Cover Feature

Executable Object Modeling with Statecharts

Statecharts, popular for modeling system behavior in the structural analysis paradigm, are part of a fully executable language set for modeling object-oriented systems. The languages form the core of the emerging Unified Modeling Language.

David Harel
The Weizmann Institute of Science and i-Logix

Eran Gery
i-Logix

Models for the development of object-oriented systems should be behaviorally expressive and rigorous as well as intuitive and well structured. Thus, any modeling approach must be detailed and precise enough to produce fully executable models and permit the automatic synthesis of efficient code in languages such as C++.

Most OO modeling methodologies specify a model through graphical notations. Entity-relationship-like diagrams typically specify object classes and their interrelationships, and there is some way to describe what objects do and how they interact. Most methodologies also adopt a state-based formalism to specify behavior, using statecharts[1] or some sublanguage thereof.

However, many methodologies fail to rigorously define the semantics of the languages. Without a rigorous semantic definition, precise model behavior over time is not well defined and full executability and automatic code synthesis is impossible. Adopting a richly expressive behavioral language like statecharts makes modeling easier, but requires great care in defining the way it integrates with the other parts of the model. Statecharts must capture not only the state of the object as a precondition to service requests, but also the dynamics of the object's internal behavior in responding to those requests and in maintaining relationships with other objects.

These issues are complicated and go beyond recommending a modeling approach or methodology—they are language design concerns, requiring rigorous mathematical underpinnings. Both syntax and semantics must be fully worked out: any possible combination of constructs must be clearly characterized as syntactically legal or illegal, and each legal combination must be given a unique and formal meaning.

To address these needs, we embarked on an effort to develop an integrated set of diagrammatic languages for object modeling, built around statecharts, and to construct a supporting tool that produces a fully executable model and allows automatic code synthesis. The language set includes two *constructive* modeling languages (languages containing the information needed to execute the model or translate it into executable code).

- *Object-model diagrams* specify system structure by identifying object classes and their multiplicities, object relationships and roles, and subclassing relationships.
- *Statecharts* describe system behavior. A statechart attached to a class specifies all behavioral aspects of the objects in that class.

In addition, we support *message sequence charts*, also called sequence diagrams, as a *reflective* language (which captures parts of the thinking that goes into building the model—behavior included—or is used to derive and present views of the model to aid analysis). Message sequence charts describe the possible ways a system behaves in terms of scenarios or use cases.[2]

Although we focus on the language set in this article, we believe our supporting tool, Rhapsody, is a critical element of our work. Rhapsody demonstrates we have a fully worked out behavioral semantics, allowing our models to be fully executable and enabling full code synthesis. The sidebar "From Structural Analysis to Object Orientation" describes some issues we faced in applying statecharts to the OO paradigm.

Our language set constitutes the core part of the Unified Modeling Language, a flexible, general-purpose modeling methodology (http://www.rational.com/uml). UML unifies three popular approaches

0018-9162/97/$10.00 © 1997 IEEE

to OO modeling—the Booch method,[3] OMT,[4] and OOSE[2]—with statecharts as its heart. It thus seems positioned to become the Object Management Group's standard modeling language. Partly as a result of our recent collaboration with the UML team, our language set and Rhapsody are consistent with UML. For example, the semantics of composite objects in UML are defined as we define them in this article.

Both the language set and Rhapsody deal with active objects and multiple-thread concurrency, but we have decided to focus on single-thread concurrency here. We have also kept the presentation of both the syntax and semantics informal, and our descriptions are not exhaustive. A complete definition of both is in preparation, including the way Rhapsody translates any syntactically legal model into executable code.

Our current implementation framework is based on C++, which is natural given its status in the OO language community. However, this is more a matter of convenience, so that models contain actions and operations written directly in the implementation language. This, in turn, makes it relatively easy to plug in a framework based on another language, such as Ada, Smalltalk, Java, or even on a set-based language.[5] However, what programming language is chosen as the implementation framework has little bearing on our modeling and analysis approach. Rhapsody supports the modeling process in its entirety, so once we chose C++ for our initial implementation, it became natural to use it for the detail level of the model, too.

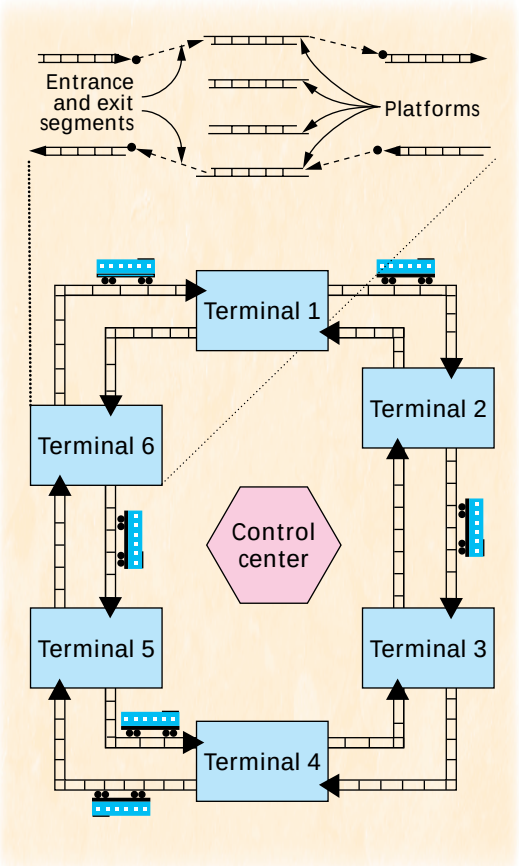

Figure 1. A railcar system. The enlargement shows the structure of each terminal.

RAILCAR SYSTEM

To explain the properties of our language set, we use the automated rail-car system in Figure 1, inspired by Vered Gafni. Six terminals are located on a cyclic path. Each pair of adjacent terminals is connected by two rail tracks, one for clockwise and one for counterclockwise travel. Several railcars are available to transport passengers between terminals. A control center receives, processes, and sends system data to various components.

As the enlargement of Terminal 6 shows, each terminal has a parking area containing four parallel platforms. Each platform can hold a single car. The four rail tracks (two incoming and two outgoing) are connected to a rail segment that can link to any one of the four platforms.

The terminal has a destination board for passenger use (not shown), containing a push button and indicator for each destination terminal. Each car is equipped with an engine and a cruise-controller for maintaining speed. The cruiser can be off, engaged, or disengaged. The car is to maintain maximum speed as long as it never comes within 80 yards of any other car. A stopped car will continue its travel only if the smallest distance to any other car is at least 100 yards. A car also has its own destination board, similar to the one in the terminal. The control center communicates with various system components—receiving, processing, and providing system data.

Three possible scenarios, or use cases,[2] stated as customer requirements, are

- *Car approaching terminal.* When the car is 100 yards from the terminal, the system allocates it a platform and an entrance segment, which connects it to the incoming track. If the car is to pass through without stopping, the system also allocates it an exit segment. If the allocation is not completed within 80 yards from the terminal, the system delays the car until all is ready.
- *Car departing terminal.* A car departs the terminal after being parked for 90 seconds. The system connects the platform to the outgoing track via the exit segment, engages the car's engine, and turns off the destination indicators on the terminal destination board. The car can then depart unless it is within 100 yards of another car; if so, the system delays departure.
- *Passenger in terminal.* A passenger in a terminal wishes to travel to some destination terminal, and

there is no available car in the terminal traveling in the right direction. The passenger pushes the destination button and waits until a car arrives. If the terminal contains an idle car, the system will assign it to that destination. If not, the system will send a car in from some other terminal. The system indicates that a car is available with a flashing sign on the destination board.

We use message sequence charts to describe such scenarios. They are especially good for describing collaborations, since one chart typically includes several objects. Using Rhapsody, system modelers can check the consistency of a chart against the model itself, provided they have used statecharts to specify model behavior (described later). Message sequence charts are also an appealing reflective formalism. For example, modelers can set up a chart to show the animated progress of interobject communication during model execution.

OBJECT-MODEL DIAGRAMS

Object-model diagrams specify a system's classes and their structural relationships. Conceptually, there is one diagram per system. However, a modeler will typically construct and view that diagram in several parts. The object-model diagram uses the same language for classes and instances.

An object-model diagram is similar to an entity-relationship diagram viewed as an object model except that it is hierarchical and features *higraph encapsulation* [6] (boxes inside other boxes), which denotes a strong composite class aggregation. Directed edges represent relationships; an undirected edge is the same as a two-way directed edge. Classes and relationships can have typical kinds of multiplicity information. For consistency with UML and other class structure notations, object-model diagrams also allow a weaker kind of aggregation—`part-of`, a special association relationship between the aggregate and its components, depicted with a diamond icon[3,4]—which our examples here do not include.

Railcar system

Figure 2a shows a partial object-model diagram for the railcar system with four main classes (two of which, `Car` and `Terminal`, are composites). The numbers in the boxes indicate multiplicity information (instances): one `ControlCenter` and six `Terminal`s. The other classes contain an asterisk, which means they can have unlimited instances. The figure also shows four many-to-one bidirectional association relationships and two unidirectional ones. When there is no relationship name or role, the instances refer to their relatives by `its`. Thus, a passenger can refer to `itsTerminal` and a terminal will have a set of `itsPassenger`s. Edges that have a role name are referred to differently; a `Car` can refer to the set of terminals it `stopsAt`, which could be different from the set of `itsTerminal`s. Directionality dictates the ability to reference instances: In Figure 2, a terminal cannot refer to `itsCar`s, for example.

To make referral along relationship links easy, we allow standard kinds of *navigation expressions*. For example, the navigation expression `Passenger -> itsCar -> stopsAt` refers to the set of terminals at which the car carrying the passenger is scheduled to stop. Also, because `System` always refers to an explicit composite object that encloses the entire model, the six `Terminal`s can be referred to from the top level of the model by `System -> itsTerminal[1:6]`.

Figure 2b shows the six components of the composite `Terminal`, and the four components of `Car`.

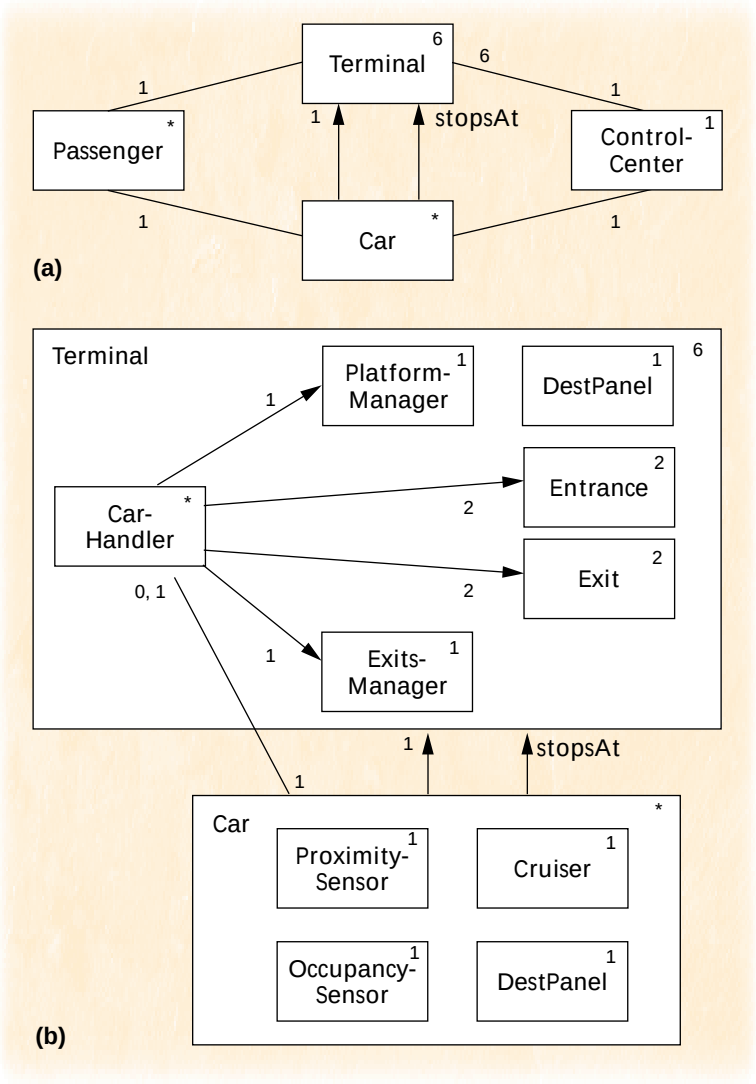

Figure 2. (a) High-level object-model diagram for the railcar system, and (b) more detailed diagram of the composites `Terminal` and `Car`.

An object model diagram also contains information that helps initialize the system's dynamic behavior.

The Entrance and Exit objects are software drivers for the relevant rail segments. PlatformManager and ExitsManager allocate platforms and exits to CarHandler. In contrast to the other classes, CarHandler is a concept that does not come from the problem domain, but is introduced by a domain expert during modeling. It handles the transactions between a Car and a Terminal. A special CarHandler is created whenever a car approaches a terminal; it is destroyed when the car departs. As such, it serves as a proxy object for the car within the terminal. The four components within Car have no links, since they do not collaborate. The primitive object ProximitySensor, for example, sends events to the Car's behavior (to its statechart) on the basis of the distance to the approached Terminal.

A composite class can refer to its components directly. We also allow direct links (and hence direct communication) between a component object and objects outside its composite parent. Figure 2a does not show the relationship between Car and CarHandler, but the more detailed Figure 2b does. We could have represented this relationship in Figure 2a as well by having an edge leading from Car to a stubbed end lying within the interior of Terminal.

Initialization

Because an object-model diagram describes classes and their structure, it appears to be concerned with static aspects only. However, it also contains information that helps initialize the system's dynamic behavior.

Defining behavioral semantics specifically enough to enable model execution and full code synthesis hinges on two things: initialization and dynamics over time. *Initialization* is the way the model starts out—which object instances are constructed at the start and how their attributes and relationships with other objects are set up. *Dynamics* concerns the way the model behaves when running. The model's status can change because of changes in the data item values, in the object's state (caused by triggers like events and operation invocation), and in the system's structure (such as the instantiation and deletion of objects and the establishment and modification of links among them).

The main tasks in initialization are to set up the initial composite structures, define associations, and supply initialization scripts where necessary.

Composite structures. When a model starts executing, its composite structures are initialized by the creation of instances from classes with fixed integer multiplicities. Creation happens recursively down the tree of composites (from a composite class to its components), and new instances are created for each instance of a composite class. Thus, in Figure 2a, one ControlCenter and six Terminals are created at the start. In Figure 2b, within each Terminal, two Entrances, two Exits, one PlatformManager, one ExitsManager and one DestPanel are created at the start. Components with unspecified multiplicity (those with asterisks) will create no instances spontaneously, so no Cars or CarHandlers, for example, are created at the start, unless a default initialization script is present, as we explain below.

This feature of composite classes remains valid beyond initialization, extending throughout the model's dynamic behavior: Whenever an instance of a composite class is created, the appropriate instances of the components with fixed multiplicities are created. (If multiplicities are specified by integer variables, the current values are used.) Similarly, when the composite is destroyed, so are all its components.

Associations. Another important part of initialization is setting up relationships, or defining how instances collaborate. If we consider associations as mechanisms to bootstrap the model, we can categorize them as unambiguous, ambiguous but bounded, and unworkable. An example of an *unambiguous* association is the link between Terminal and ControlCenter in Figure 2a. It is many-to-one, but the multiplicities on either end of the edge match those of the associated classes, both in the number of actual links and in the identity of the linked instances. Consequently, all six Terminals start out being associated with the ControlCenter: They can refer to itsControlCenter, which, in turn, can refer to all itsTerminals.

If there was no "6" at the end of the edge, the association would be *ambiguous but bounded;* any subset of the six Terminals could be associated with the unique ControlCenter but there would still be a well-defined upper bound (all six connected) and a well-defined lower bound (none connected).

An example of an *unworkable* association is the left-hand one between Car and Terminal in Figure 2a: a Terminal must be associated with each Car, but there is no information to set up this association.

In the ambiguous but bounded associations, we took a greedy approach to semantics, although Rhapsody users have some flexibility in the degree of greed. The greedy approach sets things up using canonical mappings. For example, if each class has n instances in a symmetric one-to-one relationship, the greedy approach will associate them in matching pairs, $A(i)$ to $B(i)$, for each $1 \le i \le n$. If the relationship is reflexive ($B=A$), the greedy approach matches them in cyclic order, $A(i)$ to $A(i+1)$, with $A(n+1)$ identified with $A(1)$.

The rules for setting up relationships also extend to dynamics. As the model runs, whenever objects are instantiated or destroyed, all relevant associations are reevaluated and set up as just described. Thus, for exam-

ple, if an instance of `CarHandler` is somehow created, it will be able to refer to `itsPlatformManager` and `itsExitsManager`, since those links will have become unambiguous. Also, since the multiple links to `Entrance` and `Exit` will also have become unambiguous, it should be able to refer to `itsEntrance[1]` and `itsEntrance[2]`; likewise for `Exit`s. The current version of Rhapsody does not support this dynamic reevaluation, but a future version will.

Changes may also occur in the current set of instances and their links, such as when objects are created or destroyed. These can be prescribed by several kinds of actions that can appear in statecharts, as we describe later.

Initialization scripts. For initialization to be complete, the modeler must often supply additional information. Here, we have left the number and location of `Car`s unspecified. We could have provided a multiplicity specification for `Car` in the object-model diagram, say 12, but we would have still had to resolve the ambiguous links with the six `Terminal`s—for example, where is the car located initially? (that is, what is `itsTerminal`?). Instead, we provide the implicit top-level `System` object with an *initialization script*, which is carried out once at the start. Each object may have an initialization script in its statechart, as we describe later. Here is the script for the rail-car system, decreeing the creation of 12 new `Car`s, located in adjacent pairs in the six `Terminals`:

```
for (int car = 0; car < 6; car++)
 {System -> itsCar[2*car] =
    new Car(System ->
     itsTerminal[car]);
 System -> itsCar[2*car+1] =
    new Car(System ->
     itsTerminal[car])}
```

STATECHARTS

Statecharts describe both how objects communicate and collaborate and how they carry out their own internal behavior. They must also reflect important OO issues like inheritance.

Object communication and collaboration

One of the main technical issues we faced was what mechanism to use for interobject interaction. High-level mechanisms, like events, are easier to model and are therefore more appropriate for analysis; lower level ones, like operations, are easier to translate into efficient code and are therefore closer to design.[5] In an attempt to compromise, we adopted the two mechanisms: An object can *generate an event*, which is queued, to be handed to the target server object in its turn. An object can also directly *invoke an operation* of another object, causing it to carry out an appropriate method, and perhaps return a value.

Event generation. An object—the *client*—can generate an event and address it to some other object, the *server*. To refer to the server, the client can use a

From Structural Analysis to Object Orientation

Statecharts were developed by David Harel in 1983 as a visual formalism for specifying raw reactive behavior. The next step was to integrate statecharts into a general system-modeling approach in a way that would provide fully executable models, dynamic analysis, and code synthesis—a task quite different from using statecharts on their own because the combined aspects of a system's description can be subtle and slippery.

Accordingly, Harel and colleagues built a full language set around statecharts that was based on the function-oriented, structured-analysis paradigm.[1,2] Statecharts, used for behavioral description, were closely integrated with *activity-charts*, a structured language for functional decomposition and dataflow. However, because structural analysis methods tend to suffer a discontinuity in their transition to design and reuse, many people recommended complementing function-based approaches with ones that follow the object-oriented paradigm. This change has proven to be one of the most significant recent advances in software engineering.

The basic idea is to model the structural properties of classes in a clear hierarchical manner, and to integrate the resulting description with a precise specification of behavior over time, using statecharts. Because classes represent dynamically changing collections of concrete objects (instances)—and because the structure itself is dynamically changing—the model must address the initialization of and reference to real object instances; the delegation of messages; the creation and destruction of instances; the initialization, modification, and maintenance of links that represent association relationships; and so on.

The model must also address aggregation and inheritance from a behavioral point of view. All this makes the problem of combining structure and behavior much harder in an OO-based framework. And it is particularly delicate in the realm of highly reactive systems, which are characterized not by data-intensive computation but by control-intensive, often time-critical, behavior.

References

1. D. Harel and M. Politi, *Modeling Reactive Systems with Statecharts*, McGraw-Hill, New York (to be published); for abridged version, see *The Languages of* STATEMATE, tech. report, i-Logix, Andover, Mass., 1991.
2. D. Harel et al., "STATEMATE: A Working Environment for the Development of Complex Reactive Systems," *IEEE Trans. Soft. Eng.*, Apr. 1990, pp. 403-414.

Operations are synchronous: The thread of control is passed immediately to the called object.

legal navigation expression, or use the name directly if the server is one of its components in a composite or a regular aggregation. In any case, the reference denoted generically is

```
<server> ->
   gen(<eventname>(<parameters>))
```

One special server to which a client can address an event is `this`, which stands for the client itself. In fact, if there is no server, `this` becomes the default. An interesting consequence is that expressions of the form `gen(<eventname>(<parameters>))` that appear in an object's statechart are really just standard events broadcast within and limited to the present statechart.[1]

When it is generated, the event gets queued on a system queue (a multiple-thread system will have a separate queue for each thread). Thereafter, when the client reaches a stable situation (all orthogonal components are in states, and none are left along transitions), the system resumes its continuous process of applying the events from the queue to the appropriate server objects, one by one, in order. Servers use the following simple syntax as a trigger to act on an event:

```
<eventname>(<parameters>)
```

Actual parameters represent the data that comes with the event, and the server may use formal parameters, as we describe later. Because we are dealing with a single-thread model here, the dynamic semantics of this client-server setup are somewhat easier to define, since at most one instance will be active at any time. If more than one instance can act on an event, the semantics imposes no instance order. Thus, the actual order is implementation-dependent.

Now to the important issue of event delegation. Assume that server *A* is a composite object. Who gets to respond to an event *e* addressed to *A*? Is it just *A* via its own statechart, or is it perhaps *A*'s component objects (some or all)? To address this issue, any composite class can be endowed with a simple *forwarding spec* that determines the delegation strategy for the various events. We place this spec inside the top-level state of the class's statechart, as we shall see later for `Terminal`. By default, an event not appearing in *A*'s forwarding spec will be known to *A*'s statechart only. The other two possibilities are for *A* to delegate *e* to one or more of its components explicitly or to delegate it to them all by broadcast. The syntax for these in the forwarding spec is simply `delegate(e,B)` (or `delegate(e,B,C,...)`) and `broadcast(e)`, respectively. In either case, *A*'s statechart is implicitly included. The delegation then continues inductively down the tree of composites, using the components' own forwarding specs. This delegation mechanism is not in the current version of Rhapsody, but it will appear in a future version.

This additional semantic notion for composite classes means that events can be communicated to other objects in many ways—from direct object-to-object communication to full or limited broadcast. In a full broadcast communication, for example, *e* is addressed to the `System` and `broadcast(e)` is included in all forwarding specs. Our railcar example uses default forwarding almost exclusively, meaning that events are always sent to an explicitly named object's statechart. The one exception is the event `clearDest`, which the `Terminal` delegates to its component `DestPanel`.

Events are themselves entities of the model, and can be organized in a generalization/specialization hierarchy. Thus, an object's response to a general event means that it can also respond to any of its more specialized events.

This asynchronous event-based communication mechanism supports client-server relationships in a straightforward way, and modelers do not have to worry about each aspect of sequencing. When an event is sent, the system's queuing scheduler takes care of passing it to the server, and the server deals with it at its own pace.

Operation invocation. Operation invocation causes the immediate execution of the method associated with it in the server object's statechart. The syntax of invocation is the same as that for C++ method activation:

```
<server> ->
   <operationname>(<parameters>)
```

The expression that triggers the method in the called object is just like the one for events:

```
<operationname>(<parameters>)
```

The semantics, however, is synchronous. The thread of control is passed immediately to the called object, which proceeds to execute the relevant method without delay. The client's progress is frozen until the method has executed, at which point the client picks up the thread and resumes its work. The server is deemed to have completed the method when it reaches a stable situation or when it returns a value using the implementation framework's `reply`. We thus view operations as inherently synchronous (although some other frameworks also allow asynchronous operation).

Operations are particularly beneficial when the modeler wants to closely control sequencing or when tight object synchronization is important. Efficiency

is another benefit: With direct invocation, modelers avoid the overhead of queuing and get a simpler and faster translation into an OO programming language. In fact, if we leave out events altogether, basing the dynamics on operations alone, the entire setup takes on an almost exclusive C++ flavor: The objects are really C++ objects, their interaction mechanism is as in C++, and so on.

Object behavior

Objects use a statechart to describe *modal* behavior—behavior that can be different under different circumstances or in different modes. Thus, states (or state configurations) can be viewed as abstract situations in an object's life cycle or as temporary object invariants.

Full vs. partial statecharts. Some authors[7] use statecharts mainly to specify the pre- and post-conditions of operations in the form of abstract states. Others[8] discard concurrent states (orthogonality), claiming that concurrency is already inherent in an OO system because different object instances can exist and can operate simultaneously. Some criticize the broadcast mechanism, claiming that it is unrealistic for many systems. Our adoption of statecharts is lock, stock, and barrel. That is, we use the full power of statecharts, including state hierarchy, multilevel orthogonal components, history connectors, and (internal) broadcast communication. We can justify having orthogonality and broadcasting in our statecharts on several points:

- *Concurrent objects and orthogonal states are quite different*. Orthogonality in statecharts is not necessarily for specifying components that correspond to different subobjects. It enables highly compact descriptions of complex specification logic. Modelers can use orthogonal states to decompose large state spaces naturally into independent (or almost independent) parts. They can also use them to describe complicated transactions or scenarios with many parts, in which several requests to other objects are made simultaneously as part of the treatment of some incoming request and so on. Moreover, in a multiple-thread model, different threads can execute simultaneously in different orthogonal components of the same object.
- *The broadcasting mechanism in statecharts has nothing to do with interobject communication*. It is limited to the scope of a single statechart, in a single object instance. It makes specifying complex internal behavior easier.
- *Orthogonality is crucial for inheritance*. Adding portions to a specified behavior to capture a more specific subtype can be done most easily by adding orthogonal components.

These points are especially obvious in large models.

Triggers and actions. Statecharts involve reactions of the form

```
trigger[condition]/action-list
```

all parts of which are optional, in the usual statechart manner.[1,9] Such a reaction can adorn a transition arrow or appear within a state's *reaction spec*, as in the bottom of Figure 3. In the latter case, it is reevaluated (and triggered whenever relevant) as long as the statechart is in the state in question. A *trigger* is either an event expression or an operation request. *Actions* are sequences of event-generation expressions, operation invocations, and C++ statements.

Some OO purists regard every action as a message; we take a layered view: assignments to variables are C++ statements, not messages. Conditions are also taken from C++ and, again, this is consistent with using the implementation framework language at the detailed level of the model. All these elements may use variables and expressions over data types, according to the underlying application domain.

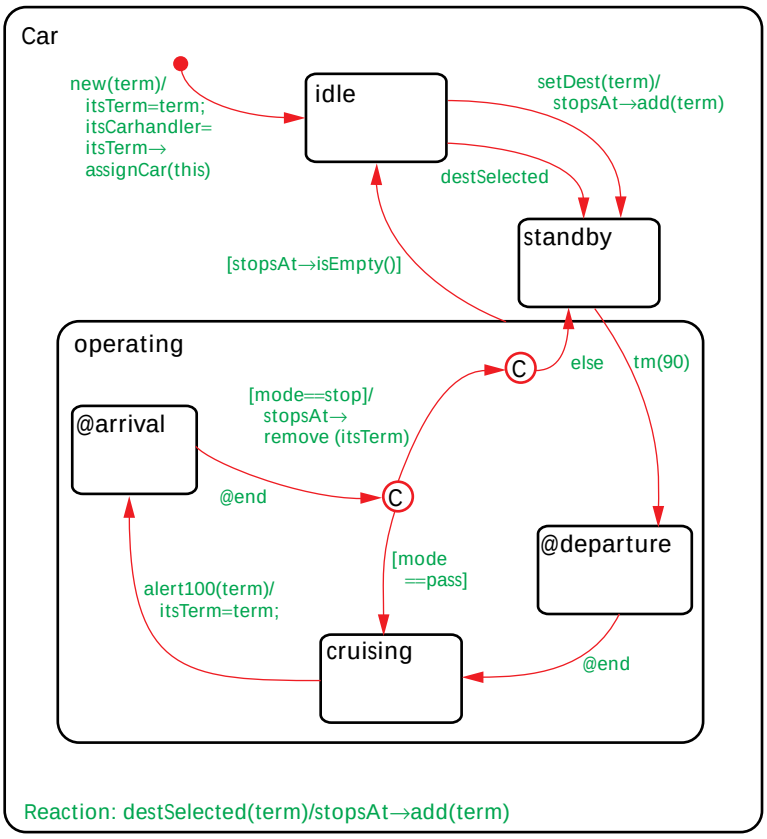

Figure 3. Top-level statechart of `Car`.

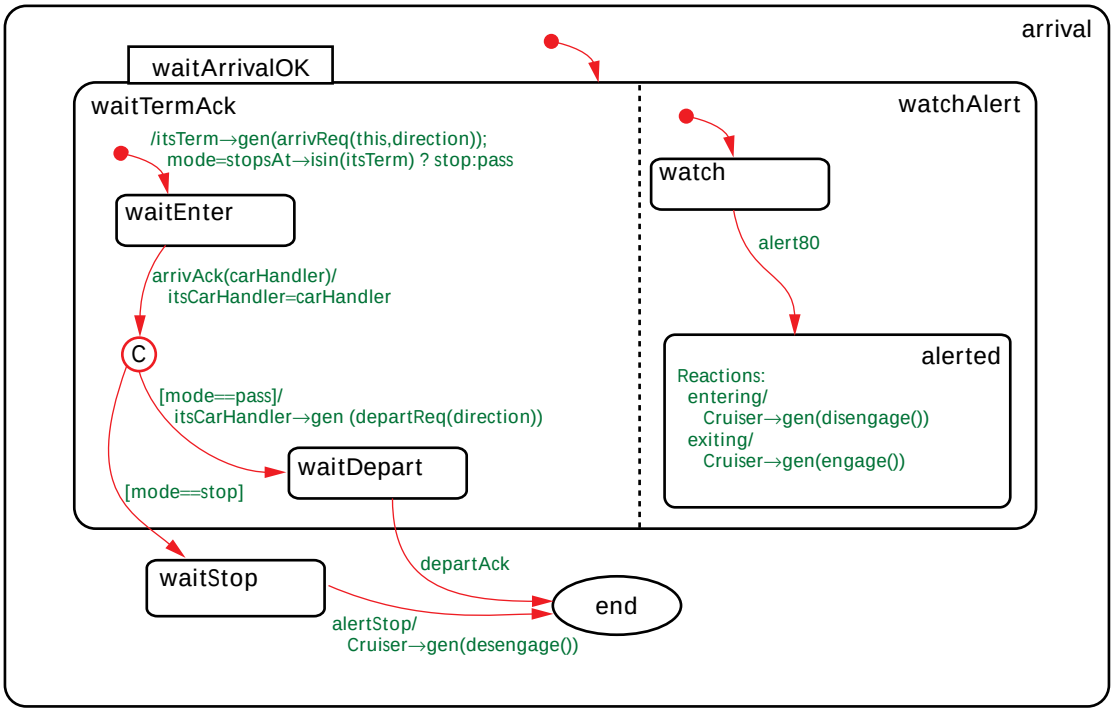

Figure 4. Arrival portion of Figure 3.

The special internal conditions of statecharts, such as `in(state)`, and various kinds of timeouts and delays, such as `tm(n)`, are also allowed.[1,9]

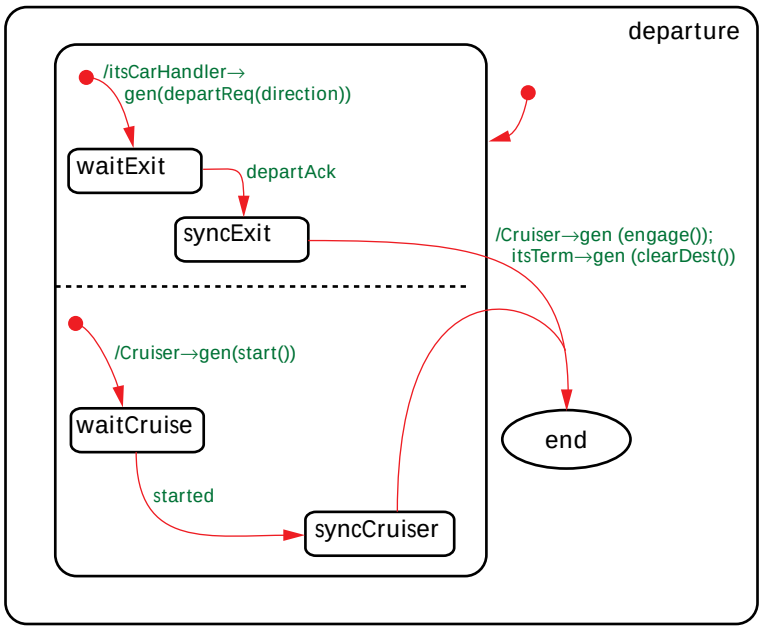

Figure 5. Departure portion of Figure 3.

Among the actions that can appear in statecharts are those for creating and deleting instances. Again, they take their format from our C++-based implementation framework:

```
<object> = new <classname>
(<parameters>)
delete <object>
```

Next, we have actions for adding and removing components from a composite instance:

```
<new component> = add<component
   name>()
remove<component name>(<component
   type>)}
```

For example, the action `stopsAt -> add(term)` appears in the statechart of `Car` in Figure 3, adding the terminal `term` to the set associated with a given car by the `stopsAt` relationship. The car is now scheduled to stop at `term` also.

Finally, the implementation framework has actions for maintaining association relationships by adding an object to the other end of a relationship and removing one from it:

```
<rolename> -> add(<objectname>)
<rolename> -> remove(<objectname>)
```

The default entrance of the statechart's top level state is the initialization entrance for any newly created instance of the object. A reaction attached to the top-level default arrow serves as an initialization script for instances of that class. (Such initialization scripts appear in Figures 3-6).

Semantics. The semantics we adopt for OO statecharts is close to what we defined for statecharts in Statemate.[10] As in Statemate, reactions to events are step by step: the events and actions generated in one transition do not take effect until the next step, after a stable situation has been reached. However, in Statemate, all triggers are constantly attentive, and generated events reach their destinations instantly. Our approach here is different and is aimed at achieving a more realistic design and implementation environment. Thus, we have omitted several features of Statemate, such as event conjunctions, which are harder to implement and do not seem to arise naturally when modeling with a practical design in mind. Also, events in OO statecharts are handed to server objects one by one from the queue, in single-event processing. Another difference is in priorities. When an event can trigger several conflicting transitions, OO statecharts give priority to lower level states. In statecharts in Statemate, higher level states take priority.

The main difference, however, is the role of transitions. In the OO setup, transitions are treated in a run-to-completion manner.[8] Thus, in contrast to Statemate, a transition here will "freeze" in mid-execution, waiting for actions to complete. Moreover, we require that all parts of a transition be fully executed before the statechart becomes stable and the system can respond to another event.

In addition, the method executed by the server in response to an operation invocation must be provided in its entirety along a transition. We require this because once the statechart enters a stable state configuration the method terminates and the thread of control returns to the calling object's statechart. In addition, parameters from events and operations are valid and available only during execution of the (possibly compound and multiple) transition within which the event or operation invocation was received. Once the statechart has stabilized, these values disappear.

From the client's view, the difference between event generation and operation invocation is very important. When the client's statechart generates an event, it retains its thread of control for the rest of the transition, running to completion until the situation stabilizes. In contrast, when the client's statechart invokes another object's operation, its execution freezes in midtransition, and the thread of control is passed to the called object. Clearly, this might continue, with the called object calling others, and so on. However, a cycle of invocations that leads back to the same object instance is illegal, and an attempt to execute it will abort.

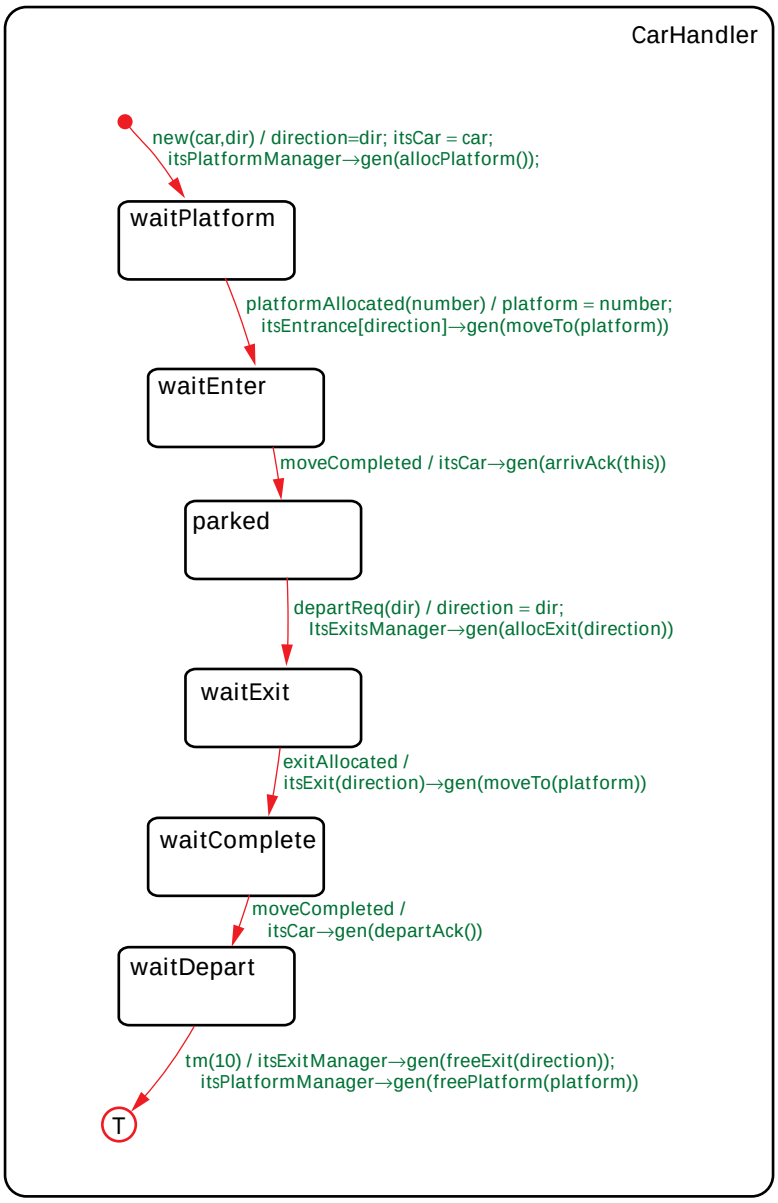

Figure 6. Statechart of `CarHandler`.

Scenario walk-through

Figures 3 through 6 give the main statecharts for the rail-car system. Figures 4 and 5 are subcharts of the statechart for `Car` in Figure 3. The @ prefix in Figure 3 denotes the presence of a more detailed statechart. Because of space limitations and for clarity of discussion, we drew the subcharts separately here. They actually plug into the `@arrival` and `@departure` blobs in Figure 3.

All the usual statechart features[1] are allowed (such as circled C's for conditional branching), as well as termination with self-destruction of the instance in question (circled *T*s). The circled *T* in Figure 6, for example, indicates that a `CarHandler` destroys itself when its task is complete.

The statecharts for `Terminal` and `ControlCenter` are modeless, containing reactions and forwarding information only. They are given respectively as

```
Reactions:
 arrivReq(car,dir)/ new CarHandler
    (car,dir);
 assignCar(car)/ reply=new
    CarHandler(car,1);
 destSelected/
    if (itsCar->isEmpty())
       (itsControlCenter)
         ->gen(sendCar(this))
Forwarding:
 delegate(clearDest, DestPanel)
```

and

```
 sendCar/
    for each (car, itscar) {
       if (car->idle()
          car->gen(setDest(sendCar
            ->term)) }
```

We now walk through the "car approaching terminal" scenario. `Car` has five main modes, as Figure 3 shows. Assume the car is in its `cruising` state, approaching a terminal. It leaves that state when it receives the event `alert100(term)` from `itsProximitySensor` (behavior not described here), alerting the car that it is 100 yards from the terminal. (As we explained earlier, `Car` actually receives the event from the system's queue manager, but for simplicity, we describe the scenario as if events are sent and received directly.) The car sets `itsTerm` to be the `term` it received as a parameter with the `alert100` event, and enters its `arrival` state, described in Figure 4.

Although the real work is carried out by `waitTermAck` (left orthogonal component), `watchAlert` (right orthogonal component) makes sure the car is more than 80 yards from the terminal. If it comes too close, it disengages its `Cruiser`, depicted by the reaction carried out upon entering the `alerted` state.

Meanwhile, the `Car` sends an arrival request to the `Terminal` by generating the event `arrivReq(this,direction)`, providing its own identity and the direction it is traveling. (We omit the internal details of `direction`, which is computed inside the `standby` state of `Car`.) The `Car` also checks if the `Terminal` it is approaching is in the set of terminals it `stopsAt`, setting the `mode` to `stop` or `pass` accordingly. The reaction at the bottom of the `Car` statechart in Figure 3 adds a terminal to the list of scheduled stops whenever a destination is selected. The `destSelected` event is generated when a passenger presses a button on either the car's or the terminal's `DestPanel`.

Inheritance: Structural or Behavioral Conformity?

What exactly does it mean for an object of type *B* to be also an object of the more general type *A*? In virtually all approaches to inheritance in the literature, the `is-a` relationship between classes *A* and *B* entails a basic minimal requirement of *interface* conformity, or subclassing. This means that we should be able to plug in a *B* wherever we could have used an *A* by requiring that *B*'s interface, what it can be asked to do, is consistent with *A*'s. It also means that *B*'s internal structure, such as its set of composites and aggregates, must be consistent with that of *A*.

But this says little about the *behavioral* conformity of *A* and *B*. It requires only that we be able to replace *B* with *A* without causing incompatibility; nothing is guaranteed about the way *B* will actually operate when it replaces *A*. In fact, *B*'s response to an event or an operation invocation might be totally different from *A*'s. In reality, subtyping, that is, guaranteeing full behavioral conformity between a type and its subtype, is technically very difficult, and much research is still needed.

Fortunately, however, most modelers do not expect the inheritance relationship between *A* and *B* to mean that *B* can do anything *A* can do and in the very same way. They are satisfied with guaranteeing that anything *A* can do, *B* can be *asked* to do—that *B looks* as if it is doing what *A* does even if it might actually be doing so differently with different results. One reason for this attitude is that inheritance is introduced largely to enable reuse, really an issue of convenience and savings: We want to be able to spend less effort (and to decrease the chance of error) when respecifying things that have already been specified for a more abstract class.

Object-oriented programming languages do not deal with abstract behavior at all, and therefore their inheritance mechanisms do not address behavioral issues. In C++, for example, a class derived from a base class can turn the original behavior upside down. In contrast, our behavior-intensive language set forces us to address the inheritance of behavior one way or another. The crucial issue, of course, is the statecharts: How should *A*'s statechart relate to *B*'s to yield some kind of conformity and encourage reuse?

The OO community generally agrees that the modeler should somehow construct B's statechart from A's, but recommendations differ. It is possible to show, very easily in fact, that none of the recommended restrictions cannot prevent behavior from changing radically. Consequently, no reasonably liberal proposal, including ours, neither can nor was meant to establish full behavioral conformity.

An `arrivReq` event causes the `Terminal` to instantiate a new `CarHandler`, with the car's identity and its direction as parameters—as shown in the reaction of `Terminal`'s modeless statechart given earlier. Moving to Figure 6, the `CarHandler` statechart starts its life by executing its initialization script, attached to the default entrance arrow. There it saves the two parameters in variables and proceeds to ask for a platform to be allocated. Once the `CarHandler` receives confirmation of allocation and a platform `number`, which it saves in `platform`, it asks for the entrance rail segment of that direction to be moved to the platform in question. Once that is confirmed, making it possible for the car to glide neatly into the terminal, `CarHandler` generates the event `arrivAck` for the car to act on, with its own identity as a parameter. The car, which waited patiently in its `waitEnter` state (Figure 4), instantiates the link to `itsCarHandler` and branches off to stop or make a `departReq` to its handler—depending on whether it is scheduled to stop at the terminal or to pass through. If it must stop, the car waits for an `alertStop` from `itsProximitySensor`, and then leaves its `arrival` state. Going back to Figure 3, `Car` removes the current terminal from its list of `stopsAt` terminals, and enters either `idle` or `standby`, depending on whether it is scheduled to visit any more terminals. If the car is to pass through the terminal (Figure 4), it waits for its `departReq` to be followed by a `departAck` from its handler, and resumes cruising (Figure 3). Upon receiving the `departReq`, the `CarHandler` (Figure 6) goes through a process like the one it went through to set up the car's entrance, causing an exit rail to be connected to the platform. It then notifies the car that all is ready by a `departAck`, waits 10 seconds, frees the exit and platform, and then self-destructs.

Inheritance

Inheritance is a key topic in the OO paradigm. We allow the `is-a` subclassing relationship between object classes to be specified in the object-model diagram in the usual way: a triangular icon on connecting edges. The main concern, however, is to establish a relationship between the statecharts of a parent class (*A*) and its inherited subclass (*B*).

Statechart restrictions. In addressing inheritance, we have adopted the approach described in the sidebar "Inheritance: Structural or Behavioral Conformity?"—careful modifiability but not full behavioral conformity—with code synthesis predominantly in mind. Thus, the restrictions we describe for constructing *B*'s statechart from *A*'s were designed to be as helpful as possible when it comes to reusing parts of the code generated from *A*.

The main guideline, which previous authors have also adopted,[5] is to base the two statecharts on the same underlying state/transition topology. Thus, *B* inherits all *A*'s states and transitions. Although these cannot be removed, certain refinements are allowed. States can be modified in three ways:

- Decompose a basic (atomic) state by `Or` (into substates) or by `And` (into orthogonal components).
- Add substates to an `Or` state.
- Add orthogonal components to any state.

The third way is the most important because it is used to enrich *A*'s behavioral capabilities. Transitions can be added to the statechart, and certain modifications are allowed in the original inherited ones: The target state of an inherited transition can be changed, even to a completely different state (not necessarily to a substate of the original state), but the source state must not be changed. This difference between source and target is somewhat less restrictive than it sounds: We can effectively change the source to a lower level state by adding a new transition with the same target but with the lower level state as the source. Because the statechart semantics gives the transition leading out of the lower state priority, we have what we want.

In addition, if the transition is labeled by `trigger[condition]/action-list`, we can modify the condition and change the `action-list` by deleting some actions (but retaining the order on those remaining) and adding new ones. Although we cannot explicitly remove a transition, we can do so implicitly by making its guard false.

Railcar system statecharts. In modeling the railcar system, suppose we want to add a maintenance car. We could do this by having a generic `Car` and two inheriting subclasses, `PassengerCar` and `MaintenanceCar`. To construct their statecharts, we would first remove the `standby` state in Figure 3 so that its entering transitions would enter `@departure` directly. This would be the statechart for the modified `Car`. Next, the statechart of `PassengerCar` would inherit the states and transitions of the modified `Car` statechart and would add the `standby` state, which, together with some additional changes, would be identical to the original `Car` statechart in Figure 3. We would then have a different version of the `Car` statechart for `MaintenanceCar`, as Figure 7 shows. The new version includes the special `manual` state, in which instructions to the engine are

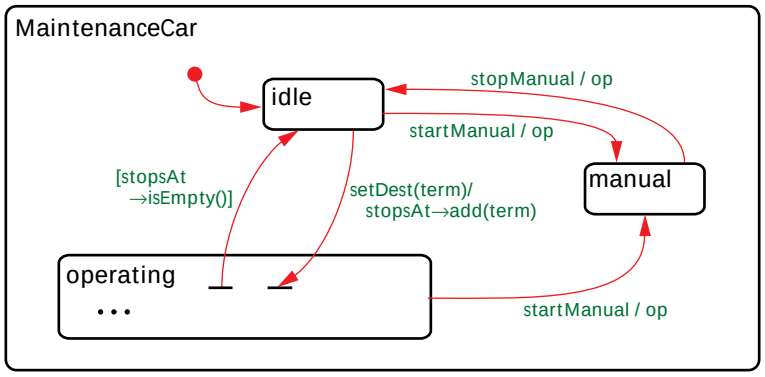

Figure 7. Partial statechart of `MaintenanceCar`.

given directly by the driver. We have left out many details in Figure 7, including some of those inherited from the modified `Car` statechart. Rhapsody lets modelers display such statecharts in more useful ways, such as showing the inherited elements in light gray and highlighting new or modified elements.

To continue the work we have described, we intend to explore several research areas. One of the more interesting is inheriting behavior. We plan to carefully investigate the various levels of behavioral conformity possible in a setup such as ours, and to address their feasibility, enforceability, and computational complexity.

Another research topic is the possible productive relationships between statecharts and message sequence charts. For example, we would like the ability to synthesize a first efficient version of the statecharts for an object model from the scenarios given in the message sequence charts.

Rhapsody, which is available from i-Logix, is another area of future work. Although we did not focus this article on Rhapsody per se, we believe we have communicated its essence through our descriptions of the language set and through our commitment to executability, analysis, and code synthesis in Statemate.[11] In addition to supporting the language set with the same dedication to such behavioral issues (ObjectTime[8] is another such tool), Rhapsody addresses many methodological issues—in ways that are in line with UML and recommendations by other authors.[3-5,8] However, we continue to enhance Rhapsody's power and usefulness and plan to extend it to best accommodate the ways engineers approach the development of complex systems.

As far as code synthesis goes, we believe we are on the right track. We hope that the code Rhapsody generates will help bring high-level modeling closer to the desired final product. ❖

Acknowledgments

We thank Michal Politi, Alex Nerst, and especially Michael Hirsch, for numerous helpful discussions and ideas.

David Harel's research was supported in part by grant 7096/3 from the Israel Academy of Sciences.

References

1. D. Harel, "Statecharts: A Visual Formalism for Complex Systems," *Sci. Computer Prog.*, July 1987, pp. 231-274; also see Tech. Report CS84-05, The Weizmann Inst. of Science, Rehovot, Israel, 1984.
2. I. Jacobson, *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison Wesley, Reading, Mass., 1992.
3. G. Booch, *Object-Oriented Analysis and Design, with Applications* (2nd ed.), Benjamin/Cummings, San Mateo, Calif., 1994.
4. J. Rumbaugh et al., *Object-Oriented Modeling and Design*, Prentice Hall, New York, 1991.
5. S. Cook and J. Daniels, *Designing Object Systems: Object-Oriented Modelling with Syntropy*, Prentice Hall, New York, 1994.
6. D. Harel, "On Visual Formalisms," *Comm. ACM*, May 1988, pp. 514-530.
7. D. Coleman, F. Hayes, and S. Bear, "Introducing Objectcharts, or How to Use Statecharts in Object Oriented Design," *IEEE Trans. Soft. Eng.*, Jan. 1992, pp. 9-18.
8. B. Selic, G. Gullekson, and P. Ward, *Real-Time Object-Oriented Modeling*, John Wiley & Sons, New York, 1994.
9. D. Harel and M. Politi, *Modeling Reactive Systems with Statecharts*, McGraw-Hill, New York (to be published); for abridged version, see *The Languages of* STATEMATE, tech. report, i-Logix, Andover, Mass., 1991.
10. D. Harel and A. Naamad, "The STATEMATE Semantics of Statecharts," *ACM Trans. Software Eng. Methodology*, Oct. 1996, pp. 293-333.
11. D. Harel et al., "STATEMATE: A Working Environment for the Development of Complex Reactive Systems," *IEEE Trans. Soft. Eng.*, Apr. 1990, pp. 403-414.

David Harel is the William Sussman professor of mathematics at The Weizmann Institute of Science and a cofounder of i-Logix Inc., Reading, Mass. His research interests are in computability and complexity theory, logics of programs, database theory, systems engineering, and visual languages. Harel received a BSc in mathematics and computer science from Bar-Ilan University, an MSc in computer science from Tel-Aviv University, and a PhD in computer science from the Massachusetts Institute of Technology. He received ACM's 1992 Karlstrom Outstanding Educator Award and is a fellow of the ACM and the IEEE.

Eran Gery is a project manager at i-Logix Israel Ltd., and is the principal software architect of the Rhapsody tool described in this article. His research interests are software architecture, modeling languages, behavioral modeling, and analysis and design methods. Gery received a BSc and an MSc in computer science from the Technion, Israel's Institute of Technology.

Contact the authors at i-Logix Israel Ltd., Rehovot 76327, Israel; harel@wisdom.weizmann.ac.il; erang@ilogix.co.il.