

# Executable Object Modeling with Statecharts

David Harel\*      Eran Gery†

Revised February, 1997. To appear in *IEEE Computer*  
(Early version in *Proc. 18th Int. Conf. Soft. Eng.*, IEEE Press, March 1996, pp. 246–257.)

**Abstract:** A behaviorally expressive set of diagrammatic languages for modeling object-oriented systems is presented. It constitutes the constructive subset of UML, and is supported by RHAPSODY, a tool that enables model execution and full code synthesis.

## 1 Introduction

This paper reports on an effort to develop an integrated set of diagrammatic languages for modeling object-oriented systems, and to construct a supporting tool. We want our models to be intuitive and well-structured, but also behaviorally expressive and rigorous. The goal is to support full executability and dynamic analysis, as well as automatic synthesis of usable and efficient code in object-oriented languages such as C<sup>++</sup>. At the heart of the modeling method is the language of statecharts [H1] for specifying object behavior, and a structured object-model language for describing classes, and their structure and inter-relationships. Objects can interact by event generation or direct invocation of operations.

Our approach is consistent with the recent UML effort [Ra]. First, statecharts have been adopted in UML and in its main precursors (the Booch method [B] and OMT [R<sup>+</sup>]) as the main medium for specifying behavior. Second, we have participated quite extensively in the UML definition effort (led by the group from Rational Corp.), in order to ensure that UML will be fully consistent with our work. In fact, our language set can be viewed as a “UML core”, constituting the constructive portion of UML, but coming complete with a fully worked out behavioral semantics and a powerful supporting tool.

In the interest of keeping the exposition in the paper manageable, we leave out some technically involved topics, such as multiple-thread concurrency and active objects, which will be described elsewhere. The supporting tool, RHAPSODY, which is available from i-Logix, Inc., will also be described in detail separately.<sup>1</sup>

---

\*The Weizmann Institute of Science, Rehovot, Israel. Email: [harel@wisdom.weizmann.ac.il](mailto:harel@wisdom.weizmann.ac.il). Research supported in part by a basic research grant from the Israel Academy of Sciences.

†i-Logix Israel, Ltd.. Email: [erang@ilogix.co.il](mailto:erang@ilogix.co.il)

<sup>1</sup>In the conference version of this paper (*Proc. 18th Int. Conf. Soft. Eng.*, IEEE Press, March 1996, pp. 246–257), we used the preliminary name O-Mate for this tool.

## 2 Background and overview

Statecharts were conceived of by the first-listed author in 1983. In the (belatedly published) paper that first presented them [H1], statecharts were portrayed in isolation, as a visual formalism for specifying ‘raw’ reactive behavior. Adopting statecharts as the behavioral component of a general system-modeling approach is quite a different matter, since the links between the various aspects of a system’s description can be subtle and slippery. Modeling approaches ought to be detailed and precise enough to enable model execution, dynamic analysis and code synthesis. For this, the language set must be rigorously defined and ‘closed up’: Any possible combination of graphical and/or textual constructs must be clearly characterized as syntactically legal or illegal, and all legal combinations must then be given unique and formal meaning.

Over a decade ago, a full language set was built around statecharts, based on the function-oriented structured-analysis paradigm (SA); see [H<sup>+</sup>, HP]. Statecharts, used for behavioral description, were closely integrated with a structured language for functional decomposition and data-flow, called activity-charts.<sup>2</sup> Since SA methods are widely regarded as suffering from a discontinuity problem in transition to design and reuse, many people recommend complementing function-based approaches with ones that follow the object-oriented (OO) paradigm. This change is one of the most significant in software engineering in recent years. Accordingly, we embarked on an effort to develop a set of languages for object modeling, built around statecharts, and to construct a supporting tool with full executability and code synthesis capabilities.

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. Since classes represent dynamically changing collections of concrete objects (instances), and since the structure itself is dynamically changing, the model must address issues like the initialization of, and the reference to, real object instances, the delegation of messages, the creation and destruction of instances, the initialization, modification and maintenance of links representing association relationships, etc. We must also address aggregation and inheritance from a behavioral point of view. All this makes the problem of combining structure and behavior much harder than in an SA-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.

The object paradigm started in the programming languages community, but was later adopted on an abstract level too, in the form of methodologies that are more appropriate for system modeling; see, for example [B, CD, R<sup>+</sup>, SGW, SM]. Most object-oriented modeling methodologies offer graphical notations for specifying the model. They typically have ER-style diagrams for specifying classes of objects and their inter-relationships, and means for describing the interface and capabilities of the objects themselves. A state-based formalism is usually adopted for specifying behavior, and all of the methodologies listed above recommend statecharts (or some sublanguage thereof) for this. However, in many cases such methodologies do not address dynamic semantics adequately, so that the precise behavior of models over time is not always well-defined. One major motivation for our work was to eliminate this crucial shortcoming.

---

<sup>2</sup>A third language, module-charts, was used to specify physical decomposition. See [H3] for the motivation and ‘philosophical’ aspects of this effort, and [HP] for a full description of the languages. This language set underlies the STATEMATE tool [H<sup>+</sup>], built to enable executability, analysis and code-generation.

Our approach involves two constructive modeling languages, *object-model diagrams* and *state-charts*, and a reflective language, *message sequence charts* (MSC)

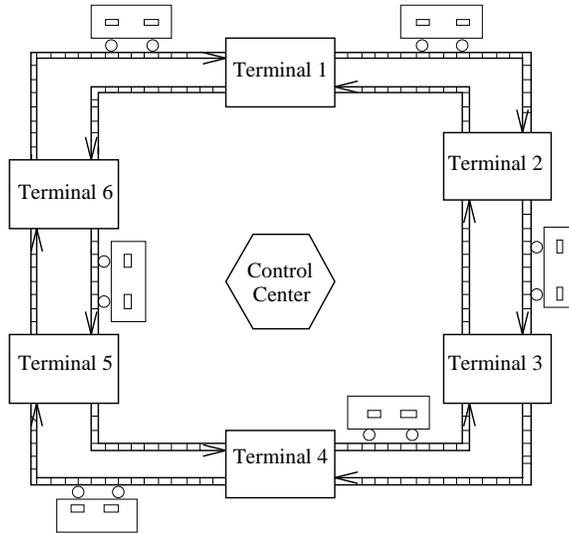


Figure 1: The rail-car system

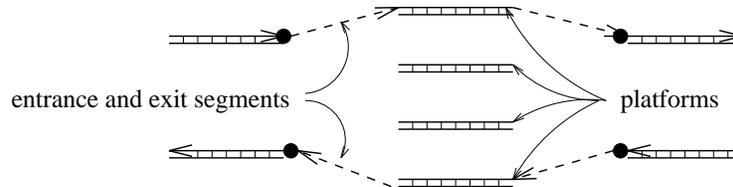


Figure 2: Inside a terminal

### 3 The rail-car example

We illustrate the approach using an automated rail-car system example (see Fig. 1).<sup>6</sup> Six terminals are located on a cyclic path, and each pair of adjacent ones is connected by two rail tracks, one for clockwise travel and the other for counter-clockwise. Several rail-cars are available to transport passengers between terminals. There is a control center that receives, processes and sends system data to the various components.

Each terminal has a parking area containing four parallel platforms, each of which can park a single car (see Fig. 2). The four rail tracks that are incident with a terminal, two incoming and two outgoing, are each connected to a rail segment that can be adjusted to link to any one of the four platforms. The terminal has a destination board for passenger use, containing a pushbutton 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 the maximal speed that will guarantee that it will never be 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 120 yards. A car also has its own destination board, similar to the one in the terminal. The control center

<sup>6</sup>The example was inspired by the specification appearing in a 1990 manuscript by Vered Gafni, titled “Automatic Transportation System”.

communicates with the various system components, receiving processing and providing system data.

Here are some typical scenarios of the system (sometimes called ‘use cases’ [J]). The first two depict interactions between a car and a terminal, and the last two between a passenger and the system:

- *Car approaching terminal:* When it reaches 100 yards from the terminal, the car will be allocated a platform and entrance segment connecting it to the incoming track. If the car is to pass through without stopping, it is allocated an exit segment too. If the allocation is not completed within 80 yards from the terminal, the car is stopped and delayed until all is ready.
- *Car departing terminal:* A car will depart the terminal after being parked for 90 seconds. In order to depart, the following operations are carried out: The platform is connected to the outgoing track by the exit segment; the car’s engine is engaged; the destination indicators on the terminal destination board are turned off. Departure then takes place, unless the track is not clear (there is a car within 100 yards), in which case departure is delayed.
- *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 presence of such a car would have been indicated by a flashing sign on the destination board.) The passenger pushes the destination button, and waits until a car arrives. When the destination button is pushed, if there is an idle car in the terminal it will be assigned to that destination, otherwise the system will send a car in from some other terminal.
- *Passenger in car:* A passenger wanting to disembark pushes the appropriate button on the car’s destination board and waits for the car to come to a halt in the destination terminal. The system will see to it that the car stops.

Such scenarios can be described beneficially using message sequence charts (MSC’s). See, for example, Fig. 3, in which the events used are described later on. As mentioned, RHAPSODY supports MSC’s, which are especially good for describing collaborations, but as a reflective modeling language: An MSC can be checked for consistency against the model itself, but the model’s behavior has to be specified using statecharts, as described in Section 7. An additional way to use MSC’s in RHAPSODY is as an appealing “reflective” formalism. When executing a model an MSC can be set up to show the progress of inter-object communication.

## 4 Object-model diagrams: Classes and their affinities

Object-model diagrams specify classes of objects and their structural relationships. An object-model diagram is an ER-like diagram that can be viewed as a UML object model [Ra]. It features higraph [H2] encapsulation that denotes a strong kind of composite class aggregation. Directed edges represent relationships, with an undirected edge abbreviating a two-way directed edge. Classes and relationships can have associated multiplicity information of the kinds available in many object-oriented methodologies. For consistency with UML and other notations for class structure, we also allow a weaker kind of aggregation, represented as in [B, R<sup>+</sup>] and elsewhere by branching arrows with a diamond-shaped icon. However, the examples in this paper do not contain weak aggregation, and

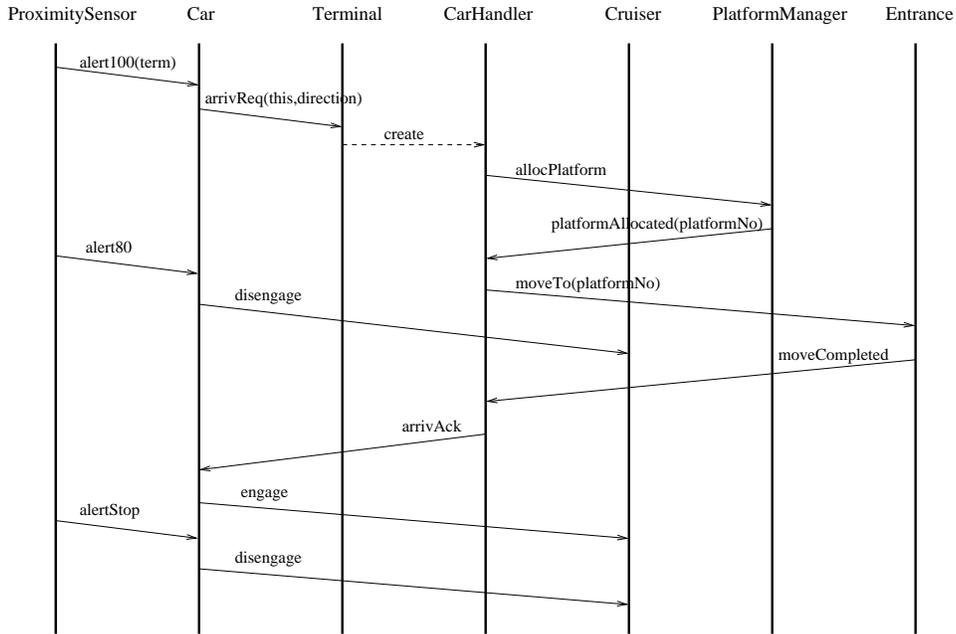


Figure 3: An MSC for the “Car approaching terminal” scenario

their semantics is indeed much weaker — essentially just that of a special association relationship, called **part-of**, between the aggregate and its components.

Two comments regarding object-model diagrams are in order here. First, in this paper, we use a rather degenerate method for presenting information about the classes in an object-model diagram. For example, we do not show the list of methods supported by an object of a given class. The **RHAPSODY** tool is much more accommodating, in the spirit of the well-known methods of [B, R<sup>+</sup>, Ra] and others. Second, we went through a lengthy period of internal deliberations as to whether we should have separate languages for classes and instances. There are obvious advantages to having the class model and instance model together, mainly in way of compactness and comprehension of the representation. However, among the strong reasons not to we might note that a separate language for concrete object instances can provide more flexibility in representing nontrivial issues of multiplicity, object creation and reference. In fact, there is no real need for a separate notation for instances: Instances are created during execution by realizing the multiplicities in class definitions and the instantiation of composite classes specified using the composite aggregation mechanism. There are many ways to show the instances and their behavior during a run of the system. In the modeling phase, on the other hand, instances play no role. We have thus decided to use a single object-model language, in which classes are described together with the information needed to create instances thereof.

Fig. 4 shows a partial object-model diagram for the rail-car system.<sup>7</sup> It shows the four main classes, with the added information that there is a single **ControlCenter** and six **Terminals**; the other classes lack multiplicity information, which means that they can have an unlimited number of instances. There are four many-one bi-directional association relationships, and two uni-directional ones. Lacking relationship names and roles, the instances refer to their ‘relatives’ by the phrase

<sup>7</sup>Conceptually, there is a single object-model diagram for the entire system, but a modeler will typically construct and view it in parts. Their union is to be taken as the chart for the entire system modeled.

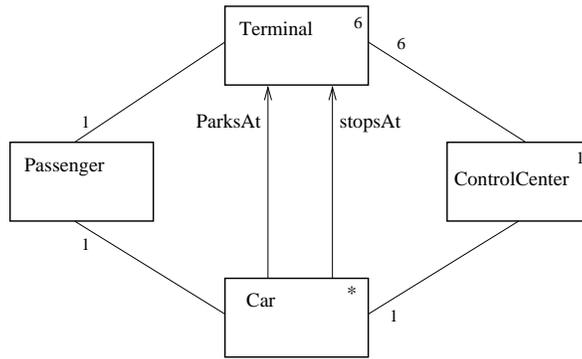


Figure 4: High-level object-model diagram for the rail-car system

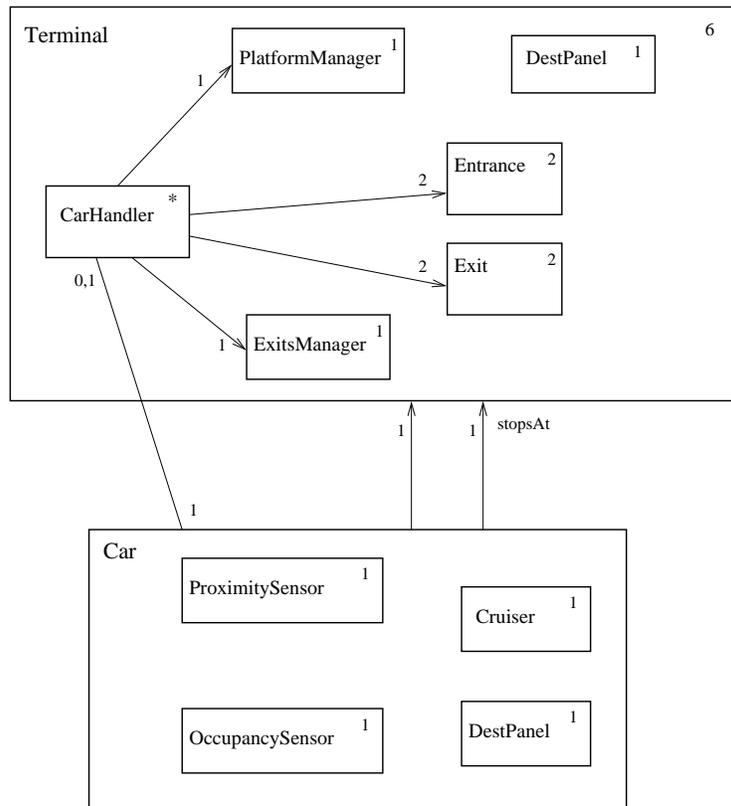


Figure 5: More of the rail-car system's object-model diagram

`its`. Thus, a passenger can refer to `itsTerminal` and a terminal will have a set of `itsPassengers`. On the other hand, one of the edges has a role name, so that a `Car` can refer to the set of terminals it `stopsAt`, which could be different from the set of `itsTerminals`. Directionality limits referencing ability: According to Fig. 4, a terminal cannot refer to `itsCars`.

To enable easy referral along relationship links, we allow standard kinds of *navigation expressions*, the full details of which will not be described here. As an example, we might want to use the navigation expression `Passenger -> itsCar -> stopsAt` to refer to the set of terminals at which the car carrying the passenger is scheduled to stop. Also, using the convention that `System` always refers to an explicit composite object that encloses the entire model, the six `Terminals` can be referred to from the top level of the model by `System -> itsTerminal[1:6]`. (We will have more to say about this C++ syntax in the next section.)

Fig. 5 shows the six components of the composite object `Terminal`, and the four components of `Car`. The `Entrance` and `Exit` components are software drivers for the relevant rail segments. The `PlatformManager` and `ExitsManager` allocate platforms and exits to the `CarHandler`. In contrast with the other classes in these figures, the `CarHandler` is a concept that does not come from the problem domain, but would probably be introduced by a domain expert during the modeling. It handles the transactions between a `Car` and `Terminal`. As we shall see, a special `CarHandler` is created whenever a car approaches a terminal, and it is destroyed when the car departs; it thus serves as a proxy object for the car within the terminal. The four components within `Car` have no links, since they do not collaborate among themselves. The `ProximitySensor`, for example, is a primitive object that sends events to the `Car`'s behavior (i.e., to its statechart) based on the distance to the approached `Terminal`. A composite class can refer to its components directly.

Note that we allow direct links (and hence direct communication) between a component object and objects outside its composite parent. Note also that the relationship between `Car` and `CarHandler` has been excluded from Fig. 4. Had we wanted it to appear there too, we could have represented it by an edge leading from `Car` to a stubbed end lying within the interior of `Terminal`.

## 5 Instantiating classed and links

Object-model diagrams describe classes and their structure, and as such, they appear to concentrate on static aspects only. However, when a model is executing, the system consists of concrete objects, i.e., instances of classes, that go through life communicating with other concrete objects along actual links. One of the reasons that it can be very difficult to describe the behavior of such a system is that, unlike hardware systems, the very topology of the objects and their relationships is often intensely dynamic.

Defining the behavioral semantics specifically enough to enable model execution and full code synthesis hinges on two main things: Initialization and dynamics over time. Initialization concerns the way the model starts out, i.e., which object instances are constructed at the start, and how their attributes and relationships with other objects are initially set up. Dynamics concerns the way the model behaves when running, and it consists of two different kinds of possible changes that can occur in the status of the model: (i) changes in the state of an object, caused by triggering occurrences like events and calls for operations, and (ii) changes in the system's structure, i.e., the instantiation and deletion of objects, and the establishment and modification of links between them.

Although statecharts are the central tool used to specify the dynamics of a model, parts of the initialization and parts of the dynamic changes to the model’s structure depend only on the information in the object-model diagram, as we now explain. The first thing to happen when a model starts executing is the initial construction of the composite structures of the model, including the creation of instances from classes that have a fixed integer multiplicity.<sup>8</sup> This happens recursively down the tree of composites (that is, from a composite class to its components, repeatedly), and new instances are created for each instance of a composite class. Thus, referring to Figs. 4 and 5, one `ControlCenter` is created at the start, as well as six `Terminals`, and within each `Terminal` there will be created two `Entrances`, two `Exits`, one `PlatformManager`, one `ExitsManager` and one `DestPanel`. Components with unspecified multiplicity will create no instances spontaneously, so that no `Cars` or `CarHandlers`, for example, are created at the start. As we shall see later, instances of these are created in two different ways.

This feature of composite classes remains valid beyond the initialization phase. In fact, it extends 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. Similarly, destruction of the composite destroys all of its components too.

Now to association relationships. When considered from the point of view of boot-strapping the model, these can be thought of as coming in three flavors: *unambiguous*, *ambiguous but bounded*, and *unworkable*. An example of an unambiguous association is the link between `Terminal` and `ControlCenter` in Fig. 4. It is many-one, but the multiplicities on either end of the edge match those of the associated classes, implying one possible interpretation, 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`, and it, in turn, can refer to all of `itsTerminals`, if it so desires. Omitting the numeral 6 from the end of the edge in the figure would have resulted in an ambiguous many-one association, since any subset of the six `Terminals` could be associated with the unique `ControlCenter`. Nevertheless, the association would have been bounded, since it has a well-defined upper bound (all six are connected) and a well-defined lower bound (none are connected). Other associations, such as the ones between `Car` and `Terminal`, are unworkable at this point, because there are simply no instances spawned on one or more ends of the link.<sup>9</sup>

Here our semantics splits into two possibilities, differing in the case of ambiguous but bounded associations — *greedy* and *nonchalant*. The greedy semantics takes the most it can, and the nonchalant semantics takes the least it can get away with. We adopt the greedy semantics in this paper, though a user of `RHAPSODY` has some flexibility. The greedy approach sets things up using canonical mappings. For example, if there are  $n$  instances in each class in a symmetric one-one relationship, the greedy approach will associate them in matching pairs,  $A(i)$  to  $B(i)$ , for each  $1 \leq i \leq n$ . If the relationship is reflexive (i.e.,  $B = A$ ), the greedy approach matches them in cyclic order,  $A(i)$  to  $A(i + 1)$ , with  $A(n + 1)$  identified with  $A(1)$ . We have worked out necessary and sufficient conditions for greedy resolution of the instantiation problem for symmetric relations, but the details are outside the scope of this paper.

These rules for setting up relationships can also be extended to the dynamics at large: Through-

---

<sup>8</sup>Our languages allow also multiplicities specified by integer-valued variables, and instance creation is carried out using those variables’ current values. Again, we do not get into the details of this feature here.

<sup>9</sup>This is another example of the informal nature of our exposition, since we provide here neither an exhaustive syntax for association links nor the details of the algorithm that resolves their three-way classification.

out the run of the model, whenever objects are instantiated or destroyed, all relevant associations should be evaluated anew and set up as above. Thus, for example, if an instance of `CarHandler` is somehow created, it will have the ability 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]`, and similarly for the `Exits`. This dynamic re-evaluation has not yet been implemented in RHAPSODY.

During the ongoing behavior of the model, changes may occur in the current set of instances and their links. These can be prescribed by several kinds of actions that can appear in the statecharts. The first two are for maintaining instances, by, respectively, creating a new instance of type `classname` (the parameters are explained later) and deleting an instance. As is our general philosophy here, they are written in the implementation framework language, which in the present version is C++:

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

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

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

Finally, we have actions for maintaining association relationships, by, respectively, adding an object to the ‘other’ end of a relationship and removing one from it:

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

For example, the action `stopsAt -> add(term)` appears in the statechart of a `Car` (see Fig. 6), with the effect of adding the terminal called `term` to the set associated with a given car by the `stopsAt` relationship; this means that the car is now scheduled to stop at `term` too.

Before we go any further, we should comment on our nongraphical syntax. The reader will have no doubt noticed that the actions are written in C++. This might not seem to deserve justification, given the status of C++ in the world of object-oriented programming languages, and the fact that it is the current target language for the code synthesis of RHAPSODY. However, the decision to use C++ as our action language bears little relationship to the ideas of the paper. As a practical matter, we had to decide on an implementation framework for the language set proposed here, and we chose one based on C++. The decision was to write actions directly in the implementation language, which makes it possible to plug in a framework based on another language, such as Ada, Smalltalk or Java, or even a set-based language (as is done, e.g., in [CD]). This would have resulted in different-looking elements in our action language, but it would not have changed anything of significance in our modeling and analysis approach. What we could not do, however, was to keep away from the specifics of the detail level completely (mainly the action language), leaving it to the reader, since we want our examples to be describable in detail, and we want our tool to support the modeling process in its entirety. Therefore, once C++ was chosen for the initial implementation, it

became natural to use C++ for the detail level of the model too, to make our action code fit the implementation framework smoothly.

Back now to the initialization of our system, which is not yet complete. In many models, additional information is needed to determine the full starting configuration of the system, and the user is expected to provide it. In our example, we have left the number and location of the `Cars` unspecified. We could have provided a multiplicity specification in the object-model diagram, specifying that there are, say, twelve `Cars`, but we would have still been left with resolving the ambiguous links with the six `Terminals`; e.g., precisely which of them is a given `Car`'s `itsTerminal` (which is really the question of where the car is located initially). What we do instead is to provide the implicit top-level `System` object with an *initialization script*, which is carried out once, at the start. In fact, each object may have an initialization script in its statechart, as we shall see later. Here is the script for the rail-car system (again, in our C++ dialect); it decrees the creation of twelve new `Cars`, located in adjacent pairs in the six `Terminals`:

```
for (int car = 0; car < 7; car++)
  System -> itsCar[2*i] =
    new Car(System -> itsTerminal[i]);
  System -> itsCar[2*i+1] =
    new Car(System -> itsTerminal[i])
```

## 6 Events and operations

Having finished with the initialization, we should now discuss statecharts and how they are used to specify behavior. At the core of the behavior are the elementary mechanisms utilized in the statecharts to effect object communication and collaboration. We have chosen two: The generation and sensing of events, and the direct invocation of operations, which trigger execution of methods. Operations are more concrete entities than events, which implies, for example, that it makes little sense to broadcast an operation. In contrast, events can be distributed widely, as we shall see, using a flexible kind of broadcasting and delegation mechanism.

An object — sometimes called the *client* — can generate an event, and address it to some other object, the *server*. The addresser must be able to refer to the addressee, and this can be done using a legal navigation expression, or directly by name if the addressee is one of its components in a composite or a regular aggregation. In any case, denoting such a reference generically, we write this in our chosen framework as:

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

One special server object to which a client can address an event is `this`, which stands for the client object itself. In fact, the omission of a server object assumes `this` by default. An interesting consequence is that expressions of the form `gen(<eventname>(<parameters>))` that appear in an object's statechart are really just the standard events of [H1], which are broadcast within, and limited to, the present statechart itself.

Upon generation, the event gets queued on a (single) system queue. Thereafter, when the client object reaches a stable situation (see the next section), 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 to act upon an event:

`<eventname>(<parameters>)`

Actual parameters represent the data that comes with the event, and the server may use formal parameters, as we shall see in the example later. That we are dealing here with a single-thread model makes the dynamic semantics of this client/server setup somewhat easier to define semantically, since at most one instance will be active at any given point in time. If more than one instance can potentially act upon an event, there is no order imposed by the semantics on which instance gets to act first. Thus, the actual order is implementation dependent.

Now to the important issue of event delegation, which is relevant in the case of a server **A** that happens to be a composite object. Who gets to respond to an event **e** addressed to **A**? Is it just **A** itself, via its own statechart, or perhaps some or all of its component objects? One possibility for approaching this issue is a publish/subscribe mechanism. We have opted for a somewhat different one. Our language set allows any composite class to 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; see Fig. 10. By default, an event not appearing in **A**'s forwarding spec at all will be known to **A**'s statechart only. The other two possibilities are for **A** to delegate the event **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 own statechart is implicitly included too. The delegation is then continued inductively down the tree of composites, i.e., from composite class to its components, using the components' own forwarding specs.<sup>10</sup>

Note how this additional semantic notion with which we endow composite classes enables events to be communicated to other objects in a wide spectrum of ways, from direct object-to-object communication to full or limited broadcast. (Full broadcast can be obtained by addressing the event **e** to the **System** and including `broadcast(e)` in all forwarding specs, including that of the **System** itself.) Our rail-car 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` in Fig. 10.

Events are themselves entities of the model, and can be organized in a generalization/specialization hierarchy.<sup>11</sup> Thus, an object's response to a general event abbreviates the fact that it responds to any of its more specialized events.

There are many methodological justifications for having such an asynchronous event-based communication mechanism in an object-oriented modeling method. They include the simple way it supports client/server relationships, and the way it frees the modeler from worrying about each and every aspect of sequencing. "Send and forget": You send an event, the system's queuing scheduler takes care of passing it on to the server, and the server may deal with it at its own pace. So much for events.

The second communication mechanism between objects involves one object directly causing

---

<sup>10</sup>The delegation mechanism is not implemented yet in Rhapsody.

<sup>11</sup>We avoid the almost philosophical question of whether events are really objects. We borrow from the theory of objects concepts we need for events, e.g., the generalization/specialization hierarchy, and leave those we don't, e.g., object behavior.

another to execute a method, by invoking one of its operations. The called object may return a value upon completing its method. The syntax of invocation is that of 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 execution of the method is completed, at which point it 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 by using the implementation framework's `reply`.

Operations are particularly beneficial in cases where the modeler wants close control over sequencing, or where tight synchronization between objects is important. But the availability of operations is crucial in our setup for other reasons too, an important one being efficiency. With direct invocation of operations, the overhead of queuing is avoided, and the translation into an object-oriented programming language is simpler and faster. In fact, as implied by the division of modeling into various stages recommended by Cook and Daniels [CD], events are more appropriate in the analysis phase, whereas operations and methods are closer to design. We predict that in real-world modeling efforts, some of the events introduced in the early stages of the development will be replaced by operation invocation as the process come closer to design, though events can serve very well for design purposes too. In fact, it is interesting to observe that if we leave out events altogether, basing the dynamics on operations and methods alone, the entire setup takes on an almost exclusive C++ flavor: The objects are really C++ objects, their interaction mechanism is as in C++, etc.

Now that we have seen the specially tailored nongraphical elements that our object-oriented statecharts may use, it is time to put everything together to obtain a fully executable model.

## 7 Statecharts for describing object behavior

The behavior of an object is specified by a statechart that can be associated with its class. We say 'can', since some objects might not need a statechart. They might delegate all their obligations to component objects using suitable forwarding specs, or the modeler may decide that their behavior is not specified inside the model, but, rather, will be taken from a ready-made library module, or from a reused component of some other system, or be given by explicit code. We term these *primitive*. This section is about nonprimitive objects.

States (or actually state configurations, since statecharts can be in multiple orthogonal state-components at any given point in time) can be viewed as representing abstract situations in the life-cycle of the object, or as temporary invariants of the object. An object needs a statechart for describing *modal behavior*, i.e., behavior that can be different under different circumstances, or in different modes.

The main methodological point we want to make here is that for such objects it is best to utilize the full statecharts language. Some authors, like those of [CHB], use statecharts mainly

for specifying the pre- and post-conditions of operations in the form of abstract states. Others discard orthogonality (concurrent states), claiming that concurrency is already inherent in an object-oriented system by virtue of different object instances existing and operating simultaneously. Some criticize the broadcast mechanism of statecharts, claiming that it is unrealistic for many systems. Our adoption of statecharts is lock, stock and barrel. First, the two kinds of concurrency are quite different. Orthogonality in statecharts is not necessarily for specifying components that correspond to different sub-objects. One of its main justifications is to enable highly compact descriptions of complex specification logic. (This succinctness can be exponential relative to the size of a concurrency-free finite state machine [DH].) It may be used 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. Second, the statechart broadcasting mechanism has nothing to do with inter-object communication; it is limited to the scope of a single statechart, in a single object instance, and is included to ease the specification of complex internal behavior. Third, orthogonality is crucial to the treatment of inheritance, since adding portions to a specified behavior in order to capture a more specific subtype can be done most easily by adding orthogonal components. Some of these points can be seen on a small scale when studying the sample statecharts given here, but they are far more evident in large models.

On a technical level, statecharts involve reactions of the form:

`trigger[condition]/action`

all parts of which are optional, in the usual statechart manner [H1,HP]. Such a reaction can adorn a transition arrow, and can also appear within the *reactions spec* of a state, in which case it is re-evaluated, and triggered whenever relevant, as long as the statechart is in the state in question. A trigger is either an event or an operation arrival expression, as discussed above. Actions are sequences of the event-generation expressions and operation invocations discussed above, and of C++ statements that we do not describe in full here. Some OO purists regard every action as a message; we take a layered point of view, where assignments to variables are C++ statements and not messages. Conditions are also taken from C++, and, again, this is part of the (arbitrary) decision to use C++ on the detail level to match our implementation framework. All these elements may use variables and expressions over data types, according to the underlying application domain. The special internal conditions of the language of statecharts, such as `in(state)`, and various kinds of timeouts and delays, such as `tm(n)`, are also allowed; see [H1,HP].

The default entrance of the statechart's top level state denotes the initialization entrance for any newly created instance of the object, and a circled T denotes termination, with self-destruction of the instance in question. Thus, for example, a reaction attached to the initialization entrance arrow of a class's statechart will serve as an initialization script for instances of that class.

As to the behavior of the statecharts themselves, since a number of semantics have been proposed for the language we ought to make some comments. The semantics we adopt for the language here is close to the one we defined for implementation in the STATEMATE tool [HN], but there are a number of differences that are derived from the object-oriented nature of the present setup. As in [HN], reactions to events are taken on a step-by-step basis, with the events and actions generated in one transition not taking effect until the next step, after a stable situation has been reached. (A stable situation is one in which all orthogonal components are in states, and none are left lingering along transitions.) However, in STATEMATE, all triggers are constantly 'attentive', and generated events

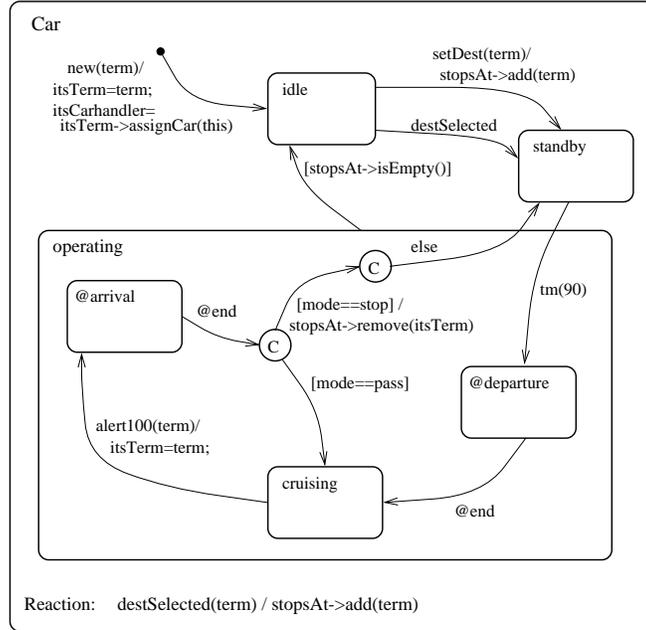


Figure 6: Top-level statechart of Car

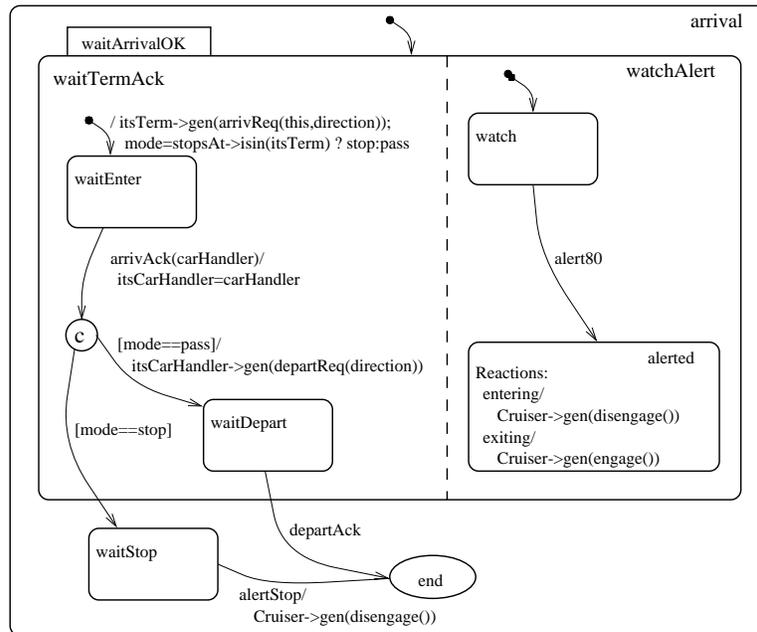


Figure 7: The arrival portion of Fig. 6

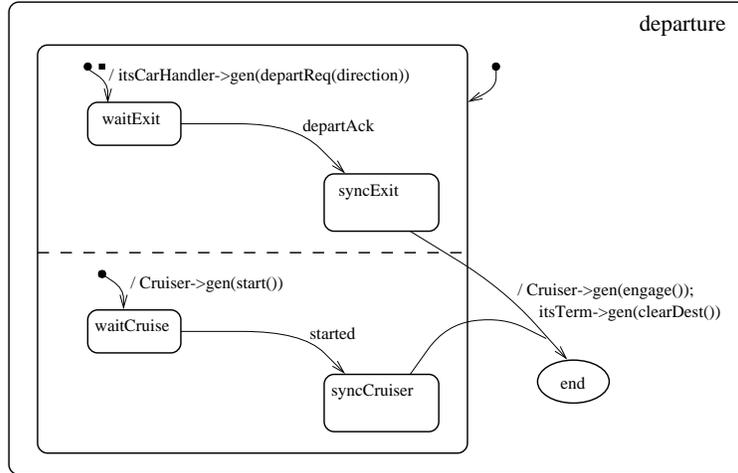


Figure 8: The `departure` portion of Fig. 6

reach their destinations instantly, implying, among other things, the need to deal with multiple simultaneous events. This is a kind of ‘zero-time’ assumption.

Here, in contrast, we have tried to set things up to form a more realistic, design/implementation framework, not just an abstract modeling one.<sup>12</sup> Thus, in the current setup, events are handed to server objects one by one from the queue, in single-event processing. Another difference is in priorities: Unlike the statecharts of STATEMATE, when an event can trigger a number of conflicting transitions we give priority here to lower level states.

However, the main difference between the way statecharts are used in a function-oriented framework such as that of STATEMATE and in the object-oriented framework proposed in this paper, is the role of the transitions. Here, events are treated in a run-to-completion manner (see, e.g., [SGW, pp. 218–219]), along transitions that can be compound (i.e., a path of adjacent arrows) and multiple (i.e., consisting of simultaneous transitions in different orthogonal components). In contrast to STATEMATE’s zero-time approach to transition execution, we require that all parts of a transition be fully executed before the statechart becomes stable and the system can respond to another event. As far as operations go, the method executed by the called object in response to an invocation must be provided in its entirety along such a transition, since once the statechart enters a stable state configuration the method terminates and the thread of control returns to the calling object’s statechart. (Of course, the method can terminate earlier, upon execution of a `reply(value)` action along the transition.) This approach to transitions is also reflected in the fact that parameters from events and operations are valid and available only during the 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.

It is worth re-emphasizing the difference between events and operations in terms of the statechart of the client object. Generating an event is something the statechart does but retains its thread of control for the remainder of the transition it is in, running it to completion until the situation stabilizes. In contrast, invoking another object’s operation freezes the statechart’s execution in

<sup>12</sup>This is why several additional features of STATEMATE have been left out of the RHAPSODY framework, such as conjunctions of events, which are harder to implement and do not seem to arise naturally when modeling with practical design in mind.

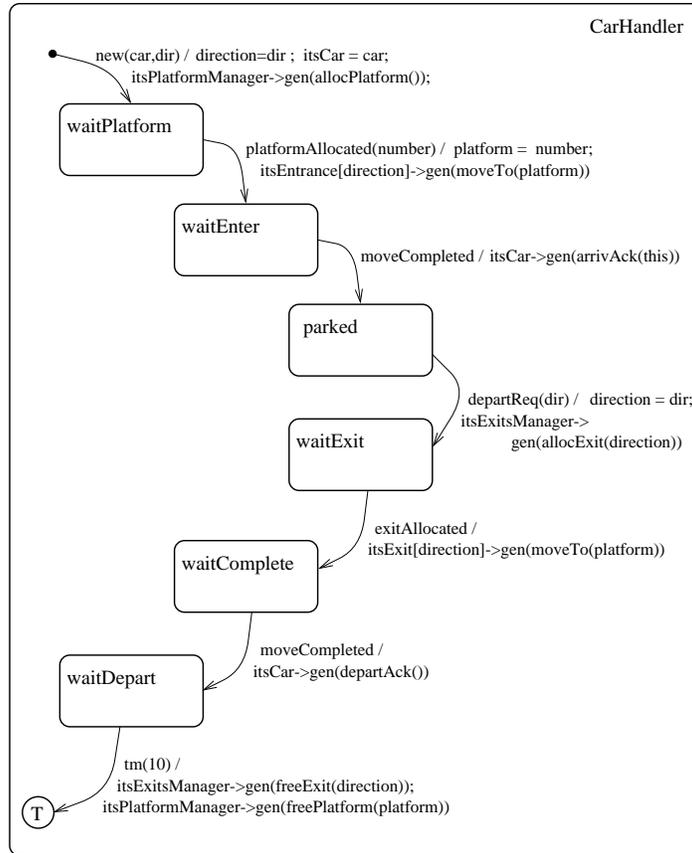


Figure 9: Statechart of CarHandler

mid-transition, and the thread of control is passed to the called object to do its thing. Clearly, this might continue, with the latter object calling others, and so on. (A cycle of calls that leads back to the same object instance is illegal, and an attempt to execute it will abort.)

## 8 Statecharts for the rail-car example

The main statecharts for the rail-car example are given in Figs. 6–10. Figs. 7 and 8 are subcharts of the statechart for `Car` given in Fig. 6. (We could have drawn these three figures as one. The sub-charts in Figs. 7 and 8 are drawn separately just for clarification, and actually should be plugged into the `@arrival` and `@departure` blobs in Fig. 6. An `@` prefixing a basic state denotes the presence of a more detailed blowup statechart.) Note the `T` icon in Fig. 9, indicating that a `CarHandler` destroys itself when its task is completed. Note also that the statecharts for `Terminal` and `ControlCenter` are modeless, containing reactions and forwarding information only.

We now walk through one of the scenarios described earlier; it helps to follow the relevant parts in the MSC of Fig. 3. The reader should be able to follow the other parts of the statecharts in a similar way quite easily. `Car` has five main modes (see Fig. 6), and assume we are in a situation where a particular car is in its `cruising` state, approaching a terminal. It leaves that state when it receives from `itsProximitySensor` (whose behavior is not described here) the event `alert100(term)`, which alerts the car that it is 100 yards from the terminal `term`. As explained

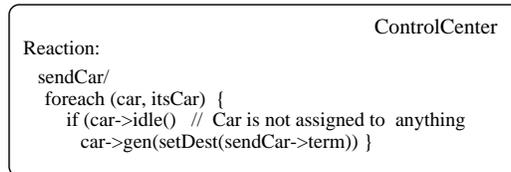
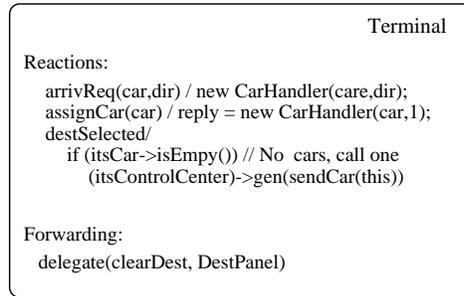


Figure 10: Two modeless statecharts

above, **Car** does not actually receive the event, but has it handed to it from the system's queue manager. Nevertheless, we shall tell the story as though 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 Fig. 6. While the real work is carried out by the left-hand orthogonal component therein, the right-hand component watches out the whole time to make 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. In the meantime, 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. (The `direction` data item is computed inside the state `standby` of **Car**, whose internal details are also omitted here.) The car also checks whether the terminal it is approaching is in the set of terminals it `stopsAt`, setting the `mode` to `stop` or `pass` accordingly.<sup>13</sup>

If we cut now to the modeless statechart of **Terminal** in Fig. 10, we see that an `arrivReq` event causes a new **CarHandler** to be instantiated, with the car's identity and its direction as parameters. Cut now to the **CarHandler** statechart in Fig. 9. It starts its life by executing its initialization script, attached to its default entrance arrow. There it saves the two parameters in variables, and proceeds to ask for a platform to be allocated. Having received confirmation of that being done and a platform `number`, which it saves in `platform`, the **CarHandler** 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, it generates the event `arrivAck` for the car to act upon, with its own identity as a parameter. The car, who waited patiently in its `waitEnter` state, instantiates the link to `itsCarHandler`, and branches off to `stop` or to make a `departReq` to its handler, depending on whether it is scheduled to make a stop at the terminal in question or simply to pass through. If it has to stop, the car waits for an `alertStop` from `itsProximitySensor`, and

<sup>13</sup>Note the reaction at the bottom of the **Car** statechart in Fig. 6, which adds a terminal to the list of scheduled stops whenever a destination is selected. The `destSelected` event can be generated by the car's `DestPanel` or by the terminal's one.

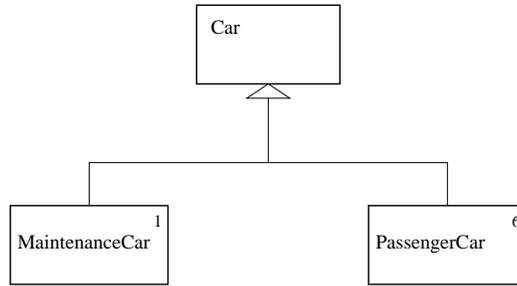


Figure 11: Two kinds of cars

then leaves its `arrival` state (and we switch back to Fig. 6), 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 it is approaching, it waits for its `departReq` to be followed by a `departAck` from its handler, and proceeds (in Fig. 6) to resume cruising. Upon receiving the `departReq`, the `CarHandler` (in Fig. 9 again) goes through a process dual to 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 and then frees the exit and platform and self-destructs.

## 9 Inheritance

Inheritance is one of the key topics in the OO paradigm. There is a large body of literature on this topic, and much of it deals with the **is-a** subtyping, or subclassing, relationship between object classes. We allow this relationship to be specified in the object-model diagram in the usual way, by the standard triangular icon on the connecting edges. Fig. 11 shows part of the object-model diagram, modified so that there are now two kinds of cars that are both subclasses of the abstract class `Car`.

But 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 *protocol conformity*, or subtyping, which roughly means that it should be possible to ‘plug in’ a  $B$  wherever an  $A$  could have been used, by requiring that  $B$ ’s protocols, i.e., what can be requested of it, are consistent with those of  $A$ . In addition, a kind of *structural conformity*, or subclassing, is softly requested, to the effect that  $B$ ’s internal structure, such as its set of composites and aggregates, is consistent with that of  $A$ .

Nevertheless, these form a weak kind of subtyping, which says little about the *behavioral conformity* of  $A$  and  $B$ . It requires only that the plugging in be possible without causing incompatibility, but nothing is guaranteed about the way  $B$  will actually operate when it replaces  $A$ . Thus we don’t have full substitutability, but merely plausibility. In fact,  $B$ ’s response to an event or an operation invocation might be totally different from  $A$ ’s. It turns out that guaranteeing full behavioral conformity between a type and its subtype is technically very difficult, and much research is still needed on this issue. Fortunately, however, behavioral conformity is too stringent in practice. Most modelers do not expect the inheritance relationship between  $A$  and  $B$  to mean that anything  $A$  can

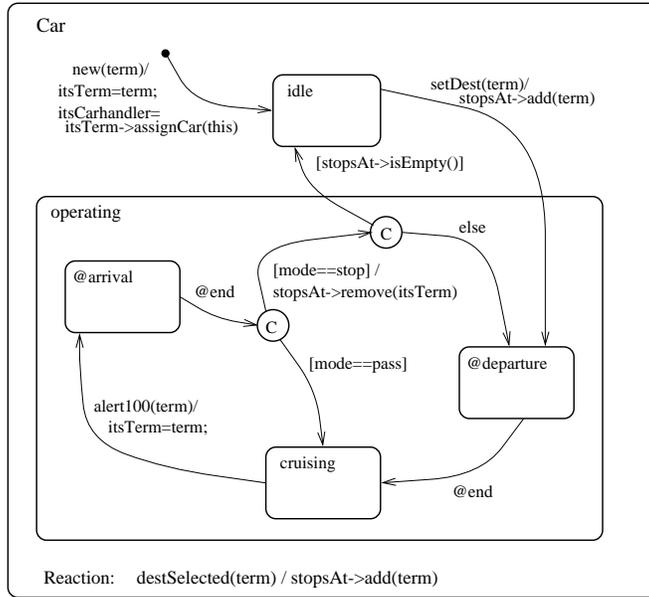


Figure 12: Statechart of the abstract class `Car`

do  $B$  can do too and in the very same way. They are satisfied with guaranteeing that anything  $A$  can do,  $B$  can be asked to do, and will look like it is doing, but it might very well do so differently and produce different results. One of the reasons for this is that, for the most part, inheritance is introduced to enable *reuse*, which is 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 paper proposes a behavior-intensive language set, which forces us to address the inheritance of behavior one way or another. The crucial issue, of course, is in the statecharts. What should the relationship be between  $A$ 's statechart and  $B$ 's, so that some kind of conformity results, and so that reuse is encouraged?

Authors who have addressed this question have felt that the modeler should somehow construct  $B$ 's statechart from  $A$ 's, with some restrictions, but their recommendations differ. For example, [CHB], [SGW], and [CD] all contain lists of such restrictions, with the recommendations of Cook and Daniels [CD] being particularly detailed. ([R<sup>+</sup>, p. 111] contains some remarks about this issue too.) It is possible to show, very easily in fact, that none of the recommended restrictions can prevent the behavior from changing radically, which means that these proposals cannot establish full behavioral conformity; and indeed they were not intended to.

We have essentially adopted this approach, but with code synthesis predominantly in mind. Thus, the restrictions described below for constructing  $B$ 's statechart from that of its parent class  $A$  were designed to be as helpful as possible when it comes to reusing parts of the code generated from  $A$  in our  $C^{++}$  implementational framework. Since we do not detail the transformation scheme here, we shall not be able to fully justify our choices in this paper.

The main guideline is to base the two statecharts on the same underlying state/transition topol-

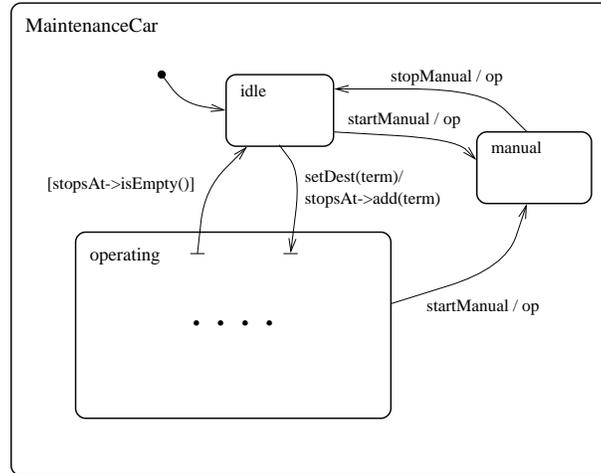


Figure 13: Statechart of `MaintenanceCar`

ogy. Thus,  $B$  inherits all  $A$ 's states and transitions, and while these cannot be removed certain refinements are allowed. States can be modified by (i) decomposing a basic (atomic) state into OR-substates or into orthogonal components, (ii) adding substates to an OR-state, and (iii) adding orthogonal components to an AND-state. As to transitions, new ones can be added to  $B$ 's statechart, and certain modifications are allowed in the original inherited ones. Specifically, the target state of an inherited transition can be changed, even to a completely different state (i.e., not necessarily to a substate of the original state, as is done in [CD]), but the source state is not to be changed. The reasons for this are, again, implementational.<sup>14</sup> In addition, if the transition is labeled by `trigger[condition]/action`, then the condition can be modified and the action can be overridden. To help highlight the difference between structural and behavioral conformity, note that although a transition is not allowed to be explicitly removed, it can be removed implicitly by making its guard false.

Let us say now that we have enhanced our object-model diagrams by the two kinds of cars, as in Fig. 11. We might then provide most of the behavior of a car in the statechart of the abstract class `Car`, as in Fig. 12. The statechart of `PassengerCar` would inherit the states and transitions of this figure, and would add the `standby` state, which, together with some additional changes, would lead to the original Fig. 6. The statechart of `MaintenanceCar` would be as in Fig. 13, including the special `manual` state, in which instructions to the engine are given directly by the driver. (In Fig. 13 we have left out many details, including some of those inherited from Fig. 11, such as the inners of the `operating` state. `RHAPSODY` enables the inherited elements to be displayed in more useful ways.)

---

<sup>14</sup>This difference between source and target is somewhat less restrictive than it sounds. We can achieve the effect of changing the source to a lower-level state by adding a new transition with the same target but the lower-level state as source. Since the semantics of our statecharts give priority to the transition leading out of the lower state, we have what we want.

## 10 Discussion

A number of research topics present themselves and seem worthy of pursuit. One of the most interesting concerns inheriting behavior. We plan to carry out a careful investigation of the various levels of behavioral conformity possible in a setup such as ours, and to address such issues as their feasibility, enforceability and computational complexity.

Another reaserach topic involves a detailed investigation of possible productive relationships between statecharts and MSC's. For example, it would be nice to be able to synthesize a first cut at the statecharts for an object model from the scenarios given in the MSC's.

A few words are in order concerning the RHAPSODY tool. Although many aspects of RHAPSODY have not been addressed in this paper, including language-related ones such as active objects and inter-object concurrency, the reader can get a pretty good feeling of its spirit by studying the language set described here, and by contemplating the dedication to executability and analysis present in its earlier sibling, STATEMATE [H<sup>+</sup>,HP]. However, it is worth mentioning that RHAPSODY addresses many methodological issues too, and in ways that are quite in line with UML and the recommendations in [B, CD, R<sup>+</sup>, SGW].<sup>15</sup> For example, the question of how to present and view overall system behavior is very important to a modeler, even though the entire system's behavior is given, in principle, by the collection of statecharts for all the object classes. Much has been said and written about this and other such topics, and RHAPSODY provides several additional features to ease the work of modelers and system analysts, such as message sequence charts.

As far as code synthesis goes, we feel that we are on the right track. It is our hope that the code generated by RHAPSODY will turn out to be useful in bringing high-level modeling closer to the desired final product.

**Acknowledgements:** We wish to thank Michal Politi and Alex Nerst, but especially Michael Hirsch, for numerous helpful discussions and ideas.

## References

- [B] Booch, G., *Object-Oriented Analysis and Design, with Applications* (2nd edn.), Benjamin/Cummings, 1994.
- [CHB] Coleman, D., F. Hayes and S. Bear, "Introducing Objectcharts, or How to Use Statecharts in Object Oriented Design", *IEEE Trans. Soft. Eng.* **18** (1992), 9–18.
- [CD] Cook, S. and J. Daniels, *Designing Object Systems: Object-Oriented Modelling with Syntropy*, Prentice Hall, New York, 1994.
- [DH] Drusinsky, D. and D. Harel, "On the Power of Bounded Concurrency I: Finite Automata", *J. Assoc. Comput. Mach.* **41** (1994), 517–539.

---

<sup>15</sup>We should also remark that in contrast to other methodologists, the authors of [SGW] have also built a tool, ObjectTime, which has similar executability and code synthesis capabilities. However, there are many differences between ObjectTime and RHAPSODY. While RHAPSODY takes a layered approach which allows ellaboration, ObjecTime is based on a closed "translative" approach. Also, ObjectTime does not allow concurrency in the statecharts.

- [H1] Harel, D., “Statecharts: A Visual Formalism for Complex Systems”, *Sci. Comput. Prog.* **8** (1987), 231–274. (Preliminary version appeared as Tech. Report CS84-05, The Weizmann Institute of Science, Rehovot, Israel, Feb. 1984.)
- [H2] Harel, D., “On Visual Formalisms”, *Comm. ACM* **31** (1988), 514-530.
- [H3] Harel, D., “Biting the Silver Bullet: Toward a Brighter Future for System Development”, *Computer* (Jan. 1992), 8–20.
- [H<sup>+</sup>] Harel, D., H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot, “STATEMATE: A Working Environment for the Development of Complex Reactive Systems”, *IEEE Trans. Soft. Eng.* **16** (1990), 403–414. (Preliminary version appeared in *Proc. 10th Int. Conf. Soft. Eng.*, IEEE Press, New York, 1988, pp. 396–406.)
- [HN] Harel, D. and A. Naamad, “The STATEMATE Semantics of Statecharts”, submitted for publication. (Revised version of “The Semantics of Statecharts”, Tech. Report, i-Logix, Inc., 1989.)
- [HP] Harel, D. and M. Politi, *Modeling Reactive Systems with Statecharts*, in preparation. (Early version titled *The Languages of STATEMATE*, Tech. Report, i-Logix, Inc. (250 pp.), 1991.)
- [J] Jacobson, I., *Object-Oriented Software Engineering: A Use Case Driven Approach*, ACM Press/Addison-Wesley, 1992.
- [Ra] Rational Corp., Documents on UML (the Unified Modeling Language), Version 1.0 (<http://www.rational.com/ot/uml/1.0/>), 1996.
- [R<sup>+</sup>] Rumbaugh, J., M. Blaha, W. Premerlani, F. Eddy and W. Lorensen, *Object-Oriented Modeling and Design*, Prentice Hall, 1991.
- [SGW] Selic, B., G. Gullekson and P. T. Ward, *Real-Time Object-Oriented Modeling*, John Wiley & Sons, New York, 1994.
- [SM] Shlaer, S. and S. J. Mellor, *Object Lifecycles: Modeling the World in States*, Yourdon Press, 1992.