ObjChart: Tangible Specification of Reactive Object Behavior*

Dipayan Gangopadhyay
I.B.M. Thomas J. Watson Research Center
P.O. Box 704, Yorktown Heights, NY 10598
dipayan@watson.ibm.com

Subrata Mitra
Department of Computer Science
University of Illinois at Urbana-Champaign
1304 West Springfield Avenue
Urbana, Illinois 61801, USA.
mitra@cs.uiuc.edu

Abstract

ObjChart is a new visual formalism to specify objects and their reactive behavior. A system is specified as a collection of asynchronously communicating objects arranged in a part-of hierarchy, where the reactive behavior of each object is described by a finite state machine. Value propagation is effected using *functional invariants* over attributes of objects. A compositional semantics for concurrent object behavior is sketched using the equational framework of Misra.

In contrast to other Object Oriented modeling notations, ObjChart uses object decomposition as the single refinement paradigm, maintains *orthogonality* between control flow and value propagation, introduces *Sequence* object which embodies structural induction, and allows tracing causality chains in time linear in the size of the system. ObjChart's minimality of notations and precise semantics make ObjChart models of systems coherent and executable.

1 Introduction

Specifying reactive systems is an important problem [Pnu86, Har87]. A large body of literature exists on various specification formalisms, for example see [MP92, CM89]. These, however, demand enough formal sophistication of a practitioner. CASE notations [Boo91, SM88], on the other hand, attempt to use pictorial descriptions to capture specifiers' conceptualization of a system under design. As Harel ([Har88a]) has eloquently argued, unless these pictorial notations are backed up by precise semantics, they remain only pretty pictures. Thus, combining the intuitive appeal of

*The term ObjChart was introduced in a video tutorial (Object-oriented paradigm, IEEE Video Press, No. 2096AV, May 1990).

visual notations with the precision of formal specification languages is an important goal.

Towards this end, we present ObjChart, a new set of visual notations for describing objects and their reactive behavior, and precise compositional semantics of behavior of ObjChart Objects.

In ObjChart formalism, a system under design is conceptualized by a set of concurrently interacting *objects* and a set of *relations* among these objects. Structurally, each object can have a set of attributes and a set of ports. An object can be decomposed into a hierarchy of constituent sub-objects. The attributes of an object or attributes of sibling objects (i.e., sub-objects within a common parent object) may be be related via *functional relations*. Each such relation is a set of functional constraints (invariants) over the related attributes.

The reactive behavior of each object is specified by associating an extended finite-state machine (FSM) with it. An FSM of an object captures the reactions of this object to received messages. Receipt of a message is treated as an event, which may cause a state transition, concomitantly initiating the reactions; reactions being asynchronous messages with limited scope. An object can send messages to itself, its immediate sub-objects, its parent object or to its ports. Ports of sibling objects can be connected by *behavioral relations*, each of which has an FSM controlling the message exchanges over these connected ports. Thus, a typical ObjChart model is a part-of hierarchy of objects with relations (both behavioral and functional) among siblings and the behavior of these objects is controlled by FSMs.

We provide compositional semantics for ObjChart objects. Although behavior of each object is controlled by a deterministic state machine, any composite object has non-deterministic behavior due to internal concurrency, with and among its sub-objects. Thus, the output reaction of a composite object cannot be expressed as a function of its input stimuli. We circumvent this problem by adopting Misra's framework of equational reasoning [Mis90]. Semantics of an ObjChart object is given by a set of equations relating functions on traces over a set of ports—i.e., sequences of observable messages received and sent by an object via these ports. Behavior of an object is the set of *smooth solutions* each of which is a trace obeying certain causality ordering[1].

Compared to the plethora of visual notations for Object-Oriented Analysis, the advantage of ObjChart formalism is that it comprises of only a minimal set of notations and the notations have precise semantics. Minimality and precision make ObjChart models *coherent and executable* – the latter property is the key to understanding and experimentation with objects and their behavior early in the analysis phase of system development. In fact, our rationale for naming ObjChart models as "tangible" is the fact that the models directly represent the conceptual entities visually and the inter-entity cause and effect relations can be perceived by observing the execution of the ObjChart models. Few of the notation-heavy modeling disciplines lend themselves easily to executable models.

In recent years, several proposals have used state machines for capturing reactive behavior of objects. For example, ObjectChart and OMT use Statechart of Harel for behavioral specification. The StateChart paradigm aims to manage state explosion

[1]Smoothness formalizes the notion that the output message of an object can depend only on the input received *prior to* sending this output.

problem of FSM based system descriptions, by structuring system description into "OR" and "AND" compositions of smaller FSMs. ObjChart models structure system behavior by object decomposition, each object requiring a small FSM. Thus, we achieve the same goals as Statecharts. Moreover, as a detailed comparison in Section 5 will show, ObjCharts have simpler semantics and much better performance in analysis—linear as opposed to exponential as compared to Statecharts.

Furthermore, treating the denotation of objects as a set of equations provides an elegant and simple framework for defining substitution and composition of objects. Behavioral equivalence can be thought of as equivalence of the sets of solutions for the respective equations. Composition by containment is given by the union of such equations. Composition by interconnection becomes union of equations with suitable port name unification. Substitution and composition are important for developing theory of types and subtypes for concurrent objects.

The rest of the paper is organized as follows. In Section 2 we introduce the constructs of ObjChart informally and illustrate them with a small example. Section 3.1 presents precisely the compositional semantics of reactive behavior of ObjChart objects using the equational framework of Misra. There we also discuss how the same framework can be used to define precisely subsumption and composition of reactive behavior. Thereafter, in Section 4, we discuss how ObjChart notations help one structure system specifications naturally by object hierarchies. We also observe how tracing causality chains along the same object hierarchy helps one organize reasoning about system behavior. In Section 5 we compare our work with other approaches that use state machines for describing object behavior.

2 ObjChart Notations

In this section, we introduce the notations of ObjChart using an example of a system to keep track of checkbook balances. All the visual notations are summarized in Figure 1.

Object

The primary construct of ObjChart is an *object* with a name and optionally with a type[2]. An object can have a set of *attributes*, each of which can hold a value of some type. An attribute a of object O can be queried and updated, respectively, by the following two constructs: $O \ll \mathit{getValue}(a,V)$[3], abbreviated optionally as $O.a(V)$, retrieves the value of a into the variable V, while $O \ll \mathit{setValue}(a,v)$ (optionally abbreviated as $O.a := v$) sets the attribute to value v. In addition, objects may have a set of named *ports* through which it sends and receives messages (to other objects). Visual notation of objects and attributes is exemplified in Figure 2. Here, the object a_RegisterEntry, modeling a typical entry in a checkbook register, has the attributes old-balance, amount and new-balance.

[2]Types in this context are just user-defined words.

[3]$a \ll m(\mathit{Args})$ denotes sending a message $m(\mathit{Args})$ to a target object a

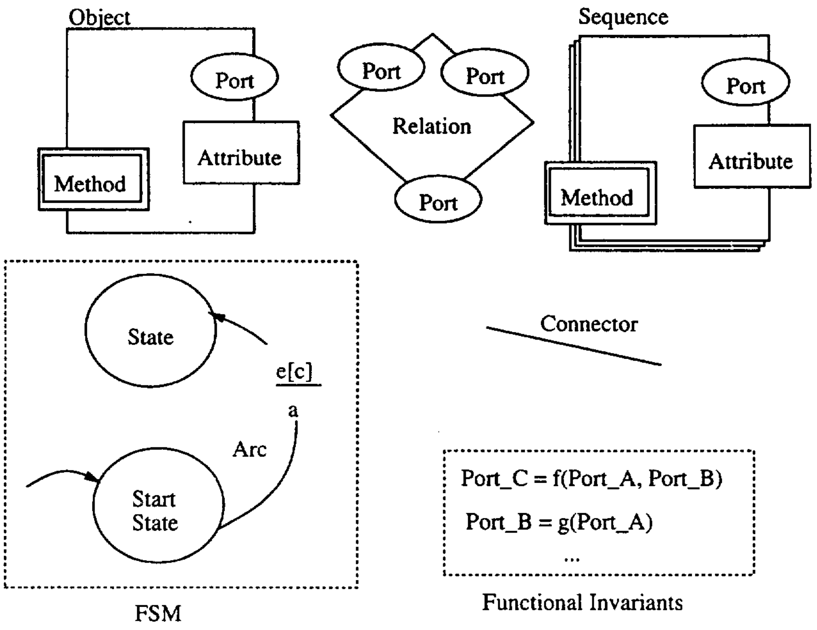


Figure 1: Basic Constructs of ObjChart

Relations

In ObjChart, *relation* objects can be used to depict both behavioral and computational dependencies among objects. In the former case, a relation object may be connected between the ports of objects, while, for the latter, attributes of objects can be connected via relations.

A relation has a set of typed *ports* and semantically embodies either some *behavioral constraint* or a set of *functional invariants* among these ports. Behavioral constraints are described using state machines (refer to the section below for details), whereas, a functional invariant is an equation of the form $p_j = f(p_1 \cdots p_k)$ which states that the value of the port p_j is a function f of the values of other ports $p_i, 1 \leq i \leq k$ and $i \neq j$. Connection of an attribute a_j to a port p_j of a relation R depicts substitution of a_j for p_j in all functional invariants for R.

Figure 3 shows an example of a relationship among attributes. The attribute `new-balance` is the difference of `old-balance` and `amount` for the object `a_RegisterEntry`. This is depicted by the relation `subtract` with ports `A`, `B` and `C` and a functional invariant $C = A - B$ among them.

State Machines

Behavior of an object is defined either by associating a *finite state machine* (FSM) or by defining *methods* in some 3GL. It can be shown that the combination of FSM used here and the functional invariants is adequate for writing any method; thus without loss of generality, we shall concentrate on behavior definition involving FSMs alone

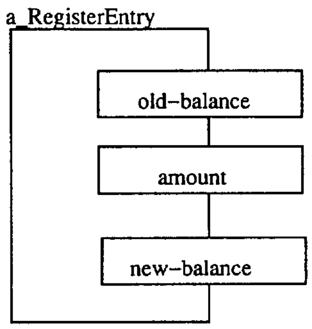


Figure 2: Object and Attribute

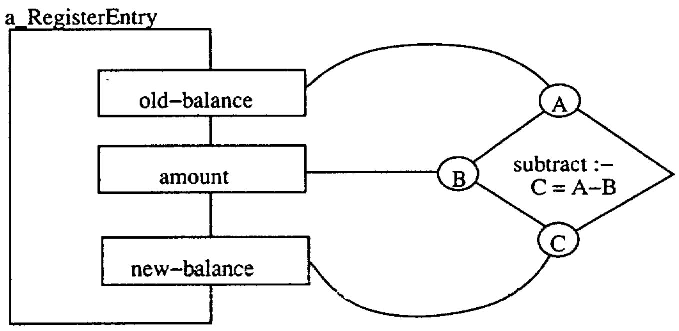


Figure 3: Functional Invariant

and use methods as "short-hand" for FSMs.

FSMs could be used either as controllers for single objects, or to coordinate message interchange between a set of sibling objects. An FSM of an individual object is meant to capture what its reactions are to the receipt of messages: the object may make state transitions and concomitantly may generate a set of other messages. On the other hand, a behavioral relation object, connecting several sibling objects, coordinates message exchanges among the connected objects by embodying a FSM. The FSM in such a relation object defines the reaction to the receipt of a message from one connected object via a port; the reaction may include a state transition and generation of messages to other ports of the relation i.e., to other connected objects.

The transitions of an FSM are depicted as labels on arcs between states. An arc label is a triple of the form ⟨Event, Condition, Actions⟩, usually written in the following syntax in our formalism:

$$\frac{\text{Event[Condition]}}{\text{Actions}}.$$

The presence of an arc labeled ⟨e, c, $\overline{a}$⟩ with source state s_1 and destination state s_2 for an object O means: If O is in state s_1 when it receives the message e, and the condition c "holds," then the object undergoes a state transition (to state s_2), sending the set of messages $\overline{a}$.

Figure 4 shows an example state machine for a single object, a register-entry; in reaction to the message `setAmount`(A), a register entry becomes "filled" by making a transition from the "blank" state, simultaneously generating a message to itself[4] to set its amount attribute. This depicts a *behavioral protocol* that a register entry, once made, cannot be altered.

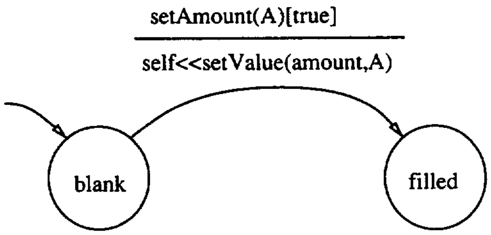


Figure 4: State Machine of a typical Register Entry

The inter-object messaging paradigm of ObjChart is as follows.

- The messages in the action part of an arc label, are *asynchronously* sent to the target objects without the sender waiting for their reply. Thus the sender is free to react to other input messages.
- The messages in the action part of an arc label can be executed in any order and the sender does not care about the order. These two aspects of ObjChart's messaging paradigm result in its ability to model object-level concurrency.
- Each recipient object process messages in a FIFO fashion. If a message is received when the object is not in the appropriate state, the message is lost.

The *condition* part of an arc label is a boolean expression involving conjunction, disjunction or negation of *predicates*. A predicate can be a simple relational comparison such as $X \leq Y$ or a *query predicate*. A query predicate is of the form $O.p(Args)$ which is true if the object O has the property $p(Args)$ for suitable bindings of the $Args$; O is limited to be one of the immediate sub-objects or an argument of the event for the arc in question. It is to be noted that such query predicates are evaluated *synchronously*, i.e., the calling object waits until a corresponding acknowledgement is received.

Both the FSM-s (i.e., actions and conditions in arc labels) and the methods of an object are restricted to having scopes only *local* to the object. These can refer to only the object itself (`self`), its environment (`parent`), its sub-objects, its ports and attributes. Furthermore, FSM-s of objects can query objects which are explicitly passed as arguments of its events during their event evaluations. Due to this locality restriction, ObjChart description of objects are very modular and are easier to analyze.

Sequence

ObjChart provides the construct *Sequence* object as a short-hand for an ordered set of identical objects. A Sequence is defined in terms of a distinguished object, its

[4]The distinguished word `self` denotes the object itself.

representative element; all elements of the sequence are structurally and behaviorally similar to the representative element. Visually a Sequence is shown as a stack of objects, where the top of the stack is the representative element. As an example, a set of register entries can be specified by a Sequence `RegisterEntries` where the representative element denotes a typical register-entry as shown in Figure 5.

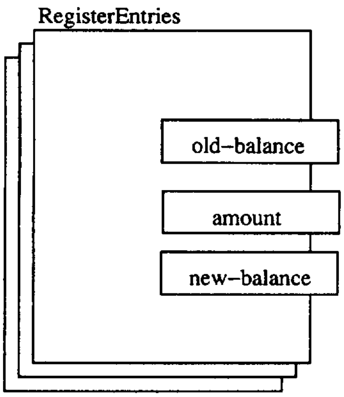


Figure 5: Sequence of Register Entries

By defining a Sequence in terms of its representative element, one can express in ObjChart an abstract schemata for potentially infinite set of objects. Moreover, ObjChart allows definition of relations between the i^{th} and $(i+k)^{th}$ elements of the sequence (for any constant k). The functional invariants specified through these relations hold at creation and update of elements. As an example, the fact that in a checkbook register, the `old-balance` of $(i+1)^{th}$ entry is equal to the `new-balance` of the i^{th} entry (for all i), is expressed via the `eqBal` relation in Figure 6.

The ability to define functional invariants over the representative element permits elegant reasoning on structure, using induction.

Sequences support the following built-in messages: The $O \ll$ *newElement* message can be used to create a new element of the Sequence O. Elements of the sequence are numbered (starting at 0). $O \ll$ *getCount*(C) returns the number of elements currently present in the sequence O. The N-th element of a Sequence O is referenced in ObjChart notations as $O(N)$.

Object Decomposition

In any large modeling effort, one would like to define objects at an abstract level first and progressively refine them into greater details incrementally. ObjChart promotes such "step-wise refinement" by decomposing objects into sub-parts. The sub-objects are only accessible via the containing composite object. In view of our previous discussion on communication ports of objects, this translates to having named communication ports between any object and its parent, and all its subcomponents. (Note that these are the ports which define the part-of hierarchy. There may be other ports which allow sideways communication between sibling objects, using behavioral relations.) As an example of object decomposition, referring to Figure 7, the checkbook

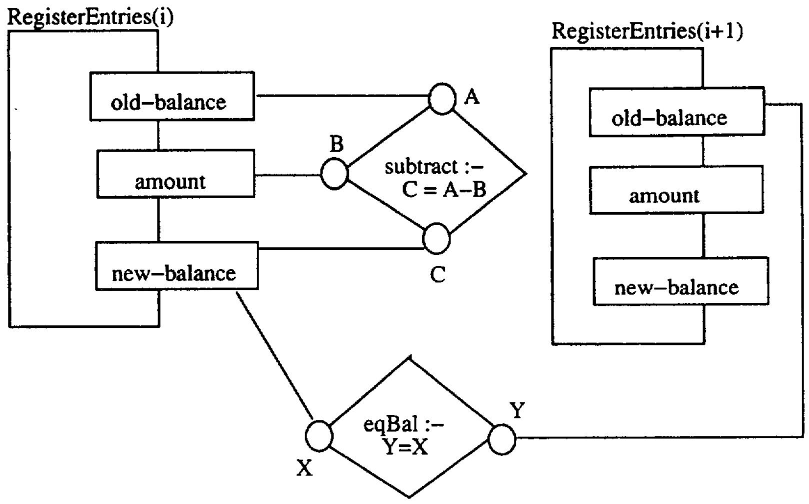


Figure 6: Dependencies between elements of RegisterEntries

register is modeled as an Object `CheckRegister` whose sub-objects are the elements of the Sequence `RegisterEntries`.

Model Execution

ObjChart models of systems are executable. Essentially execution involves interpretations of the FSMs and functional invariants, invoking the builtin messages of the various ObjChart Constructs employed. For example, the checkbook register model defined above, is executable: a `setAmount(A)` message sent to an element `RegisterEntry(i)` will compute the `new-balance` attribute of the element and set the `old-balance` attribute of the element `RegisterEntry(i+1)`. ObjChart Builder, described in [GMD93], is an environment to execute and experiment with ObjChart models.

3 Behavioral Semantics of ObjChart Objects

In this section, we provide an equational framework for defining compositional semantics of reactive behavior of ObjChart objects. We use this equational framework to define the precise meaning of behavioral subsumption of concurrent objects. Finally, we indicate how the same framework can be used to provide a semantics for Contracts.

3.1 The Equational Framework

Reactive behavior of an object is controlled by its state machine. An object may receive messages from its parent, itself, from one of its sub-objects or via one of its ports. For each such message received, its state machine determines the outcome (resulting output messages to parent, children and so forth). Furthermore, the object

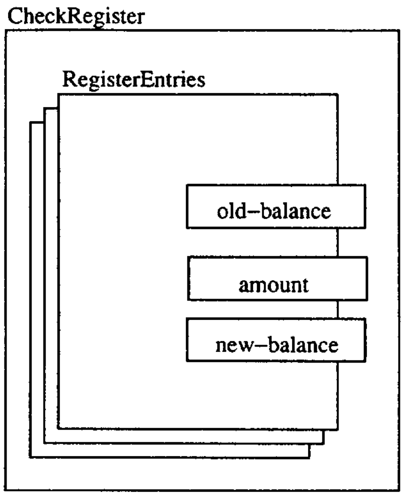


Figure 7: CheckRegister Object

may undergo state change in this process. Externally observable behavior is the relationship among the messages received from and sent to the object's environment, that is, its parent and ports, as shown in Figure 8. Each object O can communicate with its parent (through channel P for input, and P' for output), itself (through the channel marked *self*), all its sub-objects (using channels C_i for output to and channel C_i' for input from the i^{th} sub-object), and other sibling objects via named ports (which get connected to such ports of other objects using relations which describe behavioral dependencies as explained in Section 2). The external behavior of an object is the relation between the sequence of messages on channels $P \cup \mathit{Ports}$ and $P' \cup \mathit{Ports}'$. Evidently, to give a compositional semantics, this relation must be definable in terms of those for the sub-objects, and the description of the object's state machine.

Defining compositional semantics of ObjChart objects is not easy because their external behavior is in general non-functional. The main difficulty comes from their inherent non-deterministic behavior. Since in our model every object executes concurrently with others, the outcome of a given sequence of input messages may be different depending on the possible interleaving of messages from the sub-objects with the input sequence, and the order in which messages in the *action* part of an arc get executed. It is well-known that input-output relation between message sequences on channels $P \cup \mathit{Ports}$ and $P' \cup \mathit{Ports}'$ cannot be expressed functionally. In fact, there have been several proposals made in the literature of non-deterministic data-flow networks to provide compositional semantics in the presence of non-determinism, including [Jon89, Mis90]. In this section we show that these results can be adapted to provide a compositional behavioral semantics for ObjChart objects.

In the remainder of our discussion on this issue, we follow Misra ([Mis90]) more closely. Misra characterizes each component by a set of equations of the form $f(\overline{C}) \leftarrow g(\overline{C})$ ($f \leftarrow g$ for short), where $\overline{C}$ is the set of channels connected to the component of interest, and f and g are continuous functions over the traces on these channels. The behavior itself is one of the so-called *smooth solutions* of the set of equations.

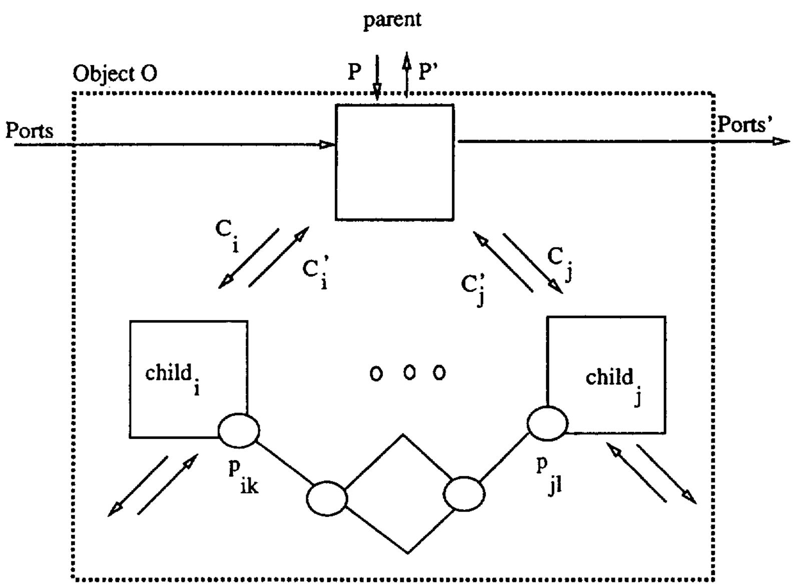


Figure 8: Object Model for Behavioral Specifications

Moreover, for deterministic components, there is a unique smooth solution, which is the least fixpoint of the equations. Given two descriptions of components $f_1 \leftarrow g_1$ and $f_2 \leftarrow g_2$, their combined description is given by $f \leftarrow g$, where f is the tuple consisting of f_1 and f_2, and similarly for g.

Following the work on nondeterministic dataflow networks, we provide the description of each ObjChart object as a network of subcomponents as shown in Figure 9. From Figure 9 we see that each object has three sub-parts (or sub-processes in the

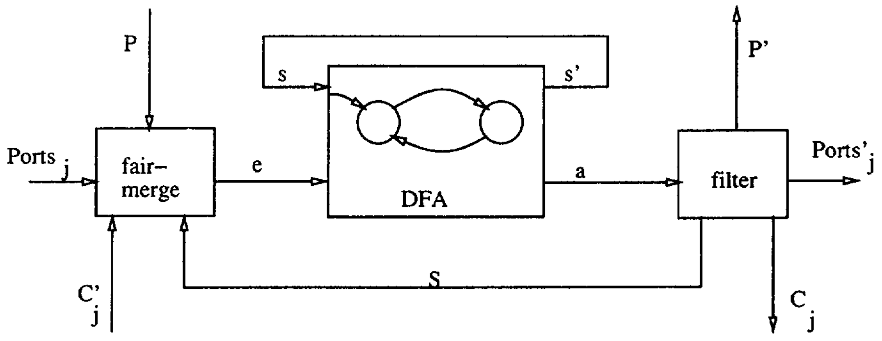


Figure 9: Subcomponents of an ObjChart Object

data-flow network terminology), namely:

Fair-merge This is a non-deterministic process which gets input from the *parent*, *self* and $C'_j, 1 \leq j \leq n$ channels, and merges them to produce a sequence of events (e)

for the state machine of the object under consideration. The following equations define the Misra style behavior of this component:

$$\text{select}(\textit{parent}, e) \leftarrow P,$$

$$\text{select}(\textit{self}, e) \leftarrow S,$$

$$\text{select}(\textit{child}_j, e) \leftarrow C'_j, 1 \le j \le n.$$

Here we have assumed that each of the channels have a sequence of values, and *select* is a function which gives the subsequence in which each element has the tag given by the first argument. Note that there have to be more components to actually attach and remove tags from the channel values, which we do not specify for simplicity. See Section 4.10 of [Mis90] for complete details.

DFA This process implements the transition table for the deterministic state machine for the current object. The actual description is given in Table 1 below. We write $\langle s, e, c, a, s' \rangle \in \textbf{transitions}(O)$ whenever there is an arc with event e, condition c and actions a from state s to state s'. Here σ is a substitution for

```
DFA : $State \times Event \rightarrow State \times Actions$
$\text{DFA}_O(\bot, e) = \langle \bot, \{\} \rangle$
$\text{DFA}_O(s, e) =$
    if $\exists.\langle s, e', c, a, s' \rangle \in \texttt{transitions}(O)$
        such that $\exists\sigma : e'.\sigma \equiv e$ and $c.\sigma$ holds, then
        $\langle s', a.\sigma \rangle$
    else
        $\langle \bot, \{\} \rangle$
```

Table 1: Transition Table for DFA

the parametric variables in e' which is a matching of e' with e (i.e., $e'.\sigma \equiv e$).

Filter This is a non-deterministic process, which receives a sequence of sets as input; each set contains elements of the form $O_j \ll e_j$. It produces a non-deterministic sequence from each such set and outputs the relevant portions of this sequence on the output channel corresponding to each target object. Figure 10 gives the detailed description of the sub-processes required for the purpose. We now

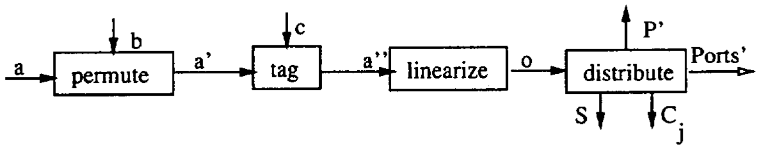


Figure 10: Description of Filter

provide the necessary equations (in the figure, we use b as a sequence of random numbers ([Mis90]), and c as a sequence of natural numbers, which is easy to generate using a counter):

$$a' \leftarrow permute(b, a),$$

$$a'' \leftarrow \langle c, a' \rangle.$$

In this description, *tag* uses natural numbers to tag its input messages such that each input set gets a unique tag, while *permute* generates a random permutation of its input set, and *linearize* is defined as in Table 2.

$$\begin{array}{l} linearize(Tag,\ Set) = \\ \quad \langle Tag, e_1 \rangle, \ldots, \langle Tag, e_n \rangle \\ (where\ e_j, 1 \leq j \leq n\ is\ the\ j^{th}\ element\ of\ Set) \end{array}$$

Table 2: Description of Linearize

Finally, *distribute* strips off the tags and the preceding object identifier O_j; it then sends the event (e_j) on the channel for O_j (which could be one of *parent*, *self*, any C_i for a child or Ports'_j for a port).

Note that *fair-merge* and *filter* are non-deterministic. This coincides with our intuition that the essential non-determinism for an object is due to (1) the potential interleaving of message sequences from its parent, its sub-objects and its ports; and (2) the order in which messages in the actions get executed. Furthermore, *fair-merge* and *filter* are duals of each other, since one merges input events to the state machine, while the other distributes outgoing actions. However, their equational descriptions are somewhat different since we are dealing with single messages for the event part, and with sets of messages for the action part.

Finally, we provide a description of the relation object (for behavioral dependencies using state machines) as shown in Figure 11. Note that this description is in fact a

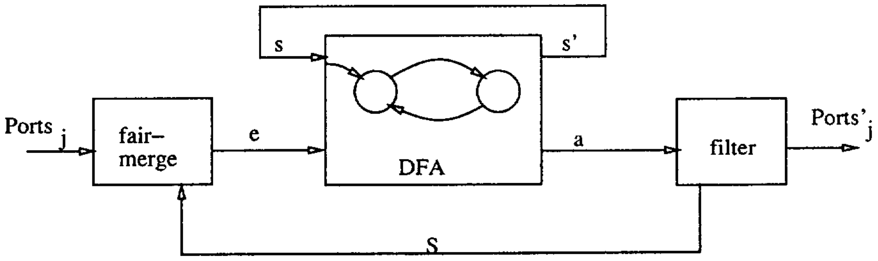


Figure 11: Description of Behavioral Relation

simplification of the one for an ObjChart object, as shown in Figure 9.

We are now ready to describe the behavioral semantics in terms of traces. A *trace* on a set of K channels is a sequence (possibly infinite) of pairs of the form $\langle c, v \rangle$, where v is a value appearing on the channel c. The *description* of an ObjChart object is the set of equations for its network of components as defined by Figures 8 and 9. The *behavioral denotation* of the object is the set of all *smooth solutions* ([Mis90]), that is, the admissible traces, of its description. The *external behavior* of an object is the projection of its behavioral denotation onto the channels $P \cup Ports$ and $P' \cup Ports'$.

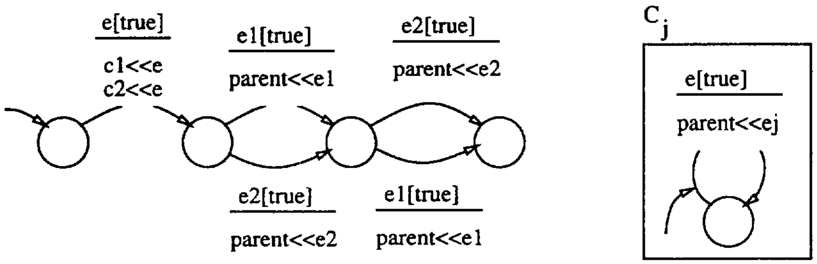


Figure 12: Example of the Trace Model

Example 1. *Consider the state machine of an object as shown in Figure 12, where the state machine on the left is for the object O which has two sub-objects $(C_j, j \in \{1,2\})$, each of which has a state machine as shown on the right. For this collection, one smooth solution is given by:*

$$\langle P, e\rangle, \langle C_1, e\rangle, \langle C_2, e\rangle, \langle C_2', e_2\rangle, \langle P', e_2\rangle, \langle C_1', e_1\rangle, \langle P', e_1\rangle,$$

which means that for the object O, the external behavior is given by:

$$\langle P, e\rangle, \langle P', e_2\rangle, \langle P', e_1\rangle.$$

Theorem 1 (Composition,[Mis90]). *Consider an object O with N sub-objects and M behavioral dependency relations between them, such that the i^{th} sub-object has a description $f_i \leftarrow g_i$ and the j^{th} relation has a description $f_{rel_j} \leftarrow g_{rel_j}$. Furthermore, let $f_{fair\text{-}merge} \leftarrow g_{fair\text{-}merge}$ be the description of the* fair-merge *component of O, $f_{DFA} \leftarrow g_{DFA}$ of the* DFA *component of O and $f_{filter} \leftarrow g_{filter}$ of the* filter *component of O. Then $f \leftarrow g$ describes O, where f is the tuple*

$$\langle f_1, \ldots, f_N, f_{rel_1}, \ldots, f_{rel_M}, f_{fair\text{-}merge}, f_{DFA}, f_{filter}\rangle$$

(and similarly g).

3.2 Object Subsumption

The equational framework presented above gives us an intuitive and elegant way to talk about substitutability and subsumption of objects. The general denotation of an object is a triple, with a structure component, a behavior component and a component for functional invariants. Since we have used equations to define both behavior (due to Misra) and functional invariants, we can in essence treat ObjChart objects as *collections of equations*, and therefore, we define object equivalence (subsumption) as the equivalence (subsumption) of the sets of solutions of these equation. That is, for two ObjChart objects O_1 and O_2, we say that O_2 subsumes O_1 if and only if the following conditions hold:

Attribute The set of external attributes of O_2 subsume those for O_1. In other words, for each attribute of O_1, O_2 has an attribute which accepts the same "type" of values.

Port The set of ports of O_2 subsume those for O_1. In other words, for each port of O_1, O_2 has a port which accepts the same "type" of messages.

Behavior Every *smooth* solution of O_1 (refer to Section 3.1) is also a smooth solution for O_2. In other words, O_2 has at least the same set of legal traces as O_1.

Functional Invariants The solutions (least fixpoint) of each functional invariant of O_1 is also a solution to the corresponding functional invariant of O_2.

From the above definition it is clear that in general it may not be possible to algorithmically decide if an object subsumes another, since we are using semantic equivalence for both behavior and functional invariants. However, we now identify a useful class of subsumption, where the subsuming object is obtained by copying and extending the subsumed object, by adding attributes, states, arcs, or actions. For such a case, it is easy to show that the subsumption problem is decidable, by comparing the two structures in question.

We illustrate this idea with a simple example, in which we extend the state machine of an alarm clock to produce one which also has the "snooze" feature, as shown in Figure 13.

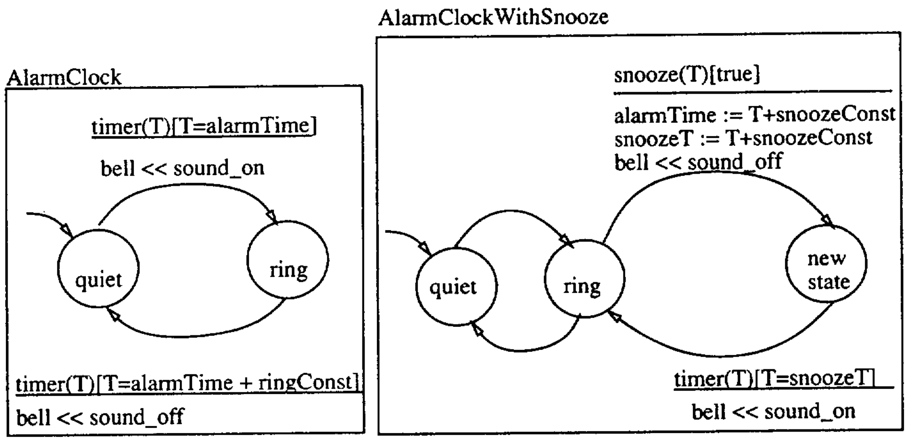


Figure 13: Alarm Clock with and without Snooze

The *copy-and-extend* paradigm exemplified here, is quite useful for the system designer, because it shows that a system may be developed incrementally by reusing and extending previously defined models. The ability to check subsumption algorithmically for such cases enable us ascertain that incrementally extended models are indeed semantically proper extension of the original ones.

3.3 Semantics of Contracts

"Contracts" was introduced in an earlier paper [HHG90] as a way of capturing multi-object behavioral collaboration. That paper did not, however, provide a precise semantics for Contracts.

The equational framework described here can provide precise semantics for Contracts. A Contract is intended to express a set of constraints on the order of message exchanges among a set of participants. These constraints are precisely the set of equations over traces where the participants are the channel names. Contract composition becomes union of the set of equations for the sub-Contracts (with possible channel name unification, whenever the output of one channel is connected to the input of another). Contract refinement has the same semantics as that of object subsumption discussed in Section 3.2.

In the visual notations of ObjChart, Contracts are depicted by the behavioral relation construct, such that the typed ports represent the Contract participant roles, and the constraints are defined by the relation's state machine. In effect, the state machine of the Contract controls the order of message exchanges between the objects connected to the ports.

4 Specification Structuring and Reasoning

This section discusses how ObjChart notations help structure system specifications naturally, using object hierarchies. We also observe how tracing causality chains along this object hierarchy helps organize reasoning about system behavior. Subsection 4.1 presents an ObjChart specification of a version of the Elevator Control System problem and Subsections 4.2 and 4.3 makes several observations about reasoning with ObjChart models.

4.1 Modeling of Reactive Systems in ObjChart

In this section we specify a version of an Elevator Control System using ObjChart notations. This example is typical of reactive systems and has been used as a test problem for judging efficacy of specification languages [Dav89]. The example shows that *hierarchical object based decomposition provides a natural structure of the specification, and keeps the state machines under manageable size.*

We first model the overall system structure as `Building` object, as shown in Figure 14. `Building` is decomposed into its sub-objects, `Elevators` and the floors. The floors themselves are decomposed into their sub-objects, namely the buttons up and down. Note that the terminal floors, `floor(0)` and `floor(m)`, are modeled separately, since terminal floors have one button each.

We represent the assembly *Elevators* as a Sequence, in which the representative element specifies an elevator, as shown in Figure 15. Each elevator has three attributes: `current-direction` of movement, `current-floor` recording the current floor number and `door` keeping the status information about its door, whether opened or closed.

Each elevator is further decomposed into its sub-objects, a set of internal buttons (`button_set`) and a request queue (`request_queue`). The request queue has the

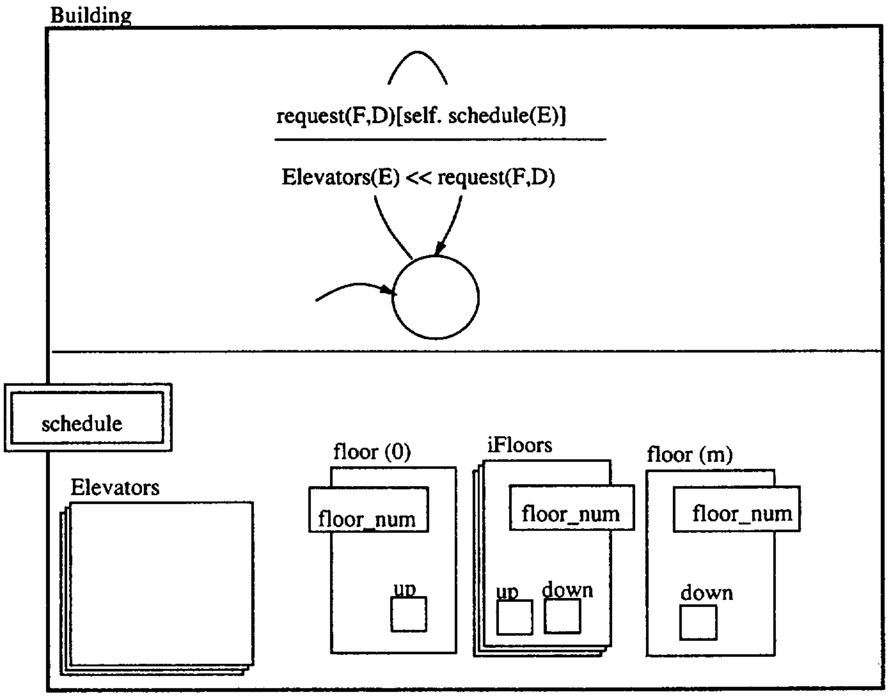


Figure 14: Overall System Structure

following four methods:

- **pending?(F,D)** queries if a particular request from a Floor F and direction D is in the queue.
- **toMove?(D)** determines if there are further requests for going in the Direction D. This method essentially implements the elevator sweeping algorithm.
- **enq(F,D)** adds to the queue the request from Floor F for going in direction D.
- **deq(F,D)** removes the request from Floor F for direction D from the queue.

Next, the state machine of Figure 16 models the reactive behavior of an elevator. An elevator could be in one of three states, namely: **dead** (this is a quiescence situation in which there are no requests to be satisfied), **stopped** (an elevator is in this state when it is actually servicing a request) and **moving** (in which the elevator is using its sweeping algorithm to continually service requests for a particular direction). The events handled by the state machine are the following:

- **request(F,D)** event denotes a request to go to floor F and direction D. These events are generated by either pushing the floor buttons or by pushing the buttons inside the elevators.

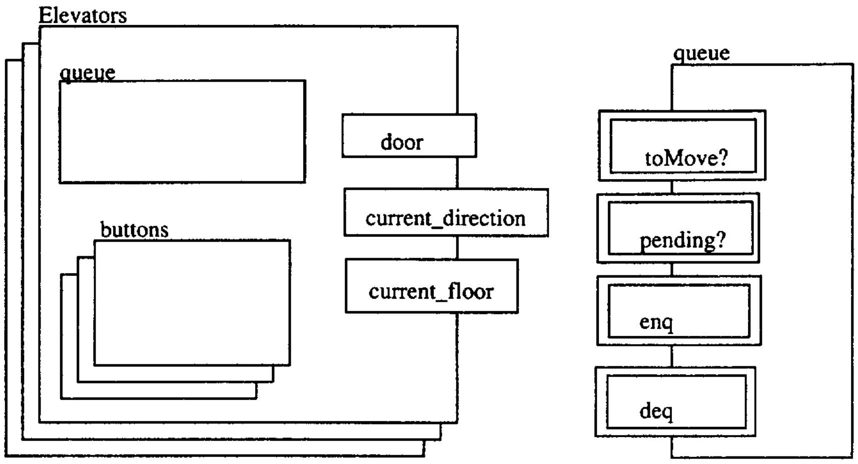


Figure 15: Sequence of Elevators

- **floor_near(F)** event is generated by the sensors (external to our model) when a floor F is approached.
- **door_close** event is generated by perhaps an external timer when certain time elapses after an elevator door is opened.

Finally, we briefly specify the reactive behavior of some of the other objects, as shown in Figure 17.

This section highlights how specification of a system is structured easily and naturally into the specifications of its constituent objects. Stepwise decomposition of objects into sub-objects results in each object with a very local scope. Such locality enables one to specify the objects independently and simply. For example, the FSM of an elevator is specified using only 3 states and is independent of the state machines of the elevator's buttons. In the next sub-section, we shall see how reasoning about a system under modeling is structured very naturally along inspecting objects locally.

4.2 Reasoning with ObjChart Models

In this section we use the elevator example developed in Section 4.1 to illustrate that in ObjChart reasoning about properties of a model can be done at a very intuitive level. *The structure of such reasoning arguments follow closely the system structure.* A typical such reasoning process about a reactive system involves making eventuality arguments and ascertaining safety properties. Eventuality arguments are effected by following causality chains through the communicating state machines, showing that progress making states are entered eventually. Safety properties of an object, on the other hand, being invariants, are established by examining all states of an object.

Consider the elevator model described in Section 4.1. A natural desire of the system analyst may be to prove the following property about the model:

> If an *up* button is pressed at floor f, eventually some elevator comes to the floor and opens the door.

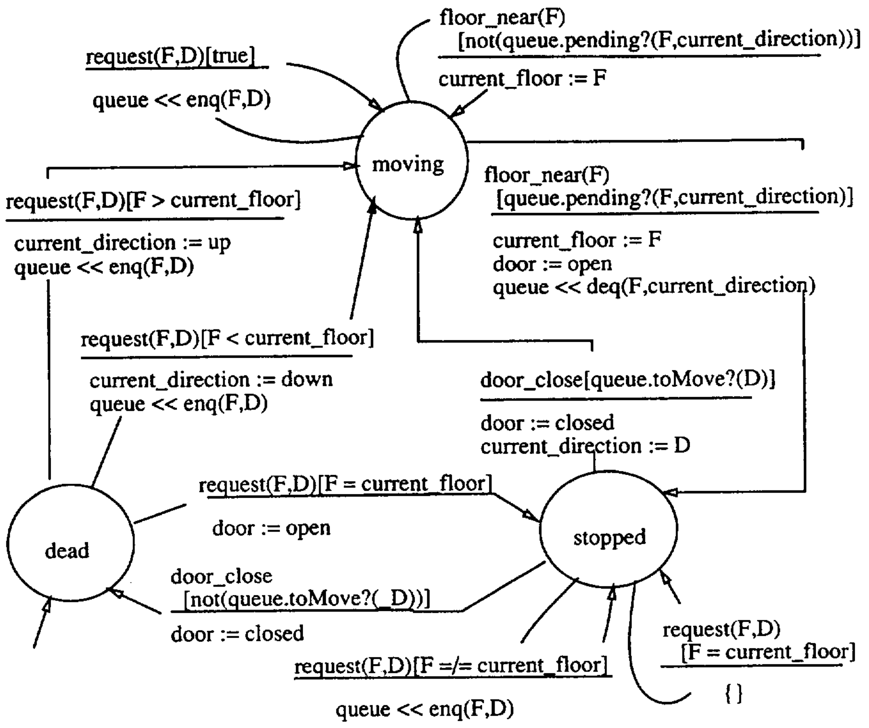

floor_near(F) [not(queue.pending?(F,current_direction))]
current_floor := F
request(F,D)[true]
queue << enq(F,D)
moving
request(F,D)[F > current_floor]
current_direction := up
queue << enq(F,D)
floor_near(F) [queue.pending?(F,current_direction)]
current_floor := F
door := open
queue << deq(F,current_direction)
request(F,D)[F < current_floor]
current_direction := down
queue << enq(F,D)
door_close[queue.toMove?(D)]
door := closed
current_direction := D
request(F,D)[F = current_floor]
door := open
dead
stopped
door_close [not(queue.toMove?(_D))]
door := closed
request(F,D)[F =/= current_floor]
queue << enq(F,D)
request(F,D) [F = current_floor]
{}

Figure 16: State machine for an individual elevator

We proceed by breaking this top-level goal into various smaller sub-goals which we solve one at a time:

1. **Every button push reaches some elevator** E. To establish this, one can trace the causality chain from the individual button (for floor f, and direction *up*), and establish that when this button is pressed, some elevator (say E) eventually gets *request*(f,*up*). For example, one possible such chain could be

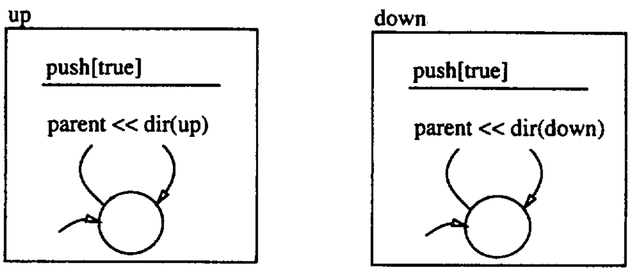
$$\text{push} \leadsto \mathit{floor}_f \ll \mathit{dir}(\mathit{up})$$

$$\mathit{floor}_f \ll \mathit{dir}(\mathit{up}) \leadsto \mathit{Building} \ll \mathit{request}(f, \mathit{up})$$

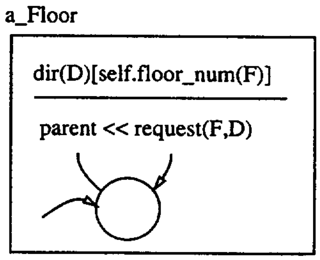
$$\mathit{Building} \ll \mathit{request}(f, \mathit{up}) \leadsto \mathit{Elevators(E)} \ll \mathit{request}(f, \mathit{up}),$$

   i.e., eventually, elevator E will receive the request (some scheduling algorithm is employed by the *building* to determine the actual value of E).[5] Refer to Figures 14 and 17 of Section 4.1 for more details on the actual structure of the model. Note that, in general, it will be necessary to establish that each object in such a causality chain is indeed in a state in which it is ready for the corresponding

[5]The notation $a \leadsto b$ means that a causes b and in fact, a is an event and b is an action in some arc label of an FSM.



FSM of the individual buttons



FSM of an individual floor

Figure 17: Specification of Behavior

message. For our example, all the intermediate objects involved, from a floor-button to an elevator, have a single state in their state machine, and so this fact is easy to ascertain.

2. **Elevator E services the message *request*(f, up) eventually.** To do so, however, we have to consider the following sub-goals:

   (a) **Elevator E never loses any request.** This is a *safety* property that can be established by inspecting all the states of the elevator (Figure 16), and noting that in each state the message *request*(f, up) is handled. Hence, the request gets immediately satisfied if E was in "dead" state at floor f or the request gets queued if the elevator was in either "moving" or "stopped" states.

   (b) **Whenever elevator E has a non-empty queue, it returns to the "moving" state.** This can be deduced from the facts that (i) *toMove?* would always hold whenever the queue is non-empty, and (ii) whenever the elevator enters the "stopped" state a *door_close* event occurs eventually. Furthermore, if the elevator was in a "dead" state, a request will send it to the "moving" state.

   (c) **In the "moving" state, the elevator makes finite progress towards satisfying *request*(f, up).** For this we assume receipt of a correct sequence of *floor_near* events whenever the elevator is in the "moving" state. As a result, the elevator may either transit to the "stopped" state or may continue in the "moving" state. In the former case, we use the previous argument to show that either the request has been satisfied, or the elevator must return to "moving" state again. In the latter case, we say that we have progressed one step closer to satisfying the request under consideration. Note that it may be the case that the elevator is actually moving away from the request under consideration, however, this is still progress, because the elevator is getting closer to its next sweep (i.e., direction change) in which the current request would be serviced.

The above arguments assume:

- All the methods of the *queue* object work correctly. In particular, the query *toMove?* is assumed to evaluate to true when the queue is non-empty. In fact, if we did specify the *queue* object completely, using its state machine, then we could have actually established this fact using similar causality argument about that state machine. However, in this paper, due to lack of space, we have simply assumed that the interface for queue is pre-defined in terms of methods which satisfy certain specifications.
- The external events (*door_close* and *floor_near*) arrive correctly whenever required.

This example shows that the ObjChart formalism allows system modelers to intuitively reason about their models by inspecting *causality chains* among the FSM-s of the objects under consideration. Furthermore, hierarchical decomposition modularizes, in most cases, such reasoning to inspecting a single object.

4.3 Orthogonality of Functional Invariants from Causality Chains

In this section we describe an example to show that our method for specifying functional invariants obviates the need of reasoning with explicit pre- and post-conditions. This example also illustrates that reasoning with functional invariants is orthogonal to reasoning with the causality chains.

The example involves specifying a bicycle, as shown in Figure 18. In our partial description of the system we have two objects, called `rear` and `front` which denote the wheels of the cycle. Each wheel has three attributes, namely: omega (for angular velocity), `radius` and `velocity`. The functional invariants describe the relation among angular and linear velocities of each wheel and the fact that the linear velocities of the two wheels must be equal. Furthermore, the WheelAssembly responds to a `rotate` message (which is generated from the pedal object in a more detailed scenario).

Suppose we want to establish that turning the pedal at an angular velocity W, will turn the front wheel at angular velocity $W * rear.radius/front.radius$.

This can be accomplished using the following two orthogonal steps:

- establish that pedal's message `rotate(W)` will set the value of `rear` object's omega attribute. That is, $rear.omega = W$. This is done by inspecting causality chain from pedal to WheelAssembly's FSM.

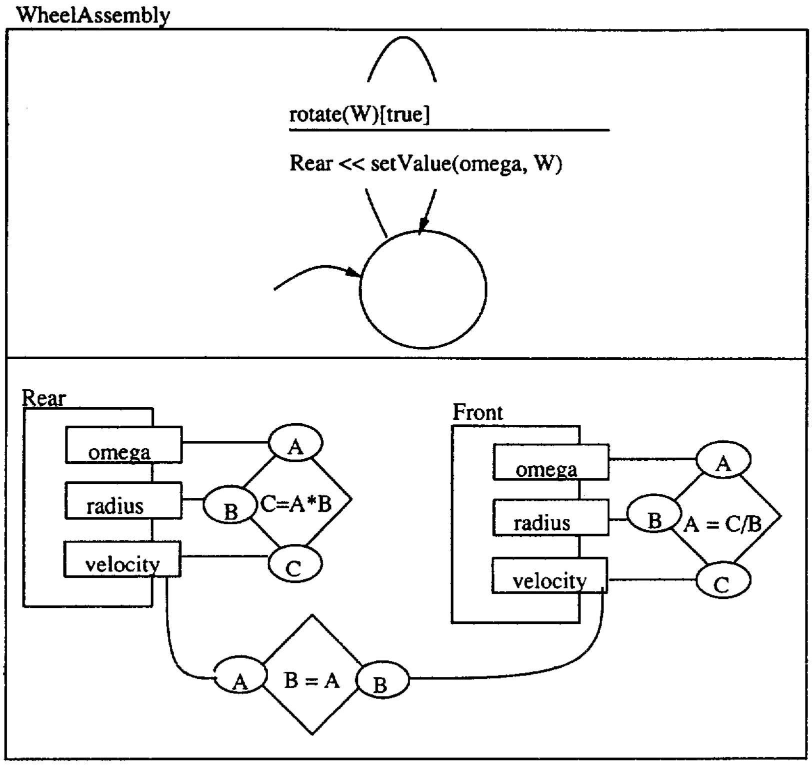


Figure 18: Orthogonality of FSMs and Functional Invariants

- establish that the angular velocities of two wheels are inversely proportional to their radii. That is, $front.omega = rear.omega * rear.radius / front.radius$. This can be established by following the chain of functional invariants.

This example illustrates the fact that reasoning about an ObjChart model is a truly orthogonal process: Follow a causality chain (any system trace) to find messages requesting value change and reason about the value network separately using techniques like function composition. In contrast, in some systems one has to explicitly deal with pre- and post-conditions in order to establish that a particular trace is valid for the object under consideration. We believe such orthogonality simplifies reasoning tremendously.

5 Comparison

In contrast to other state machine based behavior modeling languages, ObjChart uses object decomposition as the single system refinement paradigm, maintains orthogonality between control flow and value propagation, introduces *Sequence* object which embodies structural induction, and allows tracing causality chains in time linear in the size of the system. We elaborate these issues in the following sections.

5.1 Statechart

Statechart [Har87] proposes an elegant extension to FSMs so as to allow hierarchic refinement of states, decomposition of the system state machine into a set of orthogonal sub-component FSMs and multilevel concurrency utilizing broadcast communication of events among component state machines. Due to the hierarchy of states and substates, Statechart has also the notion of history nodes which remember the sub-state from the previous entry. Furthermore, chain reactions of internal events, triggered by a single external event, cannot interleave with arrival of further external events [HPSS87]. These two features seemingly complicate the semantics of Statechart—see Section 3 of [HPSS87].

In contrast, we use hierarchical object decomposition in lieu of orthogonal components for keeping the state machine of each object simple. We also use directed communication, unlike broadcast, along the decomposition hierarchy.

As shown in Section 4.2, typical reasoning for reactive systems involve chasing causality chains. Hierarchically directed communication of ObjChart allows us to analyze the effect of a chain reaction, caused by an external event, by considering only affected objects as opposed to all the objects in the system. For example, for a system of N objects where each object has at most c sub-objects, a k-step transition sequence may involve $O(c^{k+2}) = O(N)$ objects, i.e., number of inspections is linear in the size of the system. In contrast, for Statechart, analyzing a $k-$step chain reaction (i.e., a k-step micro-step sequence) requires exponentially many $(O(N^k))$ inspections. To see this consider a system which has N "AND" components. An external event e may cause state transition for any of these N subcomponent. Furthermore, each such transition may generate new events, which may cause transitions for all the N components again. Thus, each event has the potential to cause transitions for any of the N objects, and therefore, for every event one may potentially have to look at the entire system to find its effect, and thus the result. Finally, in ObjChart, communication between two objects in different parts of the hierarchy may need $O(logN)$ steps, and thus involve $O(logN)$ components. In Statecharts, although the same communication may be accomplished in a single step due to broadcast, it would still require inspections of all N components to find its effect.

Moreover, our semantics is relatively simple due to the following two facts. First, since hierarchically decomposed objects in practice need very few states, we do not have "OR" decomposition and thus, we do not have the complications due to History Nodes. Second, since we allow interleaving of external messages with those from the sub-objects, our semantics is free from the complications due to the asymmetry of macro and micro steps.

Thus, ObjChart accomplishes the same objectives as Statechart by using the object decomposition paradigm with a simpler semantics and a much better time complexity in analyzing system models.

5.2 ObjectChart

ObjectChart [BCHA90, CHB92] retains the traditional notion of classes and uses Statecharts to define the behavior of classes. They have used both "AND" and "OR" decomposition of Statecharts. It is not apparent how the orthogonal components of a Statechart within a single class can communicate and synchronize without broadcast, because their claimed directed communication is only for sending messages from one object to another and not for communication among orthogonal components within a single class.

As for operational semantics, ObjectChart is very restrictive: execution of each message (called an interaction) has to complete before the next one begins in the entire system. Moreover, execution of a service request cannot involve a nested request for another service. Thus, the concurrency allowed in Statechart formalism is not even utilized in ObjectChart's semantics.

ObjectChart also allows annotating states by state invariants (i.e., conditions holding at a given state) and associating pre- and post-conditions with service requests (message executions) in order to reason with the effects of these service requests. However, ObjectChart models have *not* achieved *orthogonality*—a trace derived from the state machines may still be disallowed if the necessary pre-conditions of the constituent service requests are not satisfied. Therefore, dealing with pre- and post-conditions destroys the intuitive simplicity of reasoning with state machines.

In contrast, we treat value propagation by functional invariants. The beauty of this approach is that these invariants always hold and therefore, any causality chain determined from inspecting state machines alone is always legal for the system. The result is that reasoning with ObjChart models consists of two truly orthogonal phases: follow a causality chain to identify any value update and then use functional invariants to determine consequent value propagation (see Section 4.3 for an example).

Finally, ObjectChart, being class based, does not directly support part-of decomposition at the object level. As such, the flatness of the traditional class model makes it difficult to express hierarchical layering of a system [BBJS92].

5.3 OMT

OMT uses three facets to describe a system, namely: the object model (E-R diagram), the dynamic model (Statechart) and the functional model (DFD). They have two major problems for describing large systems: First, their dynamic descriptions are at the class level, and therefore, like ObjectChart, the flat structure of classes is inelegant for describing hierarchically layered systems. Second, a system description gets refined along these three separate models, thus making it difficult to maintain coherence between these models [BBJS92, HC91].

As such, any system design methodology employing separate refinement hierarchies, for example, [Boo91, Har88b] and a list of others, have this inherent impediment of maintaining coherence among the different models. On the contrary, in ObjChart, object decomposition is the only system refinement paradigm. We believe that object is the fundamental element in any object oriented system, and therefore, object decomposition must be the predominant (and perhaps the sole) system refinement discipline.

6 Conclusion

In summary, ObjChart formalism aims to combine the intuitive appeal of visual notations with preciseness of formal methods. The constructs have been carefully designed, on one hand, to promote the tangible thinking in terms of objects as the sole system building-blocks and, on the other hand, to provide facilities such as functional invariants and structural induction (via Sequence object) which are found useful in reasoning.

By modeling many examples in ObjChart, we have made several encouraging observations: (1) system specifications are structured naturally as the objects of the system and (2) the same object structure is useful in guiding the reasoning process in a natural fashion. In many cases, such reasoning is very *local*, requiring inspections only within an object. Furthermore, we observe that causality chains of message passing can be reasoned *orthogonally* from reasoning with networks of functional invariants. Finally, tracing causality chains is the corner stone for reasoning with ObjChart models and can be accomplished in time linear in the size of the system. All these factors lay the foundations for reasoning with ObjChart models to scale up to practically sized systems. However, much more work is needed to realize these potentials.

Currently, we have built a prototype environment, called ObjChart Builder [GMD93], to create, edit, reuse and execute ObjChart models. In practical terms, execution of ObjChart models enable one to understand the cause and effect relations among objects. ObjChart Builder enables a system analyst to debug the conceptualization of a system directly at the level of ObjChart models by execution and inspection of the states of the objects.

In developing ObjChart notations, we have focused on object-level dynamics as the key component of system modeling and have not emphasised the static modeling features typical of class-based notations. For example, we cannot express, in ObjChart notations, disjunctive facts such as "Vehicles are either Cars or Trucks". Extending the Sequence Object construct by allowing alternative representative elements will be a potential solution.

Finally, using equations over traces as the denotation of objects provided us with an elegant framework to define object substitution and composition. However, the current framework is based on a fixed network of components and cannot handle the semantics of dynamic object creation. Semantics of Actors, on the other hand, can handle dynamic creation; however, the semantics is operational in nature. We believe that extension of the concept of smooth solutions to equations involving higher order functions may provide a general framework in which dynamic object creation can be handled.

Acknowledgement

The authors wish to thank several people who have, over the years, been involved in the Tangible Programming Project — Mukesh Dalal, Ashish Gupta, Constantine Laufer and Richard Helm. They all helped directly or indirectly in developing the ideas presented here, although the authors remain responsible for any remaining flaws. We would like to thank Rajendra Panwar and Jarir Chaar for proof-reading earlier versions of this paper. We are also indebted to Satbir S. Dhaliwal, who implemented a major portion of ObjChart Builder in C++.

References

[BCHA90] S. Bear, D. Coleman, F. Hayes and P. Allen. Graphical specification of object-oriented systems. In *Proc. OOPSLA/ECOOP '90* Vol. 25, No. 10, pages 28–37, 1990.

[Boo91] G. Booch. *Object-oriented design with applications.* The Benjamin Cummings Publishing Company Inc., 1991.

[BA81] J. D. Brock and W. B. Ackerman. Scenarios: a model of nondeterminate computation. In *Formalization of programming concepts*, LNCS 107, pages 252–259, 1981.

[BBJS92] B. Bruegge, J. Blythe, J. Jackson and J. Shufelt. Object-oriented system modeling with OMT. In *Proc. OOPSLA '92* Vol. 27, No. 10, pages 359–376, 1992.

[CM89] K. M. Chandy and J. Misra. *Parallel program design.* Addison-Wesley, 1989.

[CHB92] D. Coleman, F. Hayes and S. Bear. Introducing ObjectCharts or how to use Statecharts in object-oriented design. IEEE Transactions on Software Engineering, Vol. 18, No. 1, 1992.

[Dav89] N. Davis. Problem Set for the Fourth Int. Workshop on Software Specification and Design. In *Proc. of the Fourth IEEE Int. Workshop on Software Specification and Design*, Monterey, USA, 1989.

[GMD93] D. Gangopadhyay, S. Mitra and S. S. Dhaliwal. ObjChart-Builder: An environment for executing visual object models. IBM Technical Report. Submitted for publication. 1993.

[Har87] D. Harel. Statecharts: A visual formalism for complex systems. Science of Computer Programming, Vol. 8, pages 231–274, 1987.

[HPSS87] D. Harel, A. Pnueli, J. P. Schmidt and R. Sherman. On the formal semantics of Statecharts. In *Proc. of the Second IEEE Symp. on Logic in Computer Science*, pages 54–64, 1987.

[Har88a] D. Harel. On visual formalisms. Communications of the ACM, Vol. 31 (5).

[Har88b] D. Harel et al. Statemate: A working environment for the development of complex reactive systems. In Proc. of the Tenth Int. Conf. on Software Engg., pages 396–406, 1988.

[HC91] F. Hayes and D. Coleman. Coherent models for object-oriented analysis. In *Proc. OOPSLA '91*, Vol. 26, No. 11, pages 171-183, 1991.

[HHG90] R. Helm, I. M. Holland and D. Gangopadhyay. Contracts: specifying behavioral compositions in object-oriented systems. In *Proc. OOPSLA/ECOOP '90*, Vol. 25, No. 10, pages 169–180, 1990.

[Jon89] B. Jonsson. A Fully Abstract Trace Model for Dataflow Networks. In *Proceedings of the sixteenth annual Symposium on Principles of Programming Languages*, pages 155–165, 1989.

[MP92] Z. Manna and A. Pnueli. *The temporal logic of reactive and concurrent Systems.* Springer-Verlag, 1992.

[Mis90] J. Misra. Equational reasoning about nondeterministic processes. *Formal Aspects of Computing*, Vol. 2, pages 167–195, 1990.

[Pnu86] A. Pnueli. Application of temporal logic to the specification and verification of reactive systems: A survey of current trends. In *Current Trends in Concurrency*, LNCS 224, pages 510–584, 1986.

[RBPEL91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy and W. Lorensen. *Object-oriented modeling and design.* Prentice-Hall, 1991.

[SM88] S. Shlaer and S. Mellor. *Object-oriented systems analysis.* Yourdon Press, 1988.