ECharts: balancing design and implementation

Gregory W. Bond
AT&T Labs – Research
email: bond@research.att.com

Healfdene Goguen
AT&T Labs
email: hhg@att.com

ABSTRACT
In this paper we argue that the semantics of UML Statecharts, while adequate for design, is inappropriate for implementation. To address the issues raised, we propose a language, ECharts, that incorporates modifications to UML Statecharts. We argue that the ECharts semantics strike the right balance between supporting design and supporting implementation. We compare ECharts language features with comparable features in UML Statecharts and ROOMcharts. We draw motivating examples from an IP-based telecom services platform which uses our implementation of ECharts on a daily basis for design and implementation.

KEY WORDS
software design, real-time systems, programming languages, UML Statecharts, ROOMcharts

1 Introduction

Finite state machines have long been recognized as a natural way to visually model the behavior of event-driven systems. The stated advantage of a visual language is that models described are easier to understand than their textual counterparts. Until the introduction of Harel Statecharts [7], the prevailing modeling languages were based on flat (non-hierarchical), sequential state machines. One advantage of these languages is that their semantics are simple and deterministic. Another advantage is that they are automatically translated to code. The disadvantage is that models with large numbers of states and transitions are difficult to understand, defeating the purpose of using a visual language in the first place.

Harel Statecharts addressed this disadvantage by introducing a number of abstractions, primarily nested (hierarchical) state machines, and concurrent (orthogonal) state machines. Although originally intended only to support design, there was a natural demand to generate an implementation from a design. While this demand was later met by commercial Statecharts modeling tools, the implementations generated were often unacceptable. Since Harel Statecharts was never intended to be an implementation language, the result-
because UML Statecharts is, after all, a modeling language, and the concurrency abstraction greatly simplifies some designs. However, in bringing back concurrency UML Statecharts chose to simplify their semantics, presumably to simplify the runtime system of implementations generated from designs. Instead of synchronous concurrency, a more constrained model of interleaved atomic transition execution is used. UML Statecharts has also introduced the concepts of fork and join transitions, and sync states to provide the designer with control over concurrent state machines in a manner similar to that provided by programming languages with thread support. So it would seem that UML Statecharts attempts to satisfy both designers and implementors by striking a balance between modeling abstractions and implementation-level control.

With this in mind, we chose to adopt UML Statecharts as a design and implementation language for a telecom research project [3]. However, we quickly discovered that UML Statecharts did not provide the desired level of control over execution in an implementation generated from design. The language only supports a single transition priority rule for constraining transition execution order. As a result, many nondeterministic situations can exist in a model. The language only provides rudimentary support for reusing models, and no support at all for parameterizing models. Broadcast communication is utilized for intra-Statechart communication, which in general can result in broadcast events being lost in situations when no Statechart is ready to accept an event. Also, while it is possible for a fixed number of concurrent state machines to be statically exist within a state, it is not possible to dynamically create state machines. The former is akin to allocating a fixed number of worker threads, whereas the latter is akin to dynamically creating worker threads on an as-needed basis. When the number of active Statechart instances required is not known prior to runtime, the latter approach conserves runtime resources. Finally, we found the machine termination semantics to be awkward and inexpressive.

To address these shortcomings we have developed a UML Statecharts dialect, called ECharts, that is being used on a daily basis for designing and implementing telecom services for our project. Wherever possible, the ECharts language adheres to the UML Statecharts standard. As such, a number of aspects of ECharts can be viewed as refinements or extensions to the UML Statecharts semantics, namely the introduction of a port abstraction, the additional transition priority rules, the enhanced support for Statechart reuse, support for dynamic creation of concurrent state machines, and a refined termination semantics. Instead of using the intra-Statechart broadcast communication mechanism used by UML Statecharts, ECharts utilizes a more restricted form of communication. The only semantic element at odds with UML Statecharts is that ECharts utilizes an explicit event consumption model instead of an implicit event consumption model. Space does not permit here, but the formal compositional semantics of ECharts have been specified elsewhere in terms of structural operational semantics (SOS) rules [4].

While the changes we suggest were motivated by implementation, the extensions providing reuse also serve to enhance ECharts as a design language. Although UML Statecharts provides a primitive for including machines within a state, ECharts enables more common fragments to be identified by introducing parameters and priority rules for deciding which transition among nested machines should fire.

Because of its simplicity, UML Statecharts semantics lends itself to automated static analysis, e.g. [11]. Since design analysis is an important goal of our research project, we have taken care to ensure that all language elements in ECharts are also analyzable. Furthermore, we have taken pains to ensure the new language elements can be represented visually in the spirit of UML Statecharts.

A number of tools have been developed or are presently in development to support the ECharts language. Primary among these is a set of Java classes implementing the ECharts runtime system and a language translator which takes an ECharts program as input and translates it to any one of a number of output formats. Currently, the translator can generate Java code that serves as an implementation, and “dot” code [9] for visualization of a design. We are currently developing a translator that outputs Promela code for analyzing a design using the Spin model checking tool [8]. For an early version of ECharts we have also developed a translator [5] that outputs Reactive Module (RM) code for analyzing a design using the Mocha model checking tool [2].

We now examine the new language elements of the ECharts language, provide motivating examples of their use, and contrast them with the ROOMcharts and UML Statecharts languages.

2 ECharts

2.1 Parameterization

Any attempt to define visual languages usable for implementation must provide higher-level mechanisms for reuse. We choose to model parameterization of ECharts using a semantics roughly based on procedures and procedure calls.

More precisely, a declaration of a parameterized ECharts machine is similar to a procedure declaration:

\[
\text{fsm Chart}(T_1 a_1, \ldots, T_n a_n) = \\
\begin{aligned}
\text{body}
\end{aligned}
\]
where Chart is an identifier, T1 to Tn are types and a1 to an are formal parameters. In order to prevent infinitely recursive charts, we prevent Chart from occurring within the body of the declaration, or the bodies of any nested machines. We impose this restriction to simplify the language semantics which effectively restricts ECharts to expressing only finite state machines. However, removing this restriction permits the expression of infinite state machines, a concept that has been formally explored in [1].

Correspondingly, instantiation of a parameterized ECharts machine is similar to procedure application:

\[ \text{Chart}(v_1, \ldots, v_n) \]

where Chart is again an identifier, and v1 to vn are values.

A parameterized ECharts machine is used by declaring it as an included machine in a state declaration: these declarations can only include fully instantiated ECharts machines. Entry into a state with an included machine is modeled by allocating space for the parameters on the stack and assigning the argument values to the parameters.

UML Statecharts does not support parameterization except in the degenerative case of using the “include/” statement. This specifies that a nested state machine should be an instance of a (unparameterized) state machine, which is analogous to specifying a parameterless procedure call. Interestingly, the (admittedly informal) semantics of UML Statecharts permits a state machine to include itself, which results in an ill-formed state machine.

ROOMcharts supports its own distinct notion of parameterization. It is not possible to specify that an instance of a state machine be nested in a state. Instead, the ROOM modeling language permits nesting of parameterized “Actor” elements, which in turn, can contain state machines. By composing nested Actors, one can achieve something like parameterized state machines, although this approach is indirect and awkward.

2.2 Port Abstraction

In ECharts, messages are exchanged between a state machine and its environment via ports. A port enqueues messages arriving from the environment in FIFO order. A port message is dequeued when a transition specifying that port fires. The port abstraction provides the designer with the ability to logically partition the environment into different message sources. Like ROOMcharts, but unlike UML Statecharts, any number of ports can be defined for a state machine. UML Statecharts, on the other hand, defines only a single, implicit, port for a state machine.

2.3 Transition Priority Rules

Parameterized state machines and the port abstraction introduce the possibility of building a machine to implement a uniform behavior that is needed in distinct contexts. However, many programs can be implemented by small changes on top of a general solution. We incorporate transition priority rules in ECharts to allow us to both specify and override default behaviors, as well as reduce non-determinism.

To specify default behavior, it is assumed that a class hierarchy is defined over the message instances arriving from the environment. When two transitions are enabled for a message arriving at a port, then the transition specifying the most specific class of which the message is an instance will have a higher priority. For example, consider the ECharts fragment in Figure 1. The “Teardown” message class is a subclass of the general “DFCProtocol” message class. If a message arrives on the caller or callee port whose message class is not the “Teardown” message class (or one of its descendants), but whose message class is an descendant of the “DFCProtocol” message class, then a “DFCProtocol” transition will fire.

Although a message class hierarchy is supported in UML Statecharts, this notion of transition priority is not supported. ROOMcharts supports message types but not a type hierarchy, so this notion of transition cannot be supported. However, ROOMcharts does support the notion of message priority, so that high priority messages are dequeued from a port’s message queue before low priority messages are dequeued. This priority rule is evaluable only at run-time, since it is resolved based on the currently enqueued messages at a port. In contrast, our priority rule is statically evaluable, making it amenable to automated static analysis.

Another transition priority rule is illustrated by the sample ECharts fragment in Figure 2 at state s. In the event that a “Teardown” message (or descendant) arrives, the transition message classes of \( t_1 \) and \( t_2 \) are identical, so the previous rule does not distin-
guish between them. Instead, the transition with the more deeply nested source state takes priority: in our example, transition $t_1$ fires. The notion of nesting is generalized to the notion of “coverage” for transitions with more than one source state (see [10]). With this priority rule, it is possible to specify default behavior over a nested state machine and override this behavior for individual nested states. Both ROOMcharts and UML Statecharts support this notion of transition priority.

Another priority rule that permits overriding behavior can be understood in terms of the ECharts fragment shown in Figure 3. Transitions $t_1$ and $t_2$ from state $s_2$ have equal priority under the previous two priority rules. To distinguish between transitions in this situation we have a priority rule that gives priority to the transition defined at the higher state machine context. In this example, $t_1$ will have higher priority than $t_2$ since $t_1$ is defined in the state machine that includes states $s_1$ and $s_3$, whereas $t_2$ is defined in the state machine that is nested in $s_1$. This priority rule has proven to be useful when a parent state machine wants to override a transition defined in a nested state machine, where the nested machine is a general purpose, reusable state machine fragment. There is no such priority rule defined in either ROOMcharts or UML Statecharts.

It is possible to lexicographically order the transitions of an ECharts machine according to their relative priorities. This means that the relative priorities can be determined by inspection, or by static analysis at compile time rather than at run-time. We use static analysis as the basis of our translator to the Promela modeling language to enable model-checking ECharts.

### 2.4 Dynamic Creation of Concurrent State Machines

In real-time programming, it is often the case that only an upper-bound is known on the number of incoming events that will require responses. In such a situation it is often preferable to dynamically commit the resources required to respond to the event when it occurs, rather than committing the resources ahead of time based on a worst-case upper bound. To address this need, we introduce a new pseudostate that dynamically creates new concurrent state machine instances. The ECharts fragment shown in Figure 4 provides an example. The transition fires in response to the arrival of a “Setup” message, indicating the initiation of a new internal call. The effect of transitioning to the target “Spawn” pseudostate is to create a new instance of the nested LineFSM state machine, which will handle the new incoming call. In the example, a maximum of 100 state machine instances can exist concurrently. This is indicated by the annotation following the “include/” statement. In general, there will already exist instances of the LineFSM when a new instance is created. The semantics of transitioning to the “Spawn” pseudostate is identical to that of transitioning to a “Deep History” pseudostate (see [6]) for any existing instances. This way the execution state of any existing instances is maintained when a new instance is created. An instance is removed when a parent transition fires from an instance’s terminal state. This is discussed in more detail below.

UML Statecharts has no analogous concept. ROOMcharts, on the other hand, supports a variation of this concept. Since ROOMcharts does not support concurrent state machines, it must resort to the dynamic creation of “Actors” containing state machines. This additional level of abstraction complicates the design required to achieve the desired goal.
2.5 Intra-Statechart Communication

ECharts does not permit using the UML Statecharts event broadcast mechanism for communication between components of a state machine. The broadcast semantics is complicated and does not translate to an efficient implementation. We have also chosen not to support the port-based ROOMcharts approach since it was judged to impair our ability to automatically analyze a state machine. Instead, we provide a simple mechanism analogous to passing values on a stack upon return from a procedure call. This permits a nested state machine to return data values to a parent machine. As an example, refer to the ECharts fragment shown in Figure 5. When the transition fires, the value of local variable $x$ defined in the nested state is accessible during the execution of the transition’s action. In this case the returned value is incremented by 1 and assigned to the parent’s local variable $y$.

This is a more restrictive, simpler form of intra-state communication than those supported by UML Statecharts and ROOMcharts. Communication is only possible between a nested machine and one of its descendant machines; however, we have not yet encountered a situation where it has proven inadequate for our needs. Its benefits are that it is easy to understand and is amenable to automated analysis.

2.6 Implicit Message Deferral

ECharts semantics dictates that a message may be dequeued from its port input queue only if there exists a transition specifying that port in the current state. Consider the example ECharts fragment shown in Figure 6. The only transition that may fire from state $s_1$ is the one that fires when a message instance of $C_1$ arrives on port $p_1$. If there exists a message enqueued for port $p_2$ while in this state then it is simply ignored and remains enqueued. Only when the current state is $s_2$ will messages enqueued for port $p_2$ be considered.

Note that the existence of a transition specifying a port is a necessary condition for dequeuing a port’s message, but not a sufficient condition. In the case where there are transitions specifying more than one port in the current state, and the port input queues are non-empty, then which port to dequeue a message from is chosen non-deterministically. The important aspect of this approach is that if no transitions are defined for a port in the current state, then no message will be dequeued from that port. This semantics permits a designer to defer handling messages from a particular port until entering a state in which transitions are defined for that port. This has proven to simplify a number of designs, since state machines do not have to account for all possible environmental stimuli in every state.

In contrast to our approach, UML Statecharts and ROOMcharts support explicit event deferral. This means that a designer is required to explicitly specify that a message be re-enqueued if it is dequeued in a state in which the design is unprepared to deal with the message. In general, this forces a designer to treat message arrival as an exceptional event, complicating design.

2.7 Explicit Message Consumption

Consider the ECharts fragment shown in Figure 7, where there is only a single transition whose source state $s$ is the current state of a machine. If the next message in $p$’s input queue is not an instance of $C$ then there are a number of possible outcomes, depending on how the semantics of event consumption is defined. ECharts has chosen to deem this state machine as ill-defined, and the ECharts runtime system will raise an exception upon encountering this situation. ECharts imposes a variation of the input-enabledness constraint on state machines: if a message is dequeued for a port then a transition must fire for that message. UML Statecharts and ROOMcharts take an alternative approach to handling this situation: if a message is dequeued and no transition is enabled for the message, then the message is discarded. This approach is called implicit event consumption. We chose an explicit event consumption model since we felt that the possibility of otherwise unintentionally discarding messages was too great.

2.8 Statechart Termination

In ECharts a nested machine terminates when it is in a terminal state and a parent transitions fires. A machine is defined to be in a terminal state when it is in a state with no outgoing transitions and the cur-
rent state of all its nested state machines are also in terminal states. When a machine terminates, its machine context no longer has to be maintained. In the context of dynamically created state machines, this is the mechanism that reduces the number of concurrent state machine instances.

In addition to this basic termination mechanism, we have found it desirable to be able to define a transition whose source state are a nested machine’s terminal states. Rather than requiring that a parent machine explicitly define transitions to each of a nested machine’s terminal states, we have introduced a ‘terminal’ pseudostate which represents all of a machine’s terminal states. As shown in Figure 8, it is only necessary for a parent machine to define a single transition, in this case from $s_1$’s terminal pseudostate to $s_3$, rather than defining separate transitions from both of $s_1$’s terminal states.

UML Statecharts uses the notions of ‘final state’ and implicitly generated ‘completion’ events to support statechart termination. A statechart must explicitly define a transition to a final state in order signify termination of a machine. When a machine transitions to a final state, a completion event is generated and placed at the front of the machine’s event queue. A parent may define a transition that is triggered on a completion event. ROOMcharts does not provide any special termination abstractions.

We feel the ECharts termination support is more flexible and expressive than that provided by UML Statecharts or ROOMcharts. An ECharts designer does not have to define transitions to final states from all of a machine’s terminal states since all the machine’s terminal states are implicitly represented by its terminal pseudostate. Furthermore, as shown in Figure 8, the semantics permit a parent machine to define a transition whose source is the terminal pseudostate of an individual state machine that is nested amongst other concurrent state machines. In the example figure, the transition will fire when $s_3$ is in either of its terminal states, regardless of $s_2$’s current state. This case cannot be expressed as succinctly in either UML Statecharts or ROOMcharts.

3 Conclusions

This paper provides a comparative overview of ECharts, a visual language for behavioral design and implementation which both extends and restricts UML Statecharts. The language’s features are rooted in practical requirements we have encountered during the development of an IP telecom platform. However, these requirements are not unique to our platform—they are shared by any complex reactive system. Existing behavioral specification languages, namely ROOMcharts and UML Statecharts, are not sufficiently expressive to capture simple designs and generate efficient implementations. We feel that ECharts strikes the right balance between design and implementation while maintaining the inherent advantages of a visual language.

Future work for the language includes adding support for statechart inheritance and genericity. We are also working on a second-generation program analysis tool based on the Spin model-checking tool [8].

Acknowledgments

The authors would like to thank the other members of the Building Box project at AT&T Labs – Research for their feedback during the development of ECharts: Eric Cheung, Andrew Forrest, Franjo Ivancic (University of Pennsylvania), Michael Jackson, Hal Purdy, Chris Ramming, Thomas Wahl (University of Texas at Austin), Xiaotao Wu (Columbia University) and Pamela Zave. The authors would also like to thank Nils Klarlund and Richard Trefler, also at AT&T Labs – Research, for their contributions to the development of the model checking tool.

References


