Graphical Specification of Object Oriented Systems

Stephen Bear, Phillip Allen
Derek Coleman, Fiona Hayes

Hewlett Packard Laboratories, Bristol.

Abstract

The graphical notation *Objectcharts*, introduced in this paper, allows a developer to precisely specify the behaviour of object classes and to reason about the behaviour of particular configurations of objects.

Objectcharts combine object oriented analysis and design techniques and Harel's statecharts to give a diagrammatic specification technique for object oriented systems.

1 Introduction

The use of objects provides a flexible and productive approach to software development. However it is difficult to abstractly describe the structure and behaviour of object systems. In practice the lack of a "big picture" description means that it can be difficult to control the development process and to provide documentation for maintenance.

In this paper we present graphical techniques for representing system structure and behaviour. Structure is described by *configuration diagrams* which show class instances and their communication. *Objectcharts* define the behaviour of classes as extended state machines.

Jackson [4] points out that there are two complementary views of a system. The data view regards a system as operations acting on states. The process view focuses on sequences of events. Objectcharts use statecharts [2] in order to capture the process, or "lifecycle", view of object classes and firing-post conditions to define how class methods affect state.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1990 ACM 089791-411-2/90/0010-0028...$1.50

Objectcharts describe system components whilst the configuration diagram defines the system structure. From these descriptions it is possible to reason about overall system behaviour.

The paper first reviews our basic notions of object oriented software. Configuration diagrams and Objectcharts are developed through the use of a simple example in section three. Section four is more formal and outlines a trace semantics for reasoning about system behaviour.

2 Objects and Classes

An *object* has encapsulated state that persists over time. Statically, an object is characterised by the set of services that it provides and the set of services that it requires of other objects; its behaviour is characterised by the set of possible sequences of service requests which it could receive or generate.

In order to request a service a client must identify the object to perform it. Thus every object has a unique identifier that allows it to be referenced unambiguously.

A *service* is an abstraction of a method in object oriented programming, or a task entry in Ada. Each service has a signature which determines the number and type of the service arguments. The set of signatures of the provided services constitutes the *client interface* of the object. Similarly the *required interface* of an object comprises the signatures of its required services.

An *attribute* is a component of the state of an object. Encapsulation means that outside an object

the values of its attributes may only be changed or observed through the use of services. The objects that are used by a given object may be thought of as object valued attributes.

A *class* is a template for objects. The interfaces of an object and its behaviour are derived from its class *and* the environment of other objects with which it will interact.

These notions of object and class are quite conventional. They do not commit the design to a particular implementation language; in particular objects may be mapped onto passive encapsulated data types or active processes.

3 System Descriptions

In order to specify an object oriented system a number of views are required. We describe the objects in a system and their intercommunication by means of a *configuration diagram* (Section 3.1). Classes are specified by means of *Objectcharts* (Section 3.2). The overall behaviour of a system of objects is defined by the configuration diagram and the set of Objectcharts (Section 4).

3.1 Configuration Diagram

An object presents a set of services which can be used by other objects in its environment. The set of these services forms the *provided* interface of the object. The *required* interface of an object is the set of services that it needs other objects to provide.

A configuration diagram shows the objects in a system and their intercommunication. Objects are represented by boxes containing their instance identifiers and the class to which they belong, solid lines represent provided services and dashed lines represent required services. A solid line joining a dashed line shows possible communication between objects in the system. A trailing line shows possible communication between an object and the environment of the system. Configuration diagrams are similar to the Object Communication Model used in Shlaer and Mellor's object oriented systems analysis [5].

As an example, consider an alarm clock application which uses a window system. When the alarm 'rings' it opens a window. When the alarm is not ringing the window is closed (iconised). There are three objects: an alarm, a bell and a clock as seen in figure 1. The alarm, belonging to class Alarm Clock, allows the alarm to be set or cancelled. Once ringing, the alarm can be stopped. The alarm also shows the time of day and, if set, the alarm time. The alarm can open and close the bell window. The alarm (periodically) requests the clock, belonging to the class System Clock, for the actual time. When the alarm time is reached, the alarm causes the bell window to open. If the alarm is not stopped, the bell window is closed after a fixed duration.

The object configuration diagram partially defines the overall pattern of communication in the system. It shows the interfaces for each object and defines which objects request services from others. The object configuration diagram does not define particular control or scheduling structures. For example, it does not determine which objects are to be implemented as active objects (e.g. Ada tasks), nor which are to be implemented as passive objects.

Consider the alarm clock example described above. We know that the alarm object requires services from the bell object and the clock object, and not the other way around. We also know that the alarm provides a number of services: set, cancel, etc to an undefined environment (or main program).

One way to implement this configuration would be to make the alarm an active object communicating with an active environment and passive window and system clock.

Another approach would be to make all the objects passive. If the interface provided by the clock object was extended, then the environment could periodically prompt the alarm to send service requests to the other objects.

3.2 Extending Statecharts to Specify Object Classes

In this section we introduce Objectcharts incrementally via the alarm clock example. First we show how statecharts can capture some aspects of class behaviour. We then extend statecharts by augmenting states with attribute information. Objectcharts are extended statecharts in which the effect of state transitions on attributes are specified.

Objects have a lifecycle in which they change

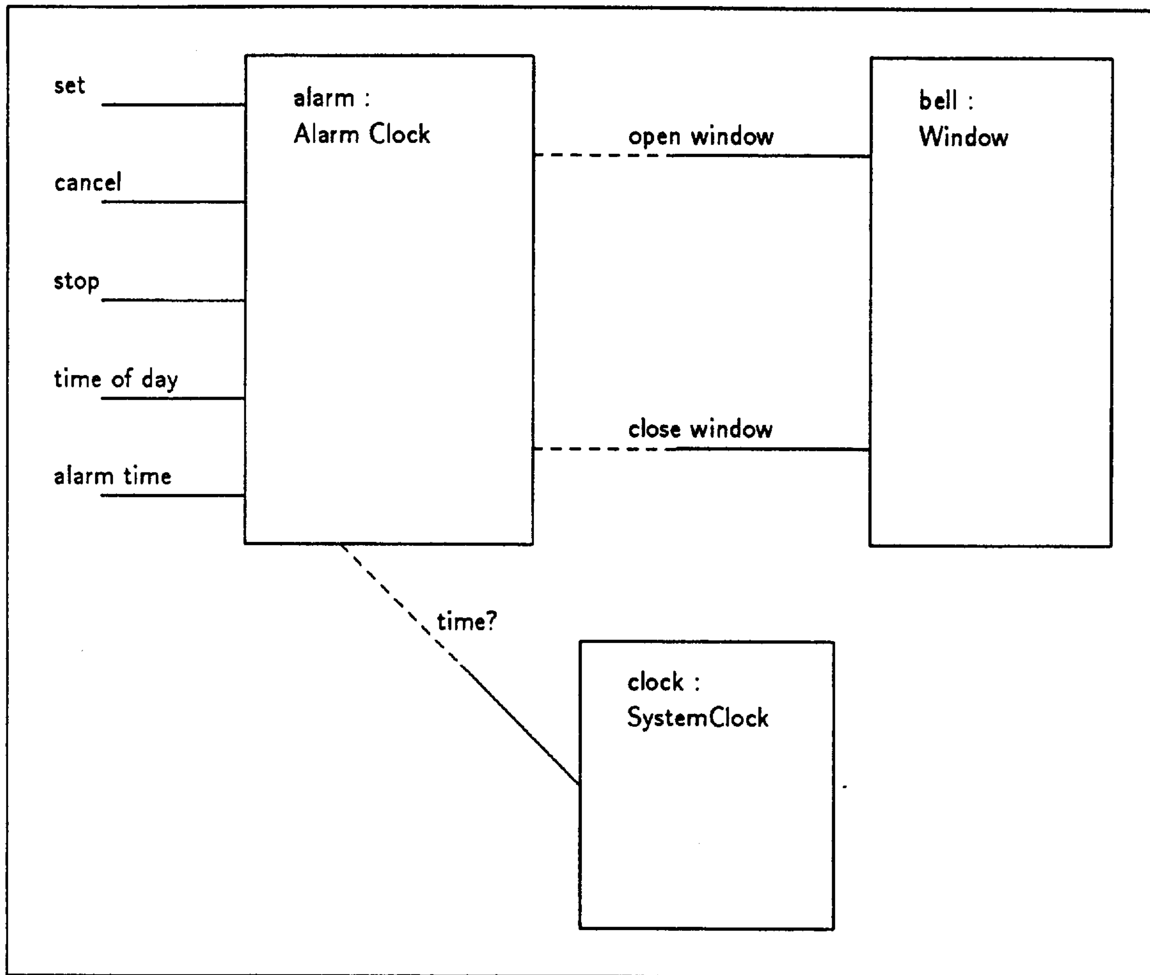

Figure 1: Alarm Clock Configuration Diagram

state as a result of providing services for clients and requiring services from other objects. The lifecycle behaviour of an object class may be expressed as a statechart.

The states of the statechart represent the various stages that an object of a class may go through. The transitions are labelled with either state changing services provided by the class or services required of other objects.

A service required of another object is prefixed by a formal name for an object providing the service. The name of the actual object is provided by the configuration diagram. If two services requests are prefixed by the same formal name then they will be sent to the same object. If two service request are prefixed by different formal names then, depending on the actual configuration, they may—or may not—be sent to different objects.

In the example the provided services *set* and *cancel* can change the state of an Alarm Clock. This gives the statechart in figure 2 which approximates the lifecycle of the instances of the Alarm Clock class.

The alarm is initially off and can be set. Once set it may be cancelled. The transition on the timeupdate state indicates that the required service *C.time?* is requested approximately every second. (In the statechart we write */C.time?*. The prefix '/' is just a reminder that *C.time?* is a required service.)

The Alarm Clock is a Harel AND-composition, so time polling happens irrespective of whether the

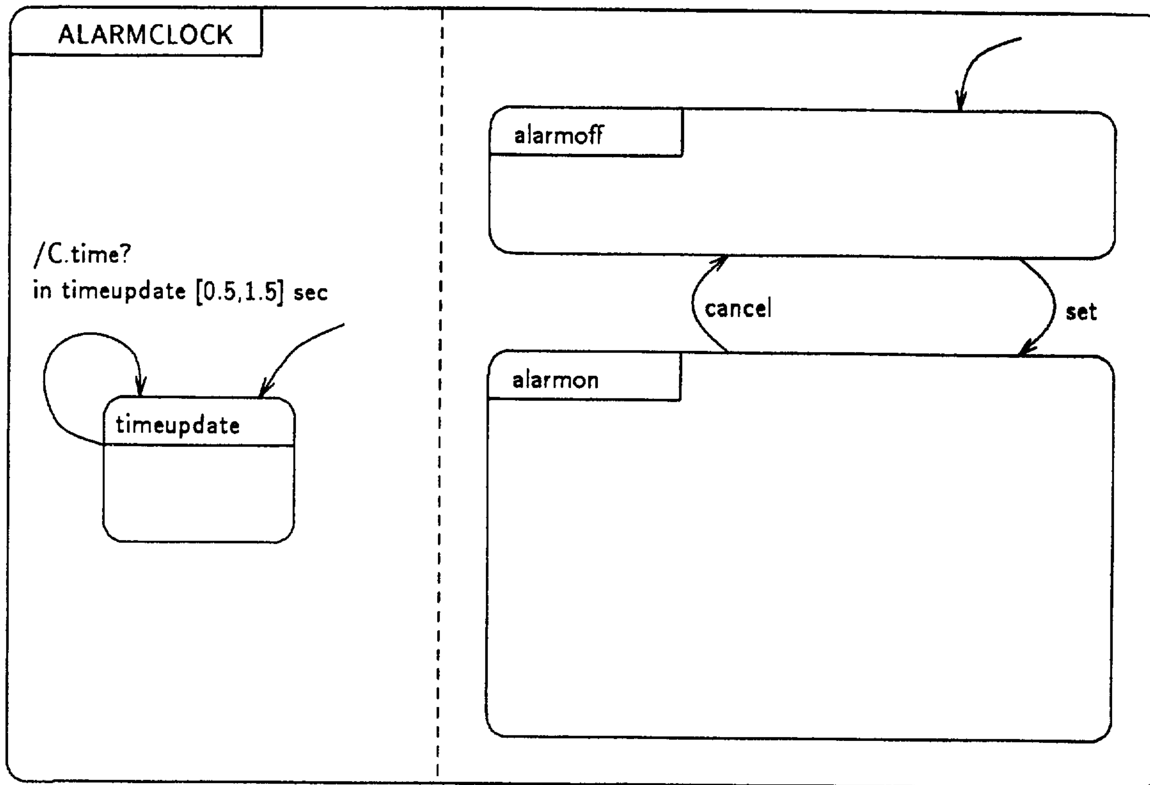

Figure 2: Alarm Clock statechart

alarm is on or off. The alarm-on state can be refined to show how an Alarm Clock interacts with other objects.

In figure 3, the alarmon state is refined into two states, quiet and ringing. In order to move from quiet to ringing the openwindow service must be requested from some object *W*. In the configuration of the example, *W* is the bell window. The transition only occurs when the alarmtime is reached, but we cannot specify that yet.

In the ringing state the window is closed after a fixed duration or when the user stops it. The alarm must be in the quiet state before an alarm setting can be cancelled.

Not all services change the state of an object, some just report on the value of attributes. These services are called *observers*. In general requests for observers, like any service, may not be allowed in all states. For example we can only request the top of a stack when it is non-empty. We enhance statecharts to carry more information by adding the name and types of the allowable observers to each state of the statechart.

We also add the necessary arguments to services; for example *set(t:time)* indicates the time, *t*, at which the alarm must ring. Parameters are partitioned into input and output parameters. Input parameters of a service may only be read by the object providing the service. Output parameters may only be updated by the providing object. We indicate all the output parameters of a service by listing them after a "|". Thus *C.time?(| t:time)* has an output parameter, *t*, and the value of *t* is set by the object *C* that provides the service *time?*.

Not all the attributes of an object need be visible. However a complete specification may be impossible without them, so we treat hidden attributes like observers and add them to states, but indicate that they are hidden by writing them in square brackets.

The enhanced statechart for the alarm clock is shown in figure 4.

The alarm clock uses the *C.time?* to define the *timeofday*. The attribute *alarmtime* is only in scope when the alarm is set. The timed transition from ringing to quiet is replaced by introducing a hidden attribute, *finish,* which characterises how long the bell window is open before it closes itself. The brackets around *finish* indicate that it is is not

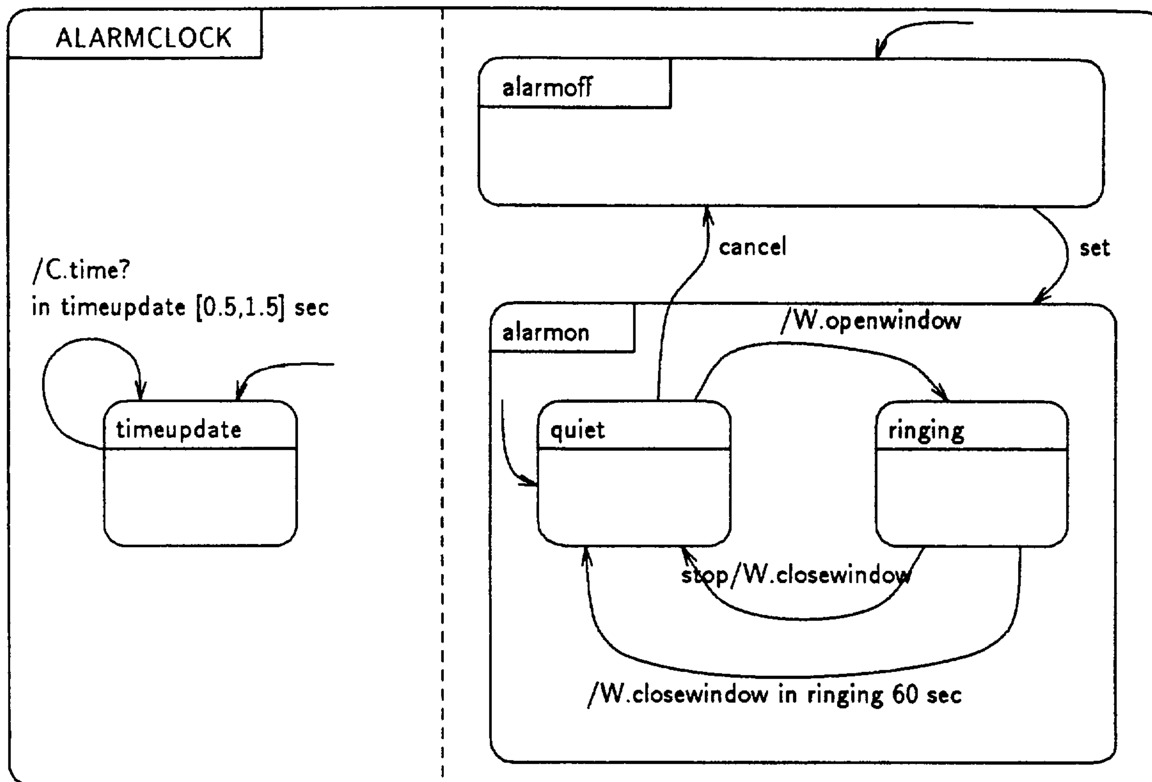

Figure 3: Refined Alarm Clock statechart

visible outside the alarm clock.

Above, we have extended statecharts with attributes and observers. The extra information allows the transitions of the machine to be specified.

A *transition specification* comprises the initial and final state names of the transition and the service name for the transition, together with a firing condition and a post-condition. A transition specification should be given for each arc in the statechart.

A firing condition describes restrictions on a transition imposed in terms of the attributes and observers of the class. The firing condition is a predicate which may mention attributes and observers. Where a service is provided which involves input parameters from the environment, the firing condition cannot mention the values of the parameters, because this might involve rejecting a communication on the basis of the values communicated.

Any attribute, observer or state name may be used in the firing conditions. Because attributes and observers exist only for certain states of the object, we use the logic of partial functions from VDM [1].

A post-condition for a transition is a predicate on the initial and final values of attributes and observers which characterises the effect of the transition. The initial value of some observer or attribute s is written $\overleftarrow{s}$. Each transition specification has the frame-condition that any attribute or observer that is not mentioned in a post-condition is unaffected by the transition.

The transitions of the alarm clock are:

- timeupdate → timeupdate: $\{true\}$ */C.time?(|t)* $\{timeofday = t\}$
- → timeupdate: $\{timeofday = 0\}$
- alarm off → alarm on: $\{true\}$ *set(t)* $\{alarmtime = t\}$
- alarm on → alarm off: $\{true\}$ *cancel* $\{\}$
- quiet → ringing: $\{alarmtime \leq timeofday < alarmtime + 1.5\}$ */W.openwindow* $\{finish = \overleftarrow{timeofday} + 60\}$
- ringing → quiet: $\{timeofday \geq finish\}$ */W.closewindow* $\{\}$

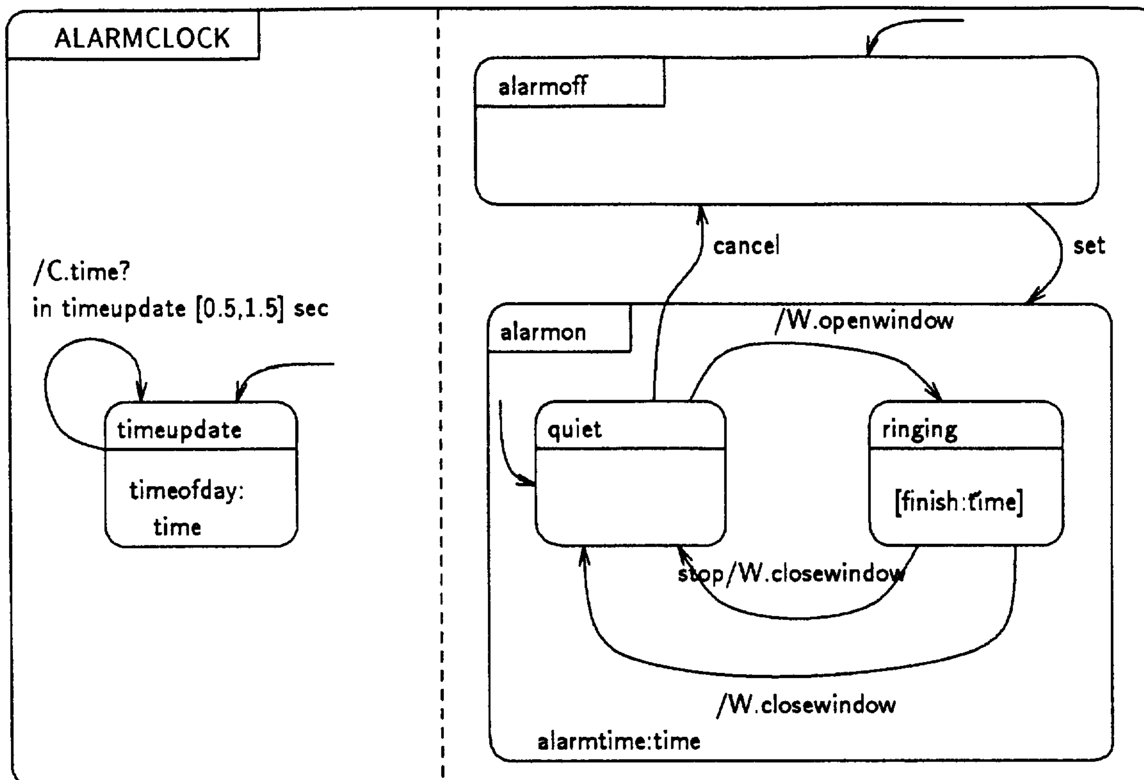

Figure 4: Alarm Clock enhanced statechart

- ringing → quiet:
  {*true*} *stop /W.closewindow* {}

When interpreting the transition specifications remember that the frame-condition ensures that the only affected attributes are those that are mentioned. The request for *time?* updates the *timeofday*, alone. When an Alarmclock object is initialised the *timeofday* is set to an arbitrary value, 0. The transition from quiet to ringing sets the bell window to be open for 60 seconds. The transition from ringing to quiet occurs when *timeofday* reaches *finish* or when the alarm is stopped, in both cases no observers or attributes are affected.

Some observers or attributes of a class may be redundant, i.e. can be derived from others. The derived observers and attributes can be specified as invariant relations on the observers and attributes used to annotate the states. An *invariant* specification comprises a state name and the relation which holds in that state.

For example an Alarm Clock may have an observer, *timetoalarm* which returns the length of time before the bell window opens; *timetoalarm* is only allowed in the quiet state, i.e. when the alarm is set but not ringing. It would be specified by

> quiet:
> {*timetoalarm* = *alarmtime* - *timeofday*}

Informally we define an *Objectchart* specification of an object class as a statechart together with a set of transition specifications and a set of invariant specifications.

3.3 Window Class and System Clock Class

We now give an Objectchart specification of a simple text window class. For a window to be used in the Alarm Clock application it has to provide only the services *openwindow* and *closewindow.* To give more examples of the Objectchart notation we consider the following services

- *move:* 'change window position on screen'
- *openwindow:* 'display text on screen (deiconise)'
- *closewindow:* 'iconise window'

- *type:* 'add a character to text contents'
- *posn:* 'position of window on screen'
- *area:* 'size of window'

The *move* operation only affects where the window is positioned on the screen. Typing is only permitted when the window is open. Closing a window reduces its size to a minimum and causes an icon to be displayed. Typing a character only changes the window contents. The Objectchart in figure 5 specifies this behaviour.

The attribute *display* represents the bit-mapped display and holds the information that appears on the physical screen. The transition specifications are as follows.

- $\rightarrow$ location: $\{posn = origin\}$
- location $\rightarrow$ location:
  $\{true\}$ $move(newposn)$ $\{posn = newposn\}$
- $\rightarrow$closed: $\{contents = nil \wedge area = minsize\}$
- open$\rightarrow$ closed:
  $\{true\}$ *closewindow*
  $\{display = icontext \wedge area = minsize\}$
- closed$\rightarrow$ open:
  $\{true\}$ *openwindow*
  $\{display = contents \wedge area = maxsize\}$
- open$\rightarrow$ open:
  $\{true\}$ $type(c)$
  $\{contents = \overleftarrow{\text{contents}}\ .\ c \wedge display = contents\}$

The value *origin* determines the default position for the window; *maxsize* and *minsize* are the areas of open windows and closed windows respectively; and *icontext* determines the text displayed in an closed window. The infix "." operator denotes right concatenation.

When in the closed state the display shows an iconic value. In the open state, the value of the display must be updated whenever *contents* is changed. This is ensured by the post-condition of the *openwindow* and *type* transitions.

The System Clock class has very simple behaviour. It provides a single service *time?* and it is always willing to accept a request for this service.

4 System Behaviour

In the previous section we gave graphical descriptions of classes and of configurations of particular instances. These descriptions were easy to understand and we exploited this to give informal explanations of the behaviour of the alarm clock system. However we did not discuss the relationship between classes and objects, and we did not explain how objects interact with each other. These issues are addressed now.

In general this is a difficult problem. In this section we show how Objectchart descriptions allow us to relate system behaviour to the behaviour of component objects.

There are two stages: first, the behaviours of individual objects are defined and then these are combined to define the behaviour of the overall system. The approach in this section is more formal than the rest of the paper, and could be omitted on a first reading.

4.1 Object Interaction

Statecharts already provide a communication mechanism, ([3]), but this is based on *broadcast* communication which is not a good model of object interaction.

Objects in a system interact by generating and receiving service requests. In an interaction, a client object generates a service request to a named server object. A client may send a request to many server objects, and a server may receive a request from many clients, but these interactions happen *one at a time.* Our model of communication reflects this directly.

Formally, we define a service request to be a triple comprising an identifier for the object which generated the request, the name of the service request with any input or output argument values, and an identifier for the object which received the request. For example, a service request s generated by object a and sent to the object b, is the triple

$$<a, s, b>$$

We model behaviour as the set of possible sequences of service requests. In this model only one interaction takes place at a time, and each interaction is completed before the next begins. We start

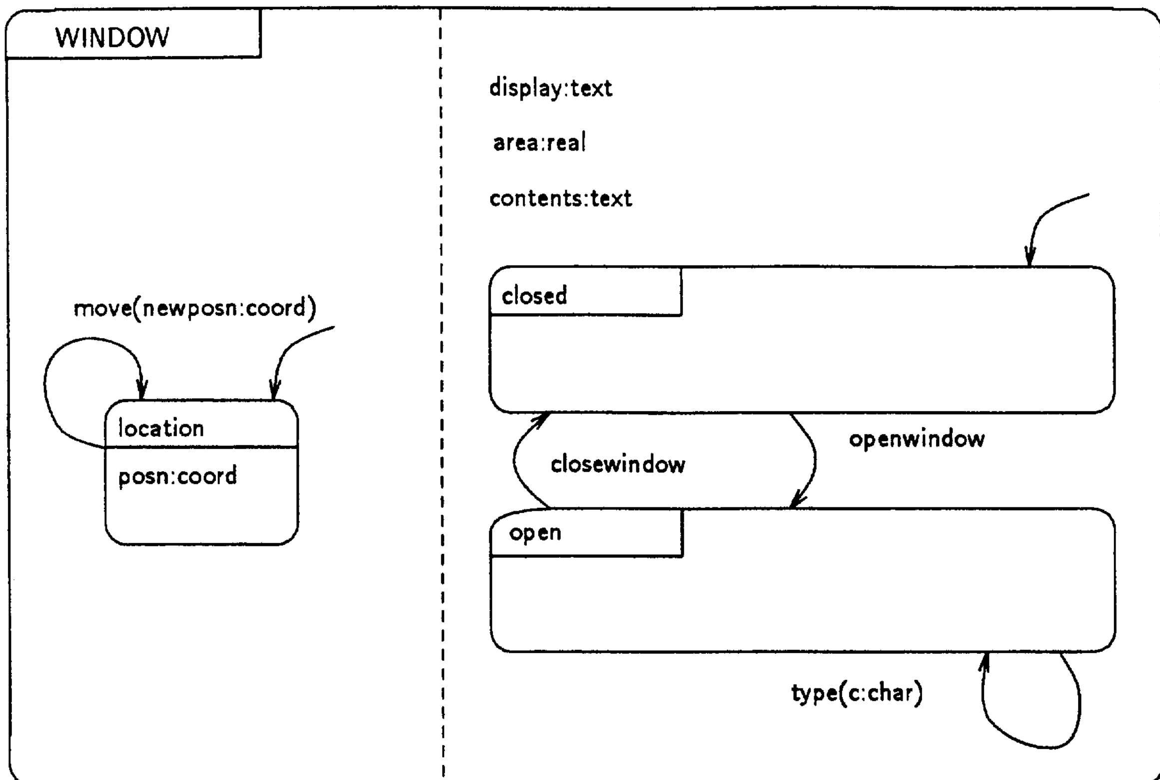

Figure 5: Window Class Objectchart

with the behaviour of a class and explain how to derive the behaviour of an individual instance.

4.2 Class Behaviour

A class definition uses *formal* names rather than actual object names. Target objects are indicated by the formal names introduced by the Objectchart and similarly, the class being defined is indicated by the name *self*. For example, the *openwindow* service request generated by the alarm *self* and sent to the bell window object *W* is the triple

<*self, openwindow, W*>

Requests for provided services are generated by anonymous clients; such clients are indicated by the symbol '*'. For example the *cancel* request received by the alarm clock is the triple

<*, *cancel, self*>

An Objectchart definition of a class introduces the services provided and required by the class, and defines—by the underlying state machine and the transition specifications—which sequences of service requests are possible, and which are not.

We call this set of possible sequences, the *traces* of the class. For a class X we write $\mathcal{T}_c$(X) and think of it as the 'behaviour' of the class.

As examples, consider the following traces in $\mathcal{T}_c$(ALARMCLOCK) where the alarm is set to ring after one second. In the first trace the alarm is stopped as soon as it starts to ring:

<*, *set(12:00:00), self*>
<*self, time?(11:59:59), C*>
<*self, time?(12:00:00), C*>
<*self, openwindow, W*>
<*, *stop, self*>
<*self, closewindow, W*>

In the second trace the alarm is allowed to ring for one minute before it stops itself:

<*, *set(12:00:00), self*>
<*self, time?(11:59:59), C*>
<*self, time?(12:00:00), C*>
<*self, openwindow, W*>
<*self, time?(12:00:01), C*>
...
<*self, time?(12:01:00), C*>
<*self, closewindow, W*>

4.3 Object Behaviour

The behaviour of a class is a *template* for the behaviour of an instance of the class. More precisely, the traces of a class are templates for the ways in which an instance of the class might communicate with other objects in its environment.

The traces of a class contain *formal* names for objects which receive requests generated by instances of the class. The configuration diagram provides the *actual* names of these objects.

The traces of an object O, $\mathcal{T}_o(\mathrm{O})$, are derived from the traces of its class by replacing formal object names with the corresponding actual object names.

In the alarmclock example, the traces of the class refer to the formal objects C and W. The configuration diagram provides the actual object names *clock* and *bell*, respectively. For example the traces from $\mathcal{T}_c$(ALARMCLOCK) given above, generate the following traces of $\mathcal{T}_o$(alarm).

<, set(12:00:00), self>*
<self, time?(11:59:59), clock>
<self, time?(12:00:00), clock>
<self, openwindow, bell>
<, stop, self>*
<self, closewindow, bell>

and

<, set(12:00:00), self>*
<self, time?(11:59:59), clock>
<self, time?(12:00:00), clock>
<self, openwindow, bell>
<self, time?(12:00:01), clock>
...
<self, time?(12:01:00), clock>
<self, closewindow, bell>

4.4 System Behaviour

A system is a particular configuration of objects. The behaviour of the system depends on the behaviour of all the component objects.

We model the behaviour of a system as a set of traces of service requests received or generated by the objects comprising the system. Roughly speaking, it is the set of all system traces such that each trace is 'possible' from the point of view of each component object.

We now give a more detailed explanation. Consider a candidate trace *tr* and some object O. We must define what it means for the trace to be possible from the object's point of view. First of all, only part of *tr* is relevant to O, this is

$$tr \rhd O$$

the subtrace of service requests which are received or generated by O. Other requests in the trace are not even noticed by O. So, we have to check whether or not the subtrace $tr \rhd O$ is a possible trace of O.

In order to check whether or not $tr \rhd O$ is a possible trace of O, we have to deal with some technical details: $tr \rhd O$ uses *actual* names for all the objects in the system. We derive a 'localised' trace, which uses the special symbols *self* and *, as follows

- in each request, the actual object name O, is replaced by the special object name *self*;
- in each request received by O, the name of the object generating the request is replaced by the special symbol '*'.

We can now state that the system trace *tr* is possible from *O's* point of view if and only if the localised trace $local(tr \rhd O)$ is in $\mathcal{T}_o(\mathrm{O})$.

The behaviour of a system $S = \{O_1, \ldots, O_n\}$ *is defined to be*

$$\mathcal{T}_s(\mathrm{S}) = \{tr \mid \forall o \in S \cdot local(tr \rhd o) \in \mathcal{T}_o(o)\}$$

the set of traces which are possible from the point of view of all the objects of the system.

From this definition, we have the following result:

> For a system S composed of a finite number of objects, and a trace *tr* of service requests, it is decidable whether the trace is a possible behaviour of S.

As an example, consider the following trace of the alarm clock system

tr =
<, set(12:00:00), alarm>*
<alarm, time?(11:59:59), clock>
<alarm, time?(12:00:00), clock>
<alarm, openwindow, bell>
<, stop, alarm>*
<alarm, closewindow, bell>

The subtraces of *tr* relevant the component objects are

tr ▷ *alarm* =
<*, *set(12:00:00)*, *alarm*>
<*alarm*, *time?(11:59:59)*, *clock*>
<*alarm*, *time?(12:00:00)*, *clock*>
<*alarm*, *openwindow*, *bell*>
<*, *stop*, *alarm*>
<*alarm*, *closewindow*, *bell*>

tr ▷ *bell* =
<*alarm*, *openwindow*, *bell*>
<*alarm*, *closewindow*, *bell*>

tr ▷ *clock* =
<*alarm*, *time?(11:59:59)*, *clock*>
<*alarm*, *time?(12:00:00)*, *clock*>

The localised versions are

local(tr ▷ *alarm)* =
<*, *set(12:00:00)*, *self*>
<*self*, *time?(11:59:59)*, *clock*>
<*self*, *time?(12:00:00)*, *clock*>
<*self*, *openwindow*, *bell*>
<*, *stop*, *self*>
<*self*, *closewindow*, *bell*>

which is in $\mathcal{T}_o$(alarm),

local(tr ▷ *bell)* =
<*, *openwindow*, *self*>
<*, *closewindow*, *self*>

which is in $\mathcal{T}_o$(bell) and

local(tr ▷ *clock)* =
<*, *time?(11:59:59)*, *self*>
<*, *time?(12:00:00)*, *self*>

which is in $\mathcal{T}_o$(clock). It follows that *tr* is a possible trace of the alarm clock system.

5 Conclusion

This paper has extended the statechart notation to deal with object oriented software. The resulting notation, Objectcharts, is an extended state machine specification technique which

- combines lifecycle behaviour of object classes with a declarative specification of their provided services;
- provides a basis for reasoning about system behaviour from component specifications.

Communication in Objectcharts is based on *service request* which is quite different from the broadcast mechanism provided by statecharts [3]. Our approach is to define communication using simple traces. In this paper we have outlined a semantics and a formalisation is in preparation.

The example in this paper is a simple one-level static system of objects and thus avoids dynamic object creation and deletion and aliasing. The design, efficient implementation and documentation of the dynamic systems can cause severe problems for OO development teams. Extending Objectcharts to deal with this is the subject of current research.

References

[1] Jones C.B. *Systematic Software Development Using VDM.* Prentice-Hall, 1986.

[2] Harel D. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[3] Harel D., Pnueli A., Pruzan-Schmidt J., and Sherman R. On the formal semantics of statecharts. In *Proceedings of the Second IEEE Symposium on Logic in Computer Science*, pages 54–64, 1987.

[4] Jackson D. Composing data and process descriptions in the design of software systems. Master's thesis, MIT, 1988.

[5] Mellor S.J. and Shlaer S. *Object Oriented Systems Analysis: Modelling the world in data.* Prentice-Hall, 1988.