Formal Validation of Hierarchical State Machines against Expectations

Ian Toyn         Andy Galloway
Dept of Computer Science,
University of York,
Heslington, York, YO10 5DD, UK
{ian,andyg}@cs.york.ac.uk

Abstract

This paper explains some analyses that can be performed on a hierarchical finite state machine to validate that it performs as intended. Such a hierarchical state machine has transitions between states, triggered by conditions over inputs, with outputs determined per state in terms of inputs. Intentions are captured per state as expectations on input values. These expectations are expressed using the same condition language as transition triggers, extended to constrain rates of change as well as ranges. The analyses determine whether the expectations are consistent and whether the state machine conforms to the expectations.

For the analyses to find no problems, the explicit expectations on the root state will be at least as strong as the implicit expectations of the state machine. One way of using the analyses is to reveal these implicit expectations.

The analyses have been automated for statecharts built with The MathWorks' Stateflow tool.

1. Introduction

In model-based design, models should be validated against intentions and corresponding code should be verified against the model. This paper is concerned with the validation of models that are built as hierarchical finite state machines (section 2). It explains some analyses that can be performed on a hierarchical finite state machine to validate that it performs as intended [6].

Each state of the state machine can be annotated with the designer’s expectations for that state, then the analyses determine whether the expectations are consistent and whether the state machine conforms to the expectations. When a tool implementing an analysis finds an inconsistency or non-conformance, a counter-example may be reported. This counter-example guides the designer to correct the state machine or to mend the expression of expectations.

For the analyses to find no problems, the expectations that have been made explicit as annotations on the root state will be at least as strong as what the state machine implicitly expects of the environment in which it operates. Further validation is required to determine whether those expectations are satisfied by the operational environment.

The analyses have been automated by extensions to The MathWorks’ Stateflow tool [18]. The user chooses when to initiate analyses. They are performed by automatic theorem proving techniques, covertly to the user. When these tools find a problem, the user is notified and usually a counter-example is presented.

The next section defines the notations to be analysed. Section 3 provides some context concerning a problem domain (modelling control systems) and a solution domain (Stateflow). Section 4 presents an example to illustrate the analyses and what they can reveal, then section 5 formalises the analyses in general terms. The extensions to Stateflow to support the analyses are illustrated in section 6. Validation of a state machine’s expectations of its environment is considered in section 7. The paper ends with assessments of the analyses and tools.

2. State machines and expectations

The hierarchical state machines for which analyses are defined in this paper comprise basic states, grouped hierarchically by exclusive-or states. States are connected by transitions. Transitions have trigger conditions over inputs, local variables and constants, which can be of integer and boolean types. Outputs and local variables have their values defined per state. Transitions may be drawn via junction connectors (but the presentation of the analyses assumes that any connectors have already been expanded away). This set of notations is a syntactic subset of what is usually called hierarchical state machines or statecharts. (Statecharts were introduced by Harel [7] as finite state machines extended with hierarchy, concurrency and communication.
They have become a well-known graphical formalism with many dialects, such as Statemate [8], UML [4] and Stateflow. In this paper, the particular concrete syntax used for these notations is that of Stateflow statecharts.

The semantics for these hierarchical state machines is transition-then-compute: the inputs, local variables and constants may trigger a transition, in which case the current configuration of active states becomes determined by the destination state of the transition; then new values are calculated for the outputs and local variables according to the latest configuration of active states.

The subset and its semantics are chosen to be easy for machines to analyse automatically and also easy for humans to understand. The syntax is similar to that of the Safecharts language [1].

An expectation is a condition over input values, local variables and constants. Each state can have several expectations as annotations on it. These expectations serve distinct roles, e.g. one of a state’s expectations says what input values would cause the state to remain active. (Other roles are explained later.) In this paper, the particular concrete syntax used for expectations is that of Stateflow conditions (as used for transition triggers), with the following extensions. For conditions \( x \) and \( y \), the implication notation \( x \Rightarrow y \) is an abbreviation for \( \sim x \vee y \). Implication is used in analyses to express that \( y \) is established by \( x \). It has lower precedence than all other condition notation. For an input \( I \), the prefix-primed notation \( 'I \) is used to refer to the last (previous) value of \( I \). Expectations about next input values can refer to last input values, allowing constraints on an input’s rate of change, e.g. \( I - 'I < 42 \). References to values before the last (as might be written using multiple prefix primes) are not allowed.

### 3. Problem and solution domains

This section provides some context that is useful for justifying why the analyses are valuable and for explaining their implementation in our tools. This context is not needed for sections 4 and 5, where what the analyses do is explained.

A control system receives inputs from sensors and sends outputs to actuators, in such a way as to control something – the controlled system. There might also be a command system, by which a human operator communicates with the control system. All of these components of the overall system might operate in different modes at different times, with their current modes being dependent on previous modes. The mode aspect of a control system can be conveniently modelled using a hierarchical state machine.

Stateflow is a tool for building, simulating, and automatically coding from such models of control systems. Stateflow works in the environment provided by The MathWorks’ Simulink tool [17], which models (not just mode aspects of) control systems as block diagrams.

A Simulink block diagram is a directed graph in which the nodes are blocks and the arcs are signals. Each block models some function within a control system. The roots of the graph are called source blocks, and the leaves of the graph are called sink blocks. Inputs from sensors are represented by source blocks, and outputs to actuators are represented by sink blocks. Each block has inputs and outputs to which signals are connected, and can also have local variables. Blocks can be arranged in hierarchically nested subsystems, and can be connected in cyclic arrangements.

This paper is concerned with Simulink block diagrams of digital control systems that operate in discrete time. At any particular point in discrete time, a block computes values for its outputs from the current values of its inputs and local variables. It also computes new values for its local variables similarly. The blocks perform their calculations synchronously, so the outputs of a whole diagram are calculated from the inputs to the diagram at that point in discrete time. Any cyclic arrangements of blocks must include unit delay blocks, where the output of a cycle becomes the input used at the next point in discrete time.

A Stateflow statechart exists as a block in a Simulink block diagram. The characteristic feature that distinguishes a Stateflow block from other Simulink blocks is that it chooses different calculations of outputs according to the current state in its statechart.

Experience has shown that the modelling of mode transitions is an area where mistakes can be difficult to detect [2, 12]. In many control systems, a large proportion of functionality is dedicated to the detection and management of mode changes, e.g. failures. Simulations (animations of particular test scenarios), as offered by Simulink and Stateflow, contribute to validation, but are rarely sufficient to cover all cases. Moreover, standards governing the development of safety-critical control systems mandate stringent evidence that models are valid [14]. Hence analyses that can validate a statechart are potentially valuable. The analyses discussed below reveal mistakes such as inappropriate trigger conditions, missing transitions, and contradictory and missing expectations. This has been shown by small industrial case studies, performed first manually and subsequently with automated tools.

### 4. Example

This section explains the analyses by telling the story of their application to an example. The example is part of a mode change manager that has been abstracted to remove incidental detail. The requirements are stated in natural language, so are inevitably somewhat vague.
4.1. Requirements

A mode change manager is required to permit a demanded mode change only when a sensed signal is below a particular limit and tests of the actuators for the mode change are passed. (Actuators may fail, and tests are needed to verify that they are operating correctly.)

4.2. State machine

Figure 1 is a hierarchical state machine for part of a mode change manager. It contains a deliberate flaw to illustrate what can be revealed by the analyses.

The state machine models a change from ModeA to ModeB. The inputs to the state machine are as follows.

- ChangeDemanded: whether a mode change is demanded
- Signal: value of sensed signal
- TestsCompleted: whether tests have been completed
- TestsPassed: whether tests have passed
- ChangeCompleted: whether mode change has completed

The outputs from the state machine are as follows.

- DoTests: whether to do tests
- ChangeMode: whether to change mode

The initial state of the model is Operating, which is inside ModeA. State Testing models the testing of the actuators for the mode change. State ModeChange models the period while the controlled system is changing from ModeA to ModeB. Trigger conditions can use notation for disjunction (\(\lor\)), conjunction (\(\land\)) and negation (\(\neg\)), as well as the usual relational operators on numbers, their relative precedences being increasing as listed. Each state should prescribe values for outputs DoTests and ChangeMode, but these definitions have been omitted from figure 1 as they are not needed by the analyses.

A more concrete model of a mode change manager might decompose Testing to a collection of separate tests, might specify how the model is to proceed after testing fails, might have additional outputs to reveal its view of the current mode, and might model more than one mode change (hence ModeB might have analogous substates to those of ModeA), possibly between more than two modes.

4.3. Analyses

The requirements (section 4.1) say that a mode change is to be permitted only when a sensed signal is below a particular limit. This requirement can be expressed in the terms of the state machine as an expectation on the value of the Signal input on entry to state ModeChange. This is called an initial assumption of the state.

**ModeChange**’s initial assumption:

\[\text{Signal} < \text{LIMIT}\]

In this paper, names that are entirely capitalised, such as LIMIT, denote constants.

**Initial assumptions established.** An analysis can be performed to verify that a state’s initial assumptions are established. This analysis determines whether the initial assumptions are established by the entering transition’s trigger in conjunction with what the transition’s source state was expecting of the next input values, i.e. the inputs that trigger the transition. If that next assumption refers to last input values, then the source state’s assumptions on the last input values are also relevant to determining whether the initial assumptions are established. Those assumptions are called the state’s last assumptions. In this example, no next or last assumptions have been mentioned for state Testing (yet), and so the initial assumptions established analysis for state ModeChange is as follows.

\[\text{TestsCompleted} \land \text{TestsPassed} \Rightarrow \text{Signal} < \text{LIMIT}\]

This is not always True. Having defined LIMIT to be the constant 250, a counter-example (as found by the tools) is as follows.

\[\text{Signal} == 250\]

The counter-example does not mention that TestsCompleted and TestsPassed are True, since being in the left-hand side of the implication allows these values to be assumed. This counter-example reveals a flaw in the state machine: the
value of \( \text{Signal} \) can change while the \( \text{Testing} \) state is active, and hence be out of the required range by the time a mode change is initiated.

One might think of moving the trigger condition \( \text{Signal} < \text{LIMIT} \) from before \( \text{Testing} \) to after \( \text{Testing} \), but that raises a question about when is it reasonable (and indeed safe) to test the actuators for the mode change. Let us assume that they should be tested only in circumstances where a mode change would be permitted, so the requirements need to be clarified with the following expectation.

\[ \text{Testing's last assumption:} \]
\[ \text{Signal} < \text{LIMIT} \]

The analysis for \( \text{ModeChange} \)'s initial assumptions established now looks like this.

\[ \text{Signal} < \text{LIMIT} \]
\[ \& \text{TestsCompleted} \& \text{TestsPassed} \Rightarrow \]
\[ \text{Signal} < \text{LIMIT} \]

This still is not always \text{True}, as it involves values of \( \text{Signal} \) at different times. A next assumption recording \( \text{Signal} \)'s expected maximum rate of increase, combined with keeping track of time and imposing a timeout in state \( \text{Testing} \), allows a worst case calculation of a safe limit to be used in a revised last assumption.

\[ \text{Testing's next assumption:} \]
\[ \text{Signal} - '\text{Signal} < \text{MAXDELTA} \]

\[ \text{Testing's last assumption:} \]
\[ \text{Signal} < \text{LIMIT} \]
\[ - (\text{TIMEOUT} - \text{TimeTesting}) \]
\[ \ast \text{MAXDELTA} \]
\[ \& \text{TimeTesting} < \text{TIMEOUT} \]

This refers to a new local variable \( \text{TimeTesting} \), which is introduced and maintained in the state machine. The trigger of the transition into \( \text{Testing} \) is revised to use the worst case calculation.

The initial assumptions established analysis, taking into account these assumptions and also the definition of how the local variable \( \text{TimeTesting} \) is updated, is now provable.

Figure 2 shows the corresponding revisions to the model, and some further revisions caused by limitations in the current tools: \( \text{ModeChangeInit} \) is introduced because initial assumptions are allowed to be recorded only on superstates (non-basic states); the local variable \( \text{TimeTesting} \) is introduced because Stateflow does not provide direct access to the time elapsed in a state, which is needed for recording expectations; and identical entering (en) and during (du) actions are required to express the desired hierarchical state machine semantics.

Next assumptions established. Having introduced last and next assumptions on \( \text{Testing} \), there are further analyses to check whether these are established. The next assumptions are established if all next input values that the parent state (\( \text{ModeA} \)) is expecting are expected by \( \text{Testing} \). If the parent state’s next assumptions refer to last input values, then \( \text{Testing} \)'s last assumptions are relevant, thus allowing \( \text{Testing} \) to expect fewer next values than its parent. This analysis leads to a new assumption being recorded.

\[ \text{ModeA's next assumption:} \]
\[ \text{Signal} - '\text{Signal} < \text{MAXDELTA} \]

The analysis for \( \text{Testing's next assumptions established} \) looks like this, and it is provable.

\[ \text{Signal} < \text{LIMIT} \]
\[ - (\text{TIMEOUT} - \text{TimeTesting}) \]
\[ \ast \text{MAXDELTA} \]
\[ \& \text{TimeTesting} < \text{TIMEOUT} \]
\[ \& \text{Signal} - '\text{Signal} < \text{MAXDELTA} \Rightarrow \]
\[ \text{Signal} - '\text{Signal} < \text{MAXDELTA} \]

Note that this is a consistency check between expectations. Also note how \( \text{Testing's next assumptions have been propagated up to its parent ModeA} \). The analysis to show that \( \text{ModeA's next assumptions are established similarly propagates them up to the root state} \). Hence they are seen to be expectations of the state machine on its environment.
**Last assumptions established.** The check for whether last assumptions are established has to be done for each possible entering transition. The last assumptions are established if all input values that trigger the particular entering transition are amongst those expected according to the last assumptions. The transition’s source state’s next assumptions can restrict those values, and so are relevant to the analysis. Similarly, the transition’s source state’s local definitions can restrict the values to be considered. If those next assumptions refer to last input values or to local variables, then the transition’s source state’s last assumptions are also relevant.

As there is only one transition entering *Testing*, there is only one last assumptions established analysis to be done for that state.

\[
\begin{align*}
  TimeTesting &= 0 \\
  &\& \text{ChangeDemanded} \\
  &\& \text{Signal} < \text{LIMIT} - \text{TIMEOUT} * \text{MAXDELTA} \Rightarrow \\
  &\quad \text{Signal} < \text{LIMIT} \\
  &\quad \quad - (\text{TIMEOUT} - \text{TimeTesting}) * \text{MAXDELTA} \\
  &\& \text{TimeTesting} < \text{TIMEOUT}
\end{align*}
\]

For suitable definitions of the constants, this is provable.

**Last assumptions preserved.** During simulations, there is an alternative to taking a transition, which is for the current state to remain active. A further analysis checks whether a state’s last assumptions are preserved when the state remains active. It involves making explicit the state preservation condition. From the simulation perspective, a state remains active if none of the exiting transitions are triggered. For a Stateflow statechart, a state preservation condition differs from a trigger condition on a reflexive transition (a transition from a state to the same state): because the state is not exited, the timer used by *before, at and after* conditions is not reset, and during (du) actions are executed rather than entry (en) actions.

Taking into account what next values are expected while in the state, a state preservation condition can be stronger than just the complement of the disjunction of the existing transitions’ triggers. For state *Testing*, the following expectation is recorded.

\[
\begin{align*}
  \text{Testing’s state preservation condition:} \\
  &\sim \text{TestsCompleted} \& \text{TimeTesting} < \text{TIMEOUT}
\end{align*}
\]

The last assumptions are preserved if all input values admitted by the state preservation condition are amongst those expected according to the last assumptions. The state’s next and last assumptions, and its local definitions, may all be relevant to the analysis.

The analysis for *Testing’s last assumptions preserved* looks like this, and it is provable.

\[
\begin{align*}
  TimeTesting &= \text{’TimeTesting} + 1 \\
  &\& \text{’Signal} < \text{LIMIT} \\
  &\quad - (\text{TIMEOUT} - \text{’TimeTesting}) * \text{MAXDELTA} \\
  &\& \text{’TimeTesting} < \text{TIMEOUT} \\
  &\& \text{Signal} - \text{’Signal} < \text{MAXDELTA} \Rightarrow \\
  &\quad \text{Signal} < \text{LIMIT} \\
  &\quad \quad - (\text{TIMEOUT} - \text{TimeTesting}) * \text{MAXDELTA} \\
  &\& \text{TimeTesting} < \text{TIMEOUT}
\end{align*}
\]

**Exiting transitions disjoint.** As well as checking that assumptions are established, there are also some checks that can be performed on the transitions of the state machine. Statecharts allow input values to trigger more than one transition exiting the same state. This includes transitions that, as drawn, exit an ancestral state. Stateflow resolves this non-determinism firstly by prioritising transitions that cause exit of an ancestral state, and lastly by taking the earliest transition found in a clockwise search around the state’s perimeter starting from the top-left corner. This reliance on layout is unfortunate. It can be avoided by requiring the triggers of exiting transitions (and the state preservation condition) to be disjoint. For this analysis, the statechart is first flattened to an equivalent statechart in which all transitions are from and to basic states. The state’s next and last assumptions, and its local definitions, may all be relevant to the analysis.

The analysis for *Testing’s exiting transitions disjoint*, excluding the state preservation condition for simplicity, looks like this.

\[
\begin{align*}
  \text{TimeTesting} &= \text{’TimeTesting} + 1 \\
  &\& \text{’Signal} < \text{LIMIT} \\
  &\quad - (\text{TIMEOUT} - \text{’TimeTesting}) * \text{MAXDELTA} \\
  &\& \text{’TimeTesting} < \text{TIMEOUT} \\
  &\& \text{Signal} - \text{’Signal} < \text{MAXDELTA} \Rightarrow \\
  &\quad \sim (\text{TestsCompleted} \& \text{TestsPassed} \\
  &\quad \& \sim (\text{TestsCompleted} \& \sim \text{TestsPassed} \\
  &\quad \quad | \text{TimeTesting} == \text{TIMEOUT}))
\end{align*}
\]

It is not provable; with TIMEOUT set to 9, a counter-example is found with the following values.

\[
\begin{align*}
  \text{TestsCompleted} &= \text{True} \\
  \text{TestsPassed} &= \text{True} \\
  \text{’TimeTesting} &= 8
\end{align*}
\]

This indicates that both exiting transitions are triggered if TestsCompleted becomes True at the same time as the timeout expires. Allowing the mode change in this case would be safe (so Stateflow’s choice is unfortunate). The triggers
4.4. Observations

The starting point for this example was an existing state machine onto which a requirement was related as an expectation on a basic state. The results of exhaustively applying the analyses include: a next assumption on the root state constraining Signal’s rate of change; definitions of constants LIMIT, MAXDELTA and TIMEOUT; and confidence that, if the mode change manager is used in an environment that satisfies those constraints, the state machine will satisfy the requirement.

More generally, the individual analyses are small enough to hope to decide automatically, and applying them together exhaustively establishes a chain of reasoning that gives valuable results. Explicit expectations of particular states, if not guaranteed by earlier states, give rise to explicit expectations of the root state, i.e. expectations of the state machine on its environment.

The expectations are readily expressible by engineers: they make explicit what is already implicitly understood, and use notation that is already familiar. The analyses are formally constructed, which makes tool support possible.

There are other ways in which the analyses can be used, not illustrated by the above example. The analyses were designed as part of the Practical Formal Specification (PFS) method [6, 10]. The PFS method advocates development of expectations in parallel with the state machine, with both contributing to the meaning of the model. For control systems that will ultimately have to cope with faults in sensors and actuators, PFS advocates modelling normal behaviour first, with expectations making explicit the bounds of normal behaviour, and then refining this model by weakening the expectations to admit faults and specifying behaviour in those circumstances. This paper aims to explain how the analyses work, not to constrain how or why they should be used.

5. Formalisation of analyses

The above examples have illustrated how making expectations explicit provides enough information for flaws to be detected. This section presents a general characterisation of expectations, and a formalisation of the analyses expressing the consistency of expectations and the conformance of the state machine to the expectations. Space does not permit formalisation of the flattening of a statechart.

5.1. Expectations

Next assumptions. For any state $S$, the next assumptions $\text{next}(S)$ specify what the next values of the inputs should be while in the state. They can relate these next values to last
values, i.e. the primed notation may be used. They may not refer to local variables.

**Last assumptions.** For any state $S$, the last assumptions $last(S)$ specify what the values of the inputs should have been for the state to be active. Primed notation is not used in writing last assumptions: they may refer to values at only one time. (Some analyses apply priming to make last assumptions apply to earlier values.) Last assumptions may refer to local variables.

**Initial assumptions.** For any state $S$ that is a superstate (not a basic state), the initial assumptions $initial(S)$ specify what the values of the inputs should have been for the state (and its initial substate) to have become active. Primed notation is not used in writing initial assumptions: they may refer to values at only one time. Initial assumptions may refer to local variables.

**State preservation condition.** For any state $S$ that is not a superstate (a basic state), the state preservation condition $spc(S)$ specifies what next values of the inputs should cause the state to not be exited. State preservation conditions may (like triggers) refer to local variables and not to primed variables.

5.2. Analyses

**Notation.** For any state $S$ except the root state, its immediately surrounding state is $parent(S)$. For any transition $T$, its trigger condition is $trigger(T)$. For any state $S$, the conjunction of its local definitions is $locals(S)$ (wherein each local definition is an equality between the local variable’s next value and an expression in terms of last values, e.g. the assignment $x = x + 1$ becomes the equality $x \equiv 'x + 1$). In referring to the last assumptions of a state $S$, the last assumptions of its ancestral states are needed too, hence $lasts(S)$ denotes $last(S) \land last(parent(S)) \land ... \land last(root)$. Priming applied to assumptions, e.g. $'lasts(S)$, causes those assumptions to refer to the last values of the inputs. For any basic state $S$ in the equivalent flattened statechart, the disjunction of the triggers of its exiting transitions is $exiting(S)$.

**Initial assumptions established.** For any transition $T$ between a source state $S$ and a destination superstate (non-basic state) $D$,

$$'lasts(S) \land next(S) \land locals(S) \land trigger(T) \Rightarrow initial(D)$$

and for any default transition $T$ to a destination superstate (non-basic state) $D$, $initial(parent(D)) \land trigger(T) \Rightarrow initial(D)$

**Next assumptions established.** For any state $S$ that is a child of another state (not the root state),

$$'lasts(S) \land next(parent(S)) \Rightarrow next(S)$$

**Last assumptions established.** For any transition $T$ between a source state $S$ and a destination state $D$,

$$'lasts(S) \land next(S) \land locals(S) \land trigger(T) \Rightarrow lasts(D)$$

and for any default transition $T$ to a destination state $D$,

$$initial(parent(D)) \land trigger(T) \Rightarrow lasts(D)$$

**Last assumptions preserved.** For any state $S$ that is not a superstate (a basic state),

$$'lasts(S) \land next(S) \land spc(S) \land locals(S) \Rightarrow lasts(S)$$

**Exiting transitions disjoint.** For any basic state $S$ in the equivalent flattened statechart, and any distinct transitions $T$ and $U$ that both exit $S$,

$$'lasts(S) \land next(S) \land locals(S) \Rightarrow \sim (trigger(T) \land trigger(U))$$

For any basic state $S$ in the equivalent flattened statechart, and any transition $T$ that exits $S$,

$$'lasts(S) \land next(S) \land locals(S) \Rightarrow \sim (trigger(T) \land spc(S))$$

**Exiting transitions complete.** For any basic state $S$ in the equivalent flattened statechart,

$$'lasts(S) \land next(S) \land locals(S) \Rightarrow spc(S) \lor exiting(S)$$

and for any state $S$ that is a superstate (a non-basic state) containing a default transition $T$,

$$initial(S) \Rightarrow trigger(T)$$

6. Supporting tools

This section describes the user interface to the tools, and says a little about how the tools are implemented.
6.1. User interface

Expectations are entered through a dialogue, as illustrated by figure 4 for the expectations of state Testing in figure 3.

The dialogue allows expectations to be prepared in tabular formats. For example, figure 4 shows the last assumptions as a table of two conjuncts. Buttons on the right of a table allow rows to be inserted and deleted, while on the left, button $R$ makes the other buttons disappear ($R$ for read-only), button $F$ folds away the table to a restorative button (to save screen space), and button $-$ converts the table to straight text. Each editable field of text has a $+$ button to convert it to one of a choice of tabular formats, offered in a pop-up menu. Tables can be nested.

As well as showing expectations, the dialogue shows output definitions and local definitions, to allow those to take advantage of tabular formats too. Buttons appear alongside them to initialise their tables with the declared output names and local variable names. This information is kept consistent with the corresponding Stateflow state actions on the statechart diagram: changes to one affect the other too.

The Choose healthiness condition menu causes the dialogue to display the expectations relevant to a particular analysis (here called a healthiness condition) concerning the state. If the Choose transition menu is used first, then the Choose healthiness condition menu instead offers to display analyses concerning that transition, and meanwhile the transition’s trigger is shown to allow a tabular format to be used for that. Figure 5 shows the same dialogue after the Exiting transitions disjoint healthiness condition has been chosen (and some buttons have been made to disappear).

All of the expectations involved in the healthiness condition can still be changed, even though a healthiness condition is now being shown. The purpose of each expectation is shown as well as its value, making the healthiness condition easier to comprehend than the analyses presented in section 4. Moreover, almost everything in the dialogue has a helpful tooltip. Note that the last assumptions are primed, and the trigger of the transition from Testing to ModeChange is shown in two parts.

Pressing the Prove it button initiates automated analysis of this healthiness condition. The result is displayed in another window: if the healthiness condition is proved, the
user is told so, otherwise, any counter-example is displayed.

The dialogue’s menu bar offers Choose state to view the expectations of another state, Copy from environment to see the names that may be used in these expectations and copy them to the clipboard for pasting, and the File menu of commands including Prove all and Try proof. The Prove all command initiates analyses for the entire model, with results displayed in another window. The Try proof command is for use when healthiness conditions are not proved automatically and no counter-example is offered: it runs the theorem prover in its interactive mode. Try proof is likely to be of use only to users with a background in formal proof. For the example of section 4, the automatic analyses succeed in every case, and so there is no need to use the Try proof command.

There is also a means of obtaining a report that lists which analyses have succeeded.

6.2. Implementation

Stateflow is not as readily extended as Simulink is, so the implementation uses new Simulink masked blocks. These record the expectations in blocks under the mask, and when opened offer dialogues for maintaining the expectations. The new blocks are packaged with a Stateflow block to form a state machine subsystem.

When an analysis is initiated, the statechart and expectations are translated to the Z formal notation [9]. Z conjectures for all of the analyses are generated [20], and proof of the requested analysis is attempted by the CADiZ theorem prover [21]. When another analysis is initiated on the same model, the previously generated Z is reused. For the example of section 4, with expectations added to all states, the formalisation of the model and 31 conjectures are generated in 13 seconds. Proofs of all 31 conjectures are found automatically in a total of 8 seconds. These measurements are from a 2.2GHz Athlon PC running Windows XP.

CADiZ uses a tactic [19] to simplify a conjecture into goals and then to apply decision procedures such as sup-inf to linear arithmetic goals (Presburger [15]), model checking to small-range finite goals (NuSMV [5]), and simulated annealing to non-linear arithmetic goals. This finds proofs automatically for most conjectures that arise. A theorem prover is used rather than merely a model checker because the data types involved can be integers with large ranges. However, changing the data types to be smaller ranges for the sake of model checking can sometimes be worthwhile.

7. What next?

Having applied the analyses successfully, further validation should be done to show that the expectations of the root state are established. Since a Stateflow statechart always appears in the environment of a Simulink block diagram, Simulink provides a context for validating the statechart. In other work, we have investigated the use of differential weakest precondition calculus in validating that a Simulink model satisfies a contract expressed over recent values of inputs and outputs. Some initial ideas were presented in [3]. Of course, that contract also should be validated against its wider context.

8. Further work

The analyses are framed around references to just next and last values. Meanwhile, our differential weakest precondition calculus can refer to earlier values, thus validating not only rates of change but also accelerations (rates of rates of change). Could the state machine analyses be extended to validate accelerations?

Since control systems often involve feedback (inputs that are derived from earlier outputs), another idea is to allow expectations to refer directly to outputs.

The analysable hierarchical state machines subset might be expanded, e.g. to include floating point types, AND states, more kinds of actions, and events.

For analysing larger models, some automated dependency analysis to minimise the quantity of re-analysis could be useful.

The representation of expectations could be done more slickly by The MathWorks within Stateflow.

9. Comparison with related work

Statecharts have been translated to many different formalisms by others for the purpose of analysis, and some have used Z, so that aspect of our work is not particularly novel. The novelty and significance of our work is the analyses that are done based on the formalisation. Others typically formulate potential invariants of a statechart by hand, and use tools to suggest counter-examples to those, making no assumptions of the environment. Our tools generate conjectures automatically, with assumptions concerning next and last input values from the environment.

Validation of Stateflow statecharts has been reported by Ford in conjunction with Reactive Systems Inc. [16]. They have experimented with a formal validation tool called Salsa. “Salsa has attributes of both a model checker and a theorem prover”, which is something that could also be said about CADiZ. Whereas we translate from Stateflow to Z automatically, they translate to SAL by hand (at least at the time of publication). Whereas we have automated a particular set of analyses, they formulate arbitrary conjectures of interest largely by hand (though they have automated an
analysis like our exiting transitions disjoint). They consider more of the Stateflow statecharts notation than we do, e.g. AND states.

The use of tables in specification and validation can be traced back via SCR to Parnas [13]. SCR-style tables have also been used in conjunction with hierarchical state machines in RSML [11]. These influenced us to offer tabular formats in our tools.

10. Conclusions

Analyses have been presented that formally validate a hierarchical state machine against expectations. The expectations are readily expressible by engineers: they make explicit what is already implicitly understood, and use notation that is already familiar. The tools automate the analyses covertly.

The analyses can be used in a variety of ways, e.g. validating presumed expectations of the environment, or determining what expectations there are on the environment for some requirements to be satisfied. Most importantly, they address a known validation need and are usable.

The analyses were first mentioned in [6]. The Z formalisation was included in a presentation to Z users in [20]. This paper has extended the analyses to cover local variables, has presented a more pedagogical example and a briefer formalisation, and has illustrated the latest tool support. The tools are now owned by High Integrity Solutions Ltd.

11. Acknowledgements

Financial support was provided by the UK MoD (contracts FBG/01046 and FBG/01168), EPSRC (grant GR/R/70590/01), the HISE research group at York, High Integrity Solutions Ltd, and UK DTI (grant PLASMA). Petra Malik provided comments on an earlier draft of this paper.

References