ECharts: From Lab to Production

Gregory W. Bond and Healfdene H. Goguen

AT&T Labs–Research
180 Park Ave., Florham Park, NJ 07932, USA
bond@research.att.com hhg@research.att.com

Abstract. This paper summarizes our experience with ECharts, a state machine language that played a key role in the design and implementation of AT&T CallVantage, a nationally deployed voice-over-IP service. The introduction outlines the history of the language, including the original design requirements for the language, our realization that existing languages did not meet the requirements, and the evolution of the language as the project moved from prototype to production software. The language features most relevant to modularity and efficient use of resources are presented, and the operational semantics is summarized. Finally, ongoing work on new language features and on using ECharts to model BoxTalk, a telecommunication service programming language, is discussed.

Key words: State machines, voice-over-IP, production environment, semantics

1 Introduction

In 1998 Jackson and Zave [1] proposed a virtual architecture to support modular composition of telecommunications services (for example, call waiting or three-way-calling) in a traditional telephony network. In the Spring of 1999, six researchers at AT&T Labs [2] began work on a project whose goal was to adapt the architecture to support multimedia communications over the Internet Protocol, including voice-over-IP, and to implement a proof-of-concept system.

As our intention was to demonstrate the feasibility of the architecture, we needed a language to implement example telecommunication services. State machines are commonly used in this application domain, since they provide a formal way of representing responses to inputs using limited historical information; they can be implemented efficiently; and static analysis and testing techniques for state machines are well developed. Additional requirements on the language included mechanisms for code reuse, scalability to large numbers of concurrently executing state machines, and that the language easily integrate with Java, the language chosen for the rest of the project.

* Tel: (973) 360-8469. Fax: (206) 333-0806.
1.1 The Shortcomings of UML Statecharts

Since the architecture dictated that individual services communicate via asynchronous message exchange, we initially considered adopting UML Statecharts [3], a standardized state machine design language, as our service description language. The benefits of UML Statecharts are that the language is visual, expressive, and that it is supported by commercial tools that perform design capture, (Java) code generation and code execution.

With regards to expressiveness, we were attracted by the language’s support for hierarchical state machines, concurrent (orthogonal) state machines, machine synchronization, and fork/join transitions. However, when we attempted to design a few non-trivial services with the language we encountered shortcomings:

- The language only supports a single transition priority rule for constraining transition execution order. As a result, many non-deterministic situations can exist in a model.
- The language only provides rudimentary support for re-using 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.
- While it is possible for a fixed number of concurrent state machines to exist statically within a state, it is not possible to create state machines dynamically. 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 unknown prior to runtime, the latter approach conserves runtime resources.
- Finally, we found the machine termination semantics to be overly complex, because the developer is required to explicitly specify a state as terminal which we considered to be redundant.

In defense of UML Statecharts, we recognize that most shortcomings exist because the language is intended to serve first as a design language, and second as an implementation language. However, our need was for a language that served first as an implementation language and second as a design language, so we concluded that UML Statecharts did not suit our needs [4].

1.2 Addressing Implementation Issues with ECharts

To address these shortcomings while maintaining the desirable properties of UML Statecharts, we developed a Statecharts dialect called ECharts. Wherever possible, the ECharts language adheres to the UML Statecharts standard. Specifically, we consider that the introduction of a port abstraction, additional transition priority rules, enhanced support for Statechart reuse, support for dynamic creation of concurrent state machines, and a refined termination semantics can be viewed as refinements or extensions to the UML Statecharts semantics.
On the other hand, 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. The explicit event consumption model ensures that events aren’t ignored unless the programmer explicitly declares them to be ignored. Semantic aspects of ECharts will be discussed in more detail later in the paper.

From 1999 to 2003, the ECharts language was used by five researchers to implement various experimental telecommunication services for the project. Initially the language supported only parameterized hierarchical state machines and concurrent machines since these features were adequate for our needs. Other features such as dynamic machines and sync states were added later only as the demand for them arose.

1.3 Moving ECharts to Production

In 2003 our project was chosen by AT&T as the basis for a consumer-oriented voice-over-IP product that is currently marketed in the U.S. as AT&T CallVantage [5]. Overnight, our project, and by implication the ECharts language, became very important. The number of people working directly on the project went from 5 to approximately 50. Within our project code repository there are now approximately 300 ECharts machines comprising approximately 23K lines of ECharts code.

The AT&T CallVantage service offers an innovative set of features and was instrumental in pushing the voice-over-IP industry to compete partly on the basis of features. In large part this is due to the pipes-and-filters understanding of telephony features as embodied in Jackson and Zave’s architecture. However, the design of the ECharts language also contributed to our ability to produce production-quality feature code under tight project deadlines. In several cases, vastly different features used common machines, with behavior changes achieved using transition priority rules and parameterization. This reuse reduced the effort of design and coding, and also reduced the number of errors encountered in the test and production environments where changes can be extremely expensive.

Performance and scalability were the major challenges we faced moving the language from the lab to a production environment. Early on, the language interpreter performance and scalability were deemed to be unacceptable to support a nation-wide service. As a result, the interpreter has undergone two major re-writes since 2003.

The first re-write significantly decreased memory utilization and garbage collection overhead by utilizing static object storage wherever possible. The translator was improved to support object access via indexed arrays rather than via sequential list lookup or hash table access. The second re-write significantly decreased CPU utilization by optimizing the algorithms used to search for candidate transitions to fire. Optimizations included caching search values and en-
coding the machine hierarchy in order to optimize machine parent-descendent query times [6].

The result of these improvements has been an overall order-of-magnitude performance improvement and major scalability improvements. These improvements permit tens of thousands of concurrent machines to run under the control of a single thread of execution in a soft real-time environment. Many such threads are employed over a number of servers to support the AT&T CallVantage service.

Apart from the performance and scalability improvements, the language was intentionally frozen for two years in order to minimize the risk of disrupting the project. During this period, however, we encouraged developers to let us know how they thought the language could be improved. Based on this feedback, we have started work on a next-generation version of the language incorporating these suggestions. This version also incorporates features demanded by new uses we have discovered for the language. We will discuss this version of the language in our closing comments.

1.4 The General Utility of ECharts

The popularity of UML Statecharts is a testament to the utility of Statecharts-based languages. ECharts builds upon this utility in three ways:

– it provides features supporting simplified or new control over a machine’s behavior, e.g. transition priorities, inter-/intra-machine communication semantics, explicit event consumption, and simplified termination semantics;
– it provides features supporting altogether new classes of behavior e.g. dynamic machines;
– it provides features supporting enhanced re-usability e.g. parameterized machines.

For these reasons, our expectation is that ECharts will be useful outside of AT&T for general-purpose asynchronous reactive systems programming. To this end, we are pleased to announce that we have obtained permission from AT&T to release the language as open source under the Common Public License.

2 Language Overview

The ECharts language, like UML Statecharts, possesses both a graphical syntax and a textual syntax. The graphical syntax for the current version of ECharts is identical to that of UML Statecharts. The ECharts textual syntax, however, is significantly different from that of UML Statecharts. UML Statecharts uses a dense XML dialect intended for storage, retrieval and exchange of Statecharts designs; it is not intended for human composition. Instead, Statecharts developers are expected to use a graphical editor for composition.

In contrast, ECharts uses a simple textual syntax intended to support human composition. We chose this approach because it represented less initial development, changes in language design were less costly because we did not need to
maintain a sophisticated graphical editor for all language features, and we believed that the language would be more likely to be adopted since text is easier for programmers to read and develop using their favorite editors.

Here is an example of an ECharts program in the textual representation.

```plaintext
fsm UnavailFSM(BoxPort boxPort, CalleePort calleePort)
{
    initial START,
    END: CalleeTeardown FSM(calleePort);
    START = boxPort?Setup /
        calleePort!Upack(thisMessage),
        calleePort!Unavail
        -> END
}
```

Although a graphical editor has not been developed for ECharts, we have developed a tool that translates textual ECharts programs to their graphical representation using the graphviz graph layout tools [7] developed at AT&T (available as open source). As an example, Figure 1 depicts the graphical representation produced by our translator of the example program given above.

![Graphical representation using graphviz](image)

**Fig. 1.** Graphical representation using graphviz

Like most commercial Statecharts products, ECharts is a meta-language: the language only supports the notions of states and state transitions, and not notions of data types or objects. For this reason it is intended to be used in conjunction with a host language. The ECharts language therefore provides a construct for embedding native code. The ECharts language is, however, independent of the native language, meaning that it is possible for ECharts to be used with different host languages. Currently, we have a tool that translates textual ECharts programs to Java programs.

Since we have not yet developed a symbolic debugger for the language, we rely on reviewing detailed logs of machine execution output by the ECharts interpreter in debugging mode. A tool is used to parse the logs and present a more abstract view of machine behavior to developers.
In the remainder of this section, we discuss language features that address the shortcomings mentioned in the introduction. Some of these language features will be given semantics in the following section.

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:

\[
\text{fsm Chart}(T_1 a_1, \ldots, T_n a_n) \\
\{ \\
\text{body} \\
\}
\]

where \text{Chart} is an identifier, \(T_1\) to \(T_n\) are types and \(a_1\) to \(a_n\) are formal parameters. In order to prevent infinitely recursive charts, we prevent \text{Chart} from occurring within the body of the declaration, or the bodies of any nested machines.

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

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

where \text{Chart} is again an identifier, and \(v_1\) to \(v_n\) 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. The state \text{END} in Figure 1 gives an example of an included machine. 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.

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. Any number of ports can be defined for a state machine: the port abstraction provides the designer with the ability to logically partition the environment into different message sources.

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 more specific class of which the message is an instance will have a higher priority. For example, consider the ECharts fragment in Figure 2. 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 a descendant of the “DFCProtocol” message class, then a “DFCProtocol” transition will fire.

Fig. 2. Specifying default behavior

Another transition priority rule is illustrated by the sample ECharts fragment in Figure 3 at state $s$. In the event that a “Teardown” message (or child) arrives, the transition message classes of $t_1$ and $t_2$ are identical, so the previous rule does not distinguish 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 [8]). With this priority rule, it is possible to specify default behavior over a nested state machine and override this behavior for individual nested states.

Another priority rule that permits overriding behavior can be understood in terms of the ECharts fragment shown in Figure 4. 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
Fig. 3. Overriding behavior (1)

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.

Fig. 4. Overriding behavior (2)

As we shall see in the overview of the semantics of the language, 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.

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 5 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, denoted by an “S” enclosed in a circle in the example, 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 [9]) 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.

![Diagram](image)

**Fig. 5.** Dynamically created state machines

### 2.5 Statechart Termination

In ECharts a nested machine terminates when it is in a terminal state and a parent transition fires. A machine is defined to be in a terminal state when it is in a state with no outgoing transitions and the current 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 transitions whose source states are a nested machine’s terminal states. Rather than requiring that a parent machine explicitly define transitions from 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 6, 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.
3 Language Semantics

Work on the operational semantics for ECharts is on-going. To date we have formalized the semantics for most features in the current version of the language. Due to space limitations we are unable to present a complete description of the semantics of the language. Instead we have chosen to present a semantic “highlight”: the semantics of transition priorities.

3.1 ECharts Transitions

An ECharts transition specifies a source state configuration and a destination state configuration. The source state configuration must be satisfied by a machine’s current state configuration for the transition to be enabled. If the transition fires, then the next current state configuration must satisfy the destination state configuration. In general, an ECharts state configuration can span concurrent states (and-machines) corresponding to “fork” and “join” transitions. Following a variation of [8], we represent a state configuration as a vector of terms.

Let StateName denote a countable set of ECharts state names, ranged over by $n$, and let $\mathcal{V}$ represent a countable set of variable names disjoint from StateName, ranged over by $X$. Let StateCfg denote the smallest set of linear terms vectors satisfying the following conditions:

1. $\mathcal{V} \subseteq \text{StateCfg}$,
2. $() \in \text{StateCfg}$,
3. $(n_1(L_1), \ldots, n_k(L_k)) \in \text{StateCfg}, k > 0$, if for all $1 \leq i \leq k$, $n_i \in \text{StateName}$, and $L_i \in \text{StateCfg}$,
4. $\forall L \in \text{StateCfg}$, for each variable $X$ in $L$, $X$ may only occur once in $L$.

In this context, a variable $X$ represents a term vector, not an individual term. We let $L$ range over StateCfg.

A substitution, ranged over by $\rho$, specifies the replacement of a variable in a term vector with another term vector. If $\rho = [X_1 \mapsto L_1, X_2 \mapsto L_2, \ldots]$, we use the notation $L_\rho$ to denote the vector $L$ whose variable $X_1$ is replaced with vector $L_1$, and whose variable $X_2$ is replaced with the vector $L_2$, and so on. For two term vectors $L_1, L_2 \in \text{StateCfg}$, $L_1$ covers $L_2$ if there exists a substitution $\rho$ such that $L_1\rho = L_2$. 

Fig. 6. Statechart termination
The set of ECharts transitions, denoted by Transition, consists of two disjoint sets: message transitions, and messageless transitions, denoted by MsgTransition and MsglessTransition, respectively. An ECharts message transition declares a message specification. In order for a message transition to fire, its message specification must be satisfied. An ECharts messageless transition does not declare a message specification.

**Message Transitions** Let Port represent a countable set of port names, ranged over by $p$, and Class represent a countable set of message class names, ranged over by $c$. Let $\triangleright=\subseteq\text{Class} \times \text{Class}$ denote the class ancestor relation, where $c \triangleright c'$ expresses $c$ is a subclass of $c'$.

An ECharts message transition is represented by a tuple $(L_s, p?c, L_d)$. Intuitively, $(L_s, p?c, L_d) \in \text{MsgTransition}$ expresses that an ECharts machine whose current state configuration satisfies the source state configuration $L_s$ may evolve to a state configuration satisfying the destination state configuration $L_d$ if the first element in port $p$’s input queue is $m$, and the class $c'$ of which $m$ is an instance is a subclass of $c$ i.e. $c' \triangleright c$.

Figure 7 shows an example message transition defined in machine $S3$ specified using the graphical syntax. The corresponding element of $\text{MsgTransition}$ is

$$((S4(X)), p?c, (S5(X)))$$

![Fig. 7. An Example Message Transition](image)

**Messageless Transitions** An ECharts messageless transition is represented by a tuple $(L_s, L_d)$, where $(L_s, L_d) \in \text{MsglessTransition}$ expresses that an ECharts machine whose current state configuration satisfies the source state configuration $L_s$ may evolve to a state configuration satisfying the destination state configuration $L_d$.

### 3.2 ECharts LTS Actions

We define the operational semantics of ECharts in terms of a labeled transition system [10]. In the context of ECharts, the set of LTS actions represents the
universe of ECharts transition conditions e.g. a transition’s source state and any guard condition.

In general, an ECharts transition is declared in a machine that is nested within another machine. The state configuration \( L \) of an LTS action represents the ECharts transition’s source state configuration \( L_s \) “lifted” up to the context of the top-most machine it is nested within. When the declaring machine is the top-most machine then \( L = L_s \). The integer \( v \) of an LTS action denotes the nesting level of the machine that the ECharts transition is declared in relative to the top-most machine. The nesting level of the top-most level is defined to be 0.

Space does not permit a formal definition of the lifting relation; however, as an example, consider again the transition shown in Figure 7. The top-level machine is \( S_1 \). The transition is defined in machine \( S_3 \) at level \( v = 1 \). Lifting the source state configuration of this transition to the level of the top-level machine gives us the state configuration \((S_3(S_4(X)))\).

The set of LTS actions associated with message transitions is defined by the set

\[
\text{MsgAct} = \{(L, p?c, v) \mid L \in \text{StateCfg}, p \in \text{Port}, c \in \text{Class} \text{ and } v \in \mathbb{N}\}.
\]

An element \((L, p?c, v) \in \text{MsgAct}\) is associated with an ECharts message transition \((L_s, p?c, L_d) \in \text{MsgTransition}\) as described above.

The set of LTS actions associated with messageless transitions is defined by the set

\[
\text{MsglessAct} = \{(L, v) \mid L \in \text{StateCfg} \text{ and } v \in \mathbb{N}\}.
\]

An element \((L, v) \in \text{MsglessAct}\) is associated with an ECharts messageless transition \((L_s, L_d) \in \text{MsglessTransition}\) as described above.

### 3.3 ECharts Transition Priorities

We now introduce two binary relations: \(\succeq^+ \subseteq \text{MsgAct} \times \text{MsgAct}\) and \(\succeq^- \subseteq \text{MsglessAct} \times \text{MsglessAct}\). The first relation imparts ECharts message transitions with priority relative to other message transitions. The second relation imparts ECharts messageless transitions with priority relative to other messageless transitions.

The relation \(\succeq^+\) is defined as a lexicographic order over the partial orders \(\succeq_1, \succeq_2, \succeq_3\), each of which ranges over \(\text{MsgAct} \times \text{MsgAct}\). \(a \succeq^+ b\) if:

1. \(a \succeq_1 b\), or
2. \(a =_1 b\) and \(a \succeq_2 b\), or
3. \(a =_1 b\), \(a =_2 b\) and \(a \succeq_3 b\), or

Let \(a = (L_a, p_a?c_a, v_a)\) and \(b = (L_b, p_b?c_b, v_b)\).

1. \(a \succeq_1 b\) if \(c_a \succeq c_b\). Intuitively, this states that a transition \(a\) referring to a message with a more specific message class \(c_a\) has higher priority than a transition \(b\) referring to a message with a more general (super-)class \(c_b\).
2. \( a \succeq_2 b \) if \( L_b \) covers \( L_a \). Intuitively, this states that a transition \( a \) referring to more specific source states \( L_a \) has higher priority than a transition \( b \) referring to more general states \( L_b \).

3. \( a \succeq_3 b \) if \( v_a \leq v_b \). Intuitively, this states that a transition \( a \) defined at a higher level in the state machine hierarchy \( v_a \) has a higher priority than a transition \( b \) defined at a lower level \( v_b \).

Examples of these rules were included earlier in the paper. The relation \( \succeq_2 \) corresponds to the only transition priority rule defined for UML Statecharts [3]. Expressing this rule in terms of term coverage is an idea introduced in [8].

The relation \( \succeq^- \) for messageless transitions is defined similarly to \( \succeq^+ \) except that only the second and third partial orders are used in the lexicographic order defining \( \succeq^- \).

4 Current and Future Work

While work on the original project continues, we have embarked on a new project whose goal is to realize a domain-specific telecom service programming language, known as BoxTalk, proposed by Zave and Jackson [11]. Unlike ECharts, which is a general purpose state machine-based language, BoxTalk explicitly supports concepts specific to telecom service programming such as notions of “calls” and “media.” Given that BoxTalk is also a state machine-based language, our approach to realizing this language is to translate BoxTalk programs to ECharts programs. Since the BoxTalk language semantics has only been informally specified, our approach has the added benefit of providing BoxTalk with a formal operational semantics.

Many of the features of ECharts developed for reuse are helpful when using the language to define a formal semantics. The dynamic creation of concurrent machines allows variables representing call state in BoxTalk to be modeled using state machines in ECharts. Parameterized machines allow common behaviors for calls to be captured in a single machine parameterized over the signal port and media ports for the call. Inter-level transitions and transition priority rules allow the core BoxTalk state machine to be translated to a very similar ECharts state machine, with transitions in the BoxTalk program translated to similar transitions extended with a vector of transitions for the concurrent call states.

As we mentioned in the introductory comments, we have started work on a new version of the ECharts language that incorporates refinements and features suggested by developers. A number of new features have also been added to support our BoxTalk translation effort.

- We have discarded the original approach to intra-machine communications based on a return-value semantics, and instead support a more general approach where machine variable values can be inspected by ancestor machines in transition guards or actions.
- We have extended the dynamic machine model to support referencing individual dynamic machine instances via a unique index assigned to each dynamic machine.
We have generalized transitions in or-machines so that they are able to reference states other than the machine’s current source or future target state.

- We have further simplified terminal state semantics.
- We have added a new facility for restricted asynchronous communications between ancestor and descendant machines.
- We have rationalized the approach to data exchange between ECharts and its host language.
- We have added an exception handling facility which complements the host language exception handling facility.

A future goal for the ECharts language is to develop a tool to perform high-level model-checking. Because of its simplicity, UML Statecharts semantics lends itself to automated static analysis, e.g., [12]. In fact, we developed a model-checker for an early version of ECharts that consisted of a translator [13] that output Reactive Module (RM) code for analysis using the Mocha model checking tool [14]. Now that the ECharts language has matured and stabilized we intend to revisit this problem.

Currently developers modify the behavior of existing ECharts machines by wrapping them in machines and using the transition priority rules to override nested transitions. A more general approach would be to support machine inheritance. Machine inheritance is supported in the new UML version 2.0 Statecharts standard and we are investigating support for this in the future.

5 Conclusion

ECharts was designed as a modular, scalable language for telecommunication service programmers. The language allowed us and our colleagues to develop reusable components quickly, and a significant amount of code written in ECharts was included in AT&T CallVantage, a nationally available voice-over-IP service. Many language features have been given a formal operational semantics. We have also devoted significant attention to optimizing the language’s performance. As such, the language should be useful in a wide range of applications outside of telecommunications, and we have obtained permission from AT&T to release the language as open source under the Common Public License.

References


