Statecharts: Some Critical Observations

Michael Breen

last updated 2004-8-5

Abstract

A critical analysis of statecharts is presented, motivated by difficulties observed in practice by the author and supported by examples. Particular attention is paid to the possible uses of hierarchy and their effects. Despite the notation's popularity, the observations indicate fundamental problems with using statecharts for specification.

1 Introduction

Since the statecharts notation [3] was introduced it has proved a popular success, being widely used by practitioners, analysed by academics, and incorporated into other languages. Occasionally, problems with the notation are pointed out but often these have been of more interest to theorists who wish to modify the semantics than to practitioners. In particular, the notion of hierarchically structuring states has been largely uncriticized.

What follows is intended to be a critique of statecharts from a user's point of view; where some of the points might seem somewhat theoretical, examples are used to show their practical implications. Though these examples are mainly taken from Harel's original paper, the observations they illustrate derive principally from the author's own experience with statecharts and other approaches based on the finite state machine model. The focus here is on the use of statecharts for specification as opposed to design, though some points are relevant to both contexts.

2 Uses of Hierarchy

Stateharts differ from other notations mainly in the concept of a hierarchy of states. States may be grouped into superstates for a variety of reasons. I identify five distinct purposes and consider each in turn.

Clustering

One reason to use hierarchy is to reduce the number of transitions required on the diagram: A single transition drawn from a superstate to a target state can replace transitions from each of the substates to the same target state. Not only does this reduce repetition, it also reduces the amount of clutter in the diagrams. This can be very useful even for a small number of states especially when the label on each transition is relatively long.

There is also the potential to allow shorter transition labels where transitions in one component are dependent on the state of another: If a transition has a condition of the form "in state1 OR in state2 OR in state3" and these states are grouped into a superstate then the condition may instead be written "in stateSuper".

Abstraction

With hierarchy, one may also abstract the contents of a superstate, moving its substates and the transitions between them to a separate, lower-level statechart. This is clearly of some practical use since a statechart may otherwise become too large to be viewed comfortably on the screen or printed on a single page.

However, one should not assume that this kind of abstraction will add to the model's understandability; indeed adverse results reported by one group of users led them to advocate minimizing the number of layers of abstraction [5]. In fact, it can be argued that there is a general problem with abstraction in finite state machine models as compared to models of other kinds [1]. Below, I describe an error that would have been much more difficult to detect had abstraction been used.

Transition Dependence

Figure 1 (a) shows two orthogonal components. The transitions in one are dependent on the other being in state *Y*. Figure 1 (b) shows how hierarchy may be used to show this dependency visually. Note the use of the history mechanism to ensure the state of the second component (whether *A* or *B*) is remembered.

The fundamental problem with history is that it confuses the two otherwise quite distinct concepts of hierarchy and parallelism (or orthogonality): As emphasized by the need to remember the "previous" substate of the superstate with a history entrance, the substates are properly regarded as those of a component with an existence independent of the superstate. The effect of an apparently innocuous little circle with a 'H' is completely to transform the semantics of the substate-superstate relationship.

Consider a statechart in which all the states are superstates whose contents have been abstracted and are shown only on separate diagrams. Suppose there are two orthogonal components, one with states *A* and *B*, the other with states *X* and *Y*. Now if the system is in states *A* and *X* then to describe the overall state of the system at a finer level of detail one could normally assume one needs to consider only the substates of these. After all, *A* and *B*, for example, are mutually exclusive and so the substates of *B* should not be relevant. But if *B* has a history entrance then this assumption is invalid since a parallel state machine has effectively been hidden at a lower level. By mixing up hierarchy and parallelism, the function of memory is shared between states and histories and the integrity of the concept of a state is weakened.

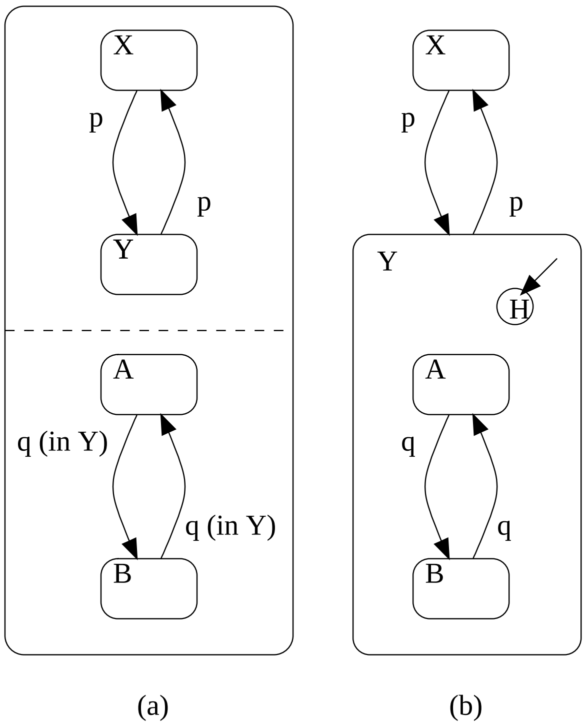

Figure 1: Hierarchy with history.

State Variable Instantiation

Hierarchy may also be used to, in effect, instantiate a new finite state machine (in statecharts terminology, an orthogonal component). Figure 2 shows a stopwatch taken from the example in [3]. In this case, the stopwatch is described either by a single state if it is stopped and reset to zero or by two states, one to indicate whether it is stopped or running and the other to tell us whether it is displaying the elapsed time or the last lap time.

Instantiation of finite state machines is normally done above the level of the finite state model. One may, for example, model the behaviour of an ob-

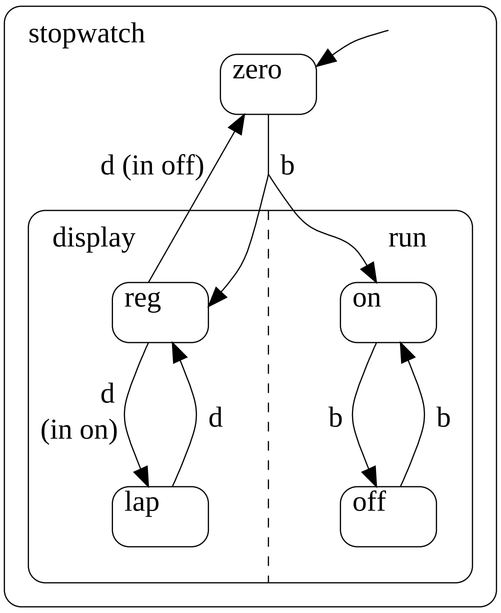

Figure 2: Stopwatch state of digital watch.

ject using a statechart and indicate in a higher level diagram or other description that one or more instances of that object can exist and specify when these are created and deleted. Although this is generally a more flexible and powerful approach, it seems unsuitable in the case of the stopwatch. For example, if the statechart completely models the state of the stopwatch for some of the time with a single state variable (*stopwatch*) then it seems preferable for it to continue to do so even when two (*display* and *run*) are needed.

However, the stopwatch example is not actually one in which a need suddenly arises for an additional variable to model the state: In state *zero*, the stopwatch is not running and it is displaying the elapsed time (which happens to be zero) and so this state really represents a combination of the two state aspects otherwise modeled separately by the *display* and *run* components. The stopwatch model could be replaced, for example, by either of those in Figure 3.

Of course, in these alternatives, *display* and *run* are still instantiated on entry to *stopwatch*, one of many states in the model of [3]. However, we could also make *display* and *run* orthogonal components at the top level of the model (we might then call the former *stopwatch-display-mode* since it reflects the actual display only in state *stopwatch*); in the full digital watch there is a deep history entrance to *stopwatch* and so the nesting of states within *stopwatch* is really a way to make the transitions between them conditional on being in this state.

In the digital watch example there are two other cases of orthogonal components appearing within a state (that is, other than at the top level - which is not really a state as such). One is to model beep test functionality, wherein the watch beeps if two buttons are pressed at the same time. Again, a state can always be associated with this function so that we may say it really belongs at the top level: whenever the system is not in the superstate containing the *beep-test* component, the beep test is implicitly in an inactive state.

The other use is in order to model a state *dead* which corresponds to the battery being removed or exhausted. Because of this, all the other orthogonal components, which would otherwise be at the top level and so always exist, are shifted down one level into an *alive* state. Once again, this does not mean that no state can be associated with these components: If the battery is dead then clearly the light is off, the beep test is inactive, the display is blank, and so on. Further, modeling the state *dead* really serves no purpose other than to demonstrate the notation. If the statechart is to serve as a design for the internal logic of the watch then clearly the state *dead* does not belong in the model. If its purpose is to describe or specify the external behaviour of the watch then including the state *dead* is hardly more useful than including a state like *broken*.

In the digital watch example there are no cases where the creation of orthogonal components on entry to a superstate corresponds to any need for an additional state variable, that is, one that has no meaning outside that superstate. In the author's ex-

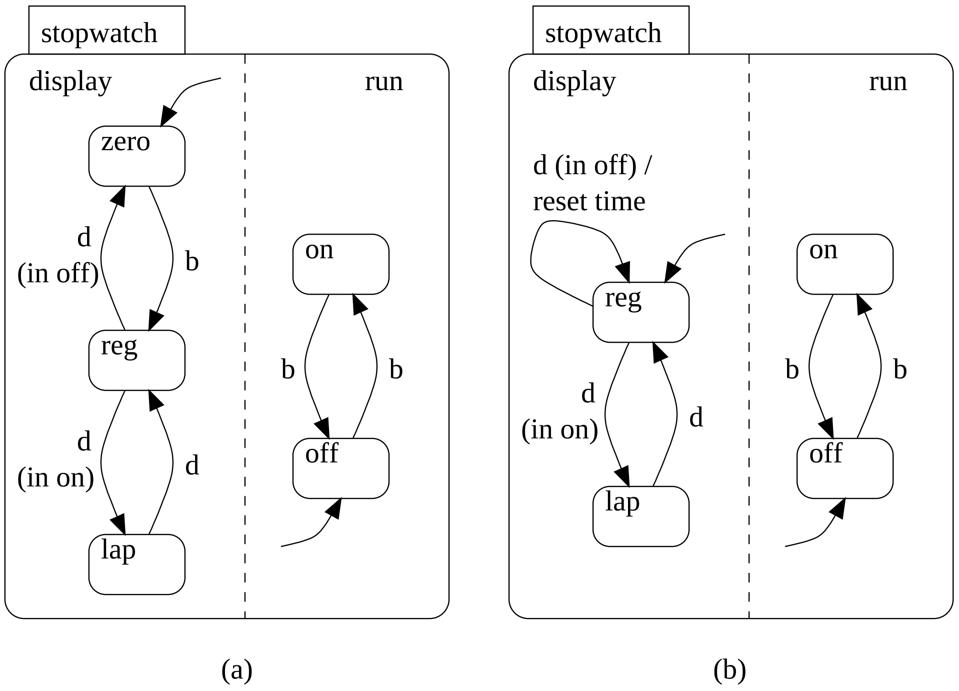

Figure 3: Alternative stopwatch models.

perience, this is always the case and so alternatives to statecharts, such as the ordinary state transition diagram, do not suffer in comparison.

Expression of a Constraint

The real effect of putting the state *zero* on a separate level, as in Figure 2, rather than keeping all the states on the same level, as in Figure 3 (a), is to express a constraining relationship: In state *zero*, the stopwatch must not be running, that is, component *run* must be in state *off*. In Figure 3 (a) it is perhaps not immediately clear that this constraint is satisfied. In Figure 2, the state *zero* in effect models not only the state of the display component but also that of the *run* component; consequently the constraint is satisfied by the semantics of the state itself.

Of course, the clarity of the expressed constraint then depends on knowing the semantics of the combined state. The interpretation of the state *zero* in Figure 2 is obvious only because stopwatches are familiar; one needs to know that in state *zero* the stopwatch does not run in Figure 2 as well as in Figure 3 (a).

This application of hierarchy would be more useful if all constraints were always modeled. In that case, the model would supply all the information on what state configurations were possible. For example, one could tell at a glance from Figure 2 that there were a total of five possible state configurations in the stopwatch: *zero* and the four combinations of the states from the *display* and *run* components. If it was subsequently found, perhaps through an automated check, that it was not possible for *display* to be *lap* when *run* was *on* then this could be flagged as an error (to be corrected either by making this configuration reachable or by replacing the orthogonal components with three separate states).

Unfortunately, if the model is more complex then specifying the possible state configurations within the statechart may lead to a fairly intricate diagram. There may be more states and many more transitions, including branching transitions perhaps with multiple branches where several partially independent aspects of the system state are involved. Compared to the alternative of simply modeling separate orthogonal components at the top level without attempting to explicitly indicate which combinations of their states are possible (at least within the statechart), this use of hierarchy may be judged undesirable or even impractical. As long as this may be so in specific instances, it cannot be made a general convention.

3 Subtlety

Adjectives like "concise," "subtle" and "powerful" seem to describe statecharts pretty well. In general, these may be regarded as positives. For example, conciseness is obviously preferable to useless repetition and many people enjoy subtlety. These attributes are strongly related to the relatively high degree of structuring possible in statecharts which means that a single aspect of a statechart model can have broad implications for behaviour.

To show the power of statecharts, Harel describes how easy it is to add a beep-test function to his digital watch model. To test the beep, the user keeps two buttons pressed at the same time. This is easily modeled by a *beep-test* component with four states: one in which neither button is pressed, two in which one of the buttons is pressed and another in which both buttons are pressed and the watch beeps; the transitions between these states occur on pressing and releasing the buttons. To neatly express the fact that the beep test only operates in certain states, it is made an orthogonal component within a superstate which encloses those states. Harel then works through a scenario to demonstrate that the model works as expected.

However, this change also introduces an error. Figure 4 shows the position of *beep-test* within the overall model (many other states and orthogonal components are omitted from this diagram). Suppose in the *time* state that one of the two buttons to activate the beep test is pressed, causing *beep-test* to enter (say) state *10*. The time of an alarm is reached causing the watch to enter *alarms-beep*. The user now releases the first button (this has no effect) and presses the second button, causing a return to state *displays*. Because of the deep-history entrance, the watch is again in state *time* and *beep-test* is again in state *10* – indicating that the first button is still pressed (which it isn't) and that the second button is not currently pressed (which it is). Pressing the first button now fails to activate the beep test. Alternatively, if the second button is released and pressed again then the beep test is incorrectly activated without pressing the first button.

Presented as in Figure 4, this error becomes obvious. However, it was not initially apparent to this author even after a fairly close examination and was only detected when converting the model into another notation. The error would be even more difficult to detect if the contents of the *time* superstate had been abstracted and shown only in a lower level statechart – for it would not be present in either statechart considered separately. By using powerful constructs which affect the behaviour in many different cases it is possible to create a very concise model. Unfortunately, as in this instance, it also becomes easier to overlook unexpected implications.

It is possible to identify other unusual scenarios involving the beep test function in which the response is questionable but, unlike the previous example, not obviously erroneous. This is another problem with subtlety: Is the behaviour in a particular case deliberate or is it an unintended side effect of structure introduced for some other purpose? This is not a problem if the statechart is merely a design but if it is also the specification then one must consult those who created the model. If they are available and can remember or at least agree on the correct response then perhaps one might compensate for the model's subtlety by recording their answer so that at least no one else will need to ask the same question. But would it

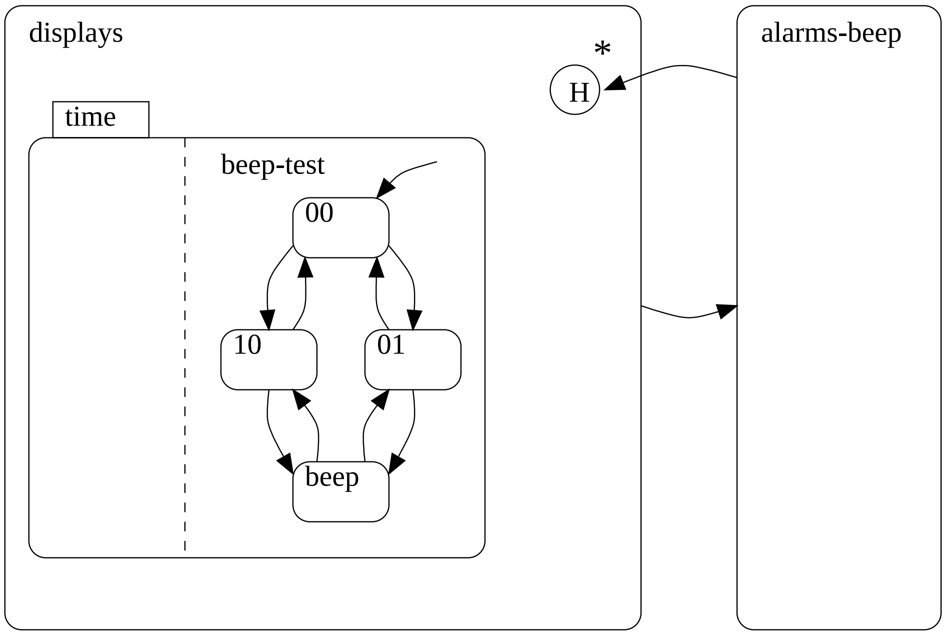

Figure 4: Beep test component in digital watch model.

have been better if the specification had been in a more explicit form to begin with?

4 Amenability to Inspection

To discover errors like that described above, one must perform simulations, either mentally or with computer assistance, to see the response of the model in various scenarios. However, this is effectively testing and for a complex model it is normally impractical to test all sequences of events which might expose an error.

Further, while the graphical nature of statecharts allows rapid random access to information, it does not naturally invite or support systematic review. In trying to proceed methodically through the states and transitions, the experience of this author is that the mind is easily led by the eye; to complete a comprehensive more-or-less linearly-ordered review in the face of a fundamentally unordered model is difficult. It is not enough to check each part separately as the hierarchical structure (and broadcast events) means the parts may be related in relatively complex ways. To check and to better understand one part of the model, one makes detours to other parts and after several detours one may lose track of what relationships have been considered thoroughly and what has been merely understood.

Also, it is easier to detect an error which is present in the model than an error of omission (unless it is a major one). A statechart model is always formally complete and remembering to check that all the necessary responses are included is difficult when one is busy checking the transitions which are present.

5 The Rubik’s Cube Effect

In statecharts, states are arranged in a strict hierarchy so that each state may be a substate of at most one immediate superstate. It would often be convenient to violate this rule. To take a simple example, if one is clustering states to reduce the number of transitions then states *A*, *B* and *C* might share the same exit but *C*, *D* and *E* may respond identically to some other event. As they overlap on state *C*, one cannot cluster both groups into superstates. When hierarchy may also be used for other purposes, the number of competing alternatives like this increases.

While recognising some semantic issues, Harel was originally convinced that allowing states to overlap would greatly enhance statecharts [3]. However, in spite of repeated requests for this from users, later work indicated that extending the notation in this way required definitions which were too complex for it to be generally recommended [4]. (Note also that is somewhat incompatible with using hierarchy for abstraction.)

To construct a statechart model one must therefore decide on a unique hierarchy from the competing alternatives one can imagine. In a complex model, what seems at first to be the best choice might appear less desirable as modeling advances. This is especially likely where the model is being constructed as part of a specification process since one does not initially know all the details of behaviour to be included.

Though the analogy is obviously exaggerated, a statechart model, as a relatively structured entity, may be compared to a Rubik's Cube: In the course of solving the cube, it is necessary to disrupt partial solutions; and as more of the existing structure embodies the desired configuration it becomes more difficult to incorporate new changes. Similarly, in a statechart one may, for example, realise that all of the transitions between the states of component *A* should be made dependent on the system being in a particular state *X* of an orthogonal component. To model this, one might move the states of *A* to make them substates of *X* and add a history entrance to this state. However, one then finds that a third component is disrupted by this change: One of its transitions contains the condition *in Y* where *Y* is a substate of *A* – but even with history the system can no longer be "in" *Y* unless it is also in *X* and so a side effect of the change is to alter the conditions in which this transition is allowed.

After the gross structural change, one may therefore need to make further adjustments to cancel out such side effects. The overall change then comprises a number of steps. Following our analogy, these steps may be likened to the sequence of moves one must remember in order to reverse the temporary disorder induced in a partially-solved Rubik's Cube. Furthermore, if one omits a step or makes some other mistake in the changes then it may be much less obvious in the statecharts than would be a square of the wrong colour on one side of the cube; the side effect described in the previous paragraph is one that might easily be missed.

The point is that during specification one should be less concerned with structural issues. Anything too much resembling the solving of a puzzle constitutes design and is best kept as a separate activity. One may then concentrate on establishing the facts of the behaviour first without the distraction of having to continually work out how best to alter the statecharts to efficiently represent this behaviour and checking that the model does successfully capture one's intentions. Though design itself may remain a challenge, design decisions are at least deferred until they can best be made, that is, when all the facts are known.

6 Broadcast Events

A well-known criticism of statecharts is of broadcast events between orthogonal components. Compared to models in which directed communication between separate parallel state machines are shown on a higher level diagram, broadcast events make it more difficult to see the relationship between different parts of the model. The same criticism can be made of the capability to refer in the condition of a transition to the state of another component. Of course, modularity can be restored very simply by limiting the use of orthogonality in a statechart and instead using separate statechart models which communicate where necessary by directed events;

the broadcast events and states within each one are invisible to the others.

7 Understandability

Event sequencing and priority can be a source of confusion in a statechart. If two transitions may be triggered by the same event, which is taken first? The answer may affect the overall response. What if a transition between substates of a superstate and a transition from the superstate itself may be taken? Issues of causality and synchrony involving conditions on events can also arise. Even if the semantics prescribes a unique response and this is remembered and understood by the reader, tracing through a causal sequence may be difficult and lead to errors. Of course, part of the problem is not the absence of a precise semantics but the presence of dozens of alternative ones (some of these were surveyed in [8]; others have been proposed since then). In general, the modeler is well advised to keep the statecharts simple enough to avoid situations where confusion may arise.

8 Why are Statecharts Popular?

The popularity of statecharts in the academic world can be attributed to the interesting problems posed by the notation's semantics and the seemingly endless potential for modifications and alternatives which semanticists like to propose and explore. Many variations have been proposed and entire theses have been written on the subject, filled with (to the practitioner) obscure theoretical results.

Nevertheless, most semantics are broadly consistent with the intuitive, operational interpretation of a practitioner working with a typical statechart. The addition of hierarchy to the ordinary state transition diagram brings more power and flexibility (if this power is akin in some respects to that of a "goto" statement in a programming language then this at least is not obvious). The graphical representation is also an attraction. Not only does it carry the promise of something that can easily be understood by following the arrows, it provides an opportunity to create something which is not only logically correct but also aesthetically pleasing.

The use of subtlety allows one to create a cunning and efficient model. Even if one is aware that one's primary objective is to describe the behaviour rather than to find the most efficient representation, there is no denying the immediate satisfaction of finding a neat way to model something.

However, if the same subtlety leads to an error, most practitioners – who might well object to using a less visually appealing but ultimately more effective approach – are likely simply to blame themselves for making the mistake. Once seen, an error in a statechart model is easily understood; one perhaps feels a little foolish for having missed it; the model is duly fixed and, lo, the fix can also be understood. Such errors are regarded as a fact of life which reviews and testing are there to find. An error which is somehow discovered after testing indicates a flaw also in the tests (even though testing every possibility is usually impractical). Further, since errors are much less likely in relatively simple models, it is easy to attribute them to the size and complexity of the model. Overall, one is unlikely to link errors to the relative subtlety of a given model, associated in its construction with a sense of satisfaction; much less to link them to any more general cognitive issues with the notation.

9 Conclusions

The problems described above will not be significant in relatively simple models. Some issues also affect other notations though perhaps to a lesser degree. For example, even if hierarchy is not used, it may be necessary to restructure a model where it becomes apparent that a single state machine would be preferable to two parallel ones which become too interdependent.

However, many problems relate particularly to the hierarchical structuring of states unique to statecharts. Of the identified uses of this feature, clustering to reduce the number of transitions in a diagram is the most convincing. Using hierarchy for abstraction also seems useful in order to limit the size of statecharts for practical reasons. However, there should be less need for this if parallel state machines are modeled using separate statecharts in the first place rather than orthogonal components or nesting within a state using the history mechanism – the most dubious of the uses of hierarchy.

Many of the above criticisms apply to the use of statecharts for specification rather than design. However, statecharts are generally regarded as a specification notation (and indeed as a kind of formal method of specification). An advertisement for one of the tools supporting statecharts claims that the statechart specification serves also as the design, allowing one to skip the design phase of a project. An alternative view, consistent with observations above, is that to use statecharts in this way is in effect to go straight to design, skipping specification.

The direction of some research has been towards simplifying statecharts to make them more suitable for specification [2]; however, it is interesting that one extended research effort into the specification of process control systems began by using statecharts, modified the notation, and ultimately proposed a completely different approach [7, 5, 6].

References

[1] M. Breen. (2003) On abstraction in finite state machine models. [Online]. Available: http://mbreen.com

[2] M. Glinz, "Statecharts for requirements specification - as simple as possible, as rich as needed," in *Proc. of ICSE 2002 Workshop: Scenarios and State Machines: models, algoritihms and tools*, 2002.

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

[4] D. Harel and C.-A. Kahana, "On statecharts with overlapping," *ACM Transactions on Software Engineering and Methodology*, vol. 1, no. 4, pp. 399–421, Oct. 1992.

[5] N. G. Leveson, M. P. Heimdahl, H. Hildrith, and J. D. Reese, "Requirements specification for process-control systems," *IEEE Transactions on Software Engineering*, vol. 20, no. 9, Sept. 1994.

[6] N. G. Leveson, M. P. Heimdahl, and J. D. Reese, "Designing specification languages for process-control systems: Lessons learned and steps to the future," in *Proc. ESEC/FSE '99, 7th European Software Engineering Conference held jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 6–10 Sept. 1999.

[7] N. Leveson, M. Heimdahl, H. Hildreth, J. Reese, and R. Ortega, "Experiences using statecharts for a system requirements specification," in *Proc. Sixth International Workshop on Software Specification and Design*, 1991.

[8] M. von der Beeck, "A comparison of statecharts variants," in *FTRTFT 94: Formal Techniques in Real-time and Fault-tolerant Systems*, ser. LNCS, W. d. R. Langmaack, H. and J. Vytopil, Eds., no. 863. Springer Verlag, 1994.