Towards Execution Models of Distributed Systems
A Case Study Of Elevator Design

John V. D'Anniballe
jvd@utrc.utc.com
(203) 727−7067
M/S 129/85

Philip J. Koopman, Jr
koopman@utrc.utc.com
(203)727−1624
M/S 129/48

United Technologies Research Center
411 Silver Lane
East Hartford, Ct. 06248

Abstract

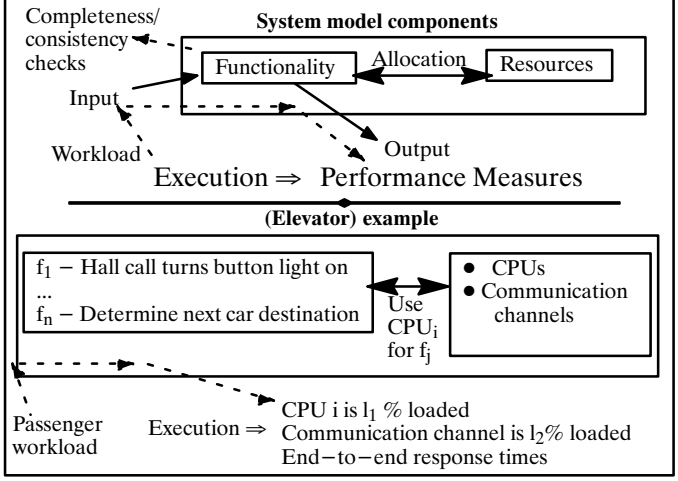
Many of United Technologies' products contain or will soon contain a distributed network of processors. In order to explore issues related to designing such systems, two different methods of modeling system functionality have been applied to a simplified elevator controller. One method results in data flow oriented models which are executable within a context of the association of units of functionality with distributed processors. Execution thus produces processing and communication workloads which can be used for design analysis. In order to allow rapid assessment of alternative designs, an automated approach was developed which allowed the units of allocated functionality to be arbitrarily fine. However, neither the data flow approach nor this automated allocation handle the complexity of large models sufficiently well. A similar elevator model has been defined with a technique which combines object−oriented analysis with formal specification. This combination avoids unnecessary complexity yet allows the model to have a formal semantics, which is necessary (but not sufficient) to achieve the key methodological requirement of executability. Future work will integrate the object−oriented and data−flow approaches into a framework which supports specification, automated fine−grain allocation, and execution for large models.

1. Introduction

The products of United Technologies include elevators, jet engines, helicopters, radars and sonars, heating and ventilating equipment, vehicle components, and electronic subsystems. All of these products contain or may soon contain distributed embedded computers. The various types of models which contribute to the development of these systems are, in general, component−oriented, and do not lay a basis for system−level tradeoffs.

The diverse methodological requirements for system−level models include the capability to effect multiple views of system functionality, fast and inexpensive construction, and automated analyses; as well as the capability to accommodate incompleteness when it is desired or necessary to suppress detail. Some of these requirements are in conflict with others. As the digital electronic content of embedded systems increases, it becomes more important to overcome the inhibitors to cost−effective system−level models.

Figure 1 − System Level Models



1.1 Execution models

Motivation. Our research program is focused on the construction of system level models which, at least conceptually, contain the components shown in Figure 1.

The main objective is to represent the functionality at a level of detail so that execution based on some workload and an allocation to resources such as CPUs and communication channels yields accurate measures of the system's performance. This would enable one important class of system−level tradeoffs.

Executability would also lay a basis for analysis of a design's robustness through observation of the effects of failures in functions and resources. Finally, and independent of any allocation and execution, the propagation of many design errors would be prevented by completeness and consistency checks on the functional representation.

Inhibitors. Two main factors inhibit the timely construction and execution of adequate models:

- the functionality component must have a formal semantics — or be straightforwardly traceable to a representation *with* a formal semantics — without having the detail (and complexity) of the actual system.
- the allocation process must not only be automated, but unobtrusive in that it cannot require significant changes to the functional representation nor dictate unnatural structuring or methods to avoid such changes.

We have made progress in both areas though neither requirement is judged to have been sufficiently met.

1.2 Elevator controller test case

A simplified, single−car, elevator controller is being used as a demonstration vehicle for the methods under development. Current high−end elevators are both physically and functionally partitioned into several different CPU boards. In a simplified model, these partitions are: sensors, signalling of lights, doors, drive/brake, motion control, and scheduling.

Section 2 discusses the application of a "baseline" method to the elevator problem. The method is based on models constructed with the SES/workbench™ [SES 92] toolset. These models are data−flow based, graphical, and executable. The baseline method is complete with respect to Figure 1 — a system level model can be executed and is additionally augmented with a character−based animation of car movement, doors, button lights, hall direction lanterns and passengers.

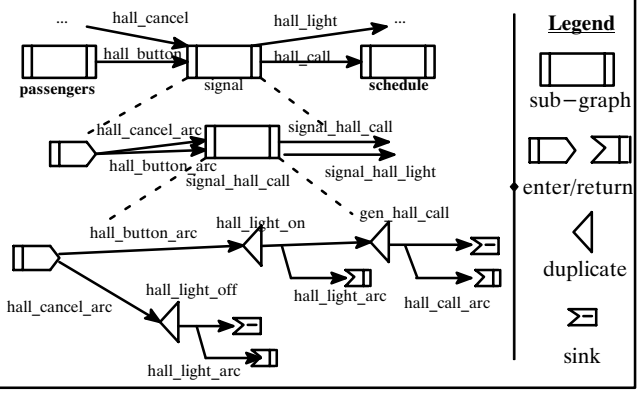
The baseline method focused the methodological issues, which are centered around the control of complexity, and motivated a manual "trial" method discussed in Section 3. This method is incomplete with respect to Figure 1 — addressing only the representation of functionality with an approach which combines techniques of object−oriented analysis and formal specification.

2. Baseline method

2.1 Functional representation

Since the approach to modeling system functionality in the baseline method is largely shaped by the commercial toolset, it is discussed here only to an extent necessary to present the main issues. Figure 2 depicts the tool's basic

Figure 2 − Directed Graph Hierarchy



representational mechanism. Within a hierarchy of directed graphs, transactions flow among nodes which are "typed" in the sense that their operation is described by grouping them into various classes, such as those which duplicate transactions, those which delay them, etc. The most general type of node is one whose semantics are entirely specified by the user in a C−like programming language. It is typical for, and in the elevator model it is the case that, a transaction represents a flow of data.

2.2 Allocation and execution

This section also avoids elaborating the detail which was required to effect execution of the elevator model and instead examines one aspect of allocation which will highlight the main issue.

Consider the classification of data flows as inter−processor or intra−processor, a basic requirement for the generation of a "communication workload" (itself a subset of the desired performance measures). The need to automate this classification in turn leads to an examination of just what it is the (sub) graphs which comprise the hierarchy represent.

If the graphs at the top level are equated with processors in the communications network, then automated classification is straightforward. Since any flows into and out of these top−level graphs would be inter−processor and all others would be intra−processor, the instrumentation required to automate classification would be localized to a relatively small portion of the model. Any change to the allocation relation, however, would impact the model at the highest levels. Thus, implicit in this approach is the expectation that a "correct" allocation is guessed at — *before* the generation of performance measures whose main purpose presumably is to help one select the allocation.

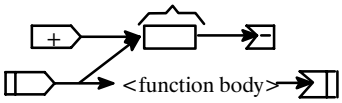
This is convincing evidence that a better situation would result by letting the top−level graphs represent "functions", or "objects" , or whatever name designates a unit of functionality. But now automated classification of the data flows becomes problematic, for now the possibility of a different classification for a flow arises at a multitude of points in the model. In particular, it arises at any point at which a transaction flows between two nodes in a graph, and where additionally it is desired to admit the possibility that these two (graph) nodes may be allocated to different processors.

One is left, it seems, with two options: either instrument the data flows, or route the flows outside of the tool's normal mechanism. Since the former would add a horrible amount of conspicuous detail to the model, an approach opting for the latter was developed and used in the elevator model. Each functional unit is characterized by the "events" (flows) it recognizes and those it generates. At initialization, these properties are "registered" so that the events can be routed to the appropriate functional unit during execution. (see Figure 3).

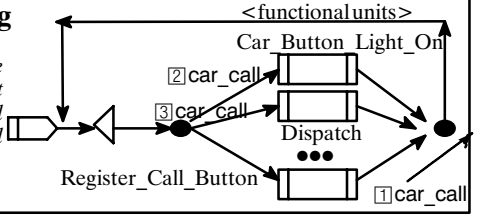
Figure 3 User−defined Routing

At t=0, all functional units declare those "events" they will receive and generate:

Functional unit : Register_Call_Button
Declare car_button in, **car_call** out



At each simulation step, the transactions which represent events are duplicated and routed to all units which have registered for them:



This technique thus delays the binding for communication between units of functionality until it is finalized by post−processing of an (instrumented) "event trace". A change in the allocation relation does not require any changes to a *given* functional representation. As explained in section 2.3.1, this is necessary but not sufficient for such allocation to qualify as unobtrusive.

2.3 Issues with the baseline method

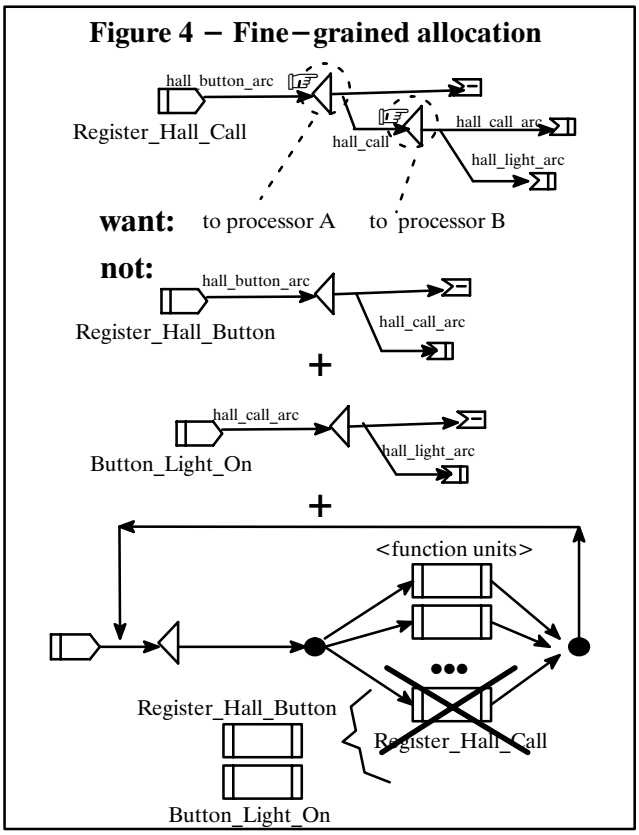
The next two sub−sections support our (subjectively derived) conclusion that both the approach to allocation and to the representation of functionality, though both are supported by automation, are inadequate for "large" models in general (and a full−function elevator controller in particular).

2.3.1 Limitations of automated allocation

To say that an allocation can change without a change to a *given* functional representation is not to say that the granularity of functionality available for allocation in a given representation is satisfactory. And if the functional representation must change in order to make the granularity satisfactory, then the allocation process is not unobtrusive.

Tool characteristics dictate that the functional units in the user−defined routing scheme discussed in section 2.2 be separate sub−graphs. Thus, for example, given a unit such as that in Figure 4:

Figure 4 − Fine−grained allocation



it is not possible to allocate the computational abstractions to separate processors unless they are made into separate sub−graphs. This can be done, of course, but at this point a figurative pencil which could group the abstractions in an arbitrary way would be much more desirable. This requirement to fragment the functional representation forfeits what little capability directed graphs have for organizing large models to begin with — presumably something related to visual proximity of dependent computations.

In short, fine−grained allocation unobtrusive enough for large models remains an unreached goal.

2.3.2 Complexity due to the representation

Some level of complexity in system−level models is unavoidable — especially in view of the requirement for a formal, executable, semantics. However, the representation approach of hierarchical directed graphs annotated with code in a third−generation language contributes to the complexity due to:

1. a failure to explicitly distinguish whether a data flow into a functional unit is providing a service or requesting a service. An "untyped" model, whereby all communication between units is classified as data flow, is appropriate for viewing a system in execution, whereas a model which distinguishes provided and required services is more appropriate for viewing a system under specification.
2. lack of a compact (and thereby lucid) representational mechanism for state machines.
3. the "low level of abstraction" provided by a third−generation (C−like) programming language.
4. the tendency for the data−flow orientation to spread detail around the model in a manner poorly suited to accommodate the (inevitable) changes which come with large models [Parnas 72].

Although these claims are not precise enough for objective consideration, they are consistent with an evaluation of a model we produced with a commercial tool which offered to "raise the level of abstraction" while providing an automatic feed into the data−flow based representation. The tool's real−time structured analysis (RTSA) [Hatley 88] approach resulted in a model judged to be only marginally "higher−level". This was likely due to the failure of the data−flow oriented RTSA to address most of the issues above (in fact all but #2).

3. Trial method

3.1 Functional representation

Given the issues with the baseline method, object−oriented analysis (OOA) techniques offered promise as an approach to modeling system functionality. The requirements for executability and for suppressing as much implementation detail as possible dictated an abstract, yet still formal, approach for specifying behavior. Several of the common OOA approaches were found wanting in this regard.

Primarily because of the absence of any reliance on data flow for the specification of behavior, the approach of [Coleman 92] was selected and adapted as required for the elevator problem. The objects in models constructed with the adapted method:

- are characterized by the services they provide and require.
- have their services precisely defined (in terms of services they require) by extended finite state machines [Harel 87] and formal transition specifications. The latter are comprised of pre and post conditions in the style of Z [Spivey 92].
- are related to one another by relationships defined with the same (Z−like) notation used by the transition specifications. This results in a clear and mathematically consistent approach when objects (in relationships) must be identified by such specifications.

The elevator test case has been recast in this formal, object−oriented framework. The Annex summarizes the method and excerpts the elevator model. The environment objects (passengers, car travel, etc.) have not been specified, and may in fact require a methodological extension for objects which must be created and destroyed dynamically.

3.2 Comparison with the baseline method

3.2.1 Allocation and execution

Neither of these notions are fully defined within the trial method and will be the subject of future work; the remarks below represent preliminary ideas.

Since the object−oriented model has a formal semantics, a notion of execution *outside* the context of allocation to processors can be put forth in a manner analogous to that in [Shlaer 92]. (Progressing from the notion to automation is problematic, however, because of the "higher−level" of Z.) The eventual definition, however, must lie *inside* the context. This implies coming to grips with the issue of fine−grained allocation discussed in 2.3.1, which becomes, in the context of the trial method, the need to admit the possibility of allocating different "parts" of an *object* to different processors.

An approach which appears promising is to define a classification based on those object "parts" which specify behavior — pre−conditions, post−conditions, and event−generation — and then implement a concept analogous to the user−defined routing (described in 2.2) to make the allocation independent of the model organization.

3.2.2 Controlling complexity

The trial method avoids unnecessary complexity in the following ways:

1. **Provided and required services.** Events and services exist in a convenient duality. They are equivalent when you expect them to be and different when you expect them to be. They are different when defining the object interface, where an object is characterized by the services it provides and requires. These services are further classified, for example, into those which change the object's state and those which do not. On the other hand, when defining state transitions, all service invocations "collapse" into events. In a typical data−flow based approach, the event (data−flow) level is the only level provided to the modeler, who must therefore repeatedly (and at the risk of error), abstract events into services whenever that view of the system is desired.

2. **Compact representation for state machines.** The specification of each object's behavior uses a natural notation for extended finite state machines.

3. **Level of annotations/specifications.** The Z−like notation allows more abstract specifications than a third−generation programming language — constructs such as functions and sequences can suppress implementation detail.

4. **Better model organization.** Conventional OOA, by virtue of associating attributes with objects, has better organizational mechanisms for "persistent data" than a data−flow approach. The adapted method improves on conventional OOA by further partitioning attributes into those which are observed, hidden, or structural. This helps distinguish which attributes are available to clients of the object, which are needed merely to complete the (formal) specification, and which are needed merely to conveniently effect relationships.

4. Issues

Near term. The most important issue for the near−term is fine−grained allocation and execution of the object−oriented representation. In addition, because the only features put into the trial method were those required specifically to specify the elevator controller, missing are several features one might expect from a "true" object−oriented method. (or from a "production−quality" method). These include:

- dynamic objects (This capability was not needed since a priori identification of all instances was adequate, given the scope of the elevator model.)
- rich semantic modeling features (e.g. aggregation, inheritance)
- a convenient syntax and rules for effecting scopes of names.

Long term. There are other broader (and interrelated) issues which must be addressed in the long−term:

1. Multiple representations − The usual interpretation of an OOA model is that it is a specification of aggregate (all objects together) behavior of a system over time and thus is in the "problem space", rather than in the "solution space" as a model of the properties of a *particular* behavior (including those properties relating to time). Because the fidelity of performance and robustness assessments may depend on a particular system *structure*, rather than aggregate behavior, it is useful to ask whether the usual interpretation, in view of such assessments, should be discarded.

An answer of yes leads to the question of what a specification looks like if not this (OOA), or whether a specification is needed at all. An answer of no seems more defensible — one would simply ascribe less risk to assessments done on a (object−oriented) design derived from an OOA.

How many representations are needed and precisely what constitutes consistency among them?

2. Time – Assuming the usual interpretation of an OOA model is not rejected, what mechanisms should it contain to specify behavior over time? Three well–thought out approaches have fundamental differences in this area. In [Shlaer 92] the actions associated with state transitions consume time; their analogues in [Coleman 92] do not. Neither approach is satisfactory: it is not even clear what time consumption *means* in a specification, nor is it clear, for the alternative, how one removes an (apparently unavoidable) simplification which makes the specification, if taken literally, impossible for any design to meet. [SPC 92] differs drastically from the first two by modeling the inputs and outputs as functions of time — required response times, for example, are straightforwardly specified within the formal framework. Is this the next step toward a better approach to time? Will this help sort out some of the issues raised in 1?

3. Rapid construction of members of a design family – Whereas rapid assessment of alternative allocations requires independence of the model organization from the allocation, rapid assessment of alternative *system models* requires independence of functional units (objects) from each other. It is naive to think that the criteria for object selection as commonly espoused and practiced (by ourselves, for example, in the trial method) will achieve the necessary independence. Can representations be constructed so that domain experts can evaluate the appropriateness of a decomposition in view of likely changes? Are information hiding and inheritance the primary means with which to address change? How is inheritance used in specifications? And how do you capture domain expertise to cut down the design space in the first place?

5. Conclusions

Models based on data flow are adequate for a view of a system in execution. It is not feasible to *design* complex systems from this basis. Required are representations which should, at minimum, characterize units by the services they provide and require, and specify behavior in a manner which is abstract yet unambiguous. Flexible, fine–grained allocation is an important issue which appears to be solvable in the near–term. This would complete a basic capability in execution models which would enable a class of system tradeoffs which require means to assess performance and robustness.

The ability to characterize system behavior in ideal terms, with detail added a little bit at a time in independent chunks, is a fundamental requirement for the control of complexity in system development and description [Parnas 86]. Basic issues with respect to the use of multiple representations and to a realistic and usable notion of time suggest it will be decades before this ability is routinely established for modeling and designing distributed systems. Execution models are an appropriate basis on which to build. In the future, execution models will be derived from higher–level representations. The trial method is a step in that direction. The methods for constructing models at the higher–levels must be built incrementally and must be directly shaped by the needs of the applications.

References

™ SES/workbench is a registered trademark of Scientific and Engineering Software, Inc.

[Coleman 92] Coleman, D., Hayes, F., and Bear, S. – Introducing ObjectCharts or How to Use Statecharts in Object–Oriented Design, IEEE Transactions on Software Engineering, Vol. 18, No. 1, January, 1992.

[Harel 87] Harel, D. – Statecharts: A Visual Formalism for Complex Systems, Sci. Computer Program, vol. 8, pp. 231–274, 1987.

[Hatley 88] Hatley, D. & Pirbhai, I. – Strategies for Real–Time System Specification – Dorset House, 1988.

[Parnas 72] Parnas, D. L. – On the Criteria to be Used in Decomposing Systems into Modules, Communications of the ACM, December, 1972.

[Parnas 86] "" – A Rational Design Process: How and Why to Fake it, IEEE Transactions on Software Engineering, Vol. 12, No. 2, February, 1986.

[Shlaer 92] Shlaer, S. and Mellor, S. – Object Lifecycles, Modeling the Word in States, Yourdon Press, 1992.

[SPC 92] Consortium Requirements Engineering Guidebook, Software Productivity Consortium. December, 1992.

[Spivey 92] Spivey, J. M. – The Z Notation, A Reference Manual, 2nd ed., Prentice–Hall, 1992.

[SES 92] SES/Workbench – User's Manual – Scientific and Engineering Software, Inc., Austin, Tx., February, 1992.

Annex
Trial method summary and excerpts

Summary

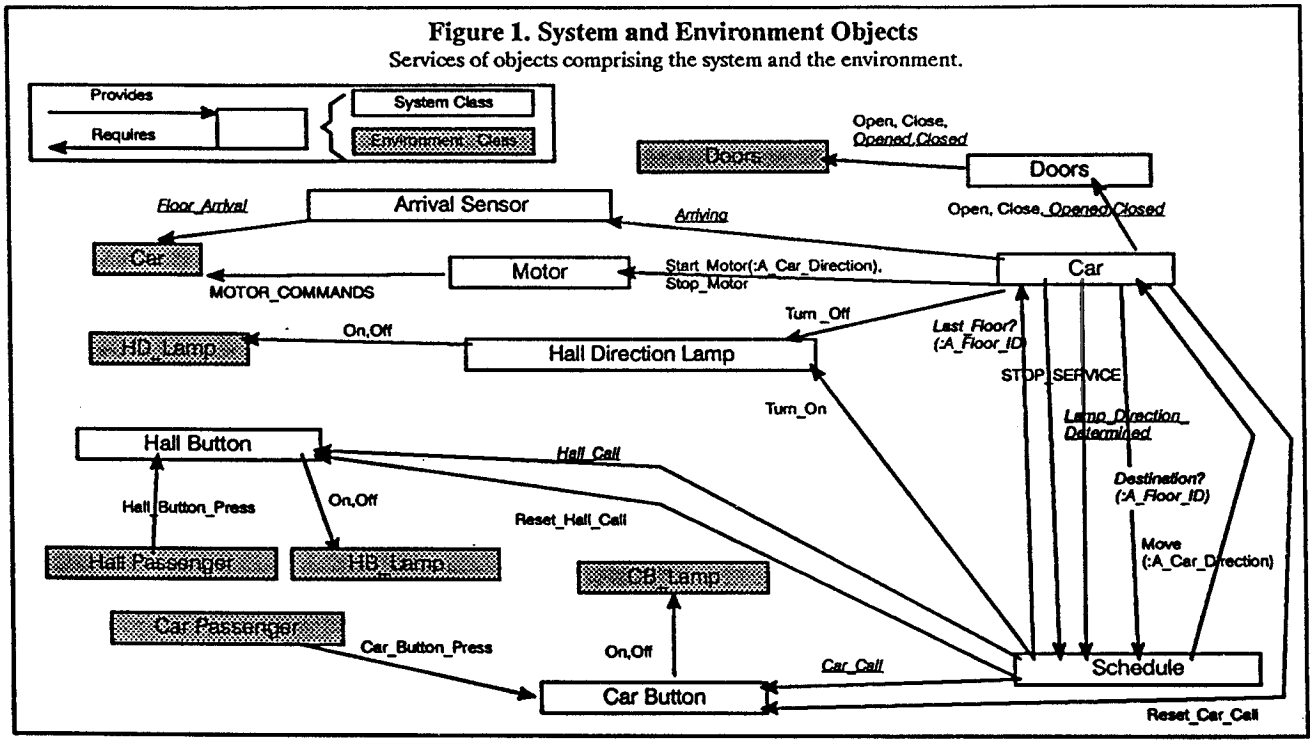
1. The model specifies the acceptable behavior of a set of interacting objects with respect to the stimuli provided by and the response to another set of (possibly interacting) objects.
2. The model consists of (object) instances which are members of classes. Classes are characterized by the services they provide and require.
3. Instances are created by an a priori process which, while merely conceptual, has well–defined effects. It results in unique object ids for the instances.
4. Instances can be related. Relationships between objects are modeled by mathematical relations between object ids.
5. Most services are mapped to events, which are the (primary) means by which objects communicate.
6. When an object receives an event, it may update its state and/or generate other events. The state is defined by an extended finite state machine and by (the state transitions' effects on) attributes.

Elevator Model Excerpts

1. *The model specifies the acceptable behavior of a set of interacting objects with respect to ... another set of ... objects.*

Figure 1. System and Environment Objects

Services of objects comprising the system and the environment.



2. *... classes are characterized by the services they provide and require.*

Figure 2 – Services provided and required by Car and Car_Button

Class	Provided Services			Required Services
	State–Changer	Observer	@T Observer	
Car	Move	*Last_Floor?*		Arriving, Start_Motor, Stop_Motor, ..., Reset_Car_Call
Car_ Button	Car_Button_Press, Reset_Car_Call	*State(Call_Active)*	*Car_Call*	On, Off

Observer services are those which do not change the object's state. For example, *Last_Floor?* provides an abstraction of Car position to its clients. "@T observers" are in the style of [SPC 92] and allow clients to observe state changes without explicitly invoking an observer service. More accurately, but still informally, *Car_Call* = @T *State (Call_Active)*, that is, an instance of the Car_Call event is generated when the Car_Button instance enters its Call_Active state. Additionally, clients of Car_Button are free to invoke an observer service (*State (Call_Active)*) which returns whether or not the instance is in the Call_Active state. The "@T" mechanism eliminates the need for "polling". To promote better independence of objects, the declaration of the State observer service explicitly denotes the (only) states the provider must "export".

3. *Instances are created by a ... process which ... has well–defined effects ... [and] results in unique object ids for the instances.*

The instantiation "process" defines the function ↑_, which allows access to the object given the object id. The "pseudo–attribute" self gives an instance access to itself. This "conceptual baggage" seems preferable to that which arises with the alternative of a value–based approach to unique object identification.

Figure 3 – Instantiation

↑_ : Object_Id → Object

↑_ = {oid:Object_Id, o:Object •
oid ∈ Arrival_Sensor_Oid => o ∈ Arrival_Sensor ∧
oid ∈ Car_Oid => o ∈ Car ∧ ...
oid ∈ Schedule_Oid => o ∈ Schedule }

Arrival_Sensor_Oid = {instance–for–arrival–sensor–1} ∪ ... ∪ {instance–for–arrival–sensor–n}

Car_Oid = ...

Object_Id = Arrival_Sensor_Oid ∪ ... ∪ Schedule_Oid

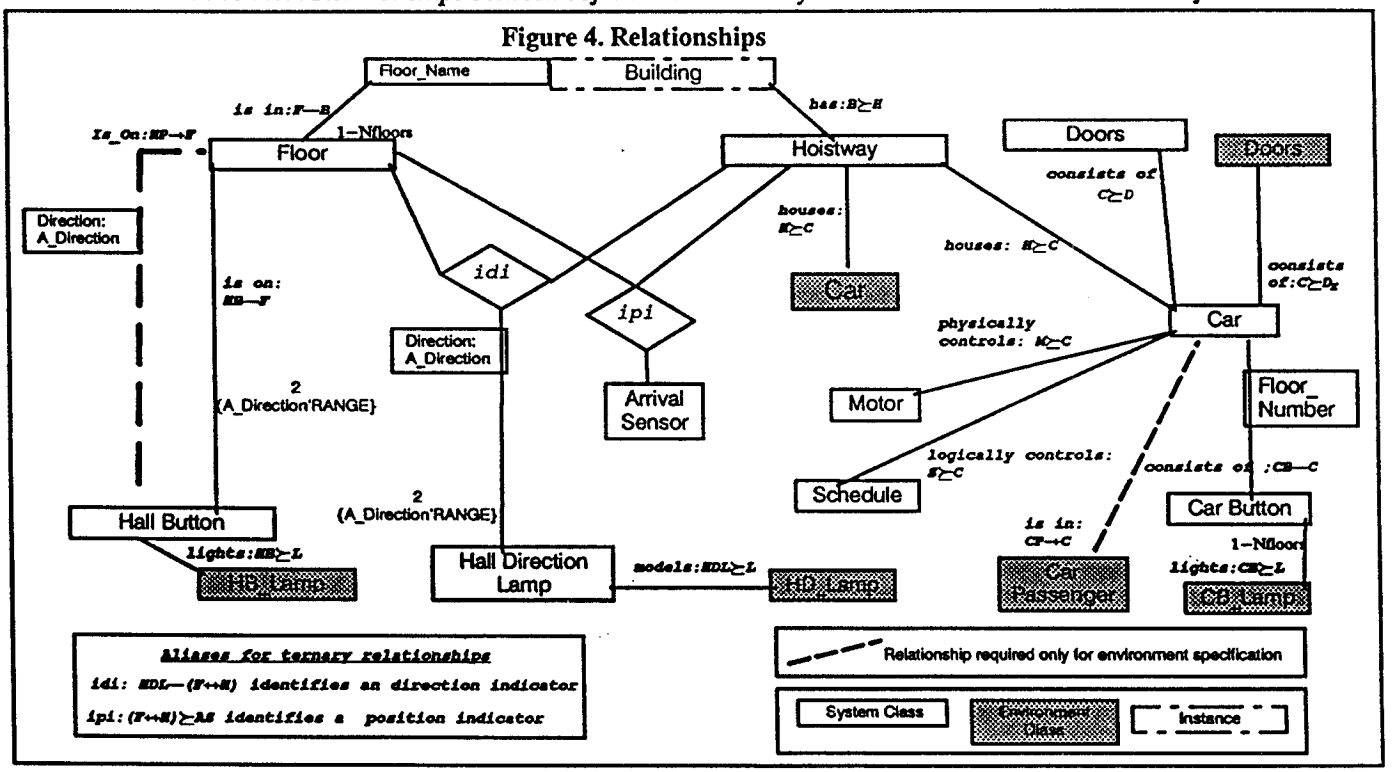
Object = {instance–for–arrival–sensor–1} ∪ ... ∪ {instance–for–schedule–1}

To Arrival_Sensor, add : self : Arrival_Sensor_Oid

To Car, add : ...

The instantiation process must satisfy any constraints specified for relationships.

4. *Instances can be related. Relationships between objects are modeled by mathematical relations between object ids.*



Arcs between classes represent relationships and are labeled, at minimum, with the relationship name and the "signature". (Some substitutes for the Z symbols are required and are listed in Table 1.) For example, Motor_physically_controls_Car is a bijective function since the car uniquely determines the motor, and vice versa. Arcs may also be labeled with indicators for cardinality and for differentiating instances. For example, the ternary relationship idi, formally defined as:

idi : Hall_Direction_Lamp_Oid — (Floor_Oid ↔ Hoistway_Oid)

Constraints are TwoLampsPerFloor ∧ LampsDifferentiatedByDirection.

TwoLampsPerFloor = ∀ (d:Hall_Direction_Lamp_Oid,f:Floor_Oid , h:Hoistway_Oid) ∈ idi • (#(idi ▶ f ↦ h) = 2)

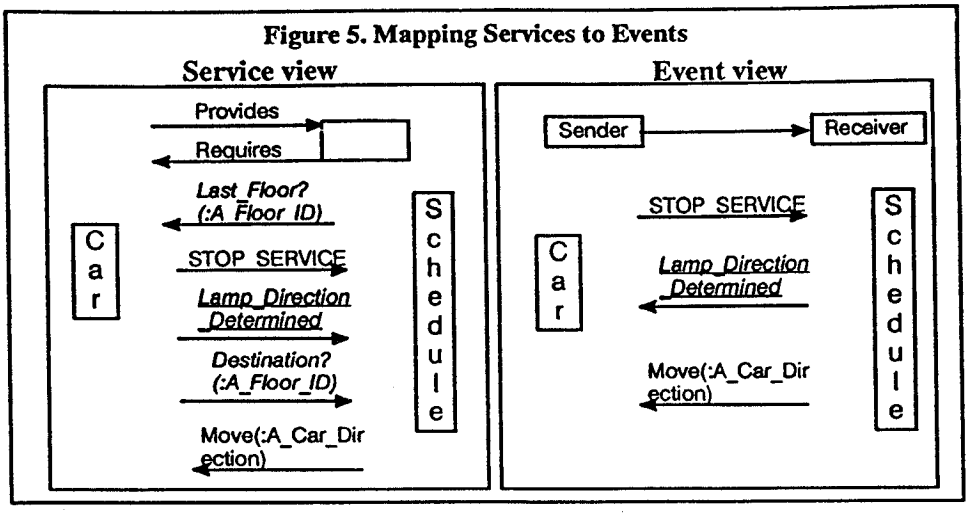
LampsDifferentiatedByDirection =

∀ (f:Floor_Oid,h:Hoistway_Oid,d_1,d_2 :Hall_Direction_Lamp_Oid) ∈ (idi ▶ f ↦ h) • (↑d_1.direction ≠ ↑d_2.direction)

is modeling the constraint that there are two lamps of (different directions) per hoistway per floor.

5. Most services are mapped to events, which are the (primary) means by which objects communicate.

Figure 5. Mapping Services to Events



State–changing services and @T observer services map to events. Events are issued by a sender, received by one or more receivers, and may cause a state transition. (Conventional observer services are excluded from the definition of events (not given here) only to make the construction rules for state transition specifications easier to define. Such services, such as Last_Floor? and Destination?, still, of course, constitute "object communication".)

6. *When an object receives an event, it may update its state and/or generate other events. The state is defined by an extended finite state machine and by (the state transitions' effects on) attributes.*

Figure 6a lists a subset of the Schedule object's services, its attributes, and StateChart. There is the intuitively obvious correspondence between the service named "Destination?" and the attribute named "destination". Figure 6c gives the specifications for a subset of the transitions (**hall–call, closer–hall–call, no–destination**) in the StateChart.

Figure 6a – Schedule : Services, Observed attributes, and StateChart

State–changing services: End_a_Stop
Observer services: *Destination? (:out A_Floor_Id), ...*
Required services: *Car_Call, Hall_Call,* Move (:A_Car_Direction), Reset_Hall_Call...
Observed attribute: destination : A_Floor_Id defined in: Going_Around

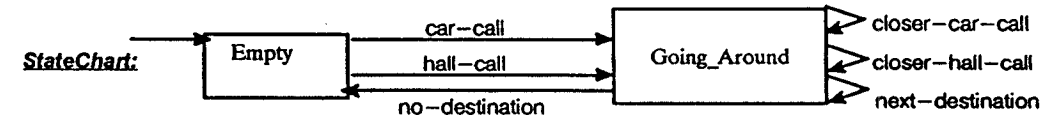


Figure 6b defines the attributes unavailable to client objects. The concept of a ring simplifies the specification of a simple scheduler and in particular allows the state machine to ascribe a single (albeit unusual) direction to car movement. It is used to a limited extent by the transitions listed in Figure 6c; it has a more profound effect on the **next–destination** transition (whose specification is not shown).

Figure 6b – Hidden Attributes

```
Ring_Domain = 1..2*(NFloors −1)
Ring_Entry : A_Floor_Id ↦ A_Direction
Ring : seq[Ring_Entry]
Ring = { 1 ↦ (2 ↦ UP), 2 ↦ (3 ↦ UP), ..., NFloors−1 ↦ (NFloors ↦ UP),
  NFloors ↦ (NFloors−1 ↦ DOWN), NFloors+1 ↦ (NFloors−2 ↦ DOWN), ..., #Ring_Domain ↦ (1 ↦ DOWN)}
    /* Each maplet in the range of the Ring sequence represents a floor and the direction the car must
    move to arrive at that floor, respectively. */
read ls as last_stop_r :        Ring_Domain defined in: All   /* last floor at which car stopped */;
read ld as last_destination_r : Ring_Domain defined in: All   /* destination */;
vf(p) = first (Ring (p))        /* "floor" part of entry */
divf(p) =second (Ring (p))      /* "direction into floor" part of entry */
```

Figure 6c

The ASSERT comments in the post−conditions for the **hall−call** and **closer−hall−call** transitions emphasize the straightforward nature of the post−conditions from the clients' view — an idle car heads for the floor where the only call is. From the Schedule object's point of view, however, things are a bit more complicated, since a hidden attribute (ld, i.e., last destination) must be updated.

If the Schedule object is in the Going_Around state, a hall call event will only change the destination if it is from a floor en route to and in the same direction as, the current destination and car direction, respectively.

The **no−destination** transition is taken when a stop is completed and there are no outstanding calls. The specifications use the relationship HallButton_is_on_Floor to get floor information for a given Hall Button object.

Figure 6c Transition Specifications

Transition States	Transition Name	Event Received	Pre−condition	Event(s) Sent	Post−conditions
...			...		...
Empty −> Going_ Around	...	...	...	...	...
	hall−call	HB.Hall_Call(f,d)	−	C.Move(divf(ld'))	destination' = vf(ld') ld'=After_[ld]_Where_[vf(:)=f] /* ASSERT: destination' = f */
Going_ Around −> Going_ Around	...	...	...	...	...
	closer− hall−call	HB.Hall_Call(f,d)	[f]Between [Last_Floor?,vf(ld)] ∧ Appropriate_Direction	−	destination' = vf(ld') ld' = ls + Dest_Increment /* ASSERT: destination' = f */
	...	...	...	...	...
Going_ Around −> Empty	no− destination	End_a_Stop	∃_Other_Floors_to_Service	HB(UP). Reset_Hall_Call; HB(DOWN). Reset_Hall_Call	{}

where f= ↑(HallButton_is_on_Floor(HB.self)).floor_name ; d = HB.direction

Terms in Post−conditions:

- Appropriate_Direction = (d = divf(ld))
- Dest_Increment = abs (f − vf(ls))
- [i]Between[e1,e2] = i > min (e1,e2) ∧ i < max (e1,e2)
- HB(mode) = ↑hb • hb:Hall_Button_Oid
  ∧ ↑hb.direction = mode ∧ ↑(HallButton_is_on_Floor(↑hb.self)).floor_name = C.Last_Floor?

Other terms:

- Car_Calls : ℘ A_Floor_Id = {↑o.floor_number • o ∈ Car_Button_Oid ∧ ↑o.State = Call_Active}
- Hall_Calls: ℘ A_Floor_Id × A_Direction = {(↑fo.floor_name ↦ ↑bo.direction) •
  (↑bo ↦ ↑fo) ∈ HallButton_is_on_Floor ∧ ↑bo.State = Call_Active}
- ∃_Other_Floors_to_Service = Car_Calls ≠ {} ∨ ({Last_Floor?} ⩤ Hall_Calls) ≠ {}
- /* After_[r:Ring_Domain]_Where[Predicate[:]] denotes the next entry in the ring where Predicate is true. The "free variable" in the predicate is indicated by ":". The value of the (After_Where) expression is of type Ring_Domain and is equal to the variable's bound value. */

  For r_i,s_i,t_i : Ring_Domain :

  □ After_[r:Ring_Domain]_Where[Predicate[:]] =
  r_1 • Predicate (r_1) and ($\neg\exists$ r_2 • Predicate(r_2) ∧ (r_2)_Is_Nearer_To(r)_Than_(r_1))

  □ (s_1)_Is_Nearer_To(s_2)_Than_(s_3) = Ring_Distance(s_2,s_1) < Ring_Distance(s_2,s_3)

  □ Ring_Distance (t_1,t_2) : Ring_Domain × Ring_Domain → 0..#Ring_Domain−1 =
  {(t_1,t_2) : if $t_1 \le t_2$ then $t_2 - t_1$ else #Ring_Domain − t_1 + t_2}

Table 1 − Substitute Z Symbols

Symbol	Name
⩤	domain anti−restriction
→	(→) function
⤀	surjective function (onto)
⤔	injective function (1−1)
⤖	bijective function (1−1 & onto)
⇸	finite function
℘	power set
▷	range restriction