Statecharts For Requirements Specification – As Simple As Possible, As Rich As Needed

Martin Glinz
Institut für Informatik, Universität Zürich
Winterthurerstrasse 190
CH-8057 Zurich, Switzerland
glinz@ifi.unizh.ch

ABSTRACT

Statecharts have evolved into a widely used instrument for specifying system behavior and interaction. Several variants of statecharts have been developed, for example, Harel's original statecharts, UML state machines or derived concepts such as the state machines in RSML.

In this paper I investigate how a statechart variant for requirements models should look if we want it to be as simple as possible, easy to understand and well suited for expressing requirements models.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications—*Languages*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Specification techniques*

General Terms

Languages, Theory

Keywords

Statecharts, statechart semantics, requirements engineering

1. INTRODUCTION

Statecharts [4] are a widespread and successful means for specifying system behavior and user-system interaction. Since their original inception by David Harel in 1987, several variants of statecharts have been introduced and there have also been several attempts to underpin the intuitive meaning of statecharts with precise semantics. Harel himself has published different versions of statechart semantics, for example [4] and [5]. UML [8] has adopted statecharts and has given them semantics that differ from Harel's in several points. However, the definition of state machines and their behavior in the OMG definition of UML is vague and incomplete in several points. In a recent paper, von der Beeck [12] has formalized UML statecharts. Leveson and Heimdahl [7] have used a variant of statecharts when creating the requirements language RSML. Glinz [1] has demonstrated the use of statecharts for specifying and integrating scenarios and has also defined the semantics of the statecharts that he uses in his approach. Von der Beeck [11] has done a comprehensive comparison of the statechart variants existing in 1994.

In this paper I examine the distinguishing features of these approaches and discuss which of them are needed and useful when using statecharts for requirements specification. Of course, we cannot model all kinds of requirements using statecharts. Naturally, statechart models will concentrate on requirements concerning dynamic system behavior and interaction. If we have other models which model aspects such as data and functionality, we expect that the statechart models and the other aspect models fit together in a smooth and well-defined way.

The goal is to arrive at statechart concepts and semantics that are *small, intuitive, easy to understand* and *suitable for specifying requirements* for system behavior and user-system-interaction.

More concretely, the goals are as follows:

(1) Ease of understanding
- Make it as simple as possible
- Avoid counter-intuitive behavior
- Avoid global coupling, enable modularity and local understanding.

(2) Suitability for Requirements Engineering
- Typical behavioral and interaction requirements must be expressible with reasonable effort
- Statechart models and models of data and functionality must smoothly fit together
- State and state transition explosion must be avoided.

2. EXAMINING THE ESSENTIAL FEATURES OF EXISTING STATECHART VARIANTS

In this section I present and assess the concepts and semantics of all those features of the existing statechart variants that are relevant for requirements models.

Hierarchical states and orthogonal states. These are the core features of statecharts that master the state and state transition explosion problem and make state machines usable in practical settings. So no statechart variant can do without.

Single event assumption. Most statechart semantics employ the so-called single event assumption, meaning that not more than one external event may happen at any given point in time. This is a quite reasonable assumption because "a point in time" is an abstract concept anyway. If we observe concurrent events in reality, we never know whether they really happened concurrently, because the smallest observable time interval is finite and greater than zero. So we can always arbitrarily serialize concurrent events without losing or distorting essential information. On the other hand, the single event assumption simplifies the definition of statechart semantics – so this is a quite useful feature.

Broadcasting events. Classic statecharts are based on event broadcasting. While this is simple and convenient for small models, it can turn into a nightmare for large ones, because it results in global coupling of all components. UML is quite unspecific in this issue: the way events are transported from their source to the event queues of the state machines where they should take effect is undefined. Within a state machine, a dispatched event is broadcast. RSML [7] uses a component model where broadcasting happens only within components, while components communicate through channels. In the UML-RT proposal [10], we also find a two-level scheme of event propagation: within a capsule, events are broadcast, whereas event propagation from one capsule to another is asynchronous and requires explicit connections.

From a software engineering viewpoint, such a two-level scheme that uses broadcasting only on the local level is clearly better than global broadcasting. It reduces coupling; events are visible only in those components where they take effect. If the source of an event lies far away from the model component where it produces an effect, the channels provide the information where the event comes from.

Synchronous event processing. Classic statecharts employ synchronous event processing. This means that the state machine immediately reacts to an external event and does all state transitions and processing of events triggered by state transitions instantaneously, i.e. in zero time. In particular, all reactions to an external event are completed before the next external event can happen (thus fulfilling the single event assumption, see above). UML queues events instead of immediately reacting to them. However, once an event is dequeued, it is processed synchronously.

Synchronous event processing may have nice formal properties, but it comes with a bunch of semantic problems (see [11]) and may lead to counter-intuitive behavior. For example, in Figure 1 we would expect that event g leads from states {R, U} to either {S, V} or {W, V}, depending on how we treat cascading events. However, in Harel's statechart semantics, event g non-deterministically leads to either {S, V} or {T, V}; the latter clearly being counter-intuitive.

The quasi-synchronous event processing and timing scheme that I proposed in [1] gets rid of most of these problems. In this scheme,

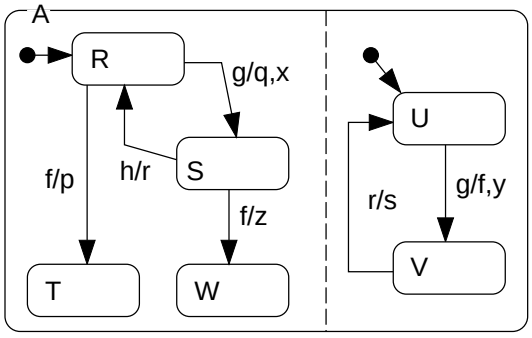

Figure 1. Event processing example

everything takes time, but the time intervals are infinitesimally short. Basically, it works as follows: when an event e happens, all transitions being enabled by e and having a source state which is currently active are taken. The source states of the triggered transitions are left immediately. A time interval ε_0 later, the destination states are entered. Then the actions of the triggered transitions are executed. Within a transition, execution follows the sequence in which the actions are specified. If more than one transition is taken concurrently, the order of transitions is non-deterministic. If an action generates an event, this event is processed completely before the next action is taken.

Example: consider statechart A in Figure 1. Suppose that all actions generate events and that A is currently in states {R, U} When event g happens at time t_e, quasi-synchronous event processing works as follows: at t_e: g happens, R is left, U is left; at $t_e+\varepsilon_0$: S is entered, V is entered; at $t_e+\varepsilon_1$: q is produced; at $t_e+\varepsilon_2$: x is produced; at $t_e+\varepsilon_3$: f is produced, S is left; at $t_e+\varepsilon_4$: W is entered; at $t_e+\varepsilon_5$: z is produced; at $t_e+\varepsilon_6$: y is produced. If the next external event happens at time $t_{e'}$, for any δ with $0 < \delta < | t_{e'} - t_e |$ holds $\varepsilon_0 < \varepsilon_1 < ... < \varepsilon_6 < \delta$, that means execution time is infinitesimally short. Alternatively f, z and y could be produced prior to q and x: the order in which concurrent transitions are processed is non-deterministic.

This is a both powerful and intuitive abstraction for requirements engineering models: things take time and happen in sequence, but we can neglect the actual (implementation-dependent) duration.

Kinds of actions. Harel statecharts have a quite simple action scheme: actions are triggered by state transitions and work synchronously, i.e. in zero time. UML, on the other hand, has introduced an elaborate action scheme, distinguishing entry actions (triggered and completed prior to entering a state), exit actions (triggered upon exiting a state and completed before proceeding to the next state) and do actions (executed while the system is in a particular state).

For modeling requirements, it suffices to have two kinds of actions: those that are triggered and completed during a state transition and those that are executed while the system is in a particular state. Having these two kinds, we can employ a simple and powerful action triggering system (see Section 3).

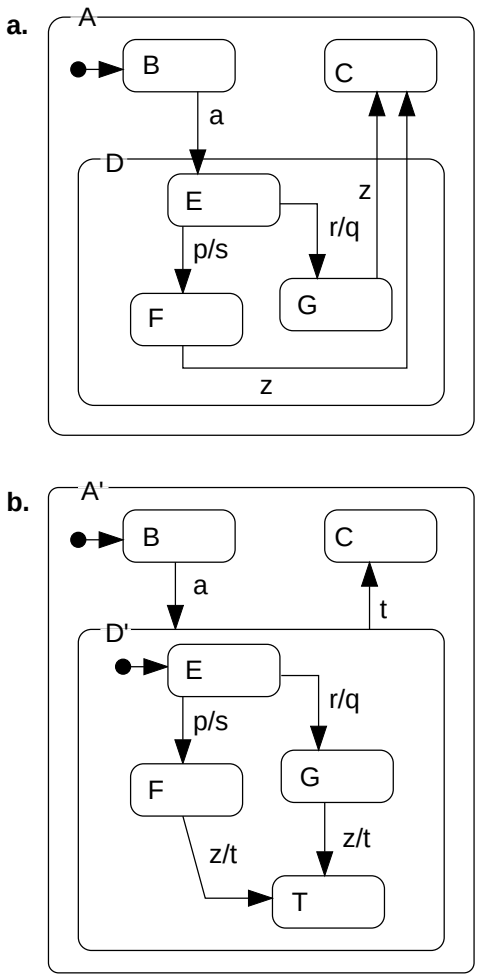

Figure 2. The inter-level transition problem
a. Statechart with inter-level transitions.
b. Equivalent statechart without inter-level transitions. Statechart A' is compositional, i.e. it can be viewed as a composition of three black box components B, C and D'.

Event queues. The UML event queues are an implementation-oriented concept that can be omitted for requirements models, thus yielding much simpler event semantics.

History. Both Harel and UML statecharts provide a history mechanism that allows easily re-entering that substate of a statechart which had been the last active one before the statechart was left.

When specifying dialogs, it sometimes may be attractive to have such a feature. However, I have never seen requirements models that crucially needed the history mechanism. On the other hand, it incurs cost in terms of comprehensibility and simplicity of semantics.

Inter-level entry and exit transitions. In most statechart variants, a transition may directly enter or exit a nested substate of a statechart, crossing several nesting layers.

While this is sometimes convenient for the model writer, it is bad for model readers and maintainers because it breaks the abstraction that comes with the nesting of statecharts. It is also bad for constructing large models from components, because it hinders or even prevents composition of statecharts. There is no urgent need for inter-level transitions: they always can be replaced by a set of layer-conforming ones (Figure 2).

State transition trigger conditions. Both Harel and UML statecharts use events and guard predicates for controlling the triggering of a transition. Usually, the trigger conditions and the triggered actions are written as annotations of the state transition arrows (cf. Figures 1 and 2). This notation quickly breaks down when it comes to complex triggering conditions. RSML introduced AND-OR tables (which are in fact just decision tables) for representing complex trigger conditions in a clear and comprehensible way[1].

For practical models, we need both: when the trigger is a single event, a decision table is overkill. On the other hand, complex conditions are easier to comprehend when written separately in tabular form.

Treating/prioritizing competing state transitions. If an active state has more than one transition that is being triggered by the same event, the statechart semantics must decide which transition to take. The simplest semantics is to take a random decision. However, this is frequently not adequate. Alternatively, one could define hierarchy-based rules (inner-level first, for example) or define transition priorities explicitly.

States and data. All major statechart variants model the overall state of a system not exclusively with states, but also with variables. These variables may be written by statechart actions and their value may be evaluated in state transition conditions. Using variables is a pragmatic way to reduce the number of required states. It allows the modeler to express the essence of behavior and interactions with states while the system memory is modeled with variables. Using variables is also a straightforward way of integrating statecharts with other models that are oriented towards data, functions, or objects. In requirements models, where we have to express all these aspects, we need variables in most domains. However, variables must be used with care: undisciplined use of variables breaks abstraction and leads to global coupling of components.

Integrating statecharts with other models. The existing statechart variants do not solve the problem of integrating statechart

[1] While disjunction of events is obvious, one may wonder how event conjunction works: with the single event assumption, a conjunction of events can never be true.
However, in a model where we have variables (see the subsection on states and data), we frequently want to specify events that happen when a set of variables takes some given combination of values. This can be expressed by a conjunctive predicate, for example mode = operational ∧ cruising ∧ speed > 100. At every point in time where such a predicate becomes true, an event is generated. It is also possible to define the conjunction of an event happening and a predicate being true as a trigger.
An AND-OR table is basically a disjunctive collection of such conjunctive predicates.

models with other (function-, data-, or object-oriented) models convincingly.

Harel models behavior and functionality as two separate models which both have a decomposition hierarchy of their own. The models communicate by references to variables and by invocation of operations [5], [6]. The separately modeled behavior hierarchy allows specification of behavior on all levels from overall global behavior to detailed local behavior and is strong if we focus on aspect separation. However, this approach is quite weak when considering aspect integration.

UML, on the other hand, considers statecharts as auxiliary models that are embedded in the specification of classifiers in order to describe their internal behavior. UML thus integrates the models of a classifier and of its behavior, which makes it easy to model local behavior. However, as UML has no true composition of components (where the composite is a higher-level abstraction of its components, see [2]), it is awkward to specify global behavior in UML.

In the UML-RT proposal [10] as well as in ROOM [9] (where UML-RT is derived from), there is a hierarchical composition of components (components are called capsules in UML-RT). However, UML-RT is in no way simple. It is an UML profile, which is eventually mapped to ordinary UML concepts. Thus it inherits all the problems that plague UML: for example, complexity, no precise semantics, and the difficulty of specifying global behavior.

In our own work [3], we have a simple integration of objects and behavior. An object decomposition hierarchy serves as the backbone of a system model. Objects may be viewed as composite states and may be refined to pure statecharts on elementary levels. So our object hierarchy forms a statechart hierarchy at the same time and allows us to model behavior and interaction at the place and on the level of abstraction where it is expressed best.

3. RECOMMENDATION FOR A REQUIREMENTS MODELING-ORIENTED STATECHART VARIANT

Based on the findings from Section 2 and considering the goals stated in the introduction, I propose the following statechart structure and semantics for modeling (functional) requirements specifications.

Basic features: *Hierarchical and orthogonal states, single event assumption.* Quite obviously, these constituent features of statecharts should be present in any serious statechart variant.

State transition syntax: *Two notations for state transition triggers, no inter-level transitions, no history*. As discussed in Section 2, it makes sense to have two notations for state transitions. For simple event-action pairs, we keep the original notation of *event/action*, which is attached to the arrow that denotes the state transition. Complex triggering conditions are written in tabular form with disjunctive columns and conjunctive rows, analogous to the AND-OR-tables of RSML. This notation also allows the formulation of state transition guards in an easy and straightforward way.

Inter-level transitions are forbidden because they break abstraction and make statechart composition hard or impossible.

History is omitted because it does add not enough value for requirements modeling in comparison to the complexity it adds.

Event processing: *Two-level distribution, quasi-synchronous event processing, quasi-synchronous and asynchronous actions, no event queues, simple prioritization scheme for competing transitions*. On a local level, broadcasting events is simple and powerful, which is good. On a global level, it breaks modularity due to global coupling, which is bad. Therefore we follow the two-level concept of RSML: Within an object, events are broadcast synchronously. From one object to another, they have to be transmitted explicitly via channels. This transmission is asynchronous.

Quasi-synchronous event processing is intuitive (everything takes time in reality), but still abstract enough for expressing requirements (we do not need to care about actual duration). Furthermore, it avoids most of the semantic problems that come with the traditional synchronous event processing.

Our concept of action processing is aligned with the concept of event processing: we distinguish quasi-synchronous actions and asynchronous actions. A quasi-synchronous action runs to completion in infinitesimally short time. Any quasi-synchronous action which is triggered in a state transition hence runs to completion within the time interval of the transition. When an asynchronous action is triggered, it is just started and then runs asynchronously. The completion of an asynchronous action is sensed by a completion event. Leaving a state terminates all asynchronous actions that have been triggered upon entering the state and have not yet completed.

Event queues are normally not needed when modeling requirements, so we omit them. In the rare cases where an event queue must be modeled, it can be modeled explicitly.

The problem of non-determinism when an event triggers two or more competing transitions is solved with a simple priority scheme that may be overridden by explicitly set priorities: By default, the innermost of the competing transitions is taken. If the competing transitions all are on the same hierarchical level, the behavior is non-deterministic. Any other scheme can be imposed by explicitly adding priorities to the transitions.

Integration: *Integration of statechart and object hierarchies.* In nearly every requirements model, we need a combination of an object model with a model of behavior and interaction. Hence we employ the concept of an object decomposition hierarchy [3], where every object is regarded to be an abstract state, too. An object may comprise other objects and/or states. States may be refined into statecharts.

The syntax of object composition is logically the same as the one for composing states in statecharts and the semantics of entering and exiting objects – triggered by events – is the same as for states in statecharts.

The integration of statecharts into an object hierarchy also helps to solve the problems of broadcasting and referencing variables. An event which is received by an object X is broadcast to all elements

that are contained in X. A neighbor of X receives the event only if it is explicitly sent to it via a channel.

The object structure also yields a natural scoping structure for names: names are visible inside out, but not vice-versa. Objects may export names in order to make them visible outside. Trigger conditions and actions can only refer to those variables that are visible for them, thus avoiding global coupling and enforcing modularity.

Figure 3 gives a rough idea how we can model requirements with an integrated object/statechart model. RoomControl is a component of a larger model that describes the requirements of a heating control system. RoomControl is an object which has two high-level modes, LocalControlOn and LocalControlOff. It further contains a data object Settings, a behavioral object Controller and a scenario ManageLocalRoomTemperature. The latter is an abstract component (indicated by three dots after its name) which may be refined elsewhere into a statechart. By embedding this scenario in the RoomControl object, we express that it belongs to the specification of RoomControl. RoomControl receives the on and off events asynchronously over the channel setOnOff from another object. Within RoomControl, events are broadcast. Settings is a data object having no explicitly modeled state. It contains variables holding the current and default temperature settings and the operations CurrentTemp and DefaultTemp that read the values of these variables. Note that variable and operation definitions are not modeled graphically. The object Controller is refined into a statechart with two parallel threads[2]. The transitions are modeled in tabular form (Figure 4). Controller also contains a variable ActualTemp. Its value is evaluated in the transition table that specifies the state transitions from Init to Modifying and from Monitoring to Modifying. Basically, this table says that if local control is on, the comparison of ActualTemp with the value given by CurrentTemp determines whether the radiator valve will be opened or closed. Otherwise, Controller uses the default temperature value obtained from Settings. The table also acts as a guard: if ActualTemp is equal to the desired value obtained from Settings, no column of the table evaluates to true and no transition is triggered.

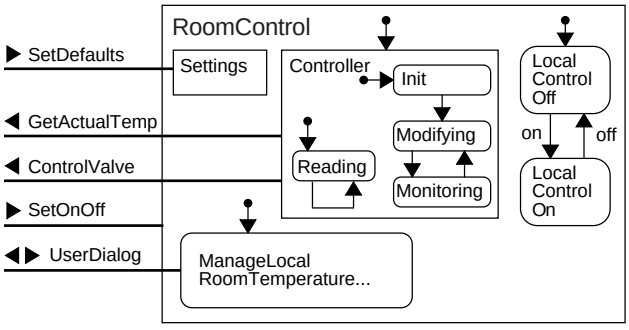

Figure 3. Example of an integrated object/statechart hierarchy[2], showing the room control requirements for a heating control system

[2] Note that we do not use dashed lines for separating orthogonal statecharts. States or objects that are not linked by a transition arrow are considered to be orthogonal.

Modifying → Monitoring

180 s IN Modifying	Y

Reading → Reading

10 s IN Reading	Y
ActualTemp = GetActualTemp()	✓

Init, Monitoring → Modifying

IN LocalControlOn	Y	N	Y	N
ActualTemp > Settings.CurrentTemp(now)	N	•	Y	•
ActualTemp < Settings.CurrentTemp(now)	Y	•	N	•
ActualTemp > Settings.DefaultTemp(now)	•	N	•	Y
ActualTemp < Settings.DefaultTemp(now)	•	Y	•	N
send open **over** ControlValve	✓	✓		
send close **over** ControlValve			✓	✓

Figure 4. Transition tables for the state transitions in the object Controller of Figure 3

4. CONCLUSIONS

Does this requirements-oriented statechart variant (ROSC) achieve the goals stated in the introduction?

- ROSC is considerably *simpler* than the two most important existing variants, Harel statecharts and UML state machines.
- The quasi-synchronous paradigm is *intuitive* and *avoids counter-intuitive behavior*.
- ROSC supports basic software engineering principles: the hierarchical object structure with scoping of names and limited broadcast *avoids global coupling* and fosters *modularity* and *local understanding*. Forbidding inter-level transitions makes statecharts *compositional* and also contributes to modularity.
- From my experience with requirements models I claim that ROSC is powerful and expressive enough for *modeling typical behavioral and interaction requirements in an easy and convenient way*.
- ROSC integrates statecharts into a hierarchical object model. In this modeling framework, *statechart models and models of interaction, behavior, functionality, and data fit together* in a smooth and well-defined way.
- The possibility of using variables *avoids state explosion*.

My proposal is not empirically validated – remember this is a position paper. Motivation and evidence stem from personal experience and from general software engineering and requirements engineering principles.

I have concentrated on requirements models only. The applicability and usability of the proposed statechart variant for other purposes, in particular for architecture and detailed design remains to be investigated.

REFERENCES

[1] Glinz, M. (1995). An Integrated Formal Model of Scenarios Based on Statecharts. In Schäfer, W. and Botella, P. (eds.): *Software Engineering – ESEC'95*. Berlin: Springer. 254-271.

[2] Glinz, M. (2000). Problems and Deficiencies of UML as a Requirements Specification Language. *Proceedings of the Tenth International Workshop on Software Specification and Design*. San Diego. 11-22.

[3] Glinz, M., S. Berner, S. Joos, J. Ryser, N. Schett, Y. Xia, (2001). The ADORA Approach to Object-Oriented Modeling of Software. In K.R. Dittrich, A. Geppert and M.C. Norrie (eds.): *Advanced Information Systems Engineering, Proceedings of CAiSE 2001*. Berlin: Springer. 76-92.

[4] Harel, D. (1987). Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming* **8** (1987). 231-274.

[5] Harel, D. and Naamad, A. (1996). The STATEMATE Semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology* **5**(4). 293-333.

[6] Harel, D. and Gery, E. (1997). Executable Object Modeling with Statecharts. *IEEE Computer* **30** (7). 31-42.

[7] Leveson, N.G., Heimdahl, M.P.E., Hildreth, H., Reese, J.D. (1994). Requirements Specification for Process-Control Systems. *IEEE Transactions on Software Engineering* **20** (9). 684-707.

[8] OMG (1999). *OMG Unified Modeling Language Specification* Version 1.3. OMG document ad/99-06-08. ftp://ftp.omg.org/pub/docs/ad/99-06-08.pdf

[9] Selic, B., Gullekson, G., Ward, P. T. (1994). *Real-Time Object-Oriented Modeling*. New York: John Wiley & Sons.

[10] Selic, B., Rumbaugh, J. (1998). *Using UML for Modeling Complex Real-Time Systems*. White Paper, Rational Software Corporation. http://www.rational.com/media/whitepapers/umlrt.pdf

[11] Von der Beeck, M. (1994). A Comparison of Statechart Variants. In H. Langmaack, W.-P. de Roever, J. Vytopil (eds.): *Formal Techniques in Real-Time and Fault-Tolerant Systems*. Berlin: Springer. 128-148.

[12] Von der Beeck, M. (2001). Formalization of UML-Statecharts. In M. Gogolla and C. Kobryn (eds.): *UML 2001 – The Unified Modeling Language, Modeling Languages, Concepts, and Tools*. Berlin: Springer. 406-421.