
transitions will not. It is also required for the combination of a triggering input and an output from
a transition to identify it uniquely. This is needed because, having triggered a transition, a tester
needs to be sure which one occurred. The requirement of being able to trigger is called \textit{t-completeness}
and that of the input-output pair being able to identify a transition is called \textit{output-distinguishability}.

These two, a requirement of absence of shared labels of transitions between states of a specification,
synchronous behaviour of an implementation (Subsection 2.3) and structural requirements (transitions
cannot directly enter default connectors and default transitions cannot be interlevel) comprise the
\textit{design for test} condition. In the example t-completeness and output-distinguishability are not satisfied
(Subsection 3.1).

The actual test data generation typically involves replacing each label in test case sequences with
a triggering input for it. For example, a possible input sequence corresponding to \{command level\} is
To construct expected outputs, the reaction of the specification to the input data sequence is computed. A test sequence \([\text{climb level}]\) applied in the initial OFF state corresponds to operation changing to \text{climb} and then to \text{none}. In the case where no transition gets triggered by some input, an implementation is expected to make no changes to observable variables and produce no events.

At the end of every test sequence the \textit{reset} label is added which brings the system into the initial configuration with the initial values of all variables. Correct and deterministic behaviour of transitions with this label from all states is also required by the testing method.

The requirements described in this section, those given in Subsection 2.3, minimality of a specification and the assumption of a bounded number of states in an implementation (Subsection 3.2) comprise the requirements for the testing method. If they are satisfied, the test set can be generated, applied and used to provide provable correctness of behaviour of an implementation to the specification as a result of testing not detecting faults [5]. It is best to ensure that these conditions are satisfied at specification time so that no changes in an implementation are needed in order to carry out testing.

4. TEST CASE GENERATION FOR STATE HIERARCHY

In this section the testing method for flat statecharts will be extended to handle hierarchical statecharts. Test data can be obtained as described above.

The most simple approach to testing a hierarchical statechart is by turning it into a behaviourally-equivalent one without substate statecharts. For example, Figure 2 depicts the result of flattening of the statechart in Figure 1. This process leaves a simple but, in practice, huge statechart. Generation of a set of test cases for it is an easy operation as described in Section 3. This is essentially what is done by [14,15].

As an alternative to the above, an approach of incremental test case development using the hierarchical structure of statecharts is proposed. It has the advantage of following the development process and thus providing a possibility of updating the set of test cases to reflect specification changes made. Reduction of the number of test cases is described in Subsection 4.2.

4.1. General case

Test case generation begins with the construction of a tuple \((\Phi, C, W)\), called the test case basis (abbreviated \textit{TCB}) for the whole statechart and every non-basic state of it separately. Afterwards, the tuples are combined (\textit{merged}) in a way described below. From the resulting tuple a set of test cases can be constructed following Equation (1).

The elements of the test case basis for the NORMAL state have been described in Subsection 3.2; those for the whole statechart of the autopilot (considering the state NORMAL to be a basic one since the whole statechart is considered separately from its contents) are as follows:

\[
\Phi_{\text{WHOLE STATECHART}} = \{\text{terminate, termination complete, do terminating}\},
\]
\[
C_{\text{WHOLE STATECHART}} = \{1, \text{terminate}\},
\]
\[
W_{\text{WHOLE STATECHART}} = \{\text{terminate}\}
\]
To obtain the elements of the resulting tuple \((\Phi^{\text{merged}}, C^{\text{merged}}, W^{\text{merged}})\), the following merging rules are proposed for the above elements with an explanation to follow:

\[
\Phi^{\text{merged}} = \Phi_{\text{WHOLE STATECHART}} \cup \Phi_{\text{NORMAL}} \\
= \{\text{terminate, termination, complete, do terminating, command, command, complete,}
\text{ do operating, level, levelling, complete, do levelling}\},
\]

\[
C^{\text{merged}} = C_{\text{WHOLE STATECHART}} \cup \{\text{path in } C_{\text{WHOLE STATECHART}} \text{ to enter NORMAL} \} \star C_{\text{NORMAL}} \\
= \{1, \text{ terminate} \} \cup \{1 \} \star \{1, \text{ command, command level}\} \\
= \{1, \text{ terminate, command, command level}\},
\]

\[
W^{\text{merged}} = W_{\text{WHOLE STATECHART}} \cup W_{\text{NORMAL}} = \{\text{level, command, complete, termination, complete}\}
\]

The rules use the fact that concatenation of an empty sequence and some sequence is that sequence, i.e. \([\emptyset] \star \{seq\} = \{seq\}\).

When constructing sets \(C\) and \(W\) of the test case basis, all interlevel transitions are ignored, because the test case basis focuses on the structure of a state on a single level of the hierarchy. For each of the interlevel transitions, it is possible to identify the lowest common OR-state ancestor of its initial and final states, called scope. Interlevel transitions can be included in sets \(\Phi\) for their scope states.

Merging rules are applied in bottom-up fashion until a merged TCB for the top-level state is constructed. For example, if the state OPERATING had a substate statechart, it would be necessary to merge its test case basis with sets constructed for NORMAL before merging TCB of NORMAL with that for the whole statechart. The resulting tuple \((\Phi, C, W)\) can be used to generate the set of test cases for the autopilot using Equation (1). It can be observed that the generation process above resulted in the sets possible for the flattened statechart in Figure 2, i.e. \(C^{\text{merged}}\) permits visiting every state, \(W^{\text{merged}}\) distinguishes every pair of them and \(\Phi^{\text{merged}}\) has labels of all full compound transitions.

The size of the set of test cases can be estimated using Equation (2) as follows:

\[
\text{Size}_T \approx (n_w + n_N)^2 \star |\Phi_{\text{WHOLE STATECHART}} + \Phi_{\text{NORMAL}}|^{m_w + m_N - n_w - n_N + 1}
\]

where \(m\) and \(n\) refer to the number of states in an implementation and of a specification, respectively (Subsection 3.2); subscript \(W\) refers to the whole statechart and \(N\) to the NORMAL state. For the autopilot, \(\text{Size}_T = 111\) (precisely) under assumption of no extra states in the implementation of the statechart.

### 4.2. OR-state refinement

A specification is usually constructed gradually. One could either develop it in a stepwise manner and, having finished it, generate code, or develop the specification and implementation in parallel. The latter means that each change in the specification is accompanied by an appropriate change in the implementation. In the case of parallel development, it is possible to avoid leaving all testing to the final pre-release stage. Additionally, such a development process can make certain kinds of implementation faults impossible, like the one depicted in Figure 4. Its faults are that transition terminate is not present from the OFF state and it enters the OPERATING one instead of TERMINATING when taken from the LEVELLING state. These faults become impossible if initially the specification in Figure 5 is built and implemented and then the statechart ‘placed’ in the NORMAL state, resulting in the system depicted in...
do_terminating

do_operating do_levelling

terminate

level

command

command_complete

levelling_complete

TERMINATING

termination_complete
do_terminating

do_operating

terminate

OFF

TERMINATING

NORMAL

LEVELLING

OPERATING

Figure 4. The faulty implementation which cannot happen if development keeps the implementation consistent with every change in the specification.

Figure 5. The top-level statechart of the autopilot.

Figure 1. This follows from the terminate transition having a precedence over any functionality inside the NORMAL state.

With the restrictions described, the size of a set of test cases and thus the size of a test set for such a refinement is greatly reduced even for the simple statechart of the autopilot. Faults related to the statechart not exiting some of the substates of NORMAL while exiting others by the terminate transition are not possible; a tester has to consider only the set of labels of transitions defined inside NORMAL when testing it (statechart specifications are required not to share transition labels between states, Subsection 3.3). This influences the rules used to construct the set of test cases using
Equation (1) for a statechart with a refined state:

\[
T = C_{\text{WHOLE\_STATECHART}} \ast (\{I\} \cup \Phi_{\text{WHOLE\_STATECHART}} \cup \cdots \cup \Phi_{\text{WHOLE\_STATECHART}}^{m_W-n_W+1}) \ast W_{\text{WHOLE\_STATECHART}}^C
\cup \{\text{path in } C_{\text{WHOLE\_STATECHART}} \text{ to enter NORMAL}\} \ast C_{\text{NORMAL}}^C
\ast (\{I\} \cup \Phi_{\text{NORMAL}} \cup \cdots \cup \Phi_{\text{NORMAL}}^{m_N-n_N+1}) \ast W_{\text{NORMAL}}^C
\]

The first part of the union tests the whole statechart treating NORMAL as a basic state. The second enters the substate statechart and tests it separately from the whole one. Numbers \(m_W\) and \(n_W\) correspond to the expected number of states in the implementation of the whole statechart and in the specification of it \((n_W = 2)\) respectively; \(m_N\), \(n_N\) are the corresponding numbers for the statechart in the NORMAL state \((n_N = 3)\).

The size of the above set of test cases can be estimated to be

\[
\text{Size}_T = \text{Size}_{\text{WHOLE\_STATECHART}}^C + \text{Size}_{\text{NORMAL}}^C
\approx n_W^2 \ast |\Phi_{\text{WHOLE\_STATECHART}}|^{m_W-n_W+1} + n_N^2 \ast |\Phi_{\text{NORMAL}}|^{m_N-n_N+1}
\]

This gives \(\text{Size}_T = 45\) (precisely), which is less than half of that obtained in Subsection 4.1. Under assumption of one or more extra states in an implementation the difference can be much larger.

5. PROBLEMS WITH TEST GENERATION FOR THE AUTOPILOT

The autopilot system does not comply with the requirements for the testing method (Subsection 3.3). This section describes how the method can be applied to it nevertheless.

The specification of the autopilot was given in Figure 1; Figure 6 shows an implementation, which has been reverse-engineered from the code with which the authors were supplied. Comparing this with Figure 1, it is clear that the implementation has a different functionality corresponding to specification labels command, command\_complete and level. Specifically, the first two have been split into three, climb\_descent, flaps\_down, flaps\_up and the corresponding completion and do\_transitions. The behaviour of the level transition is such that it can only be taken if climbing or descent is in progress. Therefore, it was coded to leave only the CLIMB\_DESCENT state.

The set of labels of transitions implemented is different from that specified and some transitions are implemented with a different domain than that specified. Consequently, making the case study system comply with requirements for the testing method requires the following:

1. Finding a correspondence between labels of the specification and that of the implementation. This is necessary for a separate testing of labels, which helps ensure the correctness of their implementations prior to the application of the test set derived above. While trivial for the case study, this is not an easy task in general because an implementation may combine labels of transitions, making it not amenable to a direct application of the statechart testing method being described. In such cases it could be possible to reverse-engineer an implementation and prove its behaviour to be compliant with a specification. This area, though, is not covered in this paper.

2. Augmenting those labels, possibly using a reverse-engineered implementation. Performing this task allows a tester to assure t-completeness and output-distinguishability. Reverse-engineering is unnecessary for systems built with testing in mind. Unfortunately, in many cases
implementations do not implement transitions directly; instead, labels are optimized for the intended behaviour. For instance, the terminate transition was implemented such that it will only occur when any of the surfaces is deflected. In such cases, detailed design documentation or code have to be used to determine triggers and/or outputs of transitions.

3. Driving an implementation through test sequences sufficiently fast to ensure that no time-dependent completion transitions become triggered. If this cannot be done, the preconditions related to time can be augmented. With this, the required synchronous behaviour of the autopilot can be achieved during testing.

4. Test outputs have to be added to all transitions of the autopilot since it is not possible to distinguish between the same transition being taken more than once consecutively and it being taken once followed by no transitions at all. The time variable also has to be under control of the tester.

Other testing requirements are satisfied.

Test case generation produces sequences of labels to trigger. Generally, one does not have to consider them earlier, in the design for test, since it is always possible to find appropriate test data. Often this means that for every transition the triggering input and expected output are chosen in advance. In practice, however, a minimal amount of augmentation is preferred and thus selection of test data could be attempted in such a way as to minimize the number of testing events, considering every specific
sequence of labels used in the set of test cases. For instance, the sequence command level often has to be followed. The definition of the level is such that it will only occur if the command is climb or descent. For this reason, it is better to use climb as a trigger for command because this allows level to be used (otherwise the level transition would have to be augmented with a testing input in order for it to be taken after command). Note that this is possible because command is the only transition which sets the internal data on which the trigger of level depends. If there were more than one, a tester would have to consider special triggers for them all.

With the proposed trigger for command, the implementation is only tested to the extent it was specified, i.e. the FLAPS_DOWN and FLAPS_UP states are not considered. This is not a limitation of the described testing method but a failure of the implementation to follow the requirements for the method. Testing can nonetheless be performed, but the scope of the implementation it covers is limited. In principle, selection of different triggers for different sequences of labels or execution of every sequence with different test data may allow one to test systems where the implementation splits transitions according to the input received. For example, test cases could be converted to test data using all of the flaps down, flaps up, climb and descent as triggers for command. While increasing the size of the test set, this would provide more thorough testing for non-compliant implementations. If it is known which transitions can be split in an implementation and in which way, the two-stage testing approach similar to the Wp method [16] could be used. Initially, testing focuses on the core of the transition structure using reliable parts of transitions; afterwards the rest of the structure is tested [5]. Applied to the case study, one could ensure that certain parts of split transitions such as climb descent and climb descent complete are reliable. Usage of these transitions and those which were not split makes it possible to test most of the transition structure. After that, flaps down, flaps up, their complement and do transitions are tested from every state. Derivation of testing requirements for this approach is left for the future.

In the description of the testing method, the distinguishing of states was based on triggering transitions and observation of whether they occurred or not. In some cases, different techniques could be used to distinguish states. Indeed, the autopilot uses dashboard lights to indicate its mode of operation and these lights could help a tester to distinguish states without taking any transitions. Observation of an input’s effect on control surfaces of an aircraft could also be used to distinguish states. This can be accomplished by periodic sampling of the position of a surface and identification of the trend in its movement. Since levelling can be expected to be performed faster in the LEVELLING state than in CLIMB_DESCENT, it would be easy to tell the two states apart.

6. CONCLUSION

The test method developed for statecharts appears to be applicable to a reasonably realistic system. In the case study the implementation was similar to the specification, but the sets of transition labels were different. These differences could be divided into two groups, labels being split and labels being implemented with different domains. The former prevented testing of the whole of the transition structure of the implementation but most of it was tested; for the latter it was impossible to derive test data from the specification itself and the implementation had to be used. In practical development of safety-critical systems, this would not be a major problem, as deviations from a specification should
be documented and these details combined with the specification are expected to permit derivation of test cases and test data.

Theoretical foundations of the testing method have been shown to be correct in [5]. Reference tool support is provided by the TestGen tool, written by one of the authors in Java.

ACKNOWLEDGEMENT

This work was funded by the DaimlerChrysler Research Laboratory (FT3/SM), Berlin, Germany.

REFERENCES