Executable Object Modeling with Statecharts

David Harel
The Weizmann Inst. of Science
Rehovot 76100, Israel
harel@wisdom.weizmann.ac.il

Eran Gery
i-Logix Israel, Ltd.
Rehovot 76327, Israel
erang@ilogix.co.il

Abstract

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. The goal is for models to be intuitive and well-structured, yet fully executable and analyzable, enabling automatic synthesis of usable and efficient code in object-oriented languages such as C^{++}. At the heart of the modeling method is the language of statecharts *for specifying object behavior, and a hierarchical OMT-like language for describing the structure of classes and their inter-relationships, that we call O-charts. Objects can interact by event generation, or by direct invocation of operations. In the interest of keeping the exposition manageable, we leave out some technically involved topics, such as multiple-thread concurrency and active objects, which will be described elsewhere.*

1 Introduction

The language of statecharts was conceived of in 1983. In the 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.

About a decade ago, an attempt was made to build a full language set around statecharts, adhering to the function-based structured analysis paradigm (SA); see [H$^+$]. Statecharts, used for behavioral description, were closely integrated with a hierarchical language for functional decomposition and data-flow, called activity-charts.[1] Since SA methods are considered to be lacking when it comes to certain aspects of system development, such as transition to design and reuse, many people recommend complementing function-based approaches with ones that are object-oriented (OO). This change is one of the most significant in software engineering in recent years. Accordingly, we have embarked on an effort to develop an set of languages for object modeling, built around statecharts, and to construct a supporting tool with full executability and code synthesis capabilities.

In this paper, we describe our appraoch to modeling the structural properties of classes in a clear hierarchical manner, and integrating the resulting description with a precise specification of the behavior using statecharts. Since classes represent dynamically changing collections of concrete objects (instances of classes), and since the structure itself is dynamically changing, the model must address issues such as the initialization of, and the reference to, real object instances, the hierarchical delegation of messages, the creation and destruction of instances, the initialization, modification and maintenance of links representing association relationships, etc. Obviously, we must also address inheritance. All this makes the problem of combining structure and behavior much harder than in an SA-based approach. 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.

2 Background and overview

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$^+$, 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-machine formalism is usually adopted for specifying behavior, and all of the methodologies listed above recommend statecharts (or some sublanguage thereof) for this.

Our approach involves two constructive modeling languages[2] — *O-charts* and *statecharts*. O-charts

[1] A third language, module-charts, was used to specify physical decomposition. See [H3] for the motivation and 'philosophical' aspects, and [HP] for a full description of these languages. This language set underlies the STATEMATE tool [H$^+$], built with executability, analysis and code-generation in mind.

[2] A language is constructive if it contributes to the dynamic

specify the structure of the system, by identifying classes of objects (i.e., object types) and their multiplicities, as well as their inter-associations and roles. Especially noteworthy in O-charts is the hierarchical depiction of a strong form of aggregation, using higraph-like encapsulation [H2]. O-charts also depict subtyping and inheritance. As far as this structural view of a system goes, the ideas and notations of OMT [R+, R] are gaining popularity among modelers, and as a consequence we have designed O-charts to be as consistent with these parts of OMT as possible. For example, our hierarchical aggregation mechanism is very similar to the composite objects of [R], but it also carries significant semantic weight.[3]

The behavior of a class in an O-chart is specified by a *controlling statechart.* These statecharts are the driving force behind the entire model, as they are used to specify all behavioral aspects of objects. They capture not only the state of the object in terms of its willingness, as a server, to respond to events or requests for services, but also the dynamics of its internal behavior in carrying out those responses, and in maintaining its relationships, as a client (or aggregate), with other objects. For all of this we utilize the full power of the statecharts language of [H1], including state hierarchy, multi-level orthogonal components, and (internal) broadcast communication.

One of the main technical issues that arise in devising the setup, concerns the choice of mechanisms to be used for inter-object interaction. There is a tradeoff between high-level mechanisms that are easier to model, and lower-level ones that are easier to translate into efficient code. We have tried to compromise, adopting a two-kind approach — *events* and *operations* — and in the process make careful distinctions between them. An object can generate an event, which is queued, to be handed to the target server object in its turn, and an object can also directly invoke an operation of another object, thus causing it to carry out an appropriate method, and perhaps return a value. Interestingly, one of the upshots of our hierarchical modeling of structure is that these interactions can be arranged to take on the form of either direct communication or broadcast.

As mentioned above, we concentrate on a single-thread approach to 'event-driven' concurrency here, which makes the exposition somewhat easier. Nevertheless, there are many subtle issues that arise in defining the semantics, and we shall touch upon some of them as we proceed. We have addressed active objects and multiple-thread concurrency too, but have decided to leave their exposition out of the present paper.

While the paper's presentation of both the syntax and the semantics of the languages is informal (for clarity) and not totally exhaustive (for brevity), the reader should keep in mind that we have a full-fledged definition. This definition is reflected in an algorithm that translates any syntactically legal model into executable code, and which is being implemented as part of the support tool we have constructed, called O-MATE. The current target language of the O-MATE code synthesis is C++, but other languages will no doubt follow. The details of the O-MATE system will also be described separately.

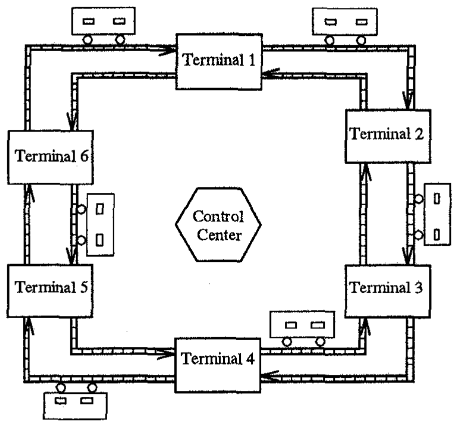


Figure 1: The rail-car system

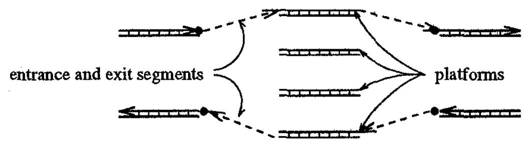


Figure 2: Inside a terminal

3 The rail-car example

We illustrate the approach using an automated rail-car system example (see Fig. 1). 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 parking bays, 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

semantics of the model. That is, its constructs contain information needed in executing the model or in translating it into executable code. Other languages can be used by the system modeler to capture parts of the thinking that go into building the model, or to derive and present views of the model to aid in analysis, but we shall not discuss these here. Excellent proposals for such accessory languages can be found in object-oriented methodology books, such as [B, CD, R+, SGW].

[3]Parts of our approach are quite consistent with the methods described in [B] and [CD] too. In particular, Cook and Daniels [CD] address some of the definitional issues relevant to this paper, and while the two approaches show some similarities, there are many differences, including the fact that [CD] does not support hierarchical specification of structure.

to link to any one of the four bays. 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 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.

4 O-charts: classes and their affinities

O-charts specify classes of objects and their structural relationships. An O-chart is an ER-like diagram that is very similar to an OMT object model [R$^+$]. It is enriched with higraph-like [H2] encapsulation, that denotes *strong aggregation* via composite classes similar to those of [R]. Directed edges represent relationships, with an undirected edge abbreviating a two-way directed edge. Classes and relationships can have associated multiplicity information of the standard kinds available in many object-oriented methodologies. For consistency with the popular versions of OMT and other notations for class structure, we also allow a weaker kind of aggregation, represented as in [B, R$^+$] and elsewhere by branching arrows with a diamond-shaped icon. However, the examples in this paper do not contain weak aggregation, and their semantics is indeed much weaker — essentially just that of a special association relationship, called `part-of`, between the aggregate and its components.

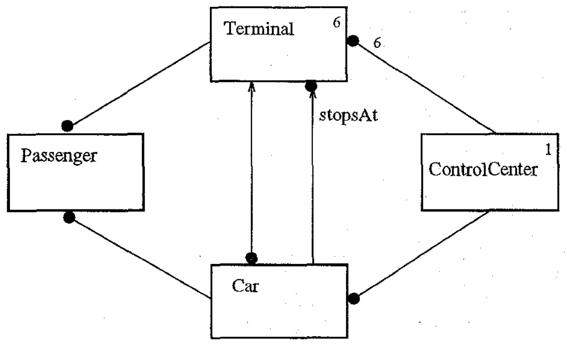


Figure 3: High-level O-chart for the rail-car system

Two comments are in order here, regarding O-charts. First, in the paper we use a rather degenerate method of presenting information about the classes in an O-chart. For example, we do not show the list of methods supported by an object of a given class. The O-MATE tool is much more flexible, in the spirit of the well-known methods of [B], [R$^+$] and others. Second, we went through a lengthy period of internal deliberations as to whether we should have separate languages for classes and concrete objects. There are obvious advantages to having the class model and instance model together, mainly in way of compactness and comprehension of the representation. However, there are also strong reasons to separate them. For example, a separate language for concrete object instances can provide more flexibility in the visual description of nontrivial issues of multiplicity, object creation and reference. If the general properties of all instances of a class have to be described in the same language as the instances themselves, the descriptive abilities become restricted. Nevertheless, we have decided to merge the two, and our O-charts thus describe classes, but also provide information on instances thereof. (In an earlier version of this paper, made available in technical report form in 1994, we decided otherwise: there was a language we called C-charts that described classes and another one called O-charts that described instances. The O-charts presented here combine both.)

Fig. 3 shows a partial O-chart for the rail-

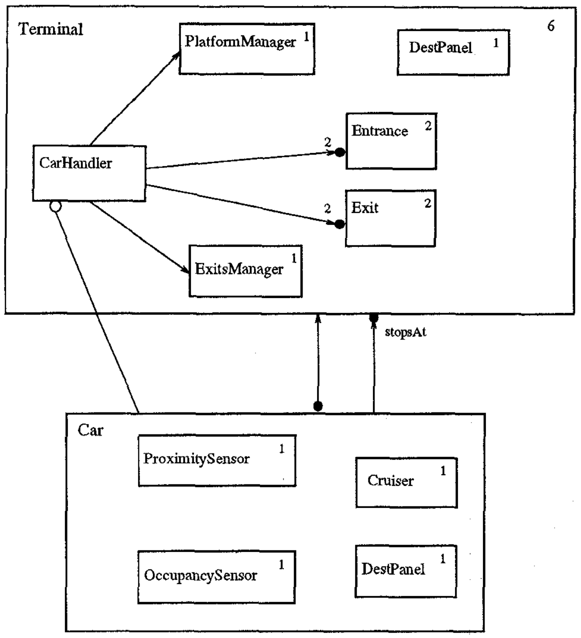


Figure 4: More of the rail-car system's O-chart

car system.[4] 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 **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. 3, 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 can use the 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. 4 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 (without the need for **its**).

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. 3. Had we wanted it to appear there too, we could have represented it in by an edge leading from **Car** to a stubbed end lying within the interior of **Terminal**.

5 Instances and actual links

O-charts describe classes and their structure, and as such, they appear to concentrate on static aspects only. However, once alive and kicking, an object-oriented 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 initialized. Dynamics concerns the way the model rolls along, and consists of two different kinds of changes that can take place in the status of the model: (i) step-by-step reactions to triggering occurrences like events and calls for operations, and (ii) changes in the structure of the model, 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 O-chart, as we now explain. The first thing to happen when a model starts executing is the initial creation of instances from classes that have a fixed in-

[4] Conceptually, there is a single O-chart 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.

teger multiplicity.[5] This happens recursively down the tree of strong aggregation, that is, from a composite class to its components repeatedly, and instances are spawned relative to each instance of a composite class. Thus, referring to Figs. 3 and 4, 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`. Classes with unspecified multiplicity will spawn 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. 3. It is many-one, but the multiplicities on either end of the edge match those of the associated classes, implying exactly 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.[6]

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 O-MATE has some flexibility.[7]

These rules for setting up relationships also extend to the dynamics at large: throughout the life of the system, whenever objects are instantiated or destroyed, all relevant associations are evaluated anew and are 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 become unambiguous, it will be able to refer to `itsEntrance[1]` and `itsEntrance[2]`, and similarly for the `Exits`.

During the ongoing behavior of the model, changes may occur in the current set of instances and their links. These can be prescribed by four 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 object:

```
new classname(parameters)
delete objectname
```

The other two are 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. 5), 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, a comment is in order, regarding our nongraphical syntax. The reader will have no doubt noticed that the actions are written in a C^{++} format. This might not seem to deserve justification, given the status of C^{++} as an object-oriented programming language, and the fact that it is the current target language for the code synthesis of O-MATE. However, the decision to use C^{++} as our action language is, in a way, an arbitrary one that bears little relationship to the ideas of the paper. As a truly practical matter, we had to decide on an implementation framework for the language set proposed here, and we chose one based on C^{++}. We will not discuss the implementation here, although there are many interesting issues that it raises, which form the technical underpinnings of the O-MATE system. We could have chosen a framework based on another language, such as Ada or a set-based language (as is done, e.g., in [CD]), which would have resulted in different-looking elements in our action language. For example, navigation expressions might have been written using dots as delimiters, rather than arrows, and keywords such as `add` and `new` might have been replaced by different ones. All this would not have changed anything

[5] 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.

[6] 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.

[7] Not all ambiguous topologies are hoplessly impossible to initialize. Some have quite reasonable solutions, and O-MATE proposes these to a user. For example, two classes with common multiplicity of n, and a one-to-one symmetric association between them, can be set up as n disjoint pairs of linked instances.

of any 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, 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 O-chart, 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 strong or weak aggregation. In any case, denoting such a reference generically by objectname, we write this in our chosen framework as:

```
objectname->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, the next step is nondeterministic, which a tool executing the model should warn the user of, and could be programmed to resolve by randomization, if so desired.

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? 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. 9. 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 strong aggregation, i.e., from composite class to its components, using the components' own forwarding specs.

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. 9.

Events are themselves entities of the model, and can be organized into an event generalization/specialization hierarchy. The syntactic way this hierarchy is described is unimportant here, but any of the standard ways this is done in the books on object-oriented methodologies will do.[8] 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 this kind of event-based communication mechanism in an object-oriented modeling method. These 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. 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 directly causing 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 similar to that of events, without the **gen**, and the called object's name must appear, thus:

```
objectname->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 different. Here the thread of control is passed immediately to the called object, which in turn carries out the method currently relevant to the operation without delay. The calling object's progress is frozen until the method is completed, at which point it picks up the thread and resumes its work. A method terminates with the called object reaching a stable situation, or with it returning a value by assigning a value to the special designated variable `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.

[8]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.

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

7 Statecharts as object controllers

The behavior of an object is specified by the controlling 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.

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 when an object is willing to accept requests, leaving its internal method of dealing with those requests to be specified in code or by other means. Others discard orthogonality (concurrent states), claiming that concurrency is already globally inherent in an object-oriented system. Yet others discard the broadcast communication of statecharts, claiming that communication is to be carried out only via messages flowing directly between objects. Our adoption of statecharts is lock, stock and barrel. First, the two kinds of concurrency are quite different. Orthogonality in statecharts is not intended 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, 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 we believe they are far more evident in large models.

On a technical level, statecharts involve reactions

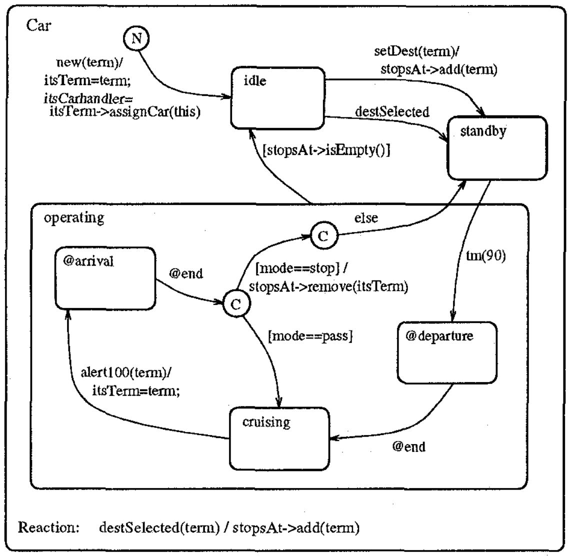


Figure 5: Top-level statechart of `Car`

of the form:

`trigger[condition]/action`

all 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^{++}-style statements that we do not describe in full here. Conditions are also taken from C^{++}, and, again, this is part of the (arbitrary) decision to use C^{++} style 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].

A special circled **`N`** in the statechart of a class denotes the initialization entrance for any newly created instance thereof, 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 there have been many semantics proposed for the language, some points ought to be made. 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. As in [HN], reactions to events are taken on a step-by-step

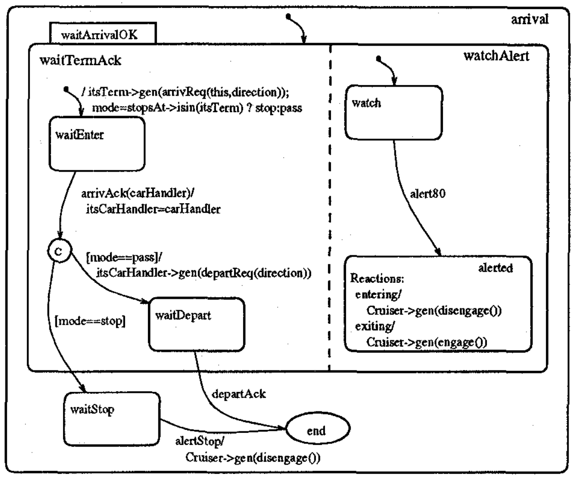


Figure 6: The **`arrival`** portion of Fig. 5

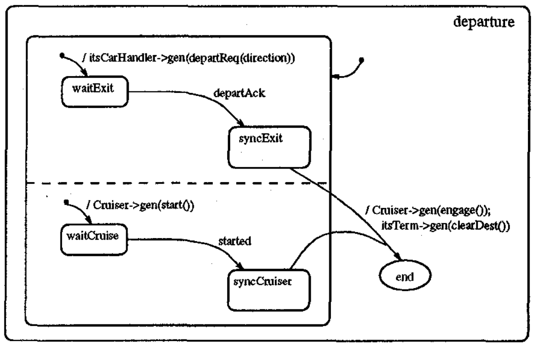

departure
/ itsCarHandler->gen(departReq(direction))
waitExit
departAck
syncExit
/ Cruiser->gen(start())
waitCruise
started
syncCruiser
/ Cruiser->gen(engage());
itsTerm->gen(clearDest())
end

Figure 7: The **`departure`** portion of Fig. 5

basis, with the events and actions generated in one transition not taking effect until the next step, after a stable situation has been reached, i.e., 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 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.[9] 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

[9]This is why several additional features of STATEMATE have been left out of the O-MATE framework, such as conjunctions of events, which are harder to implement and do not seem to rise naturally when modeling with practical design in mind.

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 both events and operations are valid 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 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. 5–9. Figs. 6 and 7 are subcharts of the statechart for `Car` given in Fig. 5. (We could have drawn these three figures as one. The subcharts in Figs. 6 and 7 are drawn separately just for clarification, and actually should be plugged into the `@arrival` and `@departure` blobs in Fig. 5. An `@` prefixing a basic state denotes the presence of a more detailed blowup statechart.) Note the `N` and `T` icons that indicate that a `Car` can get created, and that a `CarHandler` can get created and can destroy itself. The statecharts for `Terminal` and `ControlCenter` are modeless, containing reactions and forwarding information only.

We now walk through one of the scenarios described in Section 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. 5), and assume we are in a situation where

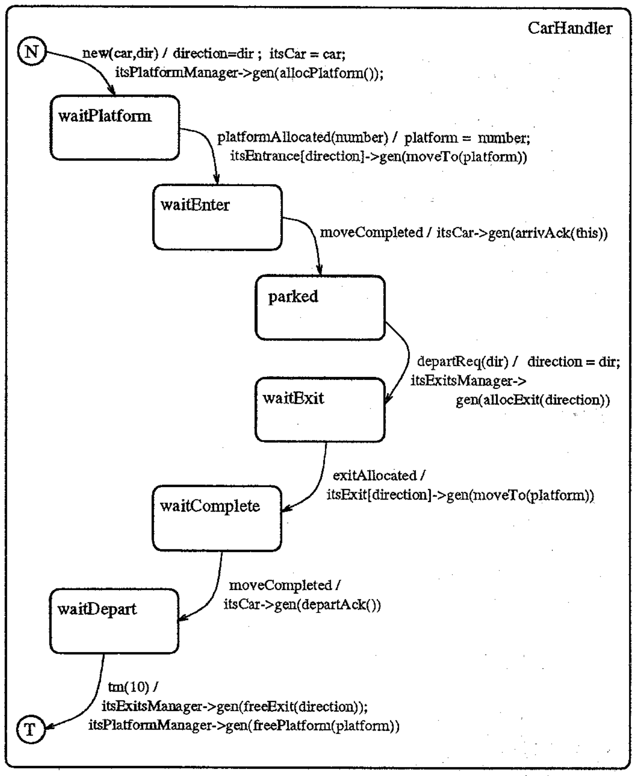
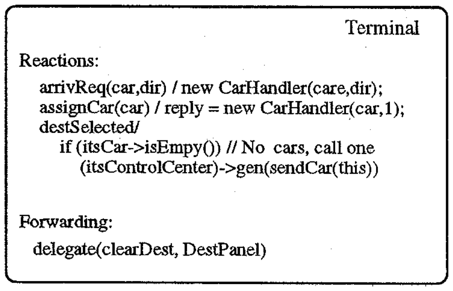
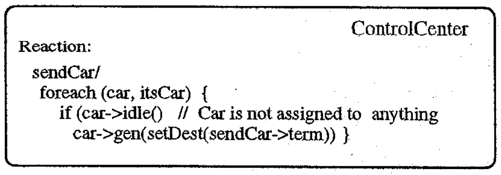

CarHandler
N
new(car,dir) / direction=dir ; itsCar = car;
itsPlatformManager->gen(allocPlatform());
waitPlatform
platformAllocated(number) / platform = number;
itsEntrance[direction]->gen(moveTo(platform))
waitEnter
moveCompleted / itsCar->gen(arrivAck(this))
parked
departReq(dir) / direction = dir;
itsExitsManager->
gen(allocExit(direction))
waitExit
exitAllocated /
itsExit[direction]->gen(moveTo(platform))
waitComplete
moveCompleted /
itsCar->gen(departAck())
waitDepart
tm(10) /
itsExitsManager->gen(freeExit(direction));
itsPlatformManager->gen(freePlatform(platform))
T

Figure 8: Statechart of `CarHandler`



```
Reactions:
  arrivReq(car,dir) / new CarHandler(care,dir);
  assignCar(car) / reply = new CarHandler(car,1);
  destSelected/
     if (itsCar->isEmpy()) // No cars, call one
        (itsControlCenter)->gen(sendCar(this))

Forwarding:
  delegate(clearDest, DestPanel)
```



```
Reaction:
 sendCar/
  foreach (car, itsCar) {
     if (car->idle()  // Car is not assigned to anything
        car->gen(setDest(sendCar->term)) }
```

Figure 9: Two modeless statecharts

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 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.[10]

If we cut now to the modeless statechart of **Terminal** in Fig. 9, 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. 8. It starts its life by executing its initialization script, attached to its **N** icon. 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 then leaves its **arrival** state (and we switch back to Fig. 5), 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. 5) to resume cruising. Upon receiving the **departReq**, the **CarHandler** (in Fig. 8 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.

[10]Note the reaction at the bottom of the **Car** statechart in Fig. 5, 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.

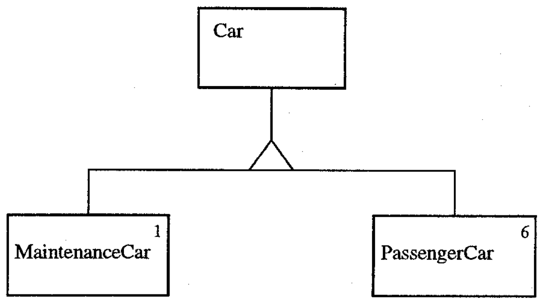


Figure 10: Two kinds of cars

9 Inheritance

Inheritance is one of the most interesting issues arising 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 O-chart in the usual way, by the standard triangular icon on the connecting edges. Fig. 10 shows part of the O-chart, 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*, 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 are consistent with those of A. In addition, we require a kind of *structural conformity*, to the effect that B's internal structure, such as its set of aggregate objects, 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 'shorting the circuit', but 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. 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 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

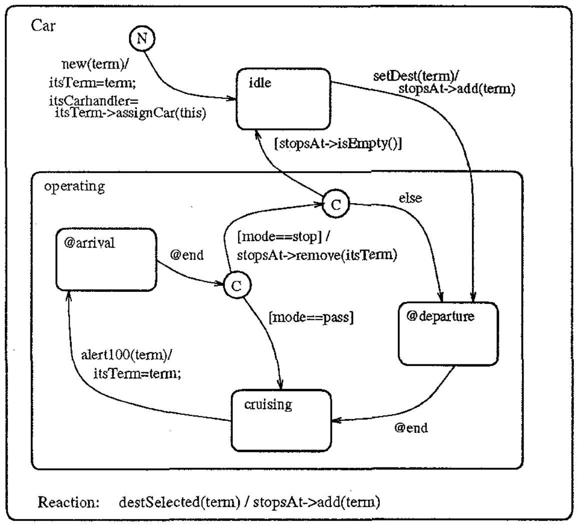


Figure 11: Statechart of the abstract class `Car`

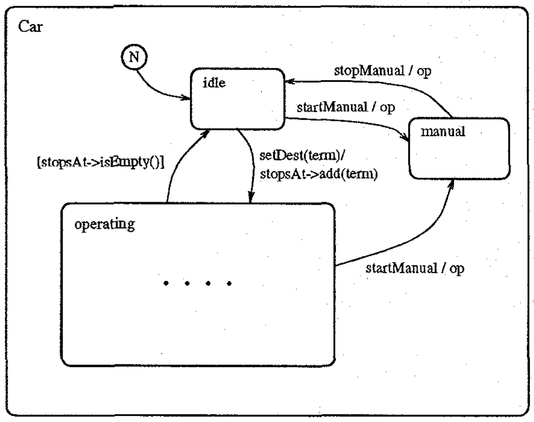


Figure 12: Statechart of `MaintenanceCar`

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 the controlling statechart. 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$^+$, 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 topology. Thus, B inherits all A's states and transitions, and these cannot be removed, but 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.[11] 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 O-charts by the two kinds of cars, as in Fig. 10. We might then provide most of the behavior of a car in the statechart of the abstract class `Car`, as in Fig. 11. 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. 5. The statechart of `MaintenanceCar` would be as in Fig. 12, including the special **`manual`** state, in which instructions to the engine are given directly by the driver. (In Fig.

[11]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.

12 we have left out many details, including some of those inherited from Fig. 10, such as the inners of the `operating` state. O-MATE enables the inherited elements to be displayed in more useful ways.)

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 complexity.

A few words are in order concerning the O-MATE tool. Although many aspects of O-MATE 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 older sibling, STATEMATE. However, it is worth mentioning that O-MATE addresses many methodological issues too, and in ways that are quite in line with the recommendations in [B, CD, R$^+$, SGW]. 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 O-MATE provides several additional languages and features to ease the work of modelers and system's analysts. An example are message sequence charts, which can be employed by an O-MATE user to help specify behavior. Although these are non-constructive charts, and do not constitute part of the formal model, O-MATE provides ways to link the information in them to the model and to check consistency between the two.

As far as code synthesis goes, while the jury is far from being in with regards to its utility, we feel that we are on the right track. It is our hope that the code generated by O-MATE 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.

[H1] Harel, D., "Statecharts: A Visual Formalism for Complex Systems", *Sci. Comput. Prog.* **8** (1988), 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** (1987), 514-530.

[H3] Harel, D., "Biting the Silver Bullet: Toward a Brighter Future for System Development", *Computer* (Jan. 1992), 8–20.

[H$^+$] 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.

[R] Rumbaugh, J., "Building boxes: Composite objects", *J. Obj. Orient. Prog.* (Oct. 1994), 12–22.

[R$^+$] 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.