An Efficient Object-Oriented Variation of the Statecharts Formalism for Distributed Real-Time Systems

Bran Selic

ObjecTime Limited
Kanata, Ontario, CANADA

Abstract

The ROOM (Real-Time Object-Oriented Modeling) methodology was developed specifically for dealing with distributed real-time systems based on the object paradigm. For describing high-level concurrent behavior of systems, ROOM uses a variation of the basic statechart visual formalism. This variation was driven by two sets of requirements. First, we wanted to integrate some of the more powerful aspects of the object paradigm, such as inheritance, with the statechart formalism. In addition, based on our own experience with finite-state machine descriptions, we felt that statecharts had a potential to be used not only for specification but also for implementation. Consequently, we filtered the set of basic statechart concepts, modifying some, removing others and introducing new ones, based on their implementability using state-of-the-art technology. The result is a highly efficient and expressive formalism which can be applied from the early phases of development (analysis) all the way to implementation. This eliminates some of the error-prone paradigm shifts which characterized traditional system development processes. This new formalism has been incorporated into a commercial toolset and has been used extensively in the design of a large number of real-time systems.

1. Introduction

Graphical finite-state machine formalisms are widely used to specify the high-level behavior of complex real-time systems because they are compact and easily understood. However, when we attempt to use them for *implementation* it often happens that much of this expressive power is lost due to the more complex nature of real-world systems. For example, a specification for a data communications protocol typically ignores various management issues (e.g., initialization, datafill, fault recovery) associated with the software entity that supports the protocol. When these are merged in with the original specification, the resulting finite-state machine graph can become incomprehensible.

Statecharts [1] is a variant of the finite-state machine formalism which reduces the *apparent* complexity of a graphical representation of a finite-state machine. This is accomplished through the addition of simple graphical representations of certain common patterns of finite-state machine usage. As a result, what might be a complex subgraph in a “basic” finite-state machine is replaced by a single graphical construct. In the remainder of this paper, we assume a basic familiarity with the statechart formalism. Comprehensive descriptions can be found in references [1] and [2].

In the majority of cases, statecharts are far more compact than equivalent flat state machines enabling the graphical representation to be extended to implementation-level systems. The benefits of this are significant: by “programming” system behavior directly in statechart form we not only reap the design and documentational benefits of a graphical representation but we also eliminate the error-prone manual step of translating a graphical design representation into an equivalent programming language implementation.

Unfortunately, as originally defined, the statechart formalism has some basic concepts which are difficult to implement for real-time systems. This applies particularly to its communication model. Our objective was to define a variant of the statechart formalism that would be more amenable to direct implementation. This meant modifying or replacing some of these inherently difficult concepts to be more in tune with currently available hardware and software technologies. For example, it is generally unrealistic to apply the concept of broadcast communications across a lossy wide-area network.

A second major requirement that motivated our work was the desire to integrate statecharts with the object paradigm [3]. Our experience with real-time system design indicated that the object paradigm was an excellent fit to the real-time domain [4]. In particular, we found that most real-time applications are more naturally expressed as net-

*Paper submitted (23/10/92) to **CHDL ‘93: IFIP Conference on Hardware Description Languages and Their Applications***, April 26-28, 1993, Ottawa, CANADA

works of cooperating objects rather than in algorithmic form. This representation is inherent in object-oriented systems. In addition, the object paradigm includes a spectrum of additional features that complement each other well. For example, inheritance, encapsulation, and polymorphism are generally useful features that have the potential for improving productivity and overall system reliability.

Over the last five years we have developed a conceptual framework for real-time system design which, among other things, allowed us to meet both of the major objectives described above. This framework is part of a system development methodology called Real-Time Object-Oriented Modeling or ROOM. A very brief overview of the fundamental aspects of ROOM is provided in the next section of this paper but readers interested in a more in-depth view should consult reference [4]. The focus of the remainder of this paper is the particular refinement of statecharts developed in ROOM and which we shall call, for want of a better name, *ROOMcharts*.

At this time there is a substantial body of experience in using ROOMcharts since they are supported by a commercial CASE tool called ObjecTime. They have been used in over fifty different real-time projects. A summary of this experience is provided in Section 5 of this paper.

2. An Overview of ROOM

ROOM is a methodology that was developed primarily for distributed real-time systems. By focusing on a particular domain, as opposed to being general purpose, we felt that we could retain a compact yet highly expressive conceptual base. We decided on an evolutionary approach so that the modeling concepts which represent the basic vocabulary of ROOM are typically generalizations or extensions of concepts already common in the real-time domain. As a result, practitioners in the domain generally find the concepts intuitive and absorb them without too much difficulty.

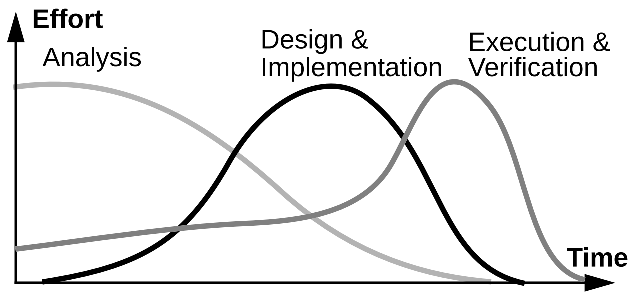
One of the fundamental objectives of ROOM is the removal of conceptual discontinuities that characterize traditional development processes. These discontinuities occur during transitions from one phase of development to another. For example, the outcome of the design phase is typically a design model. While this is used as input to the implementation model, the two models are usually not formally related.

This leads to many problems. If the design model is informal, there is no guarantee that it is consistent. Such design inconsistencies are then detected during implementation where they are often very difficult to fix because of the often overwhelming intricacies of implementation issues. Furthermore, informal models can be easily misinterpreted since their semantics are unclear. Hence design intent may not be properly conveyed. Last but not least, with an informal model there is no possibility of enforcing design decisions through implementation. It is not uncommon for implementors to "shortcut" designs in ways which impinge on the architectural integrity of a system thus jeopardizing its evolutionary potential.

ROOM avoids these pitfalls by specifying a conceptual base which is formal. In contrast to the more mathematically-oriented formalisms [5] [6] which stress expressive power at the cost of almost completely ignoring implementation concerns[1], ROOM concepts were carefully selected to be relatively easily mappable to efficient implementations while still retaining much expressive power. (The fact that the methodology is restricted to a single domain makes this easier to achieve.)

With a formal set of concepts, it becomes possible to analyze models for consistency and completeness at any point in the development cycle. This can be done by algorithmic means or by direct execution. Furthermore, since ROOM concepts range from the very abstract (e.g., layering) to the very concrete (e.g., individual low-level data structures), all of which are formally interrelated, it is possible to span the entire development cycle seamlessly with one set of concepts and thus eliminate the conceptual discontinuities.

The elimination of conceptual discontinuities has a profound effect on the way in which development is done. Traditional development processes tended to be phased with each phase focused around its own conceptual base (the analysis model, the design model, etc.). In ROOM it is more appropriate to talk about activities rather than phases of the development cycle. The three basic activities are: *analysis* which focuses on problem understanding, *design and implementation* which generates a solution to the problem, and *execution and verification* which measures the suitability of the solution to the problem.

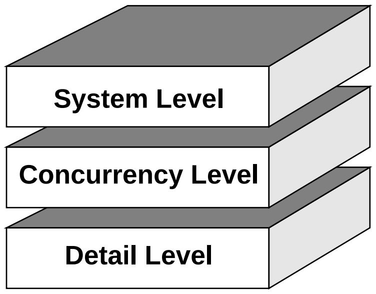


As indicated in the diagram above, at any given moment, all three activities might be performed although the emphasis changes throughout the cycle (e.g., heavy on

1. Another argument against strongly mathematical formalisms is that most software and hardware designers today do not have the necessary training to use these properly. Until this situation is rectified, rather than bemoan the state of the educational system, we feel that it is necessary to deal with current realities.

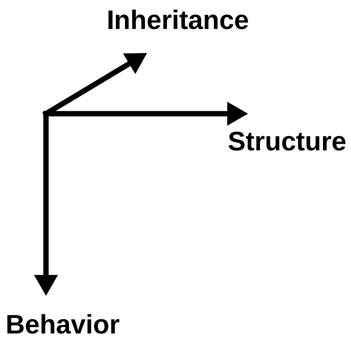
analysis in the initial part of the cycle).

For clarity, the ROOM concepts are organized around two basic paradigms. The *abstraction levels* paradigm classifies the modeling concepts according to the scope which they encompass.



At the bottom is the Detail Level which has concepts that are found in traditional non-concurrent programming languages. This is the level at which much implementation detail occurs. The next level up, the Concurrency Level, deals with aggregates of cooperating (concurrent) state machines. At the highest, System, level are concepts, such as layering, which encompass the entire system.

The second paradigm, called the *modeling dimensions* paradigm, identifies three main aspects of modeling: structure, behavior, and inheritance.



Behavior specifies the dynamic aspects of a system while structure deals mainly with architectural issues: how is the system decomposed, what is the relationship between the components, etc. Inheritance is both a reuse and an abstraction facility. In our experience, getting the inheritance done properly requires careful and dedicated effort which is why it is given equal prominence as the other two more classical dimensions of system design.

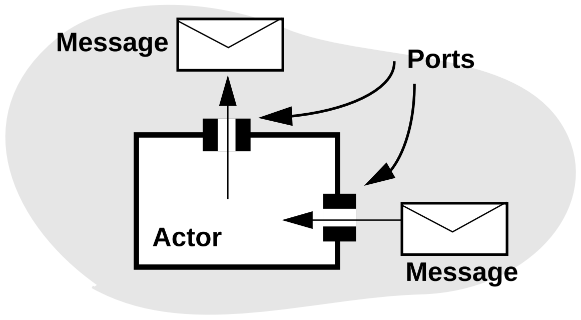
3. ROOMcharts

ROOMcharts belong at the Concurrency abstraction level. The computational model which they enforce is best described as sets of cooperating finite state machines.

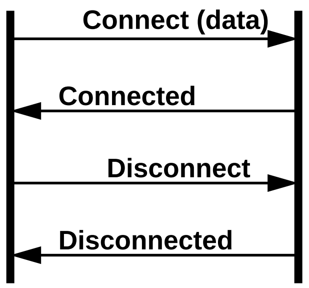
3.1. Structural concepts

The fundamental structural concept at the Concurrency level is that of an *actor*. An actor represents an active concurrent entity with a specific responsibility. Concurrency means that an actor can exist and operate in parallel with other actors in the same environment. An actor’s implementation is completely hidden from its environment and other actors by an encapsulation shell.

In order for an actor to communicate with its environment, its encapsulation shell has openings called *ports* through which information can flow in or out. The information that is exchanged is packaged into discrete units called *messages*. In our model, messages are instances of abstract data types and are the sole means of communication available to an actor. Because of the encapsulation shell, the behavior of an actor can only be deduced from the outside by observing the flow of messages on its ports. Conversely, an actor's perception of its surroundings is limited to the information received through its ports.



Each port on an actor represents one specialized interface of the actor. One of the major attributes of a port is its associated *protocol*, consisting of a set of valid *message types* which are allowed to pass through the port, and a set of valid *message exchange sequences* on that port.

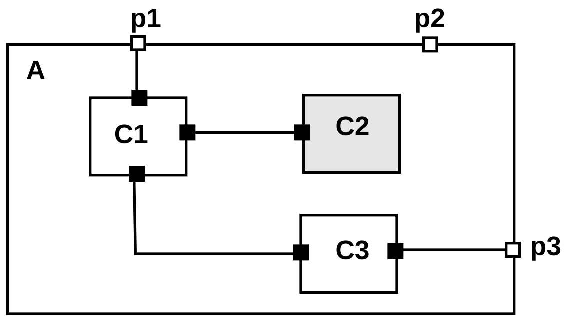


Note that this is a richer definition of an object interface than those found in most current object-oriented languages which are limited to simple data type signatures.

Since the same type of interface often appears on different actors, we have introduced the concept of a *protocol class*. A protocol class is a common specification for all port instances that respond to the same protocol. Protocol classes are organized into an inheritance hierarchy. This allows standard subclassing techniques to be used to provide abstractions or refinements of protocols as well as reuse of common protocol specifications.

Actors which implement complex functionality may have to be broken down further into component actors, each responsible for a subset of the overall functionality. (This process of decomposition can be carried to an arbitrary level.)

The concept of *bindings* is used to explicitly represent and constrain the valid communication relationships between actors. A binding is an abstraction of an underlying communication channel which carries messages from one actor to another. Bindings can be drawn only between ports that have mutually compatible protocols. Graphically, bindings are represented by undirected arcs connecting two ports on two actors in the same decomposition frame.



In general, bindings do not indicate the direction of communications. They can be either unidirectional or bidirectional depending on the characteristics of the underlying communication service.

Note that ports appearing on the containing actor's interface can be connected to internal component actors which is equivalent to the concept of delegation.

We impose a fundamental restriction that two actors in the same layer can communicate directly only if they have a binding between them. As a result, the decomposition structure, or *architecture*, of an actor explicitly captures the interactions between its component actors. This not only ensures that the architecture of large systems is directly visible, but also that it cannot be corrupted as a result of design decisions made at lower abstraction levels.

Similar to ports, each actor is an instance of some actor class which is a template for creating instances of that class. Since actor classes are also organized into class hierarchies, entire system architectures can be subclassed and reused through standard inheritance mechanisms.

Actor classes use single inheritance. Although this may appear restrictive, the ability of an actor to incorporate instances of other actor classes and pass through part of that object's interface as its own is, in our view, preferable to the intricacies of managing multiple inheritance.

In order to accommodate the highly dynamic nature of real-time systems, component actors can be specified as being *dynamic*. These are actors that are not automatically instantiated when their containing actor is created. Instead, they may created or destroyed under the control of the containing actor. In the ROOM notation, a dynamic actor is indicated by a shaded rectangle (e.g., actor **C2** in the diagram above).

3.2. Behavior

Actors are structural entities which provide the logical containers for behavior. The behavior of an actor is just one of its attributes. (Other attributes include the set of ports, the set of component actors and their interconnection, etc.) The linkage between behavior and structure is achieved through ports[1]. A port which is not bound to a container actor is called an *end port* (such as port **p2** in the previous diagram) and is directly accessible by the actor's finite-state machine.

Communication model

Note that the strong encapsulation of actors eliminates the possibility of shared variables between actors and also dictates a pure message-based communication model. The advantage of this model is that it is general enough to cover both software and hardware as well as distributed and non-distributed systems.

An *event* in this model is defined as the arrival of a message at some end port. More precisely, an event is the 4-tuple:

<port, signal, priority, data>

where *port* is the port on which the event occurred, *signal* is a unique application-specific identifier which captures the semantics of the event/message, *priority* is the dispatch priority with which the message was sent, and *data* is an optional application-specific passive object embedded in the message

Communications can be *either asynchronous* or *synchronous*. Asynchronous message communication is non-blocking: after sending a message, the sender simply continues its activity. In case of synchronous communication, the sender is blocked until the receiver replies with a message of its own. This reply message takes precedence over any other messages that may have been queued at the sender so that this communication mode is equivalent to a remote procedure call. However, at the receiving end there

1. In addition to ports, behavior has access to *service access points* and *service provision points* which pertain to the layering feature of the System abstraction level. However, in this paper we will omit discussing the layered structures in ROOM.

is no distinction between a synchronous and an asynchronous communication. The same primitive can be used to send the reply message in either case so that the receiver is effectively decoupled from the communication mode of the sender. This means that the exact same actor class can be used both in synchronous and asynchronous client-server scenarios.

Event processing model

We have chosen the *run-to-completion* programming model for the behavior of actors. In this model an actor is normally in a receiving mode during which it awaits incoming events. If an event occurs, the actor responds by performing some activity appropriate to that event and then returns to the receiving mode to await further events. If a new events occurs while an actor is still busy processing the previous one, the new event is queued by the receiving end port and will be automatically resubmitted when the actor returns to receiving mode.

When a message is sent, it is assigned a priority. The general intent is that events of higher priority get some type of precedence over events of lower priorities. However, given the variety and volatility of scheduling policies in real-time systems, the semantics of priorities are viewed as an implementation issue.

The approach taken here is that events have priorities rather than actors. This is a departure from the traditional approach found in most operating systems in which priorities are assigned to processes. In our view, given the multiplicity of interfaces of an actor, it is highly likely that a single actor will be simultaneously involved in multiple concurrent and independent threads of activity. For example, an actor could be in the process of servicing some functional request while at the same time responding to background-type maintenance queries. Not all of these activities are likely to be of the same priority so that it would be very difficult to define one priority to fit all needs. In general, in an “event-driven” system, we feel that it is more reasonable to assign priorities to events.

Behavior description

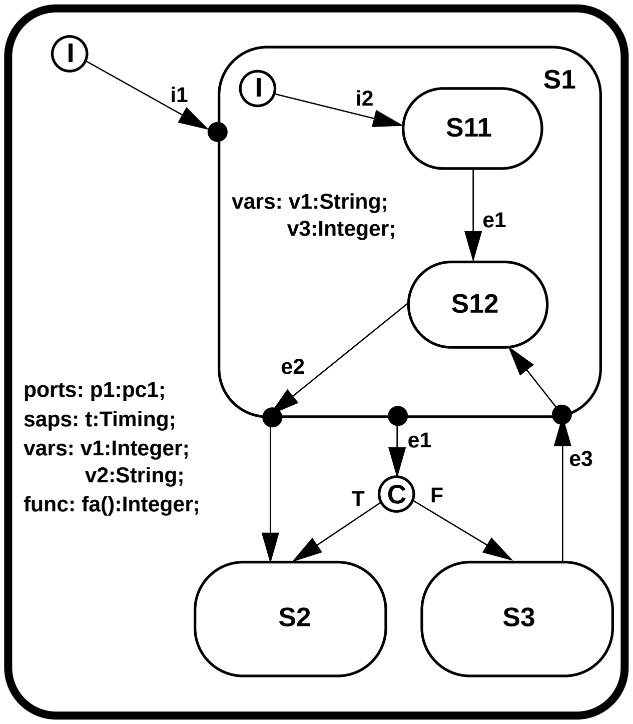
The event processing model just described is easily adapted to a finite-state machine formalism. The receiving mode of an actor can be mapped to states while event processing can be associated with the transitions. An event triggers a transition which performs the event processing.

ROOM leaves open which Detail-level programming language can be used to specify the details of event handling. The ObjecTime toolset currently supports two different object-oriented languages: C++ and a special derivative of Smalltalk that is useful for rapid prototyping (RPL).

In ROOM a finite-state machine description (ROOMchart) consists of the following attributes:

- a set of end ports,
- a set of service access points,
- a set of extended state variables,
- a set of internal functions,
- an initial point,
- a set of states,
- a set of transitions.

An example behavior description is shown below:



The end ports are the same end ports that appear on the interface of the containing actor. (Interface ports which are bound to an internal component actor are not seen by the behavior.) In this example, we have only one port defined, **p1**, which communicates via the **pc1** protocol class.

Service access points are similar to ports except that they are used to communicate with entities in adjacent layers (below and above the containing actor). For example, timing facilities are provided through service access points to a layer which can be placed beneath any actor that requires it (this is the case in the above example where service access point **t** provides access to the **Timing** service.) Like ports, service access points have an associated protocol. In fact, the distinction between a port and a service access point is a structural issue and, from the viewpoint of behavior, irrelevant. That is, the same message-based communication model is used for both service access points and ports. A message arriving on a service access point is handled in the exactly same way as a message arriving on a port.

Extended state variables are instances of passive Detail-level objects used by a finite-state machine to maintain auxiliary information that it needs to sustain between events. These objects can be manipulated by the code inside transitions and entry and exit actions (described below). Since actors are fully encapsulated, these objects are not accessible by other actors so that issues of concurrent access to these objects do not arise. The example shows two sets of variables, one set (**v1** and **v2**) in the top-level state machine and another (**v1** and **v3**) in state **S1**.

Internal functions contain common code sequences that can be shared by event handling code. The example shows a function **fa()** of type **Integer**.

The *initial point* is the source point for the initial transition. This transition is automatically taken when the enclosing actor is created. Typically, code in this transition performs initialization operations. Note that initial point is not equivalent to a state since it is transient.

States

The basic statechart formalism allows a state to be decomposed into substates. This allows complex behavior to be uncovered gradually as a series of nested behavior patterns. We have found this feature, in combination with the concepts of state history and "group" transitions to be the most useful features of basic statecharts.

A ROOMchart state is composed of the following *optional* attributes:

- a set of extended state variables,
- an entry action,
- an exit action,
- a set of (sub)states,
- a set of transitions.

One of the features that distinguishes ROOMcharts is that states *encapsulate* their contents. That is, the inside of a state is not visible from the outside. A major advantage of this feature is that it allows different refinements of a state to be captured through inheritance.

The encapsulation feature of states means that each state represents a lexical scope. The scoping rules of states follow their nesting hierarchy similar to the scoping rules of block-structured programming languages such as Pascal. The extended state variables of the containing states are accessible to their contained states but not the other way around. For example, variaable **v2** is available in all state contexts. On the other hand, variable **v3** is only available in the context of state **S1**. Also, in state **S1** the references to the variable **v1** pertain to the local variable rather than the variable with the same name in the enclosing scope.

Another consequence of encapsulation is that transitions that "cut" across state boundaries (e.g., transition **e2**) are broken into different segments. For a given transition, each of these segments is in a different scope and, hence, has access to a different set of variables and functions.

Entry actions are optional code segments that are automatically invoked whenever a transition enters the state regardless of which transition was taken into the state. In case of a segmented transition, the entry action is performed between the external and internal segments of the transition.

Exit actions are similar to entry actions except that they are activated whenever a transition is taken out of the state. In case of a segmented transition, the entry action is performed between the internal and external segments.

At execution time, the *history* of a state is the most recently visited substate of the state. ROOMcharts only support "deep" history, that is history that extends to the innermost level. Although we considered "shallow" history, we have not encountered situations in practice where it would have been indispensable.

As a notational convention, a transition that terminates on the outside border of a state automatically goes to history (e.g., transition **i1**). Hence, there is no special symbol required for history such as exists in the basic statechart formalism. If it is desired to bypass history, then the transition is simply continued (through a new transition segment) to the desired substate or to the initial point.

Note that we do not make use of the concurrent states feature of basic statecharts. This point is discussed in detail in Section 4.

Transitions

A transition can be triggered by the arrival of a message (an event). A simple trigger is specified by the 3-tuple:

<port, signal, guard>

which means that the trigger becomes enabled when the specified *signal* arrives at the specified *port* (or service access point) provided that the optional Boolean predicate guard is true. Complex triggering conditions are also possible such that the arrival of any of a set of signals on any of a set of ports can trigger a transition.

A transition can optionally have a code segment associated with it which captures the Detail-level behavior associated with event handling.

Group transitions are transitions that emanate from the border of a composite state (transition **e1** emanating from state **S1** in the example). These transitions apply equally to all substates unless explicitly overridden as explained below.

The scoping rules enforced by the nesting of states apply to transitions and their triggers as well. For example, the guard condition has access to any extended state variables or functions in its scope (recall that ports and service access points occur at the top level and are available in any scope).

In keeping with the nested scoping, transition triggers are also scoped in such a way that triggers on the innermost current state take precedence over equivalent triggers in higher scopes. In the example, if the actor is in state **S11**, then the transition `e1` originating from that state will take precedence over the group transition `e1` that emanates from state **S1**. This allows the overriding of group transitions and is a commonly used feature.

Finally, a transition can be split into multiple transition segments at a choice point (transition `e1` in the example).

3.3. Inheritance

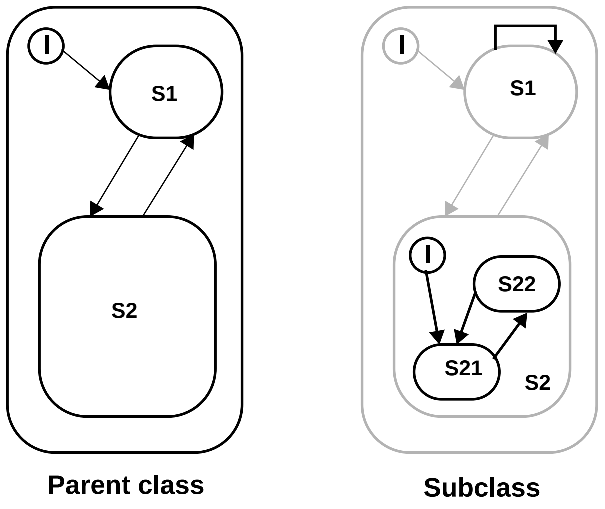
In considering how best to apply inheritance to ROOMcharts we chose to combine it with the structural inheritance scheme. That is, we view behavior as just another attribute of an actor class (albeit a complex one). Although it would have been possible to have separate class hierarchies for structure and behavior we decided against it since the structural attributes of an actor provide a context for behavior (through ports and service access points).

In ROOMcharts, a subclass automatically inherits all the behavioral attributes of its parent class. In choosing an inheritance scheme we decided against strict inheritance[1]. This means that not only can new behavioral attributes be added (e.g., new states, transitions, variables) but also that any inherited attributes can be overridden or deleted. It is our view that strict inheritance, which can be severely limiting in practical applications, does not always guarantee behavioral equivalence of the parent class and the subclass. For example, if a subclass adds just one extra transition that is triggered by a timing signal, then we can no longer claim that its behavior is equivalent, even if it is used in the exact same circumstances as the parent class.

One of the most common ways in which inheritance is used with ROOMcharts is to refine the “leaf” states of a parent class by decomposing them into substates in the subclass. The superclass captures the “gross” behavior common to all subclasses and variants then inherit that behavior but refine the detailed behavior. An example of this is a set of functionally diverse components all of which respond to a common control protocol. In this case, the control protocol would be captured by a single ROOMchart in a common superclass. Subclasses would then add their idiosyncratic behavior as decompositions of the “leaf” states of their parent.

Graphically, we use a grey rendering to indicate inherited attributes whereas local attributes are drawn with a black pen.

1. *Strict inheritance* disallows the deletion or modification of inherited attributes in a subclass on the assumption that the subclass will retain the behavior of its parent (since nothing was removed). Only addition of new attributes is allowed. Hence, it is postulated, it is possible to substitute a subclass in place of a superclass with no adverse effect.



4. Related Work

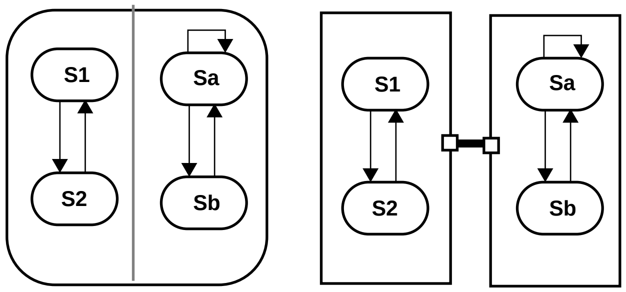
4.1. Statecharts

While statecharts were the basic inspiration for ROOMcharts, there are certain key differences between the two.

First of all , we have placed ROOMcharts in a very specific structural framework which is based on the object paradigm. Behavior descriptions are fully encapsulated within objects (actors) which have explicit multiple interfaces. Statecharts, on the other hand, do not have any formal structural boundaries or interfaces.

Related to this is the underlying event model. In statecharts, the mechanism for the generation of events is implicit and is achieved through something called “broadcast” communication which, in principle, can convey any change of state to any other component instantaneously. This vaguely-defined model is most likely intended to free the modeler from being concerned with implementation issues. However, in distributed real-time systems, communication constraints represent one of the dominant aspects of a design and it may be quite dangerous to ignore them. For example, various impossibility results [7] have demonstrated the fundamental impact that communication has on the way systems are designed and not just on the way in which they are implemented. In contrast, ROOM bases its event and communication models on a message-passing paradigm which is one of the most common paradigms found in distributed systems practice.

For the same reasons, ROOMcharts do not incorporate the concept of orthogonal concurrent states in the same object. Although this is considered as one of the principal features of basic statecharts, we have discovered that in the great majority of cases, concurrent states can be replaced by concurrent communicating actors:



Furthermore, in the latter case, the communication relationship between the concurrent components is explicit. One of the potential difficulties of the implicit communication between concurrent states is that it is possible to inadvertently create undesireable couplings between the states.

4.2. Object Modeling Technique (Rumbaugh et al.)

Object Modeling Technique (OMT) is a general system development methodology which incorporates basic statecharts as one of the techniques for specifying behavior [8]. While that work discusses inheritance for statecharts, that model of inheritance is quite different than the one given here. In OMT a substate is viewed as a “subclass” of its containing state. Hence, when a substate “inherits” a transition from its containing state, this is equivalent to saying that the transition is a group transition in ROOMchart terminology. Using this interpretation, for our example, transition **e1** emanating from state **S1** is inherited by substates **S11** and **S12**.

4.3. Objectcharts

A recently published work [9] introduces an object-oriented variant of statecharts, called *objectcharts*, that is closer to our model than basic statecharts. Similar to ROOMcharts, objectcharts are encapsulated by an object with explicit interfaces. However, there is no distinction between concurrent and non-concurrent objects. This was probably motivated by the desire to stay clear of implementation concerns. As in the case of the communication model, when dealing with distributed systems we feel that the identification of concurrency is a primary design issue and should be faced early in the development cycle.

Objectcharts, like statecharts, make use of orthogonal concurrent states although they have a more explicit communication model. However, in order to accommodate this model, they enforce restrictions in which communication cannot be chained. Finally, objectcharts do not deal with dynamic structures.

5. Experience with ROOMcharts

ROOMcharts have now been in use for over four years. In the last two years, they have been made available to a broader public through a commercial toolset called ObjecTime which supports the ROOM methodology. As of this writing over fifty different projects have used it and close to 500 people have been trained in applying them. ROOMcharts have been used on a wide variety of real-time projects including the specification of protocol standards, the high-level design of distributed systems architectures, modeling of hybrid hardware/software systems, performance modeling, and the design *and implementation* of industrial communication switching equipment.

The general feedback from the user base has been quite positive and there have been very few requests for modifications to or the addition of new concepts to the ROOMchart formalism In particular, there has been no outcry for the statechart features that we had eliminated.

However, one commonly-reported difficulty is that the partitioning of behavior into objects with embedded state machines makes the tracking of event flows which span multiple objects over time quite difficult to follow. In our view this is a significant shortcoming of the object paradigm which should be rectified in the future. One promising development in this direction can be found in the work of Prof. Buhr at Carleton University [10].

References

[1] D. Harel, “Statecharts: a visual formalism for complex systems,” *Sci. Computer Program.,* vol. 8, 1987.

[2] D. Harel, A. Pnueli, J.P. Schmidt, and R. Sherman, “On the formal semantics of statecharts,” in *Proc. 2nd IEEE Symp. on Logic of Computer Sci*., 1987.

[3] G. Booch, *Object-Oriented Design with Applications,* Redwood City, CA: Benjamin Cummings, 1991.

[4] B. Selic, G. Gullekson, J. McGee, and I. Engelberg, “ROOM: An Object-Oriented Methodology for Developing Real-Time Systems,” in *Proc. 5th Internantional Workshop on CASE,* Montreal, Canada, 1992.

[5] ISO, IS 8807, *Information Processing Systems - Open System Interconnection - LOTOS - A Formal Description Technique based on the Temporal Ordering of Observational Behaviour,* May 1989.

[6] R. Milner, *Communication and Concurrency,* Prentice-Hall, 1985.

[7] M. Fischer, N. Lynch, and M. Patterson, “Impossibility of Distributed Concensus with One faulty Process,” Journal of the ACM, April 1985.

[8] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, *Object-Oriented Modeling and Design,* Prentice-Hall, 1991.

[9] D. Coleman, F. Hayes, S. Bear, “Introducing Objectcharts or How to Use Statecharts in Object-Oriented Design,” IEEE Trans. on Soft. Eng. , vol.18, January 1992.

[10] R. Buhr, “Pictures that ‘Play’ for Designing Concurrent, Real-Time Systems,”, SCE-91-08 Dept. of Systems and Computer Engineering , Carleton University, July 1991.