On the Compositional Properties of UML Statechart Diagrams

Anthony J H Simons

Department of Computer Science, University of Sheffield
Sheffield, South Yorkshire

Abstract

This paper proposes a revised semantic interpretation of UML Statechart Diagrams which ensures, under the specified design rules, that Statecharts may be constructed to have true compositional properties. In particular, hierarchical state machines may be properly encapsulated to allow independent verification and compositional testing, something which is not possible under the current UML semantics. Certain problems regarding the formal tractability of UML Satechart Diagrams are addressed, such as the confusion over states and connectors, the flattening effect of boundary-crossing transitions, and the consequences of inverting the inter-level priority rule for handling concurrent events. A set-theoretic formal treatment of object states, events, guards and run-to-completion processing is given, describing both serial and concurrent Statecharts.

1 Introduction

One of the advantages of state machine based design models is the ability to visualise the control behaviour of a system graphically, something that is more appealing than a dense mathematical specification. For this reason, Statecharts of one kind or another have become perhaps the most acceptable means of specifying formal designs in the software industry. The UML Statechart Diagram is now an important OMG standard for documenting the behaviour of objects, components and systems.

An evolution of the Harel Statechart [1], the Statechart Diagram traces its history via the Dynamic Model of OMT [2], a version of which was subsequently incorporated in UML 1.1 [3]. After adoption by the OMG in 1997, the Statechart Diagram was subject to scrutiny during the UML 1.2 and 1.3 revisions. The current published OMG June 1999 standard defines the notation for Statecharts, giving examples of usage [4], and describes the intended semantics of Statecharts as part of the State Machine behavioural elements package (which also describes Activity Graphs) [5]. The latter sources provide perhaps the most precise descriptions that have yet been available for UML 1.3 Statecharts, superceding the gentler introduction in the UML 1.3 Users Guide [6].

1.1 Semantics of State Machines

Classical finite state machines are amenable to formal reasoning in terms of their equivalence to orders of grammar and formal language in the Chomsky hierarchy [7]. For example, a recursive language is defined by a context-free grammar and is recognisable by a pushdown automaton, a variant of a finite state automaton with a global stack. However, different Statechart formalisms are subject to a number of different semantic interpretations. These result from differences in the treatment of events as signals or functions, the static or quasi-functional nature of states, the existence of global or local memory, the discrete, continuous or interval-based nature of time and the precise meaning of the connectors between different levels of hierarchical machines [8, 5]. Statecharts typically admit the presence of read/writeable memory and allow conditions, or guards, which test this, making them equivalent to augmented transition networks, capable of recognising arbitrary context-dependent languages.

1.2 Hierarchical Modular Testing

Another property of state machines is the ability to validate the correct operation of the formal model with respect to the events handled by it; and the ability then to generate test sets which verify the correct behaviour of an implementation with respect to the formal design provided by the state machine. The basis for the state machine testing method is due to Chow [9], which guarantees the behaviour of an implementation with respect to a minimal finite state automaton, subject to assumptions about redundant states, by driving the system through its transition cover and comparing all valid and error states reached with the specification.

Holcombe and Ipate [10, 11, 12] have generalised this approach for Stream X-Machines, which are a generalisation of finite state machines with a global memory and input/output streams. X-Machines are formally equivalent to a restricted class of augmented transition networks and have Turing-equivalent computing power. Using the Holcombe/Ipate reductionist method, a design is decomposed into a hierarchy of independent Stream X-Machines, whose behaviour may be validated independently. A system based on this design is tested, using a variant of Chow's method, in a bottom-up fashion. The method provides a ground-breaking proof of correct integration [13], which *absolutely* guarantees the correct behaviour of an integrated system, on the assumption that its component parts are correct.

The importance of this cannot be under-emphasised: conventional path-based testing approaches do no more than exercise as many parts of a system as is economically feasible; and when testing is complete, no definite statements can be made about the number or location of any remaining faults, nor how serious these might be. With the Holcombe/Ipate method, systems may be tested in a divide-and-conquer fashion down to the lowest level and smallest component whose behaviour you are prepared to trust, and in which all remaining faults are necessarily to be found. Compared with other object-oriented testing methods [14, 15] which flatten hierarchical state machines and compute the transition cover for the *Cartesian product* of states and transitions, the Holcombe/Ipate method computes the transition cover for each machine independently, making *complete functional black-box testing* a tractable possibility. This divide-and-conquer method depends crucially on the design-for-test properties of the Stream X-Machine model, which include (i) compositionality - the ability to deal with each level of abstraction as an independent machine; (ii) output-distinguishability - the ability to associate the firing of each distinct transition with a uniquely observed output; and (iii) test-completeness - the ability to drive a machine through all its transitions from any state.

1.3 Design-For-Test Properties

To obtain similar testing benefits for UML Statecharts, it is necessary to examine their formal properties under decomposition. One of the principal novelties of Harel's original Statechart formalism [1] was the idea that states could be decomposed hierarchically, as illustrated in Figure 1, which models the behaviour of an automatic gearbox.

At a certain level of abstraction, the *Drive* state is considered as a single state, but at a finer-grained level, this is revealed to be made up of three forward-gear substates. Certain transitions apply to the *Drive* state as a whole (such as *drive, neutral*), whereas certain other transitions apply to the substate machine (such as *upShift*, *downShift*). To obtain the design-for-test property of compositionality, the superstate machine must be analysable in ignorance of the mechanism of the substate machine; and the substate machine must be verifiable independently of the operation of the superstate machine.

This particular example illustrates good encapsulation properties, in which the behaviour of each machine is independently verifiable, but Statecharts have also been used in ways which violate the encapsulation of substate machines. An example of this is the phenomenon of *boundary crossing*, a style in which transitions lead directly to, or from, the substates of a composite state. Identifying styles of usage which preserve the hierarchical encapsulation of state machines is therefore an important area of concern.

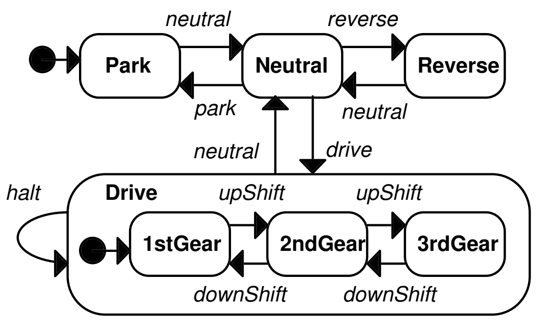

Figure 1: Harel Statechart for Automatic Gearbox

On the assumption that a single input event in the X-Machine model corresponds to a message request, resulting in a single method invocation in a Statechart, the output-distinguishable property is obtained if every distinct transition produces a distinct output value. This can be achieved by instrumenting methods during the test phase. If the names of methods do not describe unique transitions, perhaps because the same method may be invoked to reach two destination states, which are selected according to an additional guard, then a distinct output value must be chosen for each guarded version. The test-completeness property is obtained for free in the object-oriented model, since all of an object's methods may always be attempted in every state of the object (some may raise exceptions).

2 Some Features of UML Statecharts

The UML Statechart Diagram is a rich, hybrid model incorporating a number of influences that cater for different modelling preferences. Some constructions are included which are redundant, some are convenient extensions to the basic state machine model and other constructions unwittingly undermine the formal tractability of state machines [16]. While redundant and syntactically sugared forms can always be converted back to canonical forms, we should be more concerned over violations of the semantics of automata and especially constructions which prevent hierarchical encapsulation of independent state machines.

2.1 Redundant Constructions

Examples of redundancy include the provision of a separate iconic notation for *encapsulated substate machines*, which are no different from ordinary (sequential) composite states; this is recognised in [5]. The provision of *concurrent transitions* with Petri-style forks and joins (see figure 2a) is technically redundant, since it duplicates the semantics of concurrent composite states (see figure 2b). Originally, Petri-style notation was intended for the separate Activity Graphs [3], but according to the revised OMG notation document [4], p3.142, their use is now apparently allowed in Statechart Diagrams (Activity Graphs are presented separately). Both diagrams in figure 2 describe the identical forking and synchronisation behaviour in the concurrent substate machines.

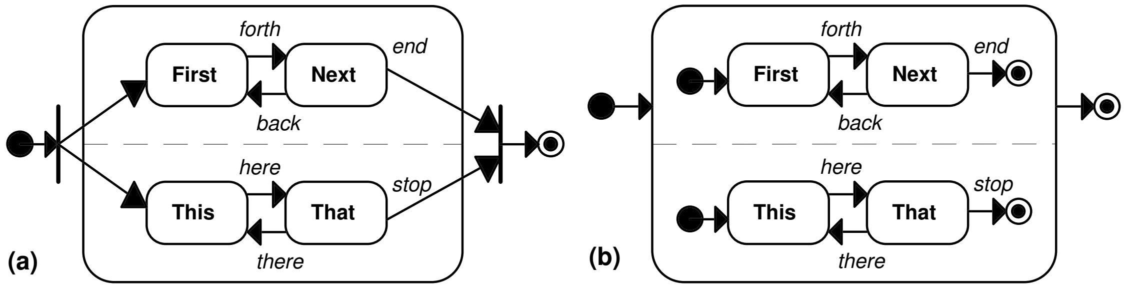

Figure 2: Equivalent UML Models for Concurrent Substate Machines

A state may also have an *internal transition* compartment, listing its distinguished *entry* and *exit* atomic actions; supplemented by an ongoing *do*-activity and aribitrary user-defined *internal transitions*. The admission of a separate class of *internal transitions* (which do not leave the superstate, therefore which do not trigger its entry and exit actions) complicates the model and is strictly unnecessary. All activity that occurs in a state can be modelled using a substate machine. So, the *do*-activity and *internal transitions* are really part of the behaviour of a substate machine, but are not treated explicitly as such in [4, 5].

2.2 Convenient Extensions

Examples of convenient extensions include the provision of *connectors* for expressing shared transition paths [4], which may be expanded in full. The path leading to a forking connector is simply expanded into as many paths as leave the fork; a symmetrical expansion is possible for joining connectors. Another extension is the admission of *conditional guards*, which are equivalent to a duplication in the number of control states [16].

Guards have the effect of introducing a certain arbitrariness into the chosen control logic. Figure 3a illustrates a temperature control system with hysteresis, in which temperature changes trigger events and timing constraints are expressed as a guards; this is reversed in figure 3b to show how the choice of event or guard is essentially arbitrary and therefore one must be careful in making assertions about the real events driving a system. Figure 3c shows that a two-state machine with guards is technically equivalent to a four-state machine with no guards. Guards are useful because they allow some aspects of the control behaviour of a system to be elided, by pushing it down into the concrete memory variables. However, it would be better if UML Statecharts had a mechanism for exposing the extra states implicit in such guards and relating these to the behaviour of the explicit state machine (see section 5.1 below).

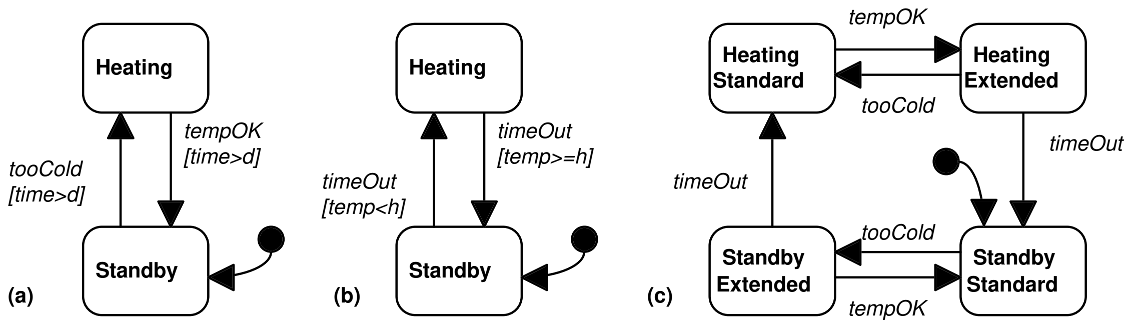

Figure 3: Guards Conceal Duplicated Control States

2.3 Inconsistent or Intractable Constructions

Examples of inconsistent constructions include the context-dependent interpretation of *pseudostates* as: states, arrows, or connectors (see sections 3.1, 3.2 below); and the special treatment of *free transitions* leaving composite states (see section 4.1 below). The freedom to indulge in arbitrary *boundary crossing*, which includes the notion of *stubbed transitions* (see figure 5) violates the clean compositional semantics of hierarchical state machines (see section 4.2). While the admission of *history states* must be regarded as permanently intractable, since recorded history conceals a product of state machines [16], it is possible to provide alternative treatments of pseudostates and free transitions, such that the notion of an *accept state* may be more cleanly defined (see section 4.3). This allows eventually for a clear distinction between forms of composite state which are strictly hierarchical, and forms which are merely convenient abbreviations for expanded machines.

3 Basic Properties of a State Machine

The UML Statechart [4] is based on the Harel Statechart [1], with certain small changes to express encapsulated object-like behaviour; and a modified execution semantics based on the queueing of events [5]. The Harel Statechart is itself a mix of *Mealy* and *Moore* classical state machines, with further extensions more characteristic of a *flowchart*. The tensions between these different models pose some challenges to a consistent formal interpretation.

3.1 Classical Machines and Flowcharts

In a classical finite state automaton, the states are quiescent vertices in the graph and all computational activity happens on the transition arcs, as events are processed. Mealy machines may be styled as transducers [7] which read an input symbol as each arc is traversed and generate an output symbol at the same time (figure 4a).

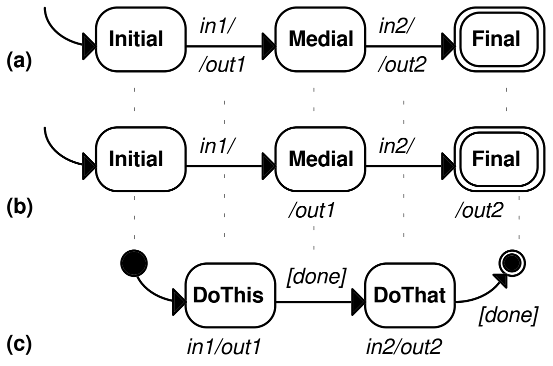

Figure 4: Comparison of (a) Mealy, (b) Moore Machines and (c) Flowchart

By contrast, the output of a Moore machine is contingent on which state it reaches, rather than which arc it is traversing. It is possible (though not necessary) to view this as the output occurring when the machine is in the destination state (figure 4b), fostering the idea that computation can happen in a state. Both Harel and UML Statecharts adopt the notion that the states are active processing stages, rather than quiescent vertices. This leads to a degenerate kind of machine in which both input and output can be processed in "states" and the transition from "state" to "state" is then automatic (figure 4c). This is a *flowchart*, not a *state machine.*

Figure 4 illustrates some of the important correspondences and differences. In particular, notice how the arcs in the Mealy automaton correspond to the processing stages in the flowchart. The arcs on the flowchart correspond to quiescent states in the Mealy machine. The flowchart completely reverses the senses of *state* and *transition*; thereby demonstrating why it is difficult to arrive at any consistent combined interpretation. In a flowchart, transition to the next processing stage is automatic, or dependent on some internal condition computed during the previous stage, rather than on any handled event, such that the behaviour of the machine at the current level of abstraction cannot be determined from the processing of events at this level. This is an important formal property which must be restored. The notion of *state* is defined formally by the varying response of a system, on different occasions, to the same event; such a view of state must be maintained in any semantics of state machines.

3.2 Contextual Interpretation of Pseudostates

The admission of non-quiescent "states" corresponding to processing stages does not otherwise pose any major theoretical problem. However, their usage necessarily forces the invention of *extra* start and finish points, which are not processing stages in the same sense. These are referred to as initial and final *pseudostates* in UML [4, 5], because their role either as first-class states or as inter-level connectors is left deliberately vague.

In a classical state machine, the *initial state* is a first-class state indicated by an initial free transition arrow (see figure 4a, 4b). In UML Statecharts, the initial state is an extra pseudostate (see figure 4c) before the first substantive "state". Likewise, the classical notion of an *accept state* (see figure 4a, 4b) is the final state reached in the machine when an event sequence has been fully processed. In UML, the final state is usually the extra pseudostate (see figure 4c) reached *after* the last substantive "state" of the machine. The comparisons in figure 4 reveal how the initial and final pseudostates, though they correspond to quiescent states in a classical machine, are equivalent to the arcs in a flowchart and are not like the other active processing "states" at all. Instead, pseudostates have the same status as mid-points reached on the arcs connecting processing stages in the flowchart.

UML pesudostates eventually have highly context-dependent interpretations [5] as either states or connectors, arising from the ambiguous treatment of "states" sometimes as the processing stages and sometimes as the quiescent points in between. At the outermost level, the initial and final pseudostates are interpreted as classical, quiescent states, before and after the machine enters its active "states". At nested levels, where the same icons are used to indicate entry and exit points from substate machines, the initial pseudostate cannot have this interpretation, since it is used like a Mealy-style initial arrow, indicating the first substantive state in the nested machine (see figure 1). The final pseudostate is nonetheless intended as a genuine Mealy accept state, indicating termination of the substate machine. Depending on the context, UML switches between the semantics of classical arrows, classical states and modern "states" (ie flowchart processes) in an unhelpful way.

While the intent of the UML 1.3 semantics document [5] is eventually to disambiguate the different context-dependent meanings and uses of the pseudostate icons, surely this is the wrong approach. The elements of a formal notation should have unambiguous, context-free interpretations: to do otherwise is to invite chaos [16]. Consider that pseudostates may only have a consistent interpretation as classical states if all the other "states" are processes. If a Statechart contains genuine quiescent states, then pseudostates have no consistent interpretation as states (what is the meaning of a pseudostate before an initial classical state?), but could be treated as indicator arrows, or connectors between different levels in a state machine hierarchy.

3.3 Required State Machine Semantics

A state machine is not a flowchart. In order to be able to apply state-based verification and testing theorems to the model, a Statechart must conform to state machine semantics. It must have proper reactive states and transitions must be triggered in response to events. Below, a number of guiding principles are introduced which help to ensure these properties.

- Principle #1: States are defined by their differential responses to the same event. You cannot define a state by the amount of time that a system may dwell in it. This rules out pseudo state machines which are merely sequences of processes strung together.
- Principle #2: Events are messages or signals, not conditions. The next state decision function should be placed on the transitions and not hidden inside processing states. This rules out simple conditional branching masquerading as event handling.

- Principle #3: The next state is computable from a state and an event. There are no hidden or implicit conditions. If a state has two or more transitions that can fire in response to an event, then the machine is non-deterministic, otherwise it is deterministic.

To be tractable under formal analysis, the states must be genuine states; that is, the response of the system to an event must be *contingent* on which state it is in. This clarifies informal definitions of states as being system conditions "which can handle events" [6], or which "may be queried by a boolean-valued function" [17]. This does not rule out processing states, so long as these states handle events. The restriction to genuine events is more severe, since it forbids branching on simple boolean-valued conditions. To do otherwise upsets the uniform event-processing semantics, because it hides the next state decision function inside the previous state. While these restrictions eliminate crude flowcharts, it is still possible to convert an *exit condition* into an *event*. In this case, the processing state must generate the internal event, which is subsequently handled by the next state decision function.

- Principle #4: The pseudostate icons are *entry and exit connectors*, not *states*. Never refer to them as states, pseudo- or otherwise. Instead, visualise them as the mid-points along transition arcs into the first, and out of the last, substantive state.
- Principle #5: An *initial state* receives a *single* half-transition from an entry connector. An initial state is not the connector itself (which is not a state), but the subsequent state.
- Principle #6: A *final state,* or *accept state* has a *single* half-transition to an exit connector. A final state is not the connector itself (which is not a state), but the preceding state. To be a legal *accept state*, it must not have any other exit branches.

This creates a semantics that is consistent across quiescent and active state interpretations; and also consistent across different levels of hierarchical composition of state machines. The *entry connector* has the sense of a half-transition, a Mealy arrow indicating the *real* initial state of the (sub-) system. If the notion of an initial quiescent state is relevant in a Moore machine with active processing states, then this must be modelled explicitly as a real state. The *exit connector* has the inverse sense of a half-transition handing back control to the higher level. The notion of a Mealy *accept state* is therefore defined as the last substantive state from which a half-transition exits. In a Moore machine, this state will nonetheless perform all its computation before terminating. An accept state (or *final state*) may not have any further exit transitions, since this would contradict the semantics of finality.

4 Compositional Properties of State Machines

In the hierarchical state machine model, an exit transition leaving a composite state's boundary is deemed to exit immediately from all of the state's substates also. In figure 1, the *neutral* transition exits the *Drive* state; this is understood to abbreviate and abstract over multiple *neutral* transitions leaving each substate *1stGear*, *2ndGear* and *3rdGear*. This is a useful feature which eventually contributes to the desired compositional property of hierarchical state machines. Operationally, it means that exit transitions leaving composite state boundaries interrupt the on-going activity of the substate machine. This semantics also guarantees that composite states are reactive states rather than locked-in processes.

4.1 Inconsistent Treatment of Free Transitions

However, there is an inconsistency in the case where the exit transition leaving a composite boundary happens to be an *unlabelled transition*. UML defines normal transitions as arrows between states that are labelled with the event that they process (classical machine), or the condition that they satisfy (flowchart) [5]. *Unlabelled transitions*, on the other hand, have the semantics of a free ride, since they are not contingent on any event or condition. A state representing a processing stage may have a free exit transition, meaning that it may be quit automatically once its associated processing has terminated. However, if such a state is a composite state with substates, there is an immediate problem in interpreting the free exit transition consistently, since under Harel's hierarchical semantics [18], a transition leaving the composite state boundary is equivalent to an exit transition from every substate. Literally, this should mean that every substate also has a free exit transition. Upon being entered, the substate machine should therefore terminate immediately!

To prevent this, UML defines a free boundary exit transition specially as the *completion transition* taken when the substate machine terminates [5]. We are asked to assume the existence of an invisible label standing for the *completion event*, which is raised by the substate machine when it terminates. To ensure that the substate machine executes to completion, UML suspends Harel's standard interpretation of a transition leaving a composite state boundary [18]. In UML, a free boundary exit transition does *not* interrupt the activity of a substate machine, but waits for its termination. This is quite significant, because it reverses the priority of inter-level event handling in the operational semantics; and means that states are not always reactive, but are sometimes locked-in processes. Again, the intent of UML's definition [5] is eventually clear, but it gives rise to machines that behave strangely in different circumstances, as the following example illustrates.

UML reverses the priority of concurrent event handling across different levels of state hierarchy with respect to Harel [5, 18]. If events arrive concurrently, then cases arise where multiple transitions could fire in machines at different nested levels. To resolve such conflicts, Harel always selects the transition in the outermost state machine, whereas UML selects the transition in the innermost state machine. In Harel's semantics, if the automatic gearbox state machine from figure 1 is in composite state *Drive* and substate *2ndGear* and then receives the events *{neutral, upShift}* simultaneously, the *neutral* transition is fired, since the outermost state machine has priority (the vehicle therefore always reacts to the driver's instructions). In UML, the *upShift* transition is preferred and fires instead, since the innermost state machine has priority (apparently ignoring the driver's instructions). However, if only a *neutral* event is received, there is no inter-level conflict and UML then interrupts the ongoing activity of the substate machine and exits to the *Neutral* state.

This has the curious consequence that a substate machine may sometimes be interrupted and sometimes not (depending on the presence, or absence of a conflicting concurrent event). The outer state machine cannot then be formally analysed in isolation. In the example above, it is impossible to tell whether a *neutral* event will be handled, or discarded due to the presence of an internal *upShift* event, which is invisible at this level of decomposition. For this reason, Harel's original priority rule is an essential part of any compositional Statechart semantics.

4.2 Boundary Crossing Violates Encapsulation

Through the notational convenience allowed by composite states, designers may produce models which, although they have the appearance of modular, encapsulated and hierarchical systems, are no less complicated than fully expanded flat state machines. One practice which immediately violates the encapsulation of hierarchical state machines is *boundary crossing*. Figure 5a illustrates an example, in which transitions both enter and leave substates directly, crossing the enclosing composite state's boundary. Graphically, the substate machine of figure 5a may even be elided, by drawing the arcs labelled *direct* and *skip* as *stubbed transitions* [4] entering and exiting the superstate, shown in figure 5b. But it would be wrong to equate this with encapsulation. In order to reason about the formal properties of this system, it would always be necessary to expand it to the flat machine shown in figure 5c, since the superstate machine is strongly, and completely, coupled to the substate machine.

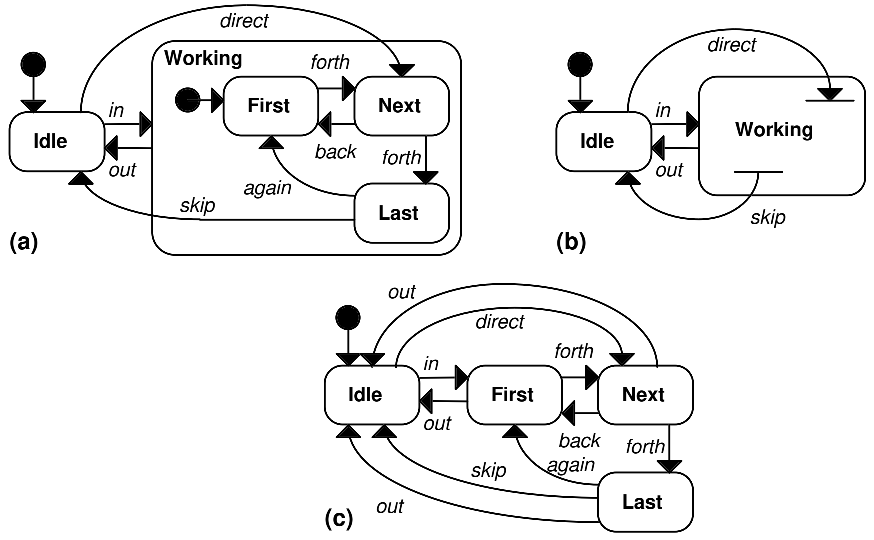

Figure 5: Logical Expansion of Boundary Crossing

Boundary crossing is clearly expected and allowed by the UML authors [6], p299, 301, 333, 437. Since this notational feature has such a disastrous impact on the encapsulation of hierarchically composed states, we might seek to outlaw it in any clean compositional model. However, the ability to compose states also serves the more mundane purpose of abbreviation. Where a group of states share a common exit transition, such as the arc labelled *out* in figure 5, it is notationally convenient to group these in a composite state, simply in order to draw the exit transition just the once. Such a composite construction cannot be understood in any hierarchical sense, nor is it technically even a *state*, but it may be considered an *abbreviation* for a set of transitions. Ideally, we should seek to distinguish genuine hierarchical states from mere abbreviations, and insist that abbreviations are always expanded to flat state machines in the formal analysis. This, however, would be at the cost of increasing the sizes of test sets in the product of states and transitions for the flattened machine, compared to the sum of states and transitions in a hierarchical machine (see section 1.2).

4.3 Boundary Crossing to Indicate Acceptance

It is sometimes interesting to examine Statecharts in which boundary crossing is present, in order to attempt to infer why the designer resorted to this strategy. Another reason for crossing boundaries appears to result from a need to indicate nested *accept states* by free exit transitions crossing the boundary [16]. According to the UML 1.3 Notation Guide [4] and Semantics [5], an accept state should be drawn using the final pseudostate icon. However, many examples exist in [6] where the accept states of a substate machine are *only* indicated using boundary-crossing free exit transitions.

After seeking an explanation for this, we eventually found that UML Statecharts can only properly represent *one* distinct accept state in a substate machine, whereas classical state machines often have multiple accept states, indicating distinct outcomes. In UML, all exit paths from multiple accept states are subsequently merged in the *single* completion transition leaving the composite state boundary. Figure 6a illustrates an example where there are two final pseudostates representing logically distinct outcomes in the substate machine, but which cannot be distinguished externally, and figure 6b distinguishes between these outcomes by boundary crossing, showing how designers are forced into this strategy in order to express what they want to say (see also [4], p 3.137 for a further example).

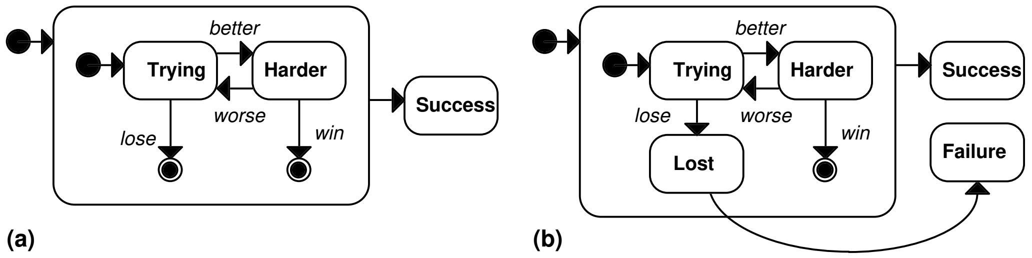

Figure 6: Encoding Multiple Accept States

The work-around is assymmetrical, which usually indicates a problem. It would surely be better if UML had a more appropriate way of indicating distinct *accept states* in a nested machine. Note that the approved use of the free exit transition in figure 6 is notwithstanding our objections raised above in section 4.1.

4.4 Required State Compositional Semantics

To facilitate the hierarchical composition of independent state machines, some further principles are introduced below. These address the problem of free transitions and the fact that their source states cannot be decomposed; the need to distinguish abbreviations from genuine composite states; and the need to distinguish multiple accept states.

- Principle #7: Hierarchical, encapsulated state machines must be independently verifiable. The behaviour of a state must be transparent at the level of its peer states; and the behaviour of its enclosed substate machine must be transparent at the next level down.
- Principle #8: A hierarchical, or composite state encapsulates a substate machine. It may not exhibit any boundary crossing, but must use a single entry connector and possibly one or more exit connectors to communicate with its substate machine.
- Principle #9: An abbreviation is not a composite state. It is a notational short-hand for describing a set of exit transitions shared by several states. It may not therefore use hierarchical entry and exit connectors and must use boundary crossing.

The criterion of independent verification is quite deliberate and powerful, giving the intended meaning to the term *encapsulation*, which cannot now just refer to the visual packaging of elements, but also must refer to the logical independence of the elements packaged. The reasons behind forbidding the use of boundary crossing for genuine composite states are clear, but the insistence that abbreviations *always* use boundary crossing is an interesting corollary. This is because an abbreviation is a notational device, not a state, therefore it is not part of any state hierarchy and cannot have entry and exit points. Communication to parts of an abbreviation must therefore go directly to those internal states. We might also want to insist that abbreviations are not given names like states; they are just enclosed regions. A given state may lie in the intersection of multiple regions, indicating that it shares the union of their transitions. Formally, such regions must always be expanded to flattened state machines.

- Principle #10: A substate machine always generates termination events. When a machine reaches one of its accept states, it must signal this to the higher level by generating the event associated with that accept state. Distinct accept states generate distinct events.
- Principle #11: Events may be simple, or compositional. Some events are messages that encapsulate argument values, which are unpacked by a substate machine. Accept states at one level may wrap up values in the event signalled to the higher level.
- Principle #12: Free transitions may only link connectors to states. No free transition may link two substantive states. The link between an entry connector and the initial state is a free transition, likewise that between an accept (final) state and an exit connector.

The requirement for accept states to generate termination events solves a number of different problems. Firstly, this means that *every* transition leaving a composite state boundary is a labelled transition, where the label corresponds to the termination event for the composite state. Secondly, a composite state may have more than one termination event, corresponding to the different accept states in the encapsulated substate machine. This strategy generalises properly to concurrent substate machines. The exit transition leaving a superstate boundary may be labelled with any synchronous conjunction of events generated by concurrent substate machines. This is irrespective of whether the substate machines have multiple accept states indicating distinct outcomes, since these can be conjoined in the desired fashion (so fixing the boundary-crossing problem in [4], p 3.137 for example). The only places where free transitions may occur is in linking entry and exit connectors with their respective initial and final (accept) states. These arrows should *never* be labelled with events, since they are not full state transitions, but merely connection points. To justify this, consider that: no further events can be handled by a machine that has reached its accept state; and no machine can handle an event before its initial state.

5 Formalisation

A state machine for an object is a tuple (S, Φ, E, R, M). The object may exist in a number of control states $s \in S$ which abstract over its concrete memory states $m \in M$, and always starts in a distinguished initial state $s_0 \in S$, $m_0 \in M$. The object reacts to events $e \in E$, the set of message requests understood by that object. The set of transitions $\phi \in \Phi$ describes the reaction of the object to an event, or the processing carried out by its methods. A transition is a maplet from a source state and event to a target state and response: $\phi = (s_i, e) \rightarrow (s_j, r)$, where $s_i, s_j \in S$ are the source and target states, e is an event and $r \in R$ is a response to the event. The set of responses is *output-distinguishable* if Φ and R are isomorphic, that is, for each $\phi \in \Phi$ there exists a unique $r \in R$ to indicate which transition was fired.

5.1 Simple and Guarded Transitions

If the behaviour of the state machine is deterministic, then only one transition can be fired in response to an event: $\forall \phi_i, \phi_j \in \Phi.\ \mathrm{dom}(\phi_i,) \cap \mathrm{dom}(\phi_j) = \emptyset$. If this is not the case, then it is possible to make the machine deterministic by extending the domain of the transition with a guard on a concrete memory state $m \in M$, which maps to an extended co-domain. In this case, the set of transition functions $\phi \in \Phi$ is considered to be replaced by $\psi \in \Psi$, such that each $\psi = (s_i, m_i, e) \rightarrow (s_j, m_j, r)$, where $m_i, m_j \in M$ are the before and after concrete states of the object's memory, and $\forall \psi_i, \psi_j \in \Psi.\ \mathrm{dom}(\psi_i,) \cap \mathrm{dom}(\psi_j) = \emptyset$.

A precise relationship relates the presence of guards on memory to elided control states. If g(m) is a boolean-valued guard on a concrete memory state $m \in M$, then the logical complement $\neg g(m)$ exists, whether or not it is explicitly notated. For some constant $s \in S$ and $e \in E$, a guarded transition $\psi(s, m, e)$ fires only if the guard g(m) is true, otherwise it fails. This and the implicit complement $\neg g(m)$ may therefore be refined by splitting the control state s into two new states: s_g, $s_{\neg g}$ which encode the guards, such that the guarded transition is replaced by the unguarded $\phi(s_g, e)$ and $\phi(s_{\neg g}, e)$. In general, many guarded transitions may exist for the same event and state. For each distinct ψ_i fired on (s, m_i, e), the memory states m_i are all that distinguish the ψ_i. This implies that guards $g_i(m_i)$ for each ψ_i must be mutually exclusive: $\forall m \in M, \forall g_i, g_j \,.\, g_i(m) \wedge g_j(m) = \text{false}$. If there are n such guards, then these either cover the concrete memory states of the object exhaustively, or there is an implicit logical complement, defined by: $\neg(g_1(m) \vee g_2(m) \vee ... \vee g_n(m))$. Accordingly, a state with n guarded transitions triggered on an event e may be refined into n states (empty complement) or n+1 states (non-empty complement) with simple unguarded transitions.

5.2 The States of an Object

At the finest level of detail, an object has many concrete memory states $m \in M$. An upper bound on the size of M may be calculated as: $\text{card}(M) = \text{card}(A_1) \times ... \times \text{card}(A_n)$, where $A_1...A_n$ are the sets of values from which the object's attributes are drawn; however, not every value combination may be meaningful. The number of abstract control states $s \in S$ is far fewer, determined by the differential response of an object to events. There are several ways in which this response may be judged: one is by examining when it is legal to invoke some of its methods; another considers when certain methods are disabled (having a legal, but null effect). A final approach considers all the state-dependent output responses of an object. These approaches yield progressively finer-grained models of control state.

Consider first that an object's methods typically have the semantics of partial functions, that is, they are not all legally executable in every state. For each $s \in S$ there exists a distinct subset of events $L_s \subseteq E$ to which the object may legally react. A subset of events L_s that are *legal* for an object in state s is defined by:

$$L_s = \{e \in E \mid \phi(s, e) \neq (s_\perp, \perp)\},$$

where $s_\perp$ is the error state and $\perp$ is the undefined response. The number of control states then follows from the number of distinct sets of legal events:

$$\forall s_i, s_j \in S, \forall L_{si}, L_{sj} \subseteq E \,.\, s_i{=}s_j \Leftrightarrow L_{si}{=}L_{sj},$$

that is, two states can only be distinguished if a distinct set of legal events exists for that state. According to this definition, if an object always responds legally to every event $e \in E$, then it only has one abstract control state. A bounded stack has three legal states and one error state, derived from the following distinct sets of legal events: *empty*$\leftrightarrow$*{push}*, *loaded*$\leftrightarrow$*{push, pop}*, *full*$\leftrightarrow$*{pop}* and *error*$\leftrightarrow\{\}$. A corollary is that each of these states also corresponds to distinct subsets of events $e \in (E - L_s)$ that are *illegal* in that state, for example (E - *{push}*) = *{pop}* is illegal in state *empty*. It may be easier to construct the control states of an object by considering when certain methods are illegal. If an object *always* responds legally to a subset $L \subseteq E$ of events, then $n = \text{card}(E - L)$ messages are illegal in some state. A maximum 2^n possible control states then may exist, calculated from the cardinality of the powerset **P**(E - L).

Consider next that a larger set of abstract control states $s \in S$ may be chosen, corresponding to distinct subsets of *valid* events $V_s \subseteq E$. A valid event $e \in V_s$ is one to which the object reacts in some way, whereas an invalid event $e \in (E - V_s)$ is one which the object ignores. A subset of events V_s that are *valid* for an object in state s is given by:

$$V_s = \{e \in L_s \mid \phi(s, e) \neq (s, r_\varnothing)\},$$

where $r_\varnothing$ indicates a trivial response generated when the event is ignored and the object does not leave the state s. Since an event may be legal, but invalid for $s \in S$, $\text{card}(V_s) \leq \text{card}(L_s)$ holds and many more distinct valid subsets $V_s \subseteq E$ may therefore be chosen than legal subsets $L_s \subseteq E$, and the greater number of control states $s \in S$ follows from this. It may be more intuitive to construct these control states by considering when certain events are either *invalid* or *illegal*: if a subset $V \subseteq E$ of events is *always* valid for an object, then $n = \text{card}(E - V)$ messages are either invalid or illegal in some state. A maximum 2^n possible control states then may exist, equivalent to the cardinality of the powerset **P**(E - V). The largest possible set of abstract control states $s \in S$ may be chosen according to the distinct output responses of the object:

$$\forall s_i, s_j \in S, \forall e \in E, \exists s_k \in S, \exists r \in R \,.\, s_i{=}s_j \Leftrightarrow \phi(s_i, e){=}(s_k, r) \wedge \phi(s_j, e){=}(s_k, r).$$

Two states can only be distinguished if they immediately yield distinct responses to an event, or else if the transition which fires in response to this event leads to distinct states (judged recursively by this rule).

5.3 Event Handling Semantics

An event handling model based on Harel's asynchronous semantics with run-to-completion is assumed [18]. The only difference is that, whereas in Harel's semantics, all events are broadcast to all hierarchically nested state machines, in the semantics given here, events are produced and consumed within the scope of particular machines, representing the idea that events are targeted at particular objects. In this model, every object *obj* is assumed to have an event-handler η, such that $\eta($*obj*$, e)$ selects a particular $\phi \in \Phi$ to fire in response to the event e. In general, η is a dispatching function which acts on an object and a conjunction of events: $(e_1 \wedge e_2 \wedge \ldots \wedge e_n)$, since we allow concurrent event signalling.

Concurrent events may be received if *obj* encapsulates concurrent substate machines, or if *obj* is a substate machine shared by concurrently-executing machines. To ensure determinism, only one transition may be enabled at a time in *obj*. This is achieved by ensuring that η is well-behaved for all combinations of concurrent events that *obj* could expect to handle. We allow event conjunctions to have first-class status as events: if $E_c \subseteq E$ is the concurrent set of events, then all possible conjunctions is given by: $e \in (\mathbf{P}(E_c) - \{\emptyset\})$, excluding the empty conjunction. Then, we assert: $\forall s \in S, \forall e \in (\mathbf{P}(E_c) - \{\emptyset\}) \;.\; \exists! \phi \in \Phi \mid \phi(s, e) \neq (s, r_\emptyset)$. This ensures that *obj* has distinct states in which it ignores all but one singleton event, or event conjunction. It is possible to synchronise concurrent subprocessing streams by labelling a transition with an event conjunction.

The hierarchical aspects of this model are captured in the execution of each method $\phi \in \Phi$, which may distribute events (send messages) to further object state machines. The sequentially-ordered execution of a method is not described in detail here, but it is modelled as a separate state machine. The objects whose methods it invokes in turn are available as elements of the memory M of the current object. Decomposing further, the behaviour of these collaborator-objects may also be modelled as state machines. This is in a similar spirit to the Holcombe/Ipate model of transition refinement [11, 12, 13] and slightly different from the Cook and Daniels model of object state machine refinement [19]. The only special consideration here is in the semantics given to the completion of processing. The termination of a subroutine (substate machine) must always signal an event to the caller (superstate machine), which in the model looks identical to an event generated at the higher level. This ensures that every object's state machine may be analysed independently. At each level of abstraction, a machine receives external events, corresponding to message requests from its clients. In selecting a particular $\phi \in \Phi$, it executes a single processing step, which may distribute further events to substate machines. These run to completion and return to the caller, signalling completion events. A completion event may in turn trigger a further step, such that the machine makes maximal progress, corresponding to a *super-step* in Harel's asynchronous semantics [18].

6 Conclusions

An alternative semantics for UML Statechart Diagrams has been presented. The semantics are designed around twelve principles. The first six principles ensure a consistent interpretation of both classical-style and flowchart-style elements of the notation. The second six principles support the intended compositional semantics.

In particular, the notion of pseudostates is abandoned in favour of a clearer notion of connectors, such that all other states are represented explicitly. The independence of nested state machines was also examined, with the goal of identifying a set of design rules to ensure that Statecharts were tractable under hierarchical and modular approaches to verification and testing, such as the Chow [9] or Ipate/Holcombe [13] methods. It was found that true composite states must be formally distinguished from abbreviations, which are regions enclosing some states sharing a set of exit transitions. Boundary-crossing transitions must be eliminated from composite states. UML's reversed priority rule for resolving inter-level concurrency conflicts was found to introduce nondeterminism in the outer state machine and so Harel's original inter-level priority rule was restored [18]. These steps together ensured that state machines could be analysed independently of each other. Problems with the interpretation of free transitions were eliminated by ensuring that all substate machines generate real termination events. This also solved the problem of indicating distinct outcomes in a substate machine, which previously could only be indicated by boundary crossing. The same approach scaled up to concurrent event handling.

Finally, a set-theoretic semantics was given in which the translation between guards and simple states was specified, the number of distinct control states of an object was explored and the event handling model was described, showing how this deals with concurrency. This model is being explored as the basis for a compositional design-for-test approach which promises to reduce the usual state explosion.

7 References

1. Harel D. Statecharts: a visual formalism for complex systems. Sci of Comp Prog 1997; 8:231-274

2. Rumbaugh J, Blaha M, Premerlani W et al. Object-Oriented Modeling and Design. Prentice-Hall, 1991

3. Rational Software Corporation. UML 1.1 Reference Manual. http://www.rational.com/uml/, 1997

4. Object Management Group. Section 3: Notation Guide, Part 9: Statechart Diagrams. In: UML 1.3 Reference Manual, OMG, 1999, pp 3.131-3.150

5. Object Management Group. Section 2: UML Semantics, State Machines. In: UML 1.3 Reference Manual, OMG, 1999, pp 2.129-2.157

6. Booch G, Rumbaugh J and Jacobson I. The Unified Modeling Language User Guide. Addison Wesley Longman, Reading MA, 1999

7. Gazdar G and Mellish C. Natural Language Processing in Prolog. Addison Wesley, Reading MA, 1991

8. von der Beeck M. A comparison of statechart variants. Lect Notes in Comp Sci 1994; 863:128-148

9. Chow T. Testing software design modeled by finite state machines. IEEE Trans Soft Eng SE-4 1978; 3:178-187

10. Holcombe W M L. X-machines as a basis for dynamic system specification. Software Engineering J 1988; March:69-76

11. Holcombe W M L. An integrated methodology for the formal specification, verification and testing of systems. Software Testing, Verification and Reliability 1993; 3(3/4):149-163

12. Holcombe W M L and Ipate F. Another look at computability. Informatica 1996; 20:359-372

13. Ipate F and Holcombe W M L. An integration testing method that is proved to find all faults. Int J Comp Math 1997; 63:159-178

14. Binder R V. The FREE approach to object-oriented testing: an overview (synthesis of four articles). http://www.rbsc.com/pages/FREE.htm, 1996

15. Kim Y G, Hong H S, Bae D H et al. Test cases generation from UML state diagrams. IEE Proc Softw 1999; 146(4):187-192

16. Simons A J H and Graham I. 30 things that go wrong in object modelling with UML. Chap. 17 in: Kilov H, Rumpe B and Simmonds I (ed) Precise Behavioral Specification of Businesses and Systems. Kluwer Academic Publishers, 1999

17. d'Souza D F and Wills A C . Objects, Components and Frameworks with UML: the Catalysis Approach. Addison Wesley Longman, Wokingham, 1998

18. Harel D and Naamad A. The STATEMATE semantics of statecharts. ACM Trans Soft Eng Method 1996; 5(4)

19. Cook S and Daniels J. Designing Object Systems, Prentice Hall, London, 1994