CALIFORNIA PATH PROGRAM
INSTITUTE OF TRANSPORTATION STUDIES
UNIVERSITY OF CALIFORNIA, BERKELEY

Object Management Systems

Aleks Ohannes Göllü

California PATH Research Report
UCB-ITS-PRR-95-19

This work was performed as part of the California PATH Program of the University of California, in cooperation with the State of California Business, Transportation, and Housing Agency, Department of Transportation; and the United States Department of Transportation, Federal Highway Administration.

June 1995

ISSN 1055-1425

Object Management Systems

by

Aleks Ohannes Göllü

B.S. (Massachusetts Institute of Technology, Cambridge) 1987
M.S. (University of California, Berkeley) 1989

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Engineering-Electrical Engineering
and Computer Sciences

in the

GRADUATE DIVISION

of the

UNIVERSITY of CALIFORNIA at BERKELEY

Committee in charge:

Professor Pravin Varaiya, Chair
Professor Felix F. Wu
Professor James Pitman

1995

The dissertation of Aleks Ohannes Göllü is approved:

Chair date

date

date

University of California at Berkeley

1995

To My Parents . . .

You wanted grandchildren,
I give you a brainchild.
Not quite the same, but hey,
there are no diapers to change.

and

To the late Bosphorus (millions B.C. - 1989 A.D.),
the blue waters (now green) I used to swim in, and
the green (now pale concrete gray) hills I used to enjoy.

Abstract
Object Management Systems
by
Aleks Ohannes Göllü
Doctor of Philosophy in Engineering – Electrical Engineering and Computer Sciences
University of California at Berkeley
Professor Pravin Varaiya, Chair

We describe a new approach for developing large-scale object-oriented software systems, called Object Management Systems (OMS). OMS are model-based applications used to simulate, evaluate, and control large-scale physical environments. Examples of such environments are transportation networks, telecommunications networks, power distribution networks, air traffic control systems, and management information systems. These environments are heterogeneous, dynamic, and distributed.

The OMS Object Model consists of the semantic data and process model components. The data model is used to specify entities, their input, output, and state attributes; their methods; and their constraints. The process model is used to specify how a collection of objects and their relationships evolve based on their state and their input-output interconnections.

The OMS development methodology consists of four stages: Domain Customization; System Architecture; Application Development; and System Test. The first two stages customize the Object Model based on the application needs and deliver a Customized Object Model. The application developers further extend the Customized Object Model to an OMS Application.

Domain customization starts with the OMS Object Model and uses it to specify the relevant components of a particular application domain: objects, their relationships, constraints, behavior, and observation and control channels; event propagation mechanisms, control strategies; and user interfaces. System architecture uses the OMS Object Model to determine the optimal partitioning of the deployed system with respect to distributed processing, distributed databasing, process and object migration strategies, concurrency control, and versioning. Application programming fills in the details of object behaviors and control strategies keeping in mind the system constraints.

OMS Applications support configuration, fault, performance, accounting, access and security, resource, and planning management functions.

We demonstrate the methodology by implementing the SmartAHS simulation framework, a Customized Object Model. SmartAHS is used to capture different Automated Highway System designs and benchmark scenarios and to generate performance metrics through micro-simulation of the designs. SmartAHS provides generic objects for modeling highways, vehicles, control and communication agents, and performance monitors. SmartAHS also provides a scheduling engine that simulates time and event-driven object behaviors. The scheduler is configurable, and it can simulate objects at different time scales.

The California PATH Program at UC-Berkeley has proposed a hierarchical control architecture that yields up to a four-fold increase in transportation capacity while enhancing safety. We demonstrate the use of SmartAHS by implementing elements of the PATH automation architecture. The resultant OMS Application is called SmartPATH.

Professor Pravin Varaiya, Chair

Contents

List of Figures

List of Tables

Acknowledgements [1]

There are so many superlatives with which I can thank Prof. Pravin Varaiya, all have been used before, repeatedly. I say as much: In his absence, I would not have returned to my PhD.

Professors Felix Wu, Jim Pitman and Jean Walrand served on my qualifying exam committee or my reading committee. Among other things, Felix read the first and least readable draft of my thesis and Jean has revolutionized the concept of presentation for all of us.

Akash, Farokh, Praveen, John-Park, Delnaz, Daren, Grace, John, Yu-Kai, and Daniel contributed to this thesis not only with their discussions but also with their work.

Heather Brown, Annie Hayflick, Ferenc Kovac, and Katherina Law were shining beacons whenever I was lost in the administrative maze.

Since I first joined, a couple of generations passed through 275m. It has always been, and still is, good company!

In return for a mere phone-call every fortnight, my parents have given me so much love and care.

Finally, I thank all my friends, who are hidden in the following lines.

Three years pursuing a green card, three years pursuing PhD,
more people, more places, more fragments ...

“Where is the window?”, 7.2, the big blue, 90° lag, mohavi, sublet, “Make sure you eat the shrimpies”, Monterrey, French Hotel, Q, shear walls, become, Cabo, Roxie, not quite housed, Sam’s Cafe, no College Park, Cinnabar, nice clay dirt, FL Keys, not informant, ikkyu, aquila, tummy-tuck, flower boxes, Orbatello, dolmas, Jupiter, “once you cross the ocean $\cdots$”, the roadblocks of Min and Max, cheese rolls, Harrah’s, latte, 1812 San Pablo, many many more knees, pıhh (with a French accent), 18 π, INFJ, lingo, pohaça, Nefeli, Grizzly, $\left(\overset{\longleftrightarrow}{>\!\!-\!\!-\!\!<}\right)$, nutrition, BBQs, PokFuLam, Za’Ha’Dun, kurzum wurzum, chello, ich möchte schwimmen gehen.

[1] This research was supported as part of the California PATH program of the University of California, in cooperation with the State of California Business, Transportation, and Housing Agency, Department of Transportation, and the Army Research Office under contract DAAH04-94-9-0026.

Chapter 1

Introduction

Large engineering systems, such as telecommunication network management, highway automation, power distribution automation, factory automation, and air traffic control systems, face the challenge of providing reliable services using scarce resources. Clients of such systems demand performance, safety, comfort, and efficiency.

The problem is often compounded by physical resources that are saturated, inefficiently utilized, or technologically outdated. In many industries, failure to improve the performance of such systems results in significant financial or social costs.

Due to the heterogeneity of the system elements and the large system size, the planning and control of such systems cannot always be done in a mathematical framework. Experimentation with the actual physical system is often not feasible; in many cases the physical system is not yet built. Furthermore, most real systems have an abundance of unstructured information, too many superfluous details, no well-defined observation and control mechanisms, and no single access location due to their distributed nature.

Large-scale, complex software applications are needed to specify, simulate, evaluate, and manage the behavior of such systems.

The specification and implementation of such applications is more of an art than a well-structured process. There is a gap between the specification and implementation constructs required to build such applications on the one hand, and the interfaces provided by database management systems and programming languages on the other hand. In the absence of the right tools, managing system complexity becomes time consuming and costly. There is a need for a formally sound semantic data and process model that captures application level constructs for such systems. Furthermore, the model has to be structured to support partial system verification whenever possible.

This thesis proposes and demonstrates an object-oriented data and process model and an end-to-end development process for specification and implementation of large-scale real-life engineering software applications, called Object Management Systems (OMS). OMS are used to simulate, evaluate, and control heterogeneous, dynamic, distributed physical environments.

The following sections of this chapter provide a summary of this thesis.

1.1 Background

Object Management Systems (OMS) are object-oriented software systems used to simulate, evaluate, and control large-scale physical environments. Examples of such environments are transportation networks, telecommunications networks, power distribution networks, air traffic control, and management information systems. These environments are heterogeneous, dynamic, and distributed.

OMS provide the following functions.[1]

Configuration Management—
the ability to specify and control the configuration of the physical environment;

[1]These functions are based on the OSI NM/Forum functional categories for network management [67].

Fault Management—
: the ability to detect faults and significant events in the physical environment, to respond to them with graceful degradation of system performance, and to recover from them;

Performance Management—
: the ability to track, optimize, and fine-tune the physical system performance;

Accounting Management—
: the ability to account for physical system usage and charge the users according to pricing policies;

Access and Security Management—
: the ability to specify and control users' access to the physical system in a multi-user operating environment;

Resource Management—
: the ability to provide an inventory of all physical system resources and to administer their maintenance schedules;

Planning Management—
: the ability to specify, simulate, and evaluate alternative physical system configurations and control policies.

Three technological factors have a profound impact on the success of OMS for complex physical environments:

- The data and process models used to describe the physical environment;
- The software tools used for implementing these models; and
- The software engineering processes followed to realize the system.

The data and process models capture the domain expertise required for describing and managing the physical environment. Typical modeling approaches use relational databases for data modeling and programming languages for process modeling. In complex application domains, the object-oriented approach is gaining popularity due to its superior modeling power. While the relational model only describes system state, the object model has the potential for describing both system state (or data) and system behavior (or the processes), in an integrated manner.[2] Yet this potential is rarely exploited in practice, and the object model is often used only for data description.

Because it has been followed for a long time, the approach based on relational databases and programming languages provides a mature set of software tools. Typically, relational databases provide an end-to-end development platform that includes the core database engine, modeling tools, such as form and report generators, and application development utilities. Furthermore, the relational model has a powerful Structured Query Language (SQL) with a sound mathematical basis [29]. A standardized set of tools with a wide applicability is possible in this approach because of the structural simplicity of the relational model consisting solely of a collection of flat, fixed-format tables. The popularity of this approach in today's applications can be attributed to the existence of these tools.

The role of tools in an object-oriented approach is even more significant since the object model is semantically richer than the relational model. However, the generality of the model itself has precluded the development of a standardized set of widely applicable tools, and the object databases have failed to converge to a standard query language such as SQL [33, 25]. The emerging object databases are of two types: those tied closely to the relational model, such as Matisse and Postgres, or those tied closely to programming languages, such as Versant. The former provide enhanced relational databases with an object interface, while the latter provide programming languages with persistent objects. Another class of tools common today are translators between objects on the

[2]Refer to the object-oriented methodologies described by Booch [11], Coad [14], Rumbaugh [16], Shlaer and Mellor [18], amongst others.

program side and relational tuples on the storage side. While these tools are useful, application-level tools that implement semantic data and process models are sorely needed.

Typical software engineering life-cycles consist of the following stages: requirements analysis, functional specification, system architecture, prototype development, system design, implementation, integration, system test, release, and maintenance. The stages up to implementation are treated as the first phase of the project and the subsequent stages up to system test are treated as the second phase of the project. Most often, the output of the first phase is a design document and the output of the second phase is the software system. A strictly waterfall approach [2] to software development treats these as sequential stages. In the object-oriented methodologies the stages within a phase and even the phases themselves are repeated cyclically to obtain the final software system.

While the software industry has gained a fair amount of expertise in managing and delivering such projects, this phased approach has inherent risks which must be analyzed carefully. Ensuring coordination between the large number of project stages leads to management overhead. There are no streamlined mechanisms to ensure that what is implemented is what was designed. Poor coordination can lead to a "disconnect" between the domain experts involved in the first phase and the system experts involved in the second phase. Finally, the staffing profile of the project is typically back-loaded, leaving little control over schedule slippage.

1.2 The OMS Approach

Standardized tools and processes can emerge if a model-based approach is adopted for the development of software systems. For example, the relational model enabled the development of standardized tools such as SQL by restricting state descriptions to a tabular format. Such a restriction of modeling power constrains the class of applications to which these tools and processes can be applied.

In the OMS approach, we focus on an important class of applications, namely management systems that are used to control the behavior of heterogeneous, dynamic, and distributed physical environments. For this class of applications, we develop the OMS Object Model, a powerful semantic data and process model. We implement this model as the SmartDB software platform and customize it for specific application domains—SmartAHS for automated highway systems, SmartPower for power distribution management, and SmartNet for network management.

The OMS software engineering process consists of the following stages:

- Domain customization;
- System architecture;
- Application programming; and
- System test.

Domain customization starts with SmartDB (or one of the customized SmartDB platforms) and uses it to specify the relevant components of the physical environment: objects, their interrelationships, constraints, behavior, and observation and control channels; event propagation mechanisms; control strategies; and user interfaces. System architecture uses SmartDB to determine the optimal partitioning of the deployed system with respect to distributed processing, distributed databasing, process and object migration strategies, concurrency control, and versioning. Application programming fills in the details of object behaviors and control strategies keeping in mind the system constraints. These stages are followed by the system test, release, and maintenance stages.

The OMS process provides significant overlap between the different stages of the software life-cycle. The first two stages deliver a customization of the OMS object model along with an application architecture, all in software. Each stage successively refines the output of the previous stage using the OMS Tool Set. This integrated environment reduces project management overhead; the risk of a "disconnect" between domain experts and system experts is absent; and the staffing profile is fairly even throughout the project.

We reemphasize that the OMS approach achieves integrated models, tools, and processes by focusing on a specific, but large, class of applications such as transportation networks, telecommunications networks, power distribution, air traffic control, and management information systems, and by following a model-based approach to development.

1.2.1 The OMS Object Model

The OMS object model is derived from two streams of theoretical development: object-oriented modeling and mathematical systems theory. We give a brief description of the model features.

State

An object's attributes describe its state, inputs, and outputs. The system is an interconnected collection of objects, and its state can be thought of as the state of individual objects along with their input-output interconnections. The system as a whole has inputs and outputs corresponding to the free inputs and outputs of objects in it.

Methods

The methods of an object are maps from its state and input to a new state and new outputs. In addition, a method can specify new objects to be created, existing objects to be deleted, new input-output connections to be made, and existing input-output connections to be removed.

Each method also specifies its triggering inputs and triggered outputs.

State Transitions

The system is activated by triggering some subset of its inputs. All object methods triggered by these inputs are executed. The outputs of these executed methods themselves trigger other methods, which are then executed, and so on.

We require and impose conditions to ensure that this method execution sequence is unique (i.e., there are no race conditions and indeterminacies) and that it terminates.

If such an execution sequence satisfies all system constraints, then the system state at the end of the sequence is committed; otherwise it is rolled back to the beginning of the sequence and the triggering input is discarded.

Constraints

Constraints of four types are defined:

- State constraints—constraints on the values of the state, inputs, and outputs of an individual object;
- Connection constraints—constraints on establishment of input-output connections between objects;
- Relationship constraints—state constraints expressed over several objects that are related through relationships; and
- Behavior constraints—state constraints that must be satisfied before and after the execution of a single method of an object.

Figure 1-a shows a sample OMS with three interconnected objects. Figure 1-b shows a triggering input applied to the OMS and the method that it triggers. Figure 1-c shows the effect of the state transition caused by this triggering input: creating a new object and connecting it to existing objects through input-output relationships, and computing the triggered output of the system. Figure 1-d shows the resulting OMS state. Note the resemblance of this OMS data and process model to integrated circuit diagrams, output feedback control systems, Petri nets, and neural networks.

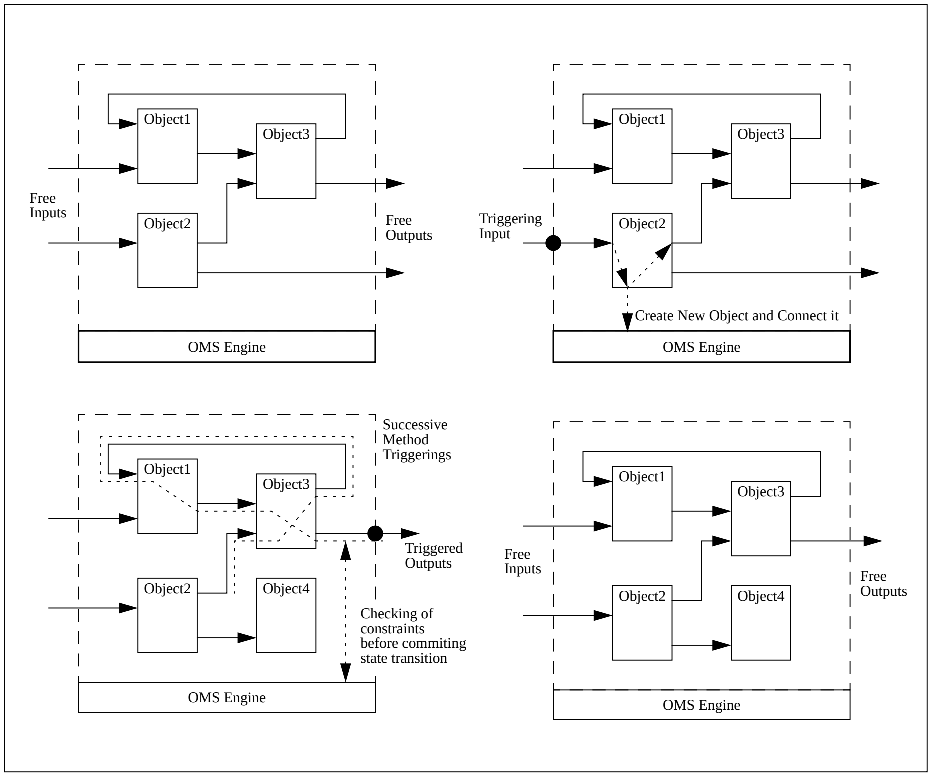
Object1
Object2
Object3
Object4
Free Inputs
Free Outputs
OMS Engine
Triggering Input
Create New Object and Connect it
Successive Method Triggerings
Triggered Outputs
Checking of constraints before commiting state transition

Figure 1.1: Sample OMS execution sequence from left top to right bottom. (a) A Sample OMS. (b) Triggering Input. (c) State Transition. (d) End State.

1.2.2 SmartDb

SmartDB is a software implementation of the OMS Object Model. In addition to the implementation of this austere model, SmartDB provides several additional features:

- It implements a relationship object and provides specific and useful relationships such as input-output, containment, views, agent-manager, client-server, and process layers; (A process layer is a collection of objects scheduled for execution at a common time- or event-granularity.)
- It implements objects such as events, sensors, actuators, schedulers, and users;
- It implements the OMS Engine, the machinery that executes the model dynamics. The OMS Engine provides an interface for creating and deleting objects, connecting and disconnecting them, triggering methods, executing state transitions, checking constraints, propagating event notifications, and providing event- and time-based scheduling; and
- It provides system architecture tools for data distribution, process distribution, object migration, process migration, packaging objects into process layers based on their scheduling requirements, versioning, concurrency control, backup and restore, schema evolution, and other utilities.

Assumed Capabilities

SmartDB is middleware built on top of a persistent storage medium.

It requires the following capabilities: persistent storage, schema generation, implicit retrieval, predicated queries, backup and restore, and commit and rollback.

The following features are desired but not essential: versioning, data distribution with location transparency, object migration, directory services, and concurrency control with locking and deadlock detection.

The following features are useful but not essential: utilities for forms and reports, and on-line schema evolution.

SmartDB functionality is limited by the availability of these features.

Customized Extensions

We have customized SmartDB to form the SmartAHS and SmartPATH platforms for highway system simulation and evaluation. SmartDB can also be customized for power distribution management and telecommunications networks. We describe briefly the features of these SmartDB extensions.

SmartAHS—
: Highway objects: lane segment, highway section, entry, exit, and zone. Vehicle objects: vehicle, engine, brakes, steering, sensors, transmitters, and receivers. Process layers: physical, regulation, coordination, link, and network. Data and processing distribution based on zones.

SmartPower—
: Links: three phase, two phase, and single phase high voltage, medium voltage, and low voltage transmission lines. Nodes: generators, transformers, loads, serial capacitors, parallel capacitors, and connectors (1–2, 2–1).

SmartNet—
: Links: channels, facilities, circuits, packets, and services. Network elements: equipment, functions, modules, multiplexors, buffers, switches, terminals, and users. Process layers: physical, data link, network, transport, session, presentation, and application; Data and processing distribution based on geographical regions. Sensors and actuators based on SNMP and CMIP protocols [71, 55].

Implementation

Currently SmartDB is implemented using the C++ programming language, Versant Object Database, Tcl/Tk user interface tool kit, and the UNIX operating system.

The SmartAHS platform represents about ten person-years of effort, and the SmartPATH platform represents another fifteen person-years of effort.

1.3 Automated Highway Systems, an OMS Application

Anybody who has driven across the Bay Bridge in San Francisco[3] in late afternoon is acutely aware of highway congestion. Congestion occurs when demand for travel exceeds highway capacity. What's worse is that during congestion highway throughput falls below capacity. Intelligent Vehicle Highway System (IVHS) proponents claim that a proper combination of control, communication, and computing technologies (3C) placed on the vehicle and on the highway can increase highway untilization and driver safety.

The PATH Program at UC-Berkeley has proposed a hierarchical control architecture that yields up to a four-fold increase in transportation capacity while enhancing safety. The architecture proposes a strategy of platooning several vehicles as they travel along the highway. The separation of vehicles within a platoon is small (2m) while separation of platoons from each other is large (60m). The movement of vehicles is realized through simple maneuvers—merge, split, lane change, entry, and exit—that are coordinated.

The automation strategy of the PATH AHS architecture is organized in a control hierarchy with the following layers:

Physical Layer—
: the automated vehicles and highways;

[3] Or any major highway in an urban area.

Regulation Layer—
: control and observation subsystems responsible for safe execution of simple maneuvers such as merge, split, lane change, entry, and exit;

Coordination Layer—
: communication protocols that vehicles and highway segments follow to coordinate their strategies for achieving high capacity in a safe manner;

Link Layer—
: control strategies that the highway segments follow in order to maximize throughput; and

Network Layer—
: end-to-end routing so that vehicles reach their destinations without causing congestion.

To avoid single-point failures and to provide maximum flexibility, the design proposes distributed multi-agent control strategies. Each vehicle and each highway segment is responsible for its own control. However, these agents must coordinate with each other to produce the desired behavior of high throughput and safety.

There is a diversity of opinion about the implementation alternatives of highway "intelligence." Various other multilayer control strategies are suggested for guiding the vehicles along the highway in a partial or fully automated fashion. None of the proposed architectures has a "closed form" mathematical representation for proper evaluation. Furthermore, the evaluation is multidimensional, including utilization, travel time, safety, comfort, implementation complexity etc.

An objective comparison of these proposals requires the existence of a uniform simulation framework in which these architectures can be specified, simulated, and evaluated. The SmartAHS customization of SmartDb provides such a framework.

The SmartPATH OMS is obtained when the PATH AHS architecture is implemented in SmartAHS.

1.3.1 Evaluation using SmartAHS

SmartAHS is used to capture different AHS designs and benchmark scenarios and to generate performance metrics through micro-simulation of the designs. The SmartPATH OMS is obtained when the PATH AHS architecture described above is implemented in SmartAHS.

SmartAHS provides generic objects for modeling highway configuration, vehicles, control and communication agents, and performance monitors. SmartAHS also provides a scheduling engine that simulates time- and event-driven object behaviors. The scheduler is configurable and it can simulate objects at different time scales. Vehicle movement, for example, may be scheduled every hundred milliseconds and roadside controllers every fifteen seconds.

SmartPATH is obtained by specifying the PATH AHS design using SmartAHS. The design is given in terms of dynamical system models such as differential equations, finite state machines, fluid flows, and queueing networks, and it also specifies sensors, actuators, transmitters, receivers, control and communication policies, and operating rules. The SmartPATH simulation setup consists of the following specifications: highway configuration, travel demand, highway automation devices, vehicle automation devices, the simulation scheduling policy, and automation device parameters. Automation devices consist of sensors, actuators, communications devices, and control agents. Simulation runs are used to collect design performance metrics such as safety, productivity, comfort, and environmental impact, generated by monitoring the system state during the simulation runs. SmartAHS can be used to optimize design performance with respect to these metrics by tuning design parameters dynamically.

SmartPATH simulation performance depends on the time-granularity of the simulation. If the integration routines used to calculate vehicle displacement are set to 5ms step size, and vehicle position on the highway is updated every 100ms, 50 vehicles can be simulated in real-time on a Sun Sparc 10 workstation. Simulation profiles indicate that 80% of the simulation time is spent on time-driven simulation of the differential equations that model vehicle dynamics.

A distributed processing version of SmartPATH is under implementation for problem scales as large as 100,000 vehicles over 1000 miles of highways.

1.3.2 Deployment using SmartAHS

Once an AHS design is simulated, evaluated, and optimized, SmartAHS can be used with hardware emulators as well as actual hardware components instead of software sensors and actuators. This aids model validation and robustness testing of the control laws. For full deployment, the regulation layer control algorithms can be deployed in vehicles and the link and network layer control algorithms can be deployed on the roadside. In this environment, SmartAHS acts as a real-time distributed operating system for command control of the deployed AHS.

1.4 Roadmap

The scope of this thesis is the design and implementation of software frameworks that facilitate the specification, simulation, and evaluation of hierarchical control architectures for diverse applications. A general methodology is developed, and the methodology is used to build a software framework, SmartAHS, intended for highway automation architecture specification and evaluation.

This thesis assumes the reader has some familiarity with object-oriented programming languages, automata theory, and databases. Chapter 2 summarizes some of the required background in these areas and provides a survey of the state of the art.

Chapter 3 consists of three application area descriptions: 1) highway automation, 2) data network management, and 3) power distribution systems.

Chapter 4 contains the OMS Object Model description and a discussion of the OMS process stages.

Chapters 5 and 6 discuss the implementation of the SmartAHS platform. Since the current SmartDb implementation is quite limited most constructs that are discussed as part of Object Model are implemented in SmartAHS.

Chapter 7 provides an overview of SmartPATH and use cases of SmartAHS.

Chapter 8 contains the conclusions.

An index is provided for all class, attribute, and method definitions in Chapters 7 and 8.

1.5 Acknowledgements

Above we've stated that SmartAHS reflects about ten person-years of effort. Clearly, the author was not the sole contributor to its implementation. As the scope and the problem size of dissertations grow, it becomes necessary that the work leading to a thesis, or several theses, is performed by a team, rather than by isolated individuals.

This dissertation not only benefited from discussions with the people listed below, but, was actually built upon their work. As such, this dissertation requires an acknowledgements section within its actual body, rather than just one in its preamble.

Grace Liu contributed to some of the early prototypes.

Yu-Kai Ng and John Park Hong implemented the graphical object editor discussed in Section 6.4.

Daren Lee implemented the graphical debugger discussed in Section 6.5.

Daniel Wiesmann implemented the code-generators for the state machine language and the parametric interfaces discussed in Sections 6.3.4 and 6.6. Daniel also ran most of the performance tests.

Praveen Hingorani implemented most of the base classes, the highway entities, the traffic entities, and the event generation and propagation mechanisms. Working with John Lygeros, he also implemented the regulation layer SmartPATH objects.

Farokh Eskafi and Delnaz Khorrmabadi started this work with their implementation of SmartPATH [36]. The future of the distributed version of SmartAHS now lies in Farokh's hands.

Akash Deshpande participated in numerous discussions that resulted in the current formulation of the Object Model.

Finally, Prof. Pravin Varaiya and Prof. Felix Wu have always provided sound advice and have patiently read many earlier drafts of this thesis.

Chapter 2

Background

The following sections provide a brief overview of relevant background and state of the art in software and control engineering.

2.1 Desirable Software Features

There are some characteristics any software system should have. These are summarized below.

2.1.1 Modularity

Modularity is a powerful design tool in software engineering or otherwise.

Myers observes, "The act of partitioning a program into individual components can reduce its complexity to some degree. ... Although partitioning a program is helpful for this reason, a more powerful justification for partitioning a program is that it creates a number of well defined, documented boundaries within the program. These boundaries, or interfaces, are invaluable in the comprehension of the program" [4].

In OMS applications it is essential that the logical counterparts of physical entities are well-encapsulated in independent modules.

2.1.2 Performance

Any software application has to meet some run-time performance requirement.

On-line control applications have to meet real-time performance criteria. In particular the sensory data has to be propagated from the physical system to the application; the data has to be processed within the application; and the control data has to be communicated back to the physical system, all within an "acceptable" time frame.

In today's software technology faster hardware is always an easy way of buying better performance. However, a good design should yield good performance given the hardware at hand.

2.1.3 Scalability

A simulation framework that can simulate ten vehicles in real time but collapses if the number of vehicles exceed one hundred, is of limited use. A good application will scale with acceptable performance as the problem size grows.

2.1.4 Openness

Any well-designed large software application consists of many modules. If the interfaces of these modules are well-defined, if the modules can be accessed directly, and if they can be replaced with third party components, an application is said to have an open architecture.

2.1.5 Robustness

A robust software application does not suffer from a single point of failure. A robust design would isolate the effects of incorrect use or localized errors.

2.1.6 Ease-of-use

The traditional approach to categorizing a modeling syntax gauges it along two axis: the first measures its expressiveness, the second measures how much a problem can be analyzed if it is modeled in this syntax.

The Chomsky language hierarchy is well understood. It consists of:

- regular languages, implemented by finite state automata;
- context-free languages, implemented by pushdown automata;
- context-sensitive languages, implemented by linear bounded automata;
- recursively enumerable sets, implemented by Turing Machines.

As we go down the list, the expressive power of a language goes up, but alas, the ability to analyze a given specification goes down. In particular most questions become undecidable. (A question is said to be decidable if there is an algorithm that takes as input an instance of the problem and determines whether the answer to that instance is "yes" or "no".)

In the context of software applications the specification language at hand usually has sufficient power to be Turing equivalent and a third question emerges: how easy is it to express a given problem in this syntax? Unfortunately there is no objective measure to answer this question for a given modeling syntax.

2.2 Semantic Modeling

Wegner [10] defines three categories of modeling paradigms:

- **Object-Based Modeling**: The modeling paradigm that requires all elements of interest to be objects with clearly defined interfaces;
- **Class-Based Modeling**: The modeling paradigm that requires all objects to belong to classes. Classes are used as templates for objects;
- **Object-Oriented Modeling**: The modeling paradigm that categorizes the classes into a inheritance hierarchy.

As we move from object to class-based modeling a distinction between *Meta-Data* and *Data* emerges. Meta-data, i.e, classes, serve as a template that define how data looks like. Instances of classes, i.e., the data, are the realization of meta-data.

As we move to object-oriented modeling, the distinction between the meta-data and data becomes weaker. After all, meta-data is also data of a given form; so classes can be considered to be instances of a *Meta-Class*. However, most object-oriented programming language implementations still impose a separation between meta-data and data. In the remainder of this thesis we observe this separation and assume that the meta-data remain static as data is instantiated and manipulated.

Object Management Systems are object-oriented. We discuss object-oriented modeling constructs in more detail in Section 2.4.

system analysis and functional specification	System Analysts, Domain Experts
system architecture and high-level design	System Architects
prototype implementation	Developers
detailed design	System Designers
development	Developers
unit testing	Developers and Testers
system integration and system testing	Developers and Testers
system release	
system support	Maintenance Team
system use	System Users

Table 2.1: Traditional Project Stages

2.3 Traditional Software Engineering Process

The development of large scale software systems can not be treated as one amalgamated task. Two main software production processes have evolved over the last decades; the traditional waterfall; [5, 6] and the circular object-oriented methodologies (OOM) [11, 16, 14, 18]. Table 2.1 summarizes the basic stages of these processes and the skill set required for each stage. Clearly different interpretations of these methodologies may omit or reorder some of these stages.

Whereas the waterfall approach treats these as sequential tasks, the OOM repeats the analysis through relase phases in several cycles until one converges to a good solution [2].

The deliverables of the analysis, functional specification, high-level design and detail design phases consist of documentation only, and all implementation is left to the subsequent phases. Typical commercial software development results in a back-loaded process and the implementation, testing, and maintenance phases use up the largest amount of resources [3].

2.4 Object-Oriented Modeling Constructs

Booch observers, “As Rentsch correctly predicted, ‘My guess is that object-oriented programming will be in the 1980s what structured programming was in the 1970s. Everyone will be in favor of it. Every manufacturer will promote his products by supporting it. Every programmer will practice it (differently). And no one will know just what it is’ [9]. Rentsch’s predictions still apply to the 1990s” [12].

Our architecture relies on the existence of object-oriented languages and databases and the abstractions they provide. In this section, we briefly highlight some of the main features of object-oriented methodology and programming language characteristics that make our design viable.

2.4.1 Entities and Instances

OO methodology makes it possible to encapsulate the characteristics and behavior of physical components as logical software objects. This organization provides natural boundaries for modularity. The logical counterpart of a particular component type is called a *class* or an *entity*; it contains *attributes* and *methods*. Each occurrence of this type of component is then represented by an *instance* of this class.

This organization is particularly useful in control and simulation software, where the software system structure has to mimic the underlying physical system. Once a mapping between physical elements and their logical counterparts is established, research for control strategies can proceed without regard to the peculiarities of the physical objects themselves. Furthermore the system can be scaled just by creating more instances.

2.4.2 Inheritance

Classes are used to categorize similar instances. *Inheritance* provides a way of categorizing classes and organizes them in a hierarchy of increasing specialization.

Inheritance provides a useful set of scoping rules that matches the "common sense" thinking in real world. It supports modularity and reuse of meta-data.

A class which has direct instances is called a *concrete class*, otherwise it is an *abstract class.* A *subclass* or a *child* inherits from a *superclass* or a *parent.* A superclass is sometimes called a *base class.* Classes without subclasses are *leaf classes.*

Monotonic Inheritance

Inheritance is a mechanism of incremental refinement. Many flavors of inheritance exist. Under monotonic inheritance, every subclass must inherit each and every attribute and method specified for its superclasses and may not cancel any of them. As part of inheritance a subclass may add attributes and methods; specialize the domains of superclass attributes; specialize the domains of method return values; and specialize the method behaviors.

2.4.3 Polymorphism

Functional polymorphism is the ability to use classes and their children interchangeably. Every car, truck, and bus is a vehicle. (Clearly the converse is false.) This gives us the ability to implement other software classes that know about the vehicle class only. These other software classes then do not have to be modified or extended, if we add the "semi" and the "taxicab" to the subclasses of vehicle.

One question remains. Assume the vehicle class provides a generic "move" method that given a jerk, computes a displacement. Assume vehicle subclasses specialize this method based on their specific dynamics. Assume a given object, say a "scheduler" in charge of moving vehicles, knows only about vehicles and invokes the move method on a vehicle, which happens to be a truck. Will the displacement be that of a vehicle or of a truck?

The answer points to the difference between dynamic and static type checking. If the implementation language provides dynamic type checking (also called dynamic binding), it will at the time of this action seamlessly determine that this particular vehicle is a truck and invoke the truck's move method.

Another form of polymorphism is the *signature polymorphism,* also called *overloading.* The syntax and sequence of arguments supplied to a function together constitute the *signature* of a function. Given two methods with the same name, signature polymorphism refers to invoking the correct method based on the argument list.

2.4.4 Extended Constructs

The constructs discussed in this subsection are not supported by any existing commercial object-oriented programming language. A language with these constructs would simplify system design, specification, and implementation.

Most of these constructs are frequently used in network management applications [62, 59]. Bapat [52] discusses them in his book that develops mechanisms for modeling communication networks through the use of practical examples.

Object Identifiers

Each object and each of its methods and attributes must have a unique object identifier, called *Distinguished Name* (DN)[1].

Most programming languages use a pointer to an object as the DN. However, the value of a pointer is not persistent and an application usually must implement a different naming convention.

[1]The DN and RDN terminology is borrowed from network management.

Assume a vehicle has a length and speed. A possible notation would use "Vehicle" as the DN of the entity, "Vehicle::Length" and "Vehicle::Speed" as the DN of the attributes.

The concept of a *Relative Distinguished Name* (RDN) is also important. The RDN uniquely identifies a component within the *Scope* of a *DN-prefix.* In the above example, "Vehicle::" is a DN-prefix that defines the scope of a particular entity. "Length" and "Speed" are unique attribute RDN's within that scope.

A more complicated example is the DN of a particular vehicle instance's length. A possible notation would be "Vehicle–Car1" for the instance, "Vehicle–Car1::Length" for the DN of the attribute value.

Virtual Attributes

A virtual attribute provides implicit access to the value of another attribute, which we call its *Source.*

A virtual attribute specification contains two components: 1) A *Scope*, an RDN, such as "Vehicle–.::Length" as in the above example and 2) a *DN-Prefix*, the DN of another instance.

In this case, if the DN-Prefix has the value "Vehicle–Car1", the value of this virtual attribute would be the length of vehicle car1.

Virtual attributes can be implemented in several ways. In an on-demand-update implementation the virtual attribute retrieves the value of its Source when accessed. It does so by constructing the Source DN from the current value of the DN-Prefix and the Scope. In an always-up-to-date scheme, the virtual attribute stores the value of its Source. In this case, the virtual attribute must be updated when the DN-Prefix or the Source are updated.

It is possible to create cyclic dependencies with virtual attributes. Depending on the virtual attribute dependencies, such dependencies can be detected at compile or at run time.

Associations

Associations are used to model binary relations. An association is very much like a bi-directional pointer, in fact in most applications a pair of pointers are used to implement associations. In the absence of an atomic association construct data integrity is compromised.

Consider the one-to-one positional relationship between two vehicles, "front/back" vehicle. Consider two vehicles v1 and v2, with v1 driving behind v2. If this relationship is implemented as an association, the moment v1 changes lane and moves behind v3, v2 will terminate the "back vehicle" relationship with v1. Otherwise, v3 may enter the "back vehicle" relationship with v1, while v2 still has v1 as its "back vehicle".

If relationships are modeled as entities an association construct is still needed to identify the relationship between the relationship entity and its participants.

2.5 Object-Oriented Methodology

Several object-oriented methodologies have emerged over the last decade. These methodologies generally provide a graphical notation that captures a design during the analysis and design stages of a project. The analysis and design notation is independent of the implementation platform, and currently no tools exist to convert the design into an implementation. Furthermore, there is no mechanism to validate a design other than through implementation.

This section serves as a summary of three such methodologies.

2.5.1 Booch

Booch states, "Object-oriented methodology is built upon a sound engineering foundation, whose elements we collectively call the *object model.* The object model encompasses the principles of abstractions, encapsulation, modularity, hierarchy, typing, concurrency, and persistence" [12].

For object-oriented analysis and high level design, he provides an extensive notation. The notation is independent of the detailed design and implementation. In fact, many notation constructs have no implementation counterparts in any programming language.

The detailed design and implementation phases are expected to find some implementation of the system captured in the notation.

The Booch notation has the following components:

- class diagrams;

  Class diagrams are used to specify classes, their attributes, methods, relationships and roles, the relationship cardinalities, class categories, containment and class nesting hierarchies, constraints, and access restrictions.

- state transition diagrams;

  A state may correspond to particular attribute value assignments of an object or it may have independent semantics. Transitions correspond to actions an object can take. State transition diagrams capture sequences of actions objects can take, based on their state. State transition diagrams allow transitions to be conditioned on attribute values and states to consist of state transition diagrams.

- object diagrams;

  Object diagrams are used to specify possible object[2] configurations of the system, and the sequence of actions objects can take in these configurations. The notation contains constructs for roles, keys, data flow, visibility, synchronization, and time budgets.

- interaction diagrams;

  Interaction diagrams provide an alternative representation of object diagrams and are used to trace the execution sequences for a given set of objects.

- module diagrams; and

  Module diagrams are used to show the allocation of classes and objects to modules in the physical design of a system.

- process diagrams;

  Process diagrams are used to show the allocation of processes to processors in the physical design of a system.

The Booch notation has constructs to represent concepts from a variety of implementation alternatives. The range includes representations for procedures, for meta-data/data distinction, or for treating meta-data as data.

In the Booch notation object relationships are denoted by arcs. However, none of the diagrams model the evolution of relationships. The instantiation relationship is denoted by arcs, but no constructs are provided to model the "deleting" relationship, nor are there any constructs that specify why and how often instantiation takes place.

Booch distinguishes the macro and micro development processes.

The macro process is the controlling framework of the micro process and consists of conceptualization, analysis, design, evolution, and maintenance. Conceptualization establishes the vision for the idea and validates its assumptions through a throw-away prototype. Analysis provides a model of the system behavior, what it does, and how it does it. Analysis is expected to deliver a specification of key desired and undesired behavior. The design creates an architecture for the evolving implementation, and establishes common tactical policies for the project. Evolution grows and changes the implementation though successive refinement, ultimately leading to the production system. Maintenance manages post-delivery evolution.

The micro process represents the daily activities of the developers. Classes and objects and their semantics, relationships, and interfaces are identified and implemented.

Booch admits that software engineering is still largely an art form.

[2]Booch uses the word object when referring to "Data". In an implementation where only instances are data, objects are instances. In an implementation where both classes and instances are data, object refers to both.

2.5.2 Rumbaugh

The Object Modeling Technique (OMT) methodology employs three kinds of models as part of the information model of the system [16]. These are:

- the object model;
- the dynamic model; and
- the functional model.

The object model is used to specify the static structure of objects.

Object diagrams (these are similar to Booch's class diagrams) are used to capture classes, attributes, methods relationships, aggregation and inheritance hierarchies and constraints.

The dynamic model is used to capture the input-output behavior of an application and the interactions among objects within an application. In Rumbaugh's approach the input-output dynamic model of a compiler is trivial: it has one input, a source file, and one output, an executable.

State diagrams are used to capture the dynamic model. Like in Booch notation, a state is an abstraction of the attribute values and the object's relationships and no formal notation is introduced to define a state. Conditional transitions are used to move from one state to another upon events; actions are generated as part of the transitions.

The functional model is used to capture the data transformations within an application. In Rumbaugh's approach the functional model of a persistent storage medium is trivial since it only stores the data but does not alter it.

Data Flow Diagrams (DFDs) are used to capture the functional model of a system. DFDs have three types of nodes: processes that transform the data, actor objects that process and consume the data, and data store objects that passively store the data. Data is moved among the nodes by data flow arrows.

Rumbaugh claims that the three models of an application are orthogonal and does not provide any constructs to connect them. However, clearly the states in a state diagram are derived from the object model and the functional transformations depend on the data and the state. This gap is expected to be filled during implementation.

Rumbaugh's notation is most suitable for object-oriented languages that distinguish meta-data and data. Unlike Booch, he does not introduce notation that supports objects that could correspond to instances, classes, or meta-classes. As such he uses class diagrams only and does not require object diagrams.

Like the Booch notation, the OMT object diagrams provide constructs to model associations and instantiation relationships. However, there is no construct to capture the "delete" relationship, nor is there any way to describe dynamics of relationship evolution, instantiation or deletion.

Rumbaugh separates system development into three stages: analysis, system design, and object design. The analysis stage captures the problem statement, and the object, dynamic, and functional models in a design document. The system design phase generates a document describing the structure of basic architecture for the system, as well as high level strategy decisions. The object design phase results in a document with detailed object, dynamic, and functional models. Implementation is not discussed.

2.5.3 Shlaer-Mellor

Like Rumbaugh, Shlaer and Mellor represent a system in three models. In their terminology these models are:

- the information model;
- the state model; and

- the process model.

The information model, discussed in great length in [18], categorizes objects into domains[3] and subdomains; describes entities, their attributes and associations, and the policies, rules, and physical laws that prevail in the real world. A graphical notation is used to depict objects and their relationships. The information model corresponds to Booch's class and object diagrams, and Rumbaugh's object model.

The state model, discussed in great length in [19], is concerned with the behaviors of objects and their relationships over time. Like Rumbaugh's dynamic model, the state model is represented by finite state machines within classes. A formal representation of state machines is not provided, rather, a set of concepts are discussed. These concepts include states, begin and end states, events, actions, input events, output events, event sequences, event synchronization, timers, and monitors that govern event synchronization.

Shlaer and Mellor do address object creation and deletion, as well as, relationship evolution. However, again, the discussion remains at the conceptual level. A state machine goes to its begin state when created, and it is deleted when it reaches its end state. Relationship evolution is represented by state machines.

The process model, described by Action Data Flow Diagrams (ADFD), further refines each action specified in the state model. The ADFD consists of processes that represent unit operations within objects; the object data stores that represent the object attributes; and arcs that represent the event and data flow.

Shlaer and Mellor separate the development process into three major stages: analysis, design, and implementation.

The analysis phase partitions the system into domains and delivers the information, state, and process models.

The OODLE (Object Oriented Design LanguagE) notation is introduced as a language-independent design notation. The notation consists of the following components:

- Class diagrams that capture the external view of individual classes;
- class structure charts that show the internal structure and operations of the class;
- dependence diagrams that depict the relationships between classes; and
- inheritance diagrams that depict the class inheritance hierarchy.

Implementation is expected to transform the analysis and design results to actual running code.

2.5.4 General Comments

Many other object oriented methodologies exist. Among others, Coad [14], Jacobson [15], Schultz [17], Wirfs-Brock [20], and Buhr [13] have their flavors of the OOM. However, like the three methodologies described above, these methodologies target the analysis phase of a project and barely touch the design. The primary products are documents that capture the analysis and design results in a general and loose notation. It is difficult, if not impossible, to streamline the conversion of these documents into actual implementations.

2.6 Architectural Concepts

This section defines some basic process and architectural concepts.

[3] Booch's modules

2.6.1 Process Organization

In a *client-server* architecture, processes have roles. A *server* responds to requests of many *clients*. The client-server architecture can be organized as a tree, i.e., a client can act as a server to other processes. In many applications the client-server architecture is used to separate the data stored in the server from the client user applications.

The *agent-manager* roles are frequently used in network management. A process that provides a direct access to a hardware component is called an *agent*. The *manager* uses the agent's services to access the hardware. Agent and manager roles can be nested. To access the hardware, an agent may actually use another agent. Furthermore, based on the operation, a process can act both as an agent and as a manager .

The *peer-to-peer* architecture removes all role semantics. It refers to two communicating processes.

2.6.2 Asynchronous vs. Synchronous Execution

Most conventional programming languages have a single execution thread. If the process state is modeled as a piece of memory, a sequence of method invocations result in a sequence of changes in the process state. Once invoked, each method assumes full control, causes some state change, and terminates. A method call usually corresponds to a line of code where the method *returns* a value and control is moved to the next line. Such a method invocation is usually called *synchronous*.

If a method call can cause state changes in a process after it returns it is called *asynchronous*. True asynchronous execution requires the spawning of other processes that somehow share the same state.

Asynchronous behavior can be implemented in a single threaded language. Usually a main control loop is used that invokes a series of methods in a loop. Some of these methods result in the creation of *Events* which are queued in some buffer. Other methods, when invoked, process the buffers and deliver the events to their recipients. In this case, event delivery takes place asynchronously.

2.7 Persistence Storage: RDBMS and OODBMS

2.7.1 RDBMS

The Entity-Relation (ER) model is well understood [27, 28, 29]. Like in the above discussion an *entity* is a thing which can be distinctly identified, and a *relationship* is an association among entities. In the ER model, relationships themselves are entities. Properties of entities are described by attributes that can have arbitrary universe of discourse.

The entity-relationship model distinguishes meta-data and data.

In most implementations a three-schema architecture is used for metadata modeling languages:

- Data Definition Language (DDL), used to define the global schema;
- View Definition Language (VDL), used to define the external schema;
- Storage Definition Language (SDL), used to define the internal schema for persistent storage.

A Data Manipulation Language (DML) is provided to manipulate the Data. The DML can be a graphical or command-line driven. It can be a high level query language or provide interfaces to programming languages.

The entity-relationship model is implemented by Relational Database Management Systems. The correspondence between the model and its implementation is summarized below:

Entity	Database Table
Instance	Table Row
Attribute Type	Column Type
Entity Attribute	Table Column
Entity Name	Table Name
Instance Identity	Primary Key Column of Table
Relationships	A secondary key column in a table, containing the primary key of relationship participant

Major RDBMS providers include Informix, Ingres, Oracle, and Sybase.

There are certain shortcomings of the RDBMS implementations. These are:

- Relationships are not implemented as entities, but rather as a reference to a primary key;
- There is no semantic support to identify the relationship type;
- There is no explicit construct for object identification. The primary key is just any other column with extended semantics;
- Attributes are restricted to atomic values. To represent sets one needs to resort to using a secondary table. To represent ordered lists, a separate column is needed to maintain the order;
- Complex entities, with many relationships and complex attributes, must be decomposed to many tables, incurring access overhead.

Finally there are certain shortcomings of the entity-relationship model itself:

- There is no behavior description;
- There is no good model for cardinality and semantic constraints;
- There is no support for event management;
- There is an inherent impedance mismatch between the entity-relationship model and programming languages.

2.7.2 OODBMS

Object-oriented databases have evolved as an extension of object-oriented methodology. As yet, there is no formal object database model, but just manifestoes [25, 33]. The correspondence between basic OODBMS and RDBMS implementation constructs are summarized below:

Class	Database Table
Instance	Table Row
Attribute Type	Column Type
Class Attribute	Table Column
Class Name	Table Name
Object Identifier	Primary Key Column of Table

Some OODBMSs provide association constructs.

Most RDBMS providers have started to provide OODB constructs. Other OODMS providers include GemStone, Matisse, O2, ODI's ObjectStore, Objectivity, Ontos, and Versant.

There are two main implementation approaches to OODBMSs.

The first approach extends an object-oriented language with persistence. A base class is used to implement methods related to persistence. Pointers are replaced with an overloaded `Link` construct that has similar syntax as pointers. Links have the additional capability of seamlessly retrieving an object from disk if the object is not in memory. Other persistent list and set attributes are provided in an extension library that the user can treat as part of the language. Most commercial databases use this approach.

This approach simplifies the integration of persistent storage into an application and removes the impedance mismatch that the ER model suffers from. However, in the absence of standard language extensions, the apparent seamless integration makes the application dependent on a particular vendor's language extensions. Furthermore, in this approach the granularity of the save and retrieve operations is usually limited to object instances. There is no support for retrieving only a subset of attributes of an object.

The second approach is more closely tied to the relational model. The database is independent of the programming language, but, provides an interface to save and retrieve objects. This approach makes it easier to treat persistence as an independent module and also provides constructs for saving and retrieving parts of an object. However, by nature of the design, there is an impedance mismatch between the persistence constructs and the programming language. All database operations have to be stated explicitly by the programmer.

The overall OODBMS model has a number of shortcomings:

- There is no relationship semantics;
- The behavior description is too unstructured;
- There is no inherent support for event management;
- There is no or limited support for aggregation and views;
- There is no standardized query language.

In the absence of more structured behavior description languages, it is unlikely that a standardized query language will ever emerge.

2.7.3 Choosing an OODBMS

There are many more services that a database management system has to provide. These are:

- Transaction management;

  transaction logs, shared transactions, long transactions, check-pointing, check-out and check-in support, atomicity, consistency.
- Concurrency management;

  optimistic and pessimistic locking, deadlock detection, version control.
- Security management;

  positive and negative authorization at class or instance level, users, user groups.
- Distribution support;

  migration, directory services, distributed indexes, location independence, local autonomy.
- Architecture independence;

  hardware, operating system, programming language independence.
- Schema evolution;

  meta-data/data independence, adding, deleting, and modifying meta-data without losing existing data.
- System management.

  replication versioning, back-up and recovery.

Last but not least databases must have good performance.

For a more complete treatment of object database features the reader is referred to [26, 32]. Some database comparisons can be found in [30, 24], however, the reader should always look for the most recent evaluations.

2.8 Functional Categories

OSI/NM Forum has identified seven functional categories for network management[4]. These categories were summarized in the Introduction and can be used for many other application areas. We now list object model requirements that result from these functional categories.

2.8.1 Configuration Management

In all applications a logical model for a physical system must be created. The logical model must capture the entities in the system, their state, their inputs, outputs, interrelations, and behavior.

Tools are needed to instantiate these entities and to create sets of instances with particular interrelationships.

The logical model captures the possible behavior of objects. Other constructs are needed to control the behavior of collections of objects.

Logical counterparts of sensors and actuators are needed to specify the event and control flow. In particular constructs are needed to specify time and event driven evolution of the system.

2.8.2 Fault and Event Management

All physical systems have normal and degraded modes of operation; in the limit they fail.

Constructs are needed to specify the different modes of operation of a system. An event creation mechanism is needed for identification of possible faults and performance degradation. Other fault recovery constructs are needed that ensure that the system remains in normal mode and that provide graceful recovery in case of faults. Finally an event notification mechanism is needed to propagate events to fault recovery constructs.

2.8.3 Performance Management

A given system has to satisfy many, possibly conflicting, performance requirements. Constructs are needed to define an evaluation criterion or evaluation criteria as needed.

Other constructs are needed to observe the behavior of a system and measure its performance based on evaluation criteria.

Finally constructs are needed that can adapt the system to changing evaluation criteria and optimize its behavior.

2.8.4 Access and Security Management

An application has many users. It is crucial to restrict the access level of individual users. Users may be allowed to access certain types of entities only, certain instances only, or specific operations of specific instances only.

2.8.5 Financial Management

The users of the system are required to pay for the rendered services. Constructs are needed to track system usage.

2.8.6 Resource Management

Constructs are needed to provide an inventory of all objects in the physical system and to administer their maintenance schedules.

2.8.7 Planning and Design Management

Macroscopic planning decisions require the ability to specify, simulate, and evaluate alternative physical system configurations and control policies.

[4]More recent work emphasizes the first five categories only.

2.9 Formal Modeling Methods

In Section 2.2 we discussed the trade-off between the expressive power of a formal modeling syntax and the possibility of analyzing a problem modeled in that syntax.

Although most of this thesis is primarily concerned with facilitating the specification and evaluation of control architectures through simulation, verification of control algorithms is always preferable. Simulation exercises the behavior of the system for a subset of possible inputs; verification exercises it for all possible inputs, and can make absolute statements about the system.

Most large systems can not be verified. However, partial verification of a system is always an option. In this case the challenge is to abstract the verifiable subset of a system in a formal manner, such that the verified abstraction still has a well-defined relation to the actual system.

In this section we provide an overview of some existing formal specification models and their use in control and verification of systems.

2.9.1 Automata

The least expressive yet most decidable and most popular formal syntax is that of Finite State Machines (FSM).

A FSM is a five-tuple: $(Q, \Sigma, q_0, \{q_f\}, \delta)$, where Q is a finite set of states, Σ is a finite set of possible events, q_0 is the initial state, $\{q_f\}$ is the set of possible final states, and $\delta : Q \times \Sigma \rightarrow Q$ is a partial function called state transition function.

To model a real system using FSMs, one sets up a correspondence between the possible states of the system and the states of the FSM. The state transition then defines the behavior of the system in response to certain events.

The language generated by a FSM is defined to be the set of all strings of events, called traces, that can be generated starting at the initial state and ending at a final state.[5]

FSM representations have been used to express the Discrete Event Control of systems. Most existing work follows a language approach. Given a desirable subset of the language generated by an automaton, certain transitions of the automaton are disabled to restrict its language to the desired subset [89]. The state based approach disables transitions to keep the automaton within a desired subset of automaton states [87].

FSMs have been widely studied. In particular one can decide whether the language generated by two FSMs are the same, or if the language of an automaton is empty.

These results make FSMs suitable for verification. If a complex system is modeled by a FSM, one can create a second FSM modeling undesirable event strings. If the language intersection of the two FSMs is empty, one can conclude that the complex system does not exhibit the undesired behavior.

Analysis and verification tools for finite automata are readily available [95, 94].

FSMs can not represent continuous evolution. Several formalisms have evolved that extend the FSM syntax with continuous components [74, 84]. Such extended FSM representations are called Hybrid Dynamical Systems [78].

Deshpande summarizes, "Hybrid systems are continuous variable, continuous time systems with a phased operation. Within each phase the system evolves continuously according to the dynamical law of that phase, and when an event occurs, the system makes a transition from one phase to the next. Thus, hybrid systems display both discrete and continuous behavior. The discrete behavior consists of abrupt transitions between phases, and the continuous behavior consists of smooth flows within phases. The terms "modes," "locations," or "places" are also used to refer to the phases of operation.

The state of the system can be thought of as a pair—the discrete state and the continuous state. The discrete state identifies a flow, and the continuous state identifies a position in it. For a transition to take place, the discrete state and the continuous state together must satisfy a condition that enables the transition. Once a transition occurs, the discrete state and the continuous state are changed abruptly. Then, until the next transition, the continuous state evolves according to the

[5]There are many other flavors of FSM and FSM language definitions [86, 76].

flow identified by the new discrete state. Thus, the discrete dynamics and the continuous dynamics influence each other" [77].

Most OMS applications are hybrid systems. The control of a continuous state dynamical system or plant is often organized in several layers. At the lower layers, the plant is regulated in the traditional manner: a controller continuously monitors the plant state or output and selects the real time control input. At the upper layers, event-based supervisors are used that issue symbolic commands and receive symbolic responses indicating either the successful execution of the command or the occurrence of an error condition.

Unfortunately very few analytic results are available for hybrid systems and many questions concerning hybrid automata are proving to be undecidable.

Most analysis tools rely on proper abstraction of the continuous components of the model and provide only partial system verification. Simulation is used to explore the full behavior of the automaton.

2.9.2 Petri-Nets and Data Flow Diagrams

Some of the ideas of the Object Model constructs proposed in this thesis are borrowed from Petri-Nets and Data Flow Diagrams.

A Petri Net is a four-tuple: (P, T, M, A), where P is the finite set of places, T is the finite set of transitions, M is the initial marking of the Petri Net, and A is two sets of directed arcs $A_i \subset P \times T$ and $A_o \subset T \times P$. A_i describes the arcs from places to transitions, A_o describes the arcs from transitions to places.

The state of a Petri Net at any instance is given by its mark, which is the number of tokens in each place. A transition t_i can fire if all places having an arc to t_i have a token. When a transition fires, it removes one token from each of its input places, and places one token in each of its output places, places to which that t_i has an arc.

There are many ways of defining Petri Net languages. It is known that they properly contain FSMs, and that they overlap with context free grammars. The question whether a mark is reachable for a given Petri Net is decidable, yet the question whether two Petri Nets have the same reachability set is undecidable.

Petri Nets have found extensive use in the industry and academia. They are widely used in many applications such as modeling of operating systems, job shops etc. Research in Petri Nets is still very active.

Data Flow Diagrams are widely used to describe the functional model of complex systems [5, 16, 18] in a graphical notation.

A Data Flow Diagram is given by a triple: (P, D, A), where P is a set of processes, representing some data transformation operation; D is a set of data stores, representing conceptual persistent data storage elements; and A is a set of data flow arcs, carrying data among processes and data stores. The traces of directed arcs between processes and data stores describe the sequence of transformations a unit data undergoes.

2.9.3 Other representations

Many other notational techniques exist for formal specification of behavior.

Calculus of Communicating Sequences (CCS) [85], Communication Sequential Processes (CSP) [79] and Finitely Recursive Processes (FRP) [82] have a different perspective than FSMs. The main concept they model is not the state of a device, but rather a process that has certain characteristics. This perspective has a more intuitive appeal if one wants to model processors that run in parallel.

Estelle, a name derived from Extended Transition Language, is a standardized formal description technique for the specification of communication protocols and services [92].

LOTOS, a name derived from Language of Temporal Ordering Specifications, is a standardized formal description technique which specifies communication systems by defining the temporal relations between its externally visible interfaces [93].

SDL, Specification and Description Language, is a language used for formal specification of real-time, interactive, distributeed systems. SDL has been used for partial specification of many telecommunication systems [91].

DEVS (Discrete Event Dynamical System) [75, 80, 88, 90] formalisms are also widely used in the computer science and artificial intelligence communities.

2.10 Sample Software Frameworks

In this section we summarize three software frameworks. Ptolemy is a data flow based simulation tool intended for signal processing applications. COSPAN is a verification tool used for verification of discrete event systems. CSIM provides basic scheduling constructs for modeling event driven evolution of processes.

None of these frameworks is suitable for implementation of object management systems, however, they may be used as components in some applications.

2.10.1 Ptolemy

Ptolemy is an object-oriented data flow based simulation tool [97]. Ptolemy objects, called stars, are represented by inputs, outputs, and input-output maps. In Ptolemy, object interaction is governed by messages that are propagated across input-output connections. The Ptolemy scheduler implements timely propagation of messages.

Ptolemy *domain*s are used to capture the various evolution modes of objects. Object evolution can be driven by time, by events, by data tokens or by a combination thereof.

A message queue domain is provided to exchange messages between objects that are not connected, but the semantic support for this domain is limited. Unconnected objects share a blank piece of memory that they can use arbitrarily.

Constructs do exists to dynamically alter the input-output connections between objects during a simulation run. However, these constructs are not well-integrated into the conceptual framework.

Ptolemy is particularly useful for specification and simulation of data flow among a static configuration of objects. It is not suitable for representing large number of objects with evolving relationships.

Object configurations are specified with a graphical editor, and monitor objects are used to observe the system output in graphical or file format.

Ptolemy does not have a well-defined and efficient interface for interacting with other simulation packages. Other constructs absent in the current implementation of Ptolemy are: dynamic activation of monitors, persistence of simulation state, and distribution support. Verification is not within the scope of Ptolemy.

Ptolemy is an active system still under development and some of these shortcomings may be addressed in future releases.

2.10.2 COSPAN

COSPAN is a general-purpose software tool for coordination-specification analysis. It provides an automaton description syntax and a set of operators to combine automata. The physical system is described by a set of automata, with language $\mathcal{L}_P$. The desired behavior of the physical system is described by a second set of automata with language $\mathcal{L}_D$. Conceptually, COSPAN verifies the correctness of the statement:

$$\mathcal{L}_P \cap (\mathcal{L}_D)^C = \emptyset \tag{2.1}$$

Here $(\mathcal{L}_D)^C$ denotes the complement of $\mathcal{L}_D$, i.e. the undesired behavior.

Har'El and Kurshan, the main developers of COSPAN, claim, "Typical applications of COSPAN include software development and hardware development for the implementation of control-intensive structures such as communication protocols; analysis of circuits (with arbitrary feedback) at transistor or gate-level for race conditions and logical correctness; analysis of array processors for functional correctness; logical analysis of discrete-event models in economics, medicine and strategic planning.

Logical analysis consist of symbolic testing of a system for quite general user-defined behavior (not simulation or execution of the system); as such, the analysis constitutes a mathematical proof (or disproof) of the stated behavior of the system" [95].

2.10.3 CSIM

CSIM [96] is a general purpose, C-based, process-oriented environment, designed to simulate a discrete event system. It is commonly used to simulate the behavior of data communication networks and VLSI designs.

The primary unit of execution in CSIM is a process. Processes are initiated by other active processes. Upon initiation, a process executes independently of its initiator. The main process is *sim* and does not require initiation.

Processes can interact by sending, receiving, broadcasting, and setting events. A process can wait for an event, or put itself on hold for a specified time interval.

CSIM provides a quasi-parallel environment, i.e., simulation time does not advance until all processes which are scheduled for the current time are activated, exercise their behavior, and are re-scheduled for a later time. However, the real or cpu time needed per unit of simulation increases with the number of processes present in the simulation.

There are experimental versions of CSIM that run on parallel machines. However, a distributed version of CSIM is not available.

2.10.4 Other

Most standard formal modeling languages such as Estelle, SDL, and LOTOS have implementations that are commercially or publically available. Current available implementations do not provide the necessary interfaces that would facilitate their efficient incorporation into a larger simulation framework.

2.11 Conclusions

We have summarized desirable software features, semantic modeling constructs; software engineering processes; object-oriented analysis, design, and programming methodologies; relational and object database management systems; state machine representations, Petri-nets, and data-flow diagrams; and software platforms that provide implementations of these various component technologies.

The applications we address in this thesis require the integration of all these component technologies into one uniform framework.

The OMS approach organizes and extends these technologies around an austere object model and an end-to-end development process.

Chapter 3

Sample Problems

Object Management Systems (OMS) are object-oriented software systems used to simulate, evaluate, and control large-scale physical environments. Examples of such environments are transportation networks, telecommunications networks, power distribution networks, air traffic control, and management information systems. These environments are heterogeneous, dynamic, and distributed.

In this chapter we provide an overview of three application domains: automated highways, power distribution automation, and network management.

3.1 Automated Highway Systems

Highway congestion is imposing an intolerable burden on many urban residents. It is estimated that lost productivity due to traffic congestion costs $100 billion each year in the United States. Alongside congestion, safety continues to be a prime concern. In 1991, 41,000 persons died in traffic accidents, and more than 5 million persons were injured. The National Safety Council estimates that in 1992, the cost of accidents totaled $157 billions.

Intelligent Vehicles and Highway Systems (IVHS) is a comprehensive program initiated by the U.S. Government under the Intermodal Surface Transportation Efficiency Act of 1991 to improve safety, reduce congestion, enhance mobility, minimize environmental impact, save energy, and promote economic productivity in the transportation system. The IVHS program combines several modern technologies, including information processing, communications, control, and electronics. IVHS has the following sub-programs.

Advanced Traffic Management Systems
: ATMS provides subsystem integration of traffic management and control systems, and performs real-time traffic control to respond to dynamic traffic conditions.

Advanced Traveler Information Systems
: ATIS acquires and analyzes information about transportation network dynamically, and communicates advisory information to travelers.

Advanced Vehicle Control Systems
: AVCS uses computers, communications, and control systems in the vehicles and the highways to enhance vehicle control.

Commercial Vehicle Operations
: CVO improves the safety and efficiency of commercial vehicle and fleet operations.

Advanced Public Transportation Systems
: APTS integrate public transportation with vehicle-highway systems by using component technologies from other functional areas.

Outside the U.S., there is substantial IVHS activity in Europe under the PROMETHEUS[1] and the DRIVE[2] projects, and in Japan under the RACS[3], AMTICS[4] and VICS[5] projects.

Modeling and simulation have been identified as important steps in realizing these transportation initiatives. The IVHS strategic plan [42] requires modeling and simulation in the following areas: urban traffic network models, traffic system models, vehicle-road models, driver-vehicle models, traffic models with dynamic traffic assignment, driving scenario simulation, and advanced vehicle control systems (AVCS) architecture simulation. Such modeling and simulation environments would address the following seven functional categories:

Configuration Management—
: the ability to specify a highway network configuration, the traffic patterns on it, and the vehicle and traffic control strategies;

Fault Management—
: the ability to detect faults and significant events such as accidents and congestion, and to respond to them with graceful degradation of highway performance and with automatic fault recovery;

Performance Management—
: the ability to track, evaluate, optimize, and fine-tune the transportation system performance;

Planning Management—
: the ability to specify and simulate alternative highway and traffic configurations and control strategies for the purpose of planning;

Resource Management—
: the ability to provide an inventory of all highway and vehicle resources and to schedule them for preventive maintenance;

Accounting Management—
: the ability to specify tolls and taxes, and to account for highway usage;

Access and Security Management—
: the ability to restrict access to information about the transportation system to authorized users only.

It is important to distinguish the actual control and communication design of an automation strategy from its simulation and evaluation. The mandate of the automation strategy is to provide cheap, safe, speedy, comfortable, and clean transportation. The objective comparison of proposed alternatives is an independent task.

A software framework in which alternative IVHS strategies can be specified, simulated, and uniformly evaluated is crucial for objective comparison of the proposed alternatives. The simulation framework would provide configuration, fault detection, performance evaluation, and planning functionalities. The requirements of such a simulation framework are discussed in Section 3.1.2

3.1.1 Underlying Concepts

The organization of highway automation will embody the following concepts:

- layered control architecture,
- coordination of distributed control agents,
- combined discrete and continuous dynamical systems, known as hybrid systems, and their control and verification,

[1] Program for European Traffic with Highest Efficiency and Unprecedented Safety
[2] Dedicated Road Infrastructure for Vehicle Safety in Europe
[3] Road/Automobile Communication System
[4] Advanced Mobile Traffic Information and Communication System
[5] Vehicle Information and Communication System

Layered Control Architecture

In the layered control architecture proposed by Varaiya and Shladover [41, 39], vehicles perform simple maneuvers such as merging into platoons, splitting from platoons, following the leader, changing lanes, and entry and exit. A vehicle executes complex end-to-end trajectories by performing a sequence of such simple maneuvers. Efficient transportation throughput is achieved by tuning traffic parameters such as platoon size and vehicle speed. The control strategies for such behavior are hierarchically organized in four layers: regulation layer, coordination layer, link layer, and network layer. These layers are shown in Figure 3.1.

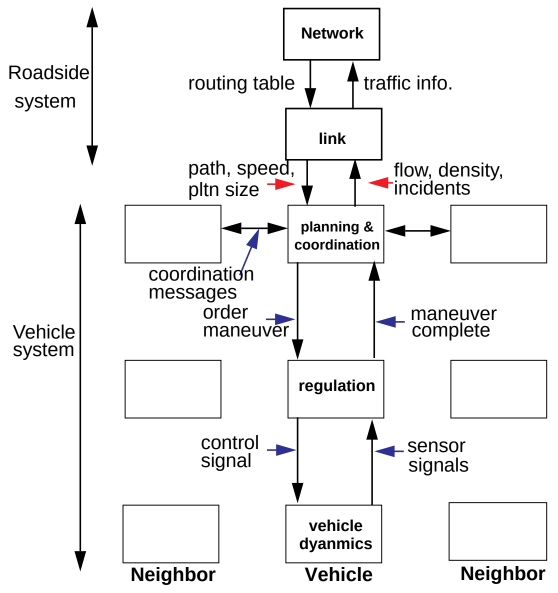

Figure 3.1: Layered Control Architecture.

Given a maneuver to perform, the vehicle follows a control strategy that regulates its dynamical behavior to a trajectory that realizes that maneuver. Such control strategies constitute the regulation layer. The maneuver to be followed by a vehicle at a given time is determined by coordinating with other vehicles in the neighborhood. The control strategies used for such coordination constitute the coordination layer. The control strategies adapt their behavior based on information about highway traffic conditions. The traffic conditions on highway segments are monitored and controlled by road-side control elements, organized in the link layer. Finally, information from individual highway segments is aggregated, and end-to-end routing and congestion control is accomplished in the network layer.

The framework should allow the specification of this and other layered control architectures.

Coordination of Distributed Control Agents

The layered scheme described above yields a distributed control strategy since each vehicle and each highway segment is responsible for its own control. At the same time, effective coordination of these distributed control agents is essential for efficiency and safety. The agents coordinate by following simple heuristic rules. For example, when a vehicle senses another vehicle ahead of it, it requests a merge with it to form a platoon. Forming a platoon increases the efficiency of the highway. However, if the leading vehicle is already a part of a large platoon, it may refuse the merge request since inordinately large platoons are potentially unsafe. Such coordination strategies are modeled using a Discrete Event System (DES) approach. The control agents communicate the discrete events to each other based on coordination protocols. Thus, communication mechanisms are essential both for gathering sensory information and for executing these coordination protocols.

The framework should allow the specification of sensors, transmitters and receivers, and of communication protocols.

Verification and Control of Hybrid Systems

Whereas the coordination strategies deal with discrete events, regulation strategies deal with continuous evolution. For example, if a merge maneuver is to be executed, then the regulation layer controller must first accelerate the vehicle, close the distance between itself and the vehicle ahead of it, and finally decelerate and follow at the same speed while maintaining a safe distance in between. Acceleration and braking, speed, and distance are continuous parameters that evolve in continuous time. Thus, the discrete coordination event corresponding to the merge command and the continuous regulation law corresponding to the merge trajectory must be dealt with together. A hybrid system approach is used to model the combined discrete and continuous behavior.

The framework should allow the specification of both discrete and continuous behavior.

3.1.2 Framework Requirements

In this Section we list the functional, modeling, and software system requirements for a software framework that would facilitate the specification, simulation, and objective evaluation of highway automation strategies. Part of the functional and modeling requirements are derived from the MITRE report [40].

Framework Users

The simulation framework will have several categories of users. These are:

- Control and Communication engineers;

  These users design, implement, and test individual control and communication components. They need the ability to define clear input and output interfaces for their components, and a straightforward mechanism for integrating their components into the overall system.

- System analysts;

  These users test and evaluate automation strategies. They need the ability of integrating various control and communication components to evaluate their collective performance.

- System Planners.

  These users select the automation strategy for deployment based on system analyst's evaluation results.

Configuration Management Requirements

The framework has to provide the following configuration management functionalities to address the users' needs.

- Ability to represent arbitrary highways;

  A set of highway objects must be provided for the creation of highway networks. A grammar or set of rules is needed that describes the valid ways of interconnecting highway objects.

- Ability to represent incoming and outgoing traffic patterns;

  Extensive work has been done for studying traffic patterns for certain highways. A mechanism must be provided to incorporate such information into the simulation. Such traffic patterns may also be generated by urban roadway simulators. Interoperability with other simulation packages is essential.

- Ability to create vehicles consisting of many components;

  Vehicles must be represented as composite objects. Different control architectures will select a different set of automation components for vehicles.

- Ability to create roadside controllers consisting of many components;

  Different control architectures will select a different set of automation components for the roadside. Therefore a composite roadside control mechanism is needed.

- Ability to have different types of vehicles on the highway;

  Not every vehicle on the highway will consists of exactly the same components. The dynamics of a truck will be different from that of an automobile.

- Ability to represent inter-vehicle and vehicle-to-roadside communication; and

  Most automation strategies will employ some sort of inter-vehicle and vehicle to roadside communication.

- Ability to define traffic rules.

  A high level specification language is needed to specify traffic rules such as "no lane change in this section". Any violations of these rules must be recorded by the framework.

Fault Management Requirements

The simulation framework does not provide fault recovery. Fault avoidance and fault recovery are part of the control and communication architecture design.

The simulation framework provides fault detection and fault creation mechanisms:

- Ability to detect accidents;

  Accidents and component failures are part of the physical system. The framework has to be able to detect accidents and take some action upon accident detection.

- Ability to create accidents.

  Since the "Falling Rock Zone" is unlikely to be an explicit part of the simulation framework, there should be a means of inducing accidents and equipment failures, in order to evaluate how various automation strategies cope with them.

Performance and Planning Requirements

The evaluation criteria for automation strategies are multidimensional, including utilization, travel time, safety, comfort, and implementation complexity, and cannot be specified a priori. The framework must support:

- Ability to collect arbitrary statistics;

  There should be a straightforward mechanism for collecting statistics about system state.

Modeling Requirements

Based on functional requirements we summarize a minimal list of entities that must be present in the simulation framework. Any highway automation architecture would depend on the existence of these entities.

- Highway representation;

  Lanes, lane splits, highways, highway splits, entries, exits are needed to represent highways. Geometric representations for curvature, incline and tilt are needed for realistic simulations.

- Vehicle representation;

  Vehicles and specializations such as car, truck, bus etc., are needed.

- Control components;

  Microsimulation takes place at vehicle component level. Representations of control components such as lateral and longitudinal controllers are needed.

- Sensors;

  Any automation strategy must rely on sensory input data to observe system state.

- Communication devices;

  An automation strategy is likely to include communications, at least at the roadside control level.

- Monitor components;

  The simulation framework is intended for evaluation. Monitors provide an interface between the simulation and evaluation components.

Concepts such as platoons and carrier plates specific to automation architectures will be added to this list as they emerge.

Validation and Deployment Requirements

The framework simulates logical models of physical objects. The framework should provide a clear path between logical and physical representations:

- Ability to validate logical models;

  The accuracy of simulation results are limited by the quality of component models. It should be possible to test their accuracy against their physical counterparts.

- Deployability of component specifications;

  In many cases the components used in simulation will correspond to planned or experimental hardware. A migration path is required to convert the software specifications into hardware elements.

Software System Requirements

In the introduction we argued that modularity, good performance, scalability, openness, and robustness are desirable characteristics for any software system. In this application these requirements appear in the following form:

- Ability to associate physical and logical representations: Modularity;

  The framework models numerous physical components. Since the "components" have to be interchangeable, their implementation, i.e., software encoding, must be self-contained.

  Each such "logical" component will be deployed as a "physical" component as the IVHS system matures. This modularity will also facilitate model validation and deployment.

- Ability to add new components to the system with minimal code rewrite: Openness, Modularity, Robustness;

  New vehicle and roadside components will be added to the framework as work progresses. The incorporation of such new components should require minimal rewrite of any other existing software.

  Since separate groups may implement various control agents and since things won't work perfectly when these agents are first integrated in one system, the simulation framework has to enforce strict boundaries between different control agents. These boundaries can then be used to identify faulty control agents and to provide fault identification.

- Ability to collect arbitrary statistics during simulation: Openness, Modularity;

  Unless the full state of the system is visible, arbitrary statistics can not be collected. As part of evaluation support, the framework should be able to *trace* the behavior of any subset of objects for future review and replay. Such tracing information can also be used for animations and for virtual reality display devices to measure driver responses.

  There are many existing software packages for processing statistical data. The framework should provide an open interface to interact with these packages.

- Ability to run simulations with acceptable performance: Performance;

  For most purposes, the simulation does not have to be real-time. Indeed, no architecture can guarantee an upper bound on the simulation time of a given object, since this time greatly depends on the amount of detail in the object model. A vehicle's position can be calculated simply as the third integral of its jerk, or, many state variables can be introduced to take the entire engine into account.

  The simulation framework, however, does impose a lower bound on the simulation performance. Assume all vehicles move forward by one meter at each time increment with no further behavior. At each time increment the framework has to obtain these displacement values for each vehicle and update their position on the highway. The time taken to perform such "bookkeeping" operations impose a lower bound by the framework for a single time increment.

  The deployed system, on the other hand, has to satisfy many real-time requirements. A real-time operating system is needed within each vehicle and on the roadside to coordinate all system components. This task is beyond the scope of the simulation framework. However, one should keep in mind that the algorithms embedded in each system component have to meet real-time requirements in the target hardware they will be deployed on.

- Ability to adjust simulation granularity: Modularity, Openness;

  The framework should allow the users to specify different levels of physical models to adjust model granularity. The framework is intended for micro-simulation, however, simulating detailed engine dynamics during flow calculations in a 500 mile highway is not very productive. The framework should also allow the user to adjust the granularity of time evolution. Time increments for vehicle position updates, or statistics collection should be variable.

- Ability to simulate up to 100.000 vehicles: Performance;

  Not every simulation run will encompass 100.000 vehicles. But, system level simulation runs must be capable of supporting large number of vehicles.

- Ability to specify system behavior in a straightforward language: Ease-of-use;

  The behavior descriptions have time and event driven components. Suitable languages are needed to express time and event driven behavior.

3.1.3 Distinguishing Problem Characteristics

We summarize framework requirements into the following four criteria that dictate certain design decisions:

- The modularity requirements dictate that we follow the object-oriented paradigm for system specification and implementation;
- The openness requirements and the need to collect arbitrary statistics dictate that the simulation state be fully visible and can be saved in a persistent storage medium;
- Performance and modularity requirements dictate that simulation is distributed; and
- Inter-vehicle communication requirements eliminate static block diagram based simulation environments.

A number of component technologies were discussed in the previous chapter. These technologies provide partial solutions, yet none meets all requirements. Object oriented methodology provides a start, but is to broad to be a solution. Databases meet some of the persistence requirements, yet, they do not provide any semantic support nor any application level specification constructs. Formal modeling methods are useful, but limited unless they are part of a broader framework.

No existing simulation environment satisfies all four criteria; nor do they have a sufficiently open architecture that allows their proper customization to the application domain.

Object Management Systems, discussed in subsequent chapters, are designed to meet these criteria in one integrated uniform framework.

3.2 Power Distribution Systems

The electric utility industry throughout the world is undergoing fundamental changes. After decades of continuing decline in the price of electricity, the 1970's mark the beginning of escalation of electricity costs. As a result, the electric power industry is being significantly restructured in order to adapt to increased competition, a harsher financial climate, new technologies, and heightened environmental concerns. Privatization of the power industry is a fact in England, Wales, Norway, and Chile, and in the US, it is being planned.

Two noteworthy features of this restructuring are:

- Organizational changes;

  Internal reorganization of utility companies to improve accountability and to encourage entrepreneurship; the entrance of independent power producers; and growing reliance on the part of regulatory authorities on market forces as a means to greater efficiency.

- Move to differentiated services.

  A reorientation of the utility industry's strategic mission from one of providing uniformly high quality power to all customers, to one, in which customer needs are assessed and services are tailored to match customer market segments.

Potential areas of major performance gains as a result of restructuring include:

- Reduced and deferred capital expenditures through better planning and equipment utilization;
- Reduced generation capacity expansion and fuel costs through greater use of wheeling, better coordinated inter-firm economic dispatch, and demand management; and
- Increased revenues through an expanded set of value-enhanced and differentiated services matched to customer needs.

Major performance improvements in power system operation can be achieved by extending automation to the level of lower voltage distribution systems and all the way to consumers. Only such automated distribution systems can accommodate future expansion of enhanced demand-side management and customer service. It is estimated that billions of dollars could be saved by operating the distribution networks more efficiently [51].

A comprehensive list of distribution automation functions is presented in [50]. Traditionally, automated power distribution network management functions are grouped under the following categories [45]:

- Supervisory Control and Data Acquisition (SCADA);

  Normal operation SCADA functions are: feeder reconfiguration to perform load balancing and loss minimization; and Voltage and Var[6] control through capacitor switching, voltage regulation, and reactive power control.

  Emergency response SCADA functions are: protection through adaptive relaying; feeder switching and sectionalizing; and customer restoration.

- Demand Side Management (DSM); and

  DSM functions include power quality control, direct load control, remote metering, real-time pricing, priority service, and customer service.

- Automated Mapping (AM), Facilities Management (FM), and Geographic Information Systems (GIS).

  AM/FM/GIS functions include maintenance and work order management.

[6]Var, i.e., Reactive power is discussed in Section 3.2.1.

These domain specific functional categories can be reorganized under the seven functional categories of Section 2.8. Normal SCADA operation functions correspond to configuration management, emergency response SCADA functions correspond to fault management, DSM functions primarily cover financial management, and AM/FM/GIS functions correspond to resource management.

Power distribution management operations and their implementation are current topics of research. The realization of these performance gains requires a vastly expanded capability to monitor and control the entire power network. The relative cost-effectiveness of proposed control, communication, and computing architectures (3C) must be assessed before costly hardware implementations are started. As in IVHS, the objective comparison of the various architectures requires the existence of a uniform software framework in which these architectures can be specified, simulated, and evaluated.

Once the distribution network architecture and the appropriate services are selected, a Distribution Network Management System is needed to manage the actual system. This application needs to monitor and control large number of points with different ownerships and different manufacturer peculiarities, spreading over wide geographic areas in the power distribution system.

3.2.1 Entities in the Distribution Network

In this section we present a summary of power distribution system entities and their basic characteristics.

The purpose of the power distribution network is to distribute power from a generator to the loads, i.e., customers. How the power flows through the distribution network is dictated by the objects it has to flow through, the physical laws governing them, and the load demands.

For a distribution network, the following assumptions are made:

- All voltages and currents are sinusoidal;
- ω, i.e, the frequency is fixed;
- power is delivered to a main distribution center, and then, it is radially distributed through feeders and laterals to various customers; and
- the transients of the distribution network are not relevant.

In a power distribution system, exact state information about every point in the network is generally not be available. Some, hopefully sufficient amount of measurement data is used to calculate or estimate the complete state of the network. The state of the system at a point of the distribution network is given by a proper subset of the following power parameters:

- V_{max}, the amplitude of the Voltage;
- I_{max}, the amplitude of the Current;
- ϕ, the phase shift between Voltage and Current, where ϕ is positive if current is lagging;
- P, the amplitude of the real (or active) power flowing radially towards ground, measured in Watts, (W);
- Q, the amplitude of the reactive (or complex) power flowing radially towards ground, measured in VoltAmperes Reactive, (VAR); and
- $R + jX$, the impedance looking radially towards ground.

Some of the relationships between these parameters are summarized below:

$$\begin{array}{lcl} V(t) & = & V_{max}\cos(\omega t) \\ I(t) & = & I_{max}\cos(\omega t-\phi) \\ p(t) & = & \frac{V_{max}I_{max}}{2}(\cos(\phi)+\cos(2\omega t-\phi)) \\ & = & P(1+\cos(2\omega t))-Q\sin(2\omega t) \\ I_{max} & = & V_{max}/\sqrt{R^2+X^2}=2\sqrt{P^2+Q^2}/V_{max} \\ P & = & \frac{V_{max}I_{max}\cos(\phi)}{2} \\ Q & = & \frac{V_{max}I_{max}\sin(\phi)}{2} \\ \phi & = & \arctan X/R = \arctan Q/P \\ \cos(\phi) & = & \frac{R}{\sqrt{R^2+X^2}} \end{array} \tag{3.1}$$

The value of $\cos(\phi)$ is referred to as the *Power Factor.* In power systems, it is desirable to have negative ϕ values. If current is leading the voltage, constant voltage levels at a load can be sustained over larger power consumption intervals, without altering voltage levels at the generators [43].

The distribution elements are best organized into three categories:

- Live Elements—all entities with power flowing through them;
- Relay Elements—all entities that can perform operations on the Live Elements and/or on the network in general; and
- Communication Elements—all entities that model the communication mechanism between the management centers and live and relay elements.

The relay and communication elements are particular to the control and communication architectures.

The live elements have the following common characteristics:

- At their input and output ports they have a value for: P, Q, V, I, R, X, and ϕ;

  Some of these values are measured, others are derived, some, such as the impedance, can be known from the physical model. Some output ports may be grounded.
- they have a transfer function that maps their inputs to outputs;
- the input ports are generally connected to output ports of other devices; and
- they have a maximum current and maximum voltage they can tolerate.

  The duration, possibly zero, these values can be tolerated and the possible results of exceeding these values must be modeled.

Live Elements can be further categorized based on their number of terminals:

- Inline elements, such as lines and serial capacitors, are two terminal devices;
- Terminal elements, such as loads and generators, have one of their terminals grounded; and
- Connectors, used to create branches and to reconfigure the distribution network, are three-terminal devices.

We now briefly discuss each category. A hierarchy of live elements is given in Figure 3.2. The details of these elements are discussed in [44].

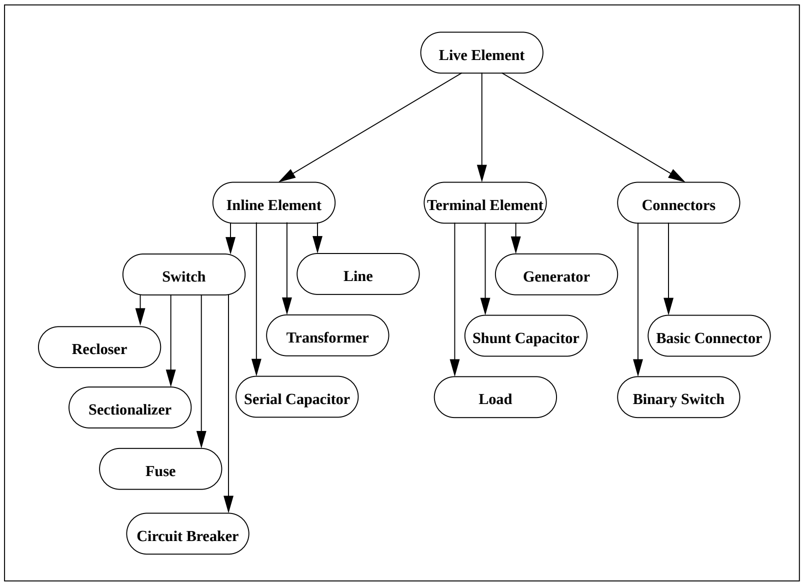

Figure 3.2: Hierarchy of Live Elements in Power Distribution

Inline Elements

Inline elements include lines, serial capacitors, transformers, and switches.

A **Line** is the basic copper wire with an associated length and impedance $R + jX$.

Serial Capacitors are used to change the power factor of the system.

Transformers are used to step up/down the voltage along the network. Common characteristics of single phase transformers include their gain, their input impedance, and their output impedance. The gain is used to regulate voltage levels. Based on transformer type, it may be set manually, remotely, or automatically by the transformer. The models for single-phase transformers are straightforward. More complex models are needed for different three phase transformers and their possible connection types [48].

A **Switch** opens or closes a circuit and changes the network configuration. The time, manner, and reason these operations are performed change based on switch type. Some switch types are:

- **Fuse**;

  A fuse "burns", i.e., opens, when current through it exceeds a certain value for a certain duration. A fuse can be opened manually. If it burns, it needs to be replaced manually.

  Sectionalizer;

  A sectionalizer can be opened and closed manually when the circuit is not live. Opening a sectionalizer during power flow may result in arching. A sectionalizer can not be controlled remotely.

  Circuit breaker; and

  A circuit breaker opens if current through it exceeds a certain value for a specified time. A circuit breaker can be opened and closed manually. Based on breaker type, it can be operated remotely. Remote operation is achieved with a relay. Circuit breakers can be operated when the circuit is live; arching is either suppressed by plasma, or it is avoided by opening the circuit during a zero crossing of the current.

  Recloser.

  A recloser is like a circuit breaker, however, it opens and closes itself. Reclosure time and algorithm may be fixed or it may be programmable.

Terminal Elements

Terminal elements have one free terminal—their second terminal is grounded. They include shunt capacitors, generators, and loads.

Shunt Capacitors are used to change the power factor of the system. Shunt capacitors are switched in and out, based on power demand.

Generators are the source of power. Their characteristics include P_{max}, Q_{max}, V generated, fuel type, cost, etc.

A **Load** is an aggregated entity. An entire lateral circuit, an apartment complex, a factory, or a toaster oven can all be modeled as loads.

Loads come in various flavors. Some loads are linear and can be represented as $R+jX$. For most loads, the value $R+jX$ is time dependent. Other nonlinear loads may be directly dependent on the voltage levels or on the price of power. Loads may have priorities. Different control strategies may be used to manage loads with various priority levels. To summarize all these characteristics a load mixture can be represented by a function $f(p,c,x,t)$, where p represents the load priority mixture, c represents the control picked for each priority level, x represents the dependencies of the load value on other parameters, and t represents the time dependency,

Connectors

Connectors are three terminal devices.

In a power distribution network, arbitrary number of branchings are possible at a given point. However, each branch effects the power flow directly and has to be kept track of by numerical algorithms [46] that solve for power parameters. A **Basic Connector** entity is used to model a branch. A basic connector has one input and two outputs.

Binary Switches are used to change the network configuration. A binary switch has two inputs and one output. At any given time, at most one input terminal is connected to the output. Special algorithms are needed to ensure that particular binary switch settings do not violate the radial network assumption.

3.2.2 Problem Nature and Requirements

The users', management, deployment, and software system requirements of power distribution systems are similar to that of highway automation systems and are not repeated here.

However, one distinguishing characteristic of the power system needs emphasis: a change in power demand at one point of the distribution network may have immediate effects on the entire distribution network.

Unlike the highway system, the distribution network does not exhibit any inherent characteristics that simplify distributed simulation. Both the flow of power as well as certain management decisions depend on detailed information deep into the distribution network.

A hierarchical control architecture is needed to simplify both the system management and the system simulation. Figure 3.3 suggests one such possible hierarchy.

Since any aggregation of loads is still a load, at the main feeder level laterals may be treated as loads. Such a lateral load can be viewed as an "agent" to the rest of the system. The feeder layer "manager" can then send management requests to these agents. This manger-agent relationship can be extended recursively as the system requirements warrant.

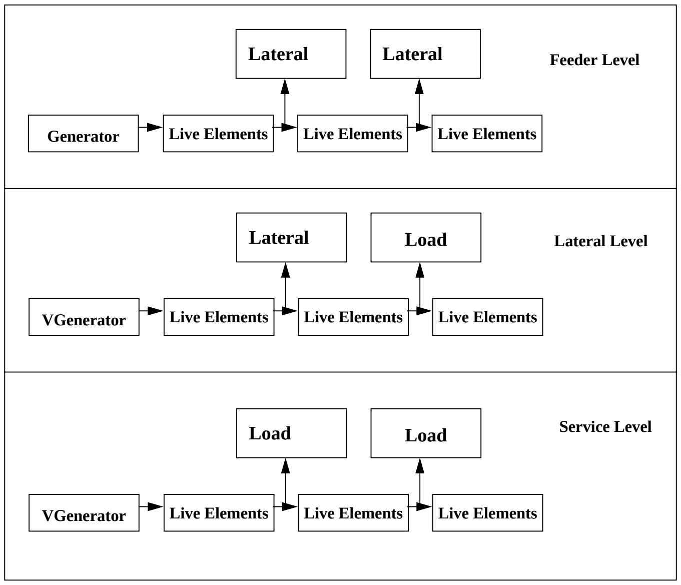

Figure 3.3: Possible Network Hierarchy

3.3 Data Network Management Systems

In the preface of his book, Bapat observes, "The growth of networking technologies in the past few years has been explosive. Digital communication networks, once limited to moving data between computers in universities and research laboratories, now permeate almost everywhere. The existing infrastructure of the worldwide telephone network, assiduously constructed over the last hundred years to provide voice communication, is undergoing a radical transformation as digital technologies are being deployed to provide improved quality and advanced services. In the world tomorrow, it is not inconceivable that a combination of underlying transport technologies —wireless, analog wireline, and digital fiber among them— will combine to form a worldwide network interconnecting offices, factories, residences, commercial establishments, transportation vehicles, and many other entities. Technical reachability will impose almost no limit; actual reachability will only be constrained by policy considerations such as security, confidentiality, service deployment costs, and service usage costs.

Clearly, the task of constructing communication networks of this magnitude is formidable. As with all complex systems, the best way to approach the construction task in a manner that is manageable and understandable is to decompose the system and create a model for it" [52].

Information resources supported by multi-vendor, multi-technology international networks are a strategic and indispensable resource in today's corporations and government agencies. These networks form the backbone of the information management systems of these agencies and enable them to communicate between their various operation cites. Automated bank tellers, credit card authorization systems, the ever-so-indispensable electronic mail and the World Wide Web all depend on communication networks. The need to maintain these networks with a high degree of reliability coupled with the need to adapt them to ever changing requirements is central to their users' success and strategic competitiveness.

The heterogeneity of the network equipment and the complexity of the services they support makes the maintenance and continual modification of the networks a daunting task. Network Management Systems (NMS) are needed to configure, secure, control, monitor, analyze, document, diagnose, and repair any of the components that constitute the network. A NMS must provide complete and efficient network management capabilities that are readily accessible to their users.

A plethora of services are available that provide varying degrees of latency and throughput for data transmission, see Figure 3.4. Some of these services, such as Asynchronous Transmission

Mode (ATM) networks, Synchronous Optical Networks (SONET), and Switched Multimegabit Data Service (SMDS) can actually be configured to meet specific service needs.

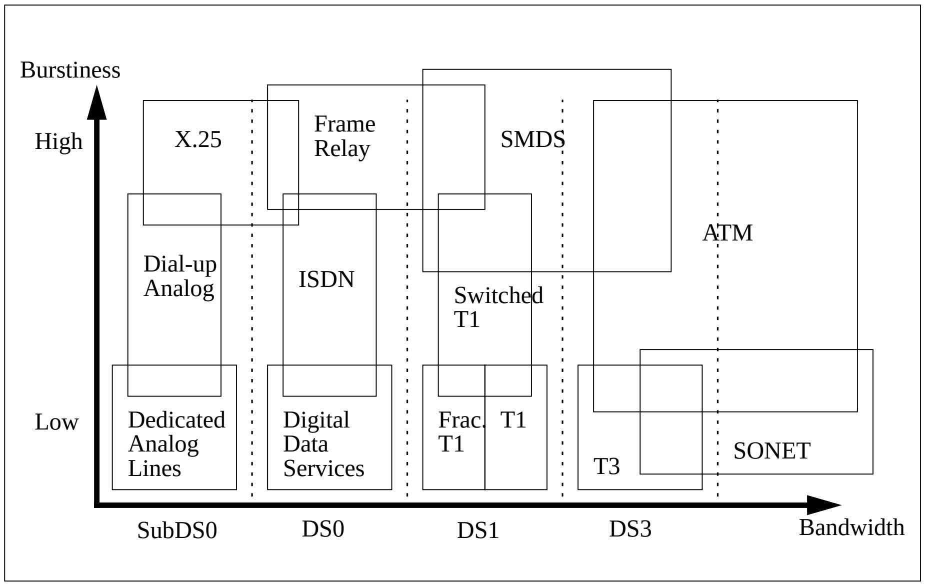

Figure 3.4: Available Network Services

3.3.1 Standards

To create some uniformity within which workable network management solutions can be implemented, a number of standard organizations have come into existence. The two main organizations are the Internet Activities Board (IAB) and the International Standard Organization's OSI/NM Forum.

While there is disagreement about the direction of these network management standards, the general assessment is that IAB's Internet protocol suit will provide the short-term solution whereas OSI will eventually supersede it to become a broadly accepted and widely applicable standard solution [65].

Internet

The Internet suite of protocols were initially sponsored by the U.S. D.O.D. and D.A.R.P.A., and grew out of ARPANET, which at the time connected a few dozen computer systems. Rose observes: "The best term to use when describing the Internet protocols is *focused*. There was a problem to solve, that of allowing a collection of heterogeneous computers and networks to communicate [...] The Internet researchers made open systems a reality by limiting the problem, gauging the technology, and, by and large, making a set of well thought out engineering decisions" [65].

The Simple Network Management Protocol (SNMP) [55, 54] is now maintained by the Internet Activities Board and has quickly become a de facto industry standard for managing local area networks. SNMP provides an application framework which runs on top of Transmission Control Protocol/Internet Protocol (TCP/ IP) [56], and User Datagram Protocol (UDP) [64].

OSI

Open Systems Interconnection/Network Management Forum is responsible for the development of a broad set of network management standards. Unlike the Internet suite of protocols, these standards are aimed at a very wide spectrum of applications.

The OSI standards include:

- The seven layer decomposition of the communication architecture [66];

  These layers (physical, data link, network, transport, session, presentation, and application) provide a widely accepted decomposition of the communication architecture.

- The Common Management Information Service (CMIS) definitions [70];

  The definitions include the agent-manager roles for OSI network management elements and a set of management operations such as get, set, create, delete, event-report, and generic actions.

- The Common Management Information Protocol (CMIP) communications standard [71];

  CMIP is the OSI counterpart of SNMP and implements the services defined by CMIS.

- The seven functional categories of Section 2.8 [67]; and

  The network management interpretation of these functional categories are discussed below.

- A standard set of managed object definitions [68];

  A set of "managed objects", such as network, circuit, service, facility, and equipment, are defined. These objects are organized in a class hierarchy. Their attributes, basic operations, and containment rules are specified. A naming convention, based on the containment hierarchy, is developed.

  Most of the OSI standards have evolved into ISO standards [69, 72, 73].

3.3.2 NMS Functional Categories

The seven functional categories of Section 2.8 correspond to the following NMS functions:

Configuration management provides the necessary functionality to add and remove network objects to the system; to change the layout and connectivity of the network; and to set the configurable parameters of network objects.

Fault management functions are responsible for monitoring and maintaining the network's health and for providing graceful fault recovery.

Performance mangement functions ensure the productive operation of the network. They measure, fine-tune, and optimize network performance.

Access and security management functions maintain the privacy and security of the user data. Since parts of the network are leased to different customers, the concepts of user, user groups, and user access levels are essential to a NMS.

Financial management functions track the network use and bill its users.

Resource Management functions provide an inventory of the hardware in the system and track their maintenance schedules.

Planning and Design Management is the process of deciding the future of a network.

3.3.3 Suppliers

Highway automation and power distribution automation, discussed in the previous sections, are emerging technologies. Network management systems, on the other hand, have seen wide use over the last decade.

Many commercial NMSs exist. Some of these applications, such as Bell Canada's MegaStream, GEC Plessy's NMCS, and Paradyne's NetCare, are element managers targeted at managing specific hardware components.

IBM's NetView, HP's OpenView, Sun's SunNet, and Novell's Netware support the SNMP protocol.

IBM's Netview and HP's OpenView claim CMIP compliance. British Telecom's Concert and AT&T's Accumaster were products aimed at supporting OSI standards.

The experiences the author gained when developing network management systems laid some of the groundwork that lead to Object Management Systems [59, 60, 61, 62].

3.3.4 The Needs

Functional Gaps

The functionality of existing NMS is mostly limited to configuration and fault management. Fault management still greatly depends on operator assistance and intuition.

Performance management generally relies on the existence of a number of predefined network configurations. These configurations are used based on the time of the day or load threshold crossings. Detailed information on the network load, actual packet travel times, and packet loss rates is not available.

Financial and resource management applications are not well integrated with other NMS functions. Independent applications are used to perform these operations. The billing of Internet use is an active research topic [58].

Most network planning tools are experimental [63, 38].

Software Tools

All NMS use a similar software architecture, see Figure 3.5. A logical object model is used to capture the elements in the network. The state of the network, derived from the logical model, is stored in a persistent storage medium, usually a RDBMS. A Management Information Base (MIB) is used to interface between the network, the persistence storage medium, and the user interface.

Most of the NMS functionality is implemented in the MIB.

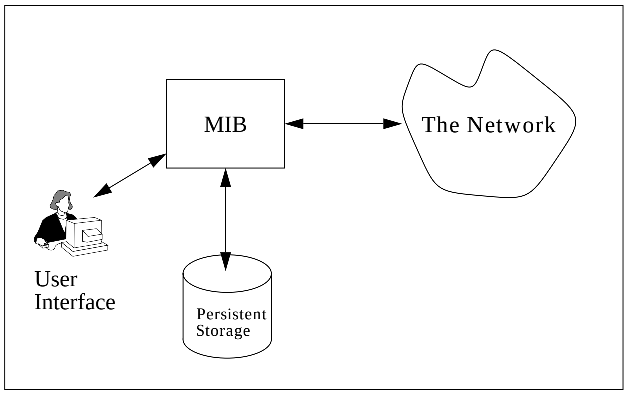

Figure 3.5: NMS Software Architecture

In Chapter 1, we stated that there is a gap between the interfaces provided by the database management systems and programming languages on the one hand, and the specification and implementation constructs required to build engineering applications on the other hand.

There are no good commercial tools that fill this gap. As a result, in a typical NMS development project, the major portion of the resources are spent on tedious implementation tasks and the actual NMS functionality is shortchanged.

Object Management Systems fill this gap and help focus project resources on the design and implementation of actual domain specific application functionalities.

Chapter 4

Object Management Systems

This chapter describes the OMS methodology. The methodology consists of four stages: 1) Domain Customization, 2) System Architecture 3) Application Development, 4) System Test.

The specification and implementation of an *Object Model* precedes these stages. The domain customizers and system architects are expected either to select an existing Object Model or to develop one that suits their application needs. The first two stages customize the Object Model based on the application needs and deliver a *Customized Object Model*, or *Customized OM*. The application developers further extend it to an *OMS Application*. The overall process is referred to as *OMS*.

In this chapter we first describe constructs that should be present in an Object Model. We distinguish between the data and process model of the Object Model. The data model is used to specify entities, their input, output, and state attributes; their methods; and their constraints. The process model is used to specify how a collection of objects evolve based on their state and input-output interconnections.

Next we describe the stages of the OMS process, and how the Object Model evolves in these stages.

Then, we look at the system functional categories and discuss which portions of the functional categories are satisfied by the Object Model in each process stage.

Finally we give evaluation criteria and a list of trade-offs for systems built with this approach.

The Object Model and the OMS process borrow concepts from system theory; object oriented analysis, design, and programming methodologies; Petri-nets, data-flow diagrams, and state machine representations. Our contribution is in selecting the right concepts and in extending and organizing them in an austere model and an end-to-end development process. The end result, the Object Management System, addresses the needs of a restricted, yet, large class of complex software systems.

4.1 Semantic Data Model

In this section we describe data model constructs that should be present in the Object Model.

Since our goal is to implement software systems for specification and control of systems, our model borrows concepts from both the object oriented paradigm and system theory. In classical system theory objects have state, inputs, outputs, and maps that define input to state and (input, state) to output behavior. These maps can be time or event driven, based on the nature of the system, and there is a domain of validity for the model. These objects are then interconnected in a static configuration. In object oriented software methodology, objects are organized in a class hierarchy. Classes consist of attributes and methods, and correspond to entity descriptions. Instances of these classes are created, interconnected, modified, and deleted as needed.

We combine these concepts. As in the object oriented methodology, we specify and implement entities in a class hierarchy that supports monotonic inheritance, functional polymorphism, and dynamic binding. As in system theory we have an input-output representation of objects.

In OMS, entities consist of the following components:

- name;
- state attributes;
- input attributes;
- output attributes;
- a set of methods from (input, state) to (state, output);
- a set of constraints that state, input and output attributes have to satisfy.

Figure 4.1 shows the canonical representation of an entity. The concept of an entity or a class is quite universal and was discussed in Section 2.4.1. We use the two terms class and entity interchangeably. Entities are logical representations of their physical counterparts. Vehicles, Lanes, RS232 Ports, Cats are all entities. Based on application requirements entities abstract and encapsulate the relevant characteristics of their physical counterparts.

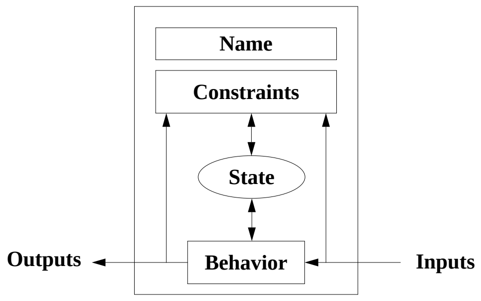

Figure 4.1: Basic Entity

Entities are organized in a class hierarchy. A subclass inherits all components of its superclass. It can add constraints, methods, and input, output, and state attributes. It can also change the methods, and add new inputs and outputs to the methods.

Entities represent the meta-data and their specification is assumed to be static. Instances are the realizations of entities and are created and deleted as needed. All instances of a given entity constitute the extent of that entity.

In the following subsections, we describe the components of entities in more detail. We use the term object when we want to refer to both entities and instances.

4.1.1 Object Identifier or Distinguished Name (DN)

Every object, and every component of an object must have a unique identifier. Every entity, every instance of an entity, every attribute of an instance of an entity, etc. has a Distinguished Name.

A particular naming convention has to be selected in the domain customization and system architecture stages that supports the use of RDNs[1].

4.1.2 Attributes

Attributes can be arbitrarily complex, they can be atomic or list valued, they can contain DN values. Attributes only exist as components of an object. Entities define the name and the domain of each attribute. Instances assign them a particular value.

List attributes provide the constructs needed to treat each list element as an independent attribute. The number of elements in a list attribute is part of the list attribute value.

[1]DNs and RDNs were discussed in 2.4.4.

The particular object model implementation should provide a basic set of attribute domain definitions and supply a mechanism for defining new ones.

All attributes must provide an `Update` method for setting their value.

State Attributes

Every object has an internal state, given by a list of attributes. The weight of a person, the speed of a car are such state attributes.

The state is entirely internal to an object and can not be accessed by other objects.

If the DN of an instance is stored as a state attribute, its value cannot be used to access that instance directly. A connection with that instance must be created explicitly as discussed in Section 4.2.1.

Input and Output Attributes

The inputs and outputs represent the information shared by objects. This information may semantically correspond to shared state or the control one object exerts upon another.

Each input can be connected to at most one output. An output can be connected to multiple inputs. An input takes on the value of the output it is connected to. The domain of an output attribute must be a subset of the domain of the input attribute to which it is connected.

We emphasize again that list attributes must provide the necessary mechanisms that treat each element in the list as a separate input/output. Each element in an input list attribute can be connected to a different output.

Consider a `Statistic` entity with a list of input `Salaries` and an `AverageSalary` output. The number of salaried employees in the department is variable; for each employee an independent `Salary` input is needed that can connect to that employee's salary output.

The input/output attributes have the following components:

Value	`output-DN`	For Inputs only: The DN of the output to which an input is connected.
	`input-list-DN`	For Outputs only: The list of input DNs to which an output is connected.
Methods	`Connect`	used to establish connections.
	`Disconnect`	used to terminate connections.
	`Enable`	accessible to methods used to enable them.

The input-output attribute methods will be discussed in Section 4.2.1.

4.1.3 Methods

The behavior of objects is given by a set of methods. Each method specifies a subset of the named input attributes of an entity as its input and a subset of the named outputs as its output. Each method defines a subset of its inputs as enabling inputs. This specification is called the *signature* of a method.

The signature of a method, and the map from input to output values are specified as part of the entity.

Since "an enabling input is enabled" is not ideal terminology, we borrow from Petri-Nets and say "a token is placed on an enabling input" to mean "an enabling input is enabled".

Methods are executed in instances. When all enabling inputs of a method in a given instance have a token, the method accesses the values of the input and state attributes of the instance, derives the new state and output values, and sets the state and output attributes of the instance to the derived values. A method can choose to place tokens on a different subset of its outputs in each execution.

The language used to specify the maps from inputs to outputs can be arbitrary; the domain customization and system architecture stages may choose to specialize it.

4.1.4 Constraints

Constraints are used to specify both the valid states and the state transitions of instances. Like methods, they are specified as part of the entity and executed in instances. They can be thought of as methods that map input, output, and state attribute values to `TRUE` or `FALSE`. State constraints restrict the universe of discourse of state, input, and output attributes. State transition constraints restrict the value assignments of methods.

Methods specify what an entity does, constraints specify what it should not do. Constraints are usually specified during the domain customization and system architecture stages. The application developers then have the responsibility of implementing methods that obey the constraints.

Note that since input/output attributes store the DN of the output/input to which they are connected as part of their value, constraints can be used to restrict possible input-output connections.

The Object Model may or may not provide an exception mechanism. The decision about what to do when a constraint is violated is delegated to domain customization and system architecture stages.

Example Constraints

Example Constraints are:

- State constraints;

  These are constraints on the value of an attribute or a set of attributes.

  1) Let `Month` be an entity with an integer attribute `numDays` corresponding to the number of days in a month. There is no 32nd day in any month.

  2) Let `Month` be an entity with an integer attribute `numDays` corresponding to the number of days in a month, and a string attribute `name` corresponding to the name of the month. 31st of February does not exist.

  3) Let `Employee` be an entity with a DN attribute `Manager`. The value of the DN can only belong to a manager instance and not to a month.

- Cardinality and participation, or connection, constraints;

  These are constraints on the cardinality of list attributes.

  1) Let `myManager` be a List-DN attribute of an `Employee`. Each employee must have at least one manager.

  2) Let `myEmployees` be a List-DN attribute of a `Manager`. Each manager must have at least three and at most ten employees.

- State transition, or behavior, constraints;

  1) Qualitative: Let `Car` have two attributes `EngineState` and `Speed`. Speed can not change if the engine is turned off.

  2) Quantitative: Let `Car` have an attribute `position`. A car can not move ahead more than fifty meters within one transition.

4.2 Semantic Process Model

The data model provides the constructs to define entities and their components. The process model addresses the following issues: 1) where are all the objects located; 2) how are instances created, deleted, connected, and disconnected; 3) how are outputs propagated; and 4) how are the initial inputs provided.

Objects are grouped into domains. An instance can be part of one and only one domain, whereas an entity can be part of several domains.

The Object Model provides a special object *OMSEngine* that is in charge of managing a domain. Each OMSEngine manages the evolution of the objects in its *domain* and provides methods to create, delete, connect and disconnect them.

The OMSEngine has the following components:

Inputs	`Create`	List attribute. Consists of an entity DN and other values required for initialization.
	`Delete`	List attribute. Consists of an instance DN and other values required for termination.
	`Connect`	List attribute. Consists of two attribute DNs, `input-provider` and `output-consumer`.
	`Disconnect`	List attribute. Consists of two attribute DNs, `input-provider` and `output-consumer`.
	`EnableRun`	List Boolean attribute. Used to enable the method `Run`.
Outputs	`Data`	The value of this attribute contains information about the last execution sequence.
Methods	`DoCreate`	Method to create instances.
	`DoDelete`	Method to delete instances.
	`DoConnect`	Method to connect objects.
	`DoDisconnect`	Method to disconnect objects.
	`Run`	Method to carry out execution sequences

We now discuss the process model and the OMSEngine components in more detail.

4.2.1 Object Configuration

The `Connect` and `Disconnect` attributes are used to change the input-output configuration of the system. These attributes contain two attribute DNs `input-provider` and `output-consumer`.

To establish input-output connections, an object must have a `Connect` output attribute connected to the `Connect` input attribute of the OMSEngine[2]. To establish a connection the object sets the values of `input-provider` and `output-consumer` and enables its output. Input-output connections are terminated the same way by using the `Disconnect` attribute.

The `Create` and `Delete` attributes are used to create and delete instances of entities. The create attribute contains an entity DN, `class-name`, the delete attribute contains an instance DN, `instance-name`. Both attributes may contain other data used for initialization or termination of instances.

To create an instance of an entity, an object must have a `Create` output connected to the `Create` input of the OMSEngine. The object sets the `class-name` and enables its output. Instances are deleted the same way by using the `Delete` attribute and providing the `instance-name`.

Comments

`Create` and `Delete` can be implemented as entity methods. An object that wants to create an instance must then have an output attribute connected to the `Create` input attribute of the respective entity[3]. To delete an instance an object must have an output attribute connected to the `Delete` input attribute of the instance.

These methods are expected to implement the necessary bookkeeping required for the creation and deletion. Such bookkeeping includes proper initialization of the state of the instance as part of creation, establishing/terminating the input-output connections of the new/late instance, and notification of the OMSEngine.

Whenever possible, create and delete methods should be implemented in the domain customization and system architecture stages. Note that an object can create an instance of an entity only if it has an output connected to that entity.

An OMS Application consists of many interacting OMSEngines. The Object Model must provide constructs for connecting the inputs and outputs of objects in different domains. This is easily accomplished if the `input-provider` and `output-consumer` are not restricted to be in the

[2] Recall that the `Connect` and `Disconnect` input attributes of the OMSEngine are list attributes. Thus many objects can connect to this input.

[3] Unlike all other input attributes, the `Create` input attribute is part of the entity itself.

domain of the same OMSEngine, and the OMSEngine has the ability to locate objects in other domains.

4.2.2 The Method `Run`

Based on our definition in Section 4.1.3, if a method executes, it may result in the execution of other methods until no more executions are possible. We call this an *execution sequence*.

In its most general form the execution sequence of a method may not be unique, can be cyclic, and infinite. Tokens may remain on inputs after termination. Furthermore, connect and disconnect outputs discussed above may alter the execution path itself. The Object Model must impose restrictions to ensure that execution sequences are unique and finite. A *valid* execution sequence that obeys these restrictions is called a *transition*.

The OMSEngine carries out execution sequences when its `Run` method is invoked. This method determines if and what methods are enabled on the placement of tokens, propagating the output tokens of these methods, connecting and disconnecting inputs and outputs, creating and deleting instances, checking that the execution sequence is valid, checking that the transition satisfied all the constraints, and accepting or rejecting the inputs.

Comments

The following restrictions ensure finite and unique execution. When a method is executed it consumes all its input tokens, simultaneously updates its outputs, and simultaneously places tokens on its outputs. Every input connected to this output is updated and delivered one token simultaneously. If an execution enables several other methods, these are executed in arbitrary sequence. An execution sequence is *valid* if the executed methods had at most one update operation on their inputs (if first one input is updated and then another, this is considered two updates) and no connect and disconnects take place on inputs and outputs of executed methods. At the end of a transition all remaining tokens are removed. Roughly speaking this restricts execution sequences to directed trees.

Tokens are bookkeeping constructs; unconsumed tokens are removed at the end of each transition so as not to carry state information between transitions. State information shared between transitions can be encapsulated within an entity's state, inputs, or outputs.

There are several other ways of ensuring uniqueness. One could define an execution sequence to be valid if no input of a method is modified after it has been executed, generalizing execution sequences to directed acyclic graphs. In this case one also must assume that if the execution of a method enables several other methods, these are executed simultaneously.

The execution sequence that a method generates depends on the internal state and inputs of objects. As such, even if all input-output connections of two systems are identical, the execution sequences may differ. We emphasize that assuring valid execution sequences without further restrictions is not a trivial task. The domain customization and system architecture stages are expected to provide a mechanism to ensure valid execution sequences and, if necessary, provide an exception handling mechanism to deal with execution sequences that are not valid.

One may finish an execution sequence and then check if the constraints are satisfied, or one may require that constraints are satisfied at all times during the execution sequence. If the transition is considered to be an "atomic" operation constraints should be checked at the end of the execution sequence. In this case if several methods of an object are executed within one transition, the object is allowed to temporarily violate its constraints.

Checking for constraints adds a performance penalty to the system. Furthermore, during development stages certain constraints are bound to be violated due to unimplemented behavior. The OMSEngine should provide an option to turn constraint checking off.

4.2.3 Sequencer Entities

Consider an initial collection of objects in the domain of an OMSEngine, with a number of unconnected inputs and outputs. Using the OMSEngine interface, any entity can connect to these inputs, change their values, and place tokens on them, and invoke the `Run` method of the OMSEngine, possibly resulting in a transition.

Next, we need a construct with the ability to specify a sequence of transitions in the domain of an OMSEngine.

The *Sequencer* objects are used to specify sequences of transitions. This object is described in Figure 4.2.

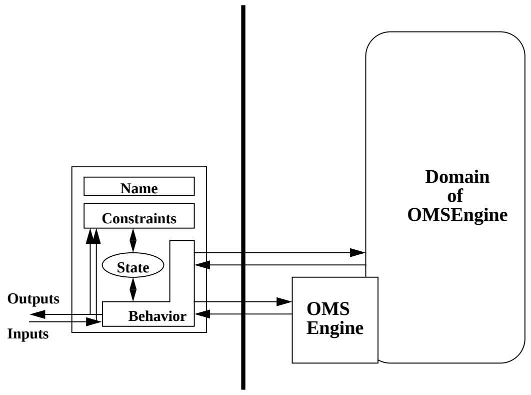

Figure 4.2: A Sequencer Object

A sequencer connects to the `EnableRun` input and `Data` output of an OMSEngine. The domain of the OMSEngine with which a sequencer interacts is called the *subdomain* of the sequencer. The sequencer also connects to a number of input and output attributes in its subdomain. It places tokens on select inputs in its subdomain and invokes the `Run` method of the OMSEngine. This results in a transition and produces a number of outputs. Based on output values, the sequencer determines the next set of tokens to be placed.

Sequencers themselves are in the domain of an OMSEngine and behave like any other entity in that domain. However, their methods do not complete execution until they have executed the proper transitions in their subdomain.

Sequencers are the key to creating hierarchical abstractions. What appears to be a single transition within the domain of one OMSEngine may require several transitions in the domain of another. A scheduling mechanism for time and event driven behavior that uses sequencers and three domains is discussed in Section 4.3.4.

The exact structure of Sequencer methods is chosen in the domain customization and system architecture stages based on application needs. One may choose state machines, programming constructs such as sequences, while-loops, if-then-else constructs, and, if needed, build an exception mechanism. The allowed nesting patterns for the domains and subdomains also depend on the nature of the application.

4.2.4 Other Services

So far we have provided the key ingredients for an Object Model. There are several other services that are needed to build OMS. These services are:

- Persistence;

  The state of an OMS should be stored in some persistent storage medium.

- Query Language;

  Data stored in the persistent storage medium should be accessible through a query language.

- Transactions, checkpointing, versioning;

  These are standard persistent storage constructs. In particular, one should be able to save the state history of instances.

- Multithreading, Concurrency;

  A construct is needed that can enable several, possibly conflicting, execution sequences simultaneously.

- Object locking mechanisms;

  In presence of multithreading, a locking mechanism is needed to guarantee the atomicity of transitions.

- Parallelization;

  It may be possible to parallelize the execution of an execution sequence. In presence of multithreading, the execution of the transitions can be parallelized.

- Distribution Support;

  We have not specified how to access a given OMSEngine. A construct is needed to access OMSEngines independent of their location.

- Object migration;

  Instances should be able to move from the domain of one OMSEngine to another.

- Entity evolution;

  Although we assumed that the entity definitions are static, eventually a mechanism is needed to add, remove, and modify entity definitions as application needs change.

- Verification.

  Whenever possible components of the OMS should be verified. It may be possible to check if the execution sequences designed in the domain customization and system architecture stages are always valid. Or, it may be possible to verify whether a given set of methods always fulfill the entity constraints.

These constructs are expected to be provided either by the Object Model or by the Customized OM based on application needs.

4.3 Extended Constructs

In this section we present further constructs that provide semantic support and simplify system specification. Although they can be composed from the constructs of the previous section, a given Object Model implementation may provide them as atomic constructs.

Virtual attributes, associations, and relations are part of the data model. Time and event driven scheduling and event generation are part of the process model.

4.3.1 Virtual Attributes

Virtual attributes were discussed in 2.4.4. In OMS virtual input attributes can be used to create implicit input-output connections between objects. The `DN-Prefix` of a virtual input attribute is set to the value of another input DN attribute of the entity, its `Scope` is set to the RDN of an output attribute. The value of the `output-DN` of the input attribute is then derived from the `DN-Prefix` and the `Scope`.

Virtual attributes are not used for state or output attributes. Using them as state attributes would implicitly convert state into inputs. Using them as outputs would cause consistency problems if a virtual input were connected to a virtual output.

If virtual attributes are used, the Object Model has to provide an implementation that ensures proper output propagation.

The use of virtual attributes greatly simplifies system specification.

Assume each `Vehicle` has a `Regulator`. Assume this relationship is captured by their respective `myRegulator` and `myVehicle` input DN attributes. Assume the Regulator has two inputs `RSpeed` and `RAcceleration` which should be connected to the `Vehicle` outputs `VSpeed` and `VAcceleration`.

Connecting a regulator instance to a vehicle instance requires 1) Setting the `myRegulator` and `myVehicle` attributes, 2) Connecting the `Speed` and the `Acceleration` attributes. Virtual attributes save the specification of the second step. If the `Regulator`'s `RSpeed` and `RAcceleration` inputs are virtual, with `myVehicle` as their `DN-prefix`, and `VSpeed` and `VAcceleration` as their `Scope`, setting the value of `myVehicle` attribute implicitly and consistently establishes the input-output connections.

4.3.2 Association and List-Association Attributes

In the example above the semantics of the problem requires that to maintain consistency if the `myVehicle` attribute of a Regulator is updateded, the `myRegulator` attribute of the previous and current Vehicle instances are updated as well. A construct like an association discussed in Section 2.4.4 is needed to simplify this operation and to ensure data integrity.

An association consists of two input attributes, one in each participating entity, and ensures the consistency of these attributes. Assume that the `myVehicle` and `myRegulator` attributes in the above example are associations. If a Vehicle "CarX" has "RegulatorY" as its `myRegulator`, then "RegulatorY" is guaranteed to have "CarX" as its `myVehicle`.

The particular association specification and implementation in an Object Model cannot violate the assumption that an input is connected to at most one output. An input can either be implicitly connected to the association, or to an outside output, but not both. We now discuss two possible association semantics.

Directed Association

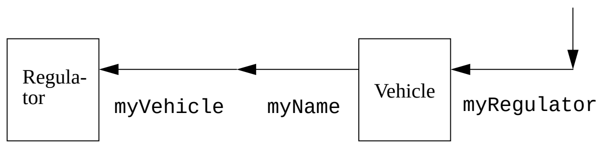

Figure 4.3: Directed Association

In a directed association only one input attribute is implicitly connected to the association, the other input is "open" and can be connected to another output.

The association implicitly adds a DN output attribute to the entity with the open input. This attribute, `myName`, carries the DN of the instance. When the input attribute of this entity is updated, the association reconnects the `myName` output as required.

Symmetric Association

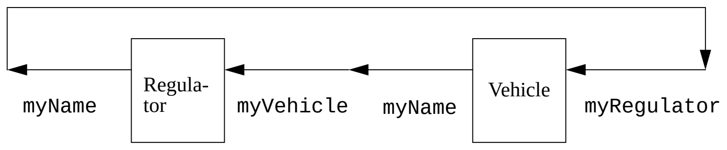

Figure 4.4: Symmetric Association

In a symmetric association both inputs are implicitly connected to the association output. As such, an external mechanism is required to atomically establish/terminate the necessary input-output connections. This mechanism can be provided by the OMSEngine, or it can be implemented as a method in some entity.

4.3.3 Relationships

A relationship is an ordered association between entities. The Vehicles in a Lane, Vehicle adjacent to another Vehicle, and a game consisting of two teams, a referee, a field, and a score are relationships.

The association and list-association constructs of the previous subsection can express ordered on-to-one, one-to-many, and many-to-many relationships. However, they have no semantics and cannot express more complex relationships such as a game.

Relationships are entities. A designated subset of their inputs keep track of the relationship participants. These are association or list-association attributes whose domain is restricted to the DN values of instances participating in the relationship.

At this point we should revisit Figure 4.1 and further categorize the input attributes. Each entity has a list of input attributes that keep track of the relationships it participates in. These are association attributes and are not necessarily used by any method. Each entity has a number of virtual input attributes. These inputs can not be connected to outputs directly, rather they create implicit interconnections. The remaining inputs have no particular semantics.

We now define some example relationships.

Membership

Class managers keep track of the extent, i.e., the members, of an entity.

Views

Views are useful constructs for providing access control to the outputs of an object. The View of an instance may make selected outputs of its source instance accessible via its own outputs.

Input-Output Relations and Switches

Input-Output Relationships (IORs) are used to specify the connection structure of instances participating in the relationship. Their use is similar to that of virtual attributes. They are initialized with a list of instances, and establish the necessary interconnections.

Switches are similar to IORs. However, they accept the output of the participating instances as inputs and implement methods that determine how to propagate them.

Switches enable us to create arbitrary input-output configurations without having to alter the source and target entity specifications. Assume an entity x has three inputs, sometimes inputs one and two may be supplied by entity y, sometime inputs two and three by entity z. Furthermore in future we may introduce entity w, that supplies input one and three.

Composition or Containment

A complex entity can be composed of several other entities, its components. In this case the complex entity contains its components.

A given entity can be contained in at most one entity.

Containment is sometimes used to specify the distinguished name of an instance. However, this is not an option if the container of an instance can change over its lifetime.

Aggregations

Aggregations provide us with the means of creating higher levels of abstractions of a system.

Assume that the `Lane` entity has the output `ContainedVehicles`, a List-DN attribute. Assume that the `Vehicle` entity has the output `speed`. We define the `LaneStatistic` relationship as follows.

`LaneStatistic`

Inputs	`myLane`	An association attribute whose domain is restricted to Lane instance DNs.
	`theVehicles`	A virtual List-DN attribute, with `myLane` as `DN-prefix` and `ContainedVehicles` as `Scope`.
	`theSpeeds`	A virtual List-float attribute, with `theVehicles` as `DN-prefix` and `speed` as `Scope`.
Outputs	`averageSpeed`	A float attribute.
Methods	`Go`	With input `theSpeeds` and output `averageSpeed`. When enabled, calculates and outputs the average speed.

The outputs of the LaneStatistic instances can be the only outputs of a domain. The user of these outputs does not need to know how these statistics are generated. They can be results of a microsimulation, actual physical data, or any other set of statistics.

4.3.4 Time and Event Driven Behavior

The Object Model constructs discussed so far do not enforce any particular implementation of time or event driven behavior. The Object Model should provide constructs for time and event driven scheduling that can be customized and configured by the Customized OM.

Below we provide a possible implementation of time and event driven evolution in three OMSEngine layers.

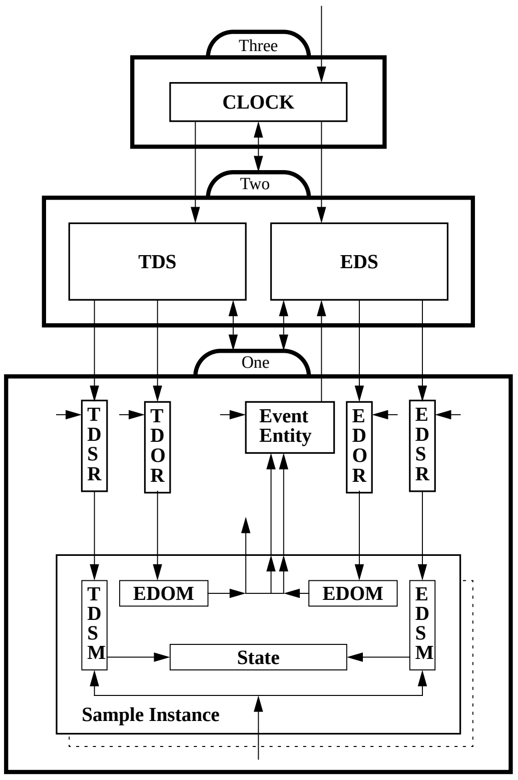

Figure 4.5: Time and Event Driven Evolution Example

OMSEngine One

The domain of OMSEngine One contains the bulk of the system. It has four special relationship entities `TDSR, TDOR, EDSR, EDOR`, an `Event Entity`, and a base class `Base`. All other entities in OMSEngine One monotonically inherit from `Base`.

`TDSR, TDOR, EDSR, EDOR` are bookkeeping Relations. They manage Time/Event Driven State/Output methods of instances. Each relationship entity has the same structure. We describe the TDSR entity:

Inputs	`myInstances`	List-association attribute shared with the `myTDSR` input of instances. Maintains a list of all the instances in the domain.
	`Enabling`	DN attribute. Enabling input of the method `Go`. Its value corresponds to the instance whose corresponding method should be enabled. Its value can be set to `ALL` denoting all instances.
Outputs	`EnableList`	List-Boolean attribute. Each element of this attribute is connected to the `EnableTDSM` input of an instance.
Methods	`Go`	With `Enabling` as its enabling input and `EnableList` as its output. When enabled it places a token on the output connected to the instance specified by its input value.

The `Event Entity` has the following components:

Inputs	`Instances`	List-association attribute shared with `myEventEntity` input of instances. Maintains a list of all the entities in the domain.
	`CreateList`	Virtual List-Boolean attribute, with `Instances` as its `DN-prefix` and `Create` as its `Scope`. Setting this value to true, denotes a create request from the source instance.
	`TargetList`	Virtual List-DN attribute, with `Instances` as its `DN-prefix` and `Target` as its `Scope`. Its value specifies the target of the Event instance to be created.
	`Enabling`	Boolean attribute. The enabling input of the method `Go`.
Outputs	`EventTargetList`	List-DN attribute. Contains the list of instances with a newly created Event.
Methods	`Go`	A single method with `CreateList`, `TargetList`, and `Enable` as inputs, and `EventTargetList` as output.

The method `Go` implements the following functionality:

1) For each `CreateList` element set to true, it creates an `Event`;

2) Ensures that each newly created event enters a relationship with its target instance, by connecting the input attribute `myTarget` of the Event instance and the input attribute `myEvent` of the target instance;

3) Sets `EventTargetList` to the DNs of all instances with a newly created Event.

The `Event` instances have the following components.

Inputs	`myTarget`	Association shared with the `myEvents` attribute of an instance.
Outputs	`mySource`	DN attribute. Its value is set at creation to the source of the event.
	`type`	Integer attribute. Identifys the event type.

The `Base` entity has the following components:

Inputs	`myEventEntity`	Association-attribute shared with the `myInstances` input of the `Event Entity`. Set to the DN of the Event Entity.
	`myTDSR`	Association-attribute shared with the `myInstances` input of a TDSR. Set to the DN of the `TDSR` instance.
	`myTDOR`	Association-attribute shared with the `myInstances` input of a TDOR. Set to the DN of the `TDOR` instance.
	`myEDSR`	Association-attribute shared with the `myInstances` input of a EDSR. Set to the DN of the `EDSR` instance.
	`myEDOR`	Association-attribute shared with the `myInstances` input of a EDOR. Set to the DN of the `EDOR` instance.

	EnableTDSM	Virtual attribute, with myTDSR as DN-prefix, and EnableList as its Scope. Enabling input for the Method TDSM.
	EnableTDOM	Virtual attribute, with myTDOR as DN-prefix, and EnableList as its Scope. Enabling input for the Method TDOM.
	EnableEDSM	Virtual attribute, with myEDSR as DN-prefix, and EnableList as its Scope. Enabling input for the Method EDSM.
	EnableEDOM	Virtual attribute, with myEDOR as DN-prefix, and EnableList as its Scope. Enabling input for the Method EDOM.
	myEvents	List-association attribute shared with myTarget input of Events. Contains Event instance DNs.
	eventTypes	Virtual List-integer attribute with myEvents as its DN-prefix and type as its Scope.
Outputs	Create	Boolean attribute. Connected to the CreateList input of the Event Entity.
	Target	DN-attribute. Connected to the TargetList input of the Event Entity.
Methods	TDSM	With EnableTDSM its enabling input. Represents time driven evolution. It is a method from Inputs to State.
	TDOM	With EnableTDOM its enabling input. Represents time driven evolution. It is a method from (EnableTDOM, State) to Outputs. In particular it is connected to the Create and Target outputs.
	EDSM	With EnableTDSM, myEvents, eventTypes as input. Represents event driven evolution. It is a method from Inputs to State.
	EDOM	With EnableTDOM as its enabling input. Represents time driven evolution. It is a method from (EnableEDOM, State) to Outputs. In particular it is connected to the Create and Target outputs.

The inputs of the domain of OMSEngine One are the Enabling inputs of the four relationship entities. The output is the EventTargetList output of the Event Entity.

OMSEngine Two

It consists of two Sequencing entities TDS, EDS, (Time and Event Driven Schedulers respectively) both treating the first OMSEngine as their subdomain.

The TDS has the following components:

Subdomain	OMSEngine One	
Inputs	EnableTDS	Boolean attribute. Enabling input of the method Go.
Subdomain Outputs	EnableTDSR	DN attribute. Connected to the Enabling input of the TDSR instance.
	EnableTDOR	DN attribute. Connected to the Enabling input of the TDOR instance.
Methods	Go	Has EnableTDS as its enabling input.

The Method Go has the following functionality:

1) Sets the EnableTDSR to “ALL” and places a token on it;

2) Upon completion of this transition, it sets the EnableTDOR to“ALL” and places a token on it;

3) Upon completion of this transition the method execution terminates.

The EDS has the following components:

Subdomain	OMSEngine One	
Inputs	EnableEDS	Boolean attribute. Enabling input of the method Go.
Subdomain Inputs	EventTargetList	List-DN attribute. Connected to EventTargetList of EventEntity.

Subdomain Outputs	`EnableEDSR`	DN attribute. Connected to the `Enabling` input of the EDSR instance.
	`EnableEDOR`	DN attribute connected to the `Enabling` input of the EDOR instance.
	`EnableEventEntity`	Boolean attribute. Connected to the `Enabling` input of the `Event Entity`.
Methods	`Go`	Has `EnableTDS` as its enabling input.

The method `Go` has the following functionality:

1) It enables the `Event Entity`.

2) It reads the `EventTargetList` output of the `Event Entity` and for each element in this list:

a) It sets `EnableEDSR` to the value of the element, and enables `EnableEDSR`.

b) Upon completion of this transition, it sets `EnableEDOR` to the value of the element, and enables `EnableEDOR`.

4) It repeats the first two steps till the `Event Entity` outputs an empty `EventTargetList`.

5) Upon completion of this transition does not generate any further tokens.

The inputs of the domain of OMSEngineTwo are the `EnableTDS` and `EnableEDS` inputs of the `TDS` and `EDS` instances. OMSEngineTwo has no outputs.

OMSEngine Three

The third OMSEngine consists of one Sequencing entity, with a single method. When enabled with an integer value n, this method goes in a loop of length n, enabling in each loop the time and event driven scheduler of the second OMSEngine.

That's the overall time and event driven model.

In this example, if the Event Entity keeps creating events the EDS executes infinite number of transitions in its subdomain. If this is not an acceptable behavior, one can restrict the number of transitions the EDS executes when enabled, or specify a mechanism for termination after some number of transitions.

Other applications may have entities with different time scales of evolution, or they may require several iterations until the time driven evolution from all inputs to outputs may reach a steady state. The number of such iterations may be state dependent. Again, appropriate sequencer entities can be specified in the domain of OMSEngine Two to implement the desired behavior.

4.3.5 Event Generation

Using these constructs one can represent event generation and propagation.

Events are generated only if some object requests their creation. For event generation one has to specify the proper entities that decide when to create the event.

Events will be propagated if some object chooses to propagate them. For event propagation one has to specify the proper entities that have the necessary information to decide how and to which objects to propagate the events.

Finally, if an object needs to be notified of the creation of an event, it has to enter into a relationship with the source for that event. If an object wants to be notified of events that meet specific conditions only, then a filter entity has to be specified that implements these conditions. The filter can then enter a relationship with the source of the events and propagate the event to the object if the conditions are met.

Domain Customization	System Analysts, Domain Experts
System Architecture	System Architects
Application Development, unit testing	Development Engineers
System Integration and System Test	Testers and All other members

Table 4.1: Stages and Key Players in OMS

System Support	Maintenance Team
OMS Operation	Day-to-day system users
System Supervision	Managers, Supervisors
Physical System Operation	Field Operators

Table 4.2: Key OMS Users

4.4 OMS Process Phases

In this section we describe the different stages of specification and implementation of an Object Management System and what each stage is expected to deliver. Our discussion emphasizes off-line OMSs targeted at simulation and evaluation.

The OMS process consists of four major stages. These stages and the key players of each stage are summarized in Table 4.1. Project management is beyond the scope of this thesis.

The domain customizer and system architect work closely together. They provide an Object Model specification and implementation that provides the constructs discussed earlier in this chapter.

Using the Object Model the domain customizer specifies all the entities in the system, their inputs, outputs, relationships, constraints, and method interfaces. The system architect organizes the entity specifications in a hierarchy of OMSEngines. Together they design the transition structure. The system architect is also responsible for the specification and implementation of other constructs such as exception handling, distribution, concurrency, etc., as the application needs dictate. The deliverable of these two stages is the *Customized OM*. Unlike in other methodologies, the Customized OM has an implementation, i.e., it is a running system with skeletal functionality.

The application developers implement entity behaviors, to meet the constraints specified by the domain customizers. They specialize the base class entities provided by the first stage and extend the customized OMS and to an *OMS Application*.

System test stage tests whether the OMS Application runs without violating any constraints for allowed input patterns. This stage is not discussed in this thesis.

The OMS process is not strictly sequential. Although the first two stages have to deliver a Customized OM before application development starts, this deliverable is not cast in stone. As application development progresses, the customized OMS evolves with the emerging needs of the final application.

The OMS-based management system has several types of users summarized in Table 4.2.

In the following subsections we discuss the activities that take place in each stage, their deliverables, and the users of the OMS.

4.4.1 Domain Customization

The domain customizer specifies the overall data and process model.

Several methodologies are available for specifying the data model of a system. The domain customizer picks his/her favorite and starts the OMS process by identifying the entities in the system, their state, input, and output attributes, their relationships, their constraints and their expected behavior.

The domain customizer decides how far these entities are specified and implemented as part of the Customized OM and how the OMS Application will specialize them for complete implementation.

Once a first cut of the data model is available, the process model needs to be examined. For each entity several key questions should be answered:

- Identify cause of creation and deletion;
- Identify relationship evolution;
- Identify time scale of evolution;
- Specify constraints;
- Identify modes of operation;
- Ensure satisfaction of functional requirements and user needs.

Creation and Deletion

Recall that objects can be created and deleted only by objects that have the proper input-output connections with the OMSEngine for these operations.

The domain customizer specifies the create/delete relations of objects. Failure to properly restrict the objects that have create/delete power results in haphazard evolution of the system.

The domain customizer also specifies when and how frequently objects are created and deleted. Instances of certain entities may be created once at system initialization time and never deleted. Other instances may be created and deleted very frequently.

Instance creation requires proper initialization of all input-output connections. Instance deletion requires proper termination of all input-output connections. The domain customizer works with the system architect to provide the proper constructs for object creation and deletion. If possible, the domain customizer specifies the create and delete methods of all entities. Subclasses specified by application developers may have to implement their own create and delete methods.

Relationship Evolution

Recall that two objects enter an input-output, or any other sort of, relation if a connect output is sent to the OMSEngine on their behalf. Such connections can be established only by objects that can send the corresponding outputs to the OMSEngine. Furthermore these objects need to have sufficient information to establish proper connections.

The domain customizer specifies the objects that are in charge of establishing/terminating relations. Failure to properly restrict the objects that have connect/disconnect power results in haphazard evolution of the system.

The domain customizer also specifies when and how frequently relationships change. Most systems have a set of objects and relationships that are either entirely static, or evolve at a much slower time scale than the rest of the system.

The domain customizer works with the system architect to provide the proper constructs for relationship creation. Whenever possible special modules are provided to initialize the static relationships.

Time Scale of Evolution

State evolution can be time or event driven. The domain customizer decides how to represent time and event driven evolution.

Time driven entities change state with the passage of time. Clearly for a time driven object it is impossible to simulate every state change. Such simulation has to take place at some meaningful sampling rate. The domain customizer has to determine this rate, i.e., the time scale of evolution of time driven entities.

Event driven entities change state upon certain events. The domain customizer has to specify the structure for event generation and event propagation. The domain customizer may be able to define all events and messages or may delegate this task to application developers. In the physical world, events have to be delivered through some medium. If a detailed model of the medium is not part of the simulation, the domain customizer may choose to assign a time delay for event delivery. The response of an entity to an event may not be instantaneous. The domain customizer determines the event response time of entities.

In hybrid simulations, time scale applies to event driven objects as well. Delivery of events does not need to take place at every clock tick.

There is a trade-off between the clock-rate and simulation performance. It is the responsibility of the domain customizer to work with the system architect and to select the right time scale of evolution for each entity. Sometimes, a sliding time scale must be used due to problem requirements. Updating vehicle positions every hundred milliseconds may be acceptable in normal mode of operation. However, if detailed accident simulation is part of the requirements, vehicles that come together closer than one meter may have to change their time scale of evolution.

The domain customizer may choose to specify the time and/or event driven behavior of select physical entities. Most behavior specification is delegated to application development.

Identification of Operation Modes

Failures and undesired behavior are bound to be part of any physical system. The domain customizer identifies the ideal and degraded modes of operation of the system.

The domain customizer designs entities that can identify failures and take corrective action. The detailed specification and implementation of the behavior of these entities may be delegated to application development.

Functional Requirements and User Needs

The entire data and process model is developed with the functional requirements and user needs in mind. This step merely performs a final check to ensure that these requirements and needs are indeed satisfied.

The domain customizer ensures that the Customized OM and OMS Application are addressing the needs of the users. A canonical advice to each domain customizer is to provide each user with as much information as necessary but as little information as possible. The provided operations have to be at the right level of granularity. For example, for frequent operations, simple and quick methods are needed. It is important to understand what the users want to change, why they want to change it, and how often they want to change it.

4.4.2 System Architecture

Working with the domain customizer the system architect first selects/develops a specification syntax for the Object Model, and develops an implementation of it. Given today's computer technology the Object Model implementation starts with the selection of a hardware platform, an operating system, a programming language, and a database management system.

Based on the application needs an of-the-shelf software tool may be available to implement the Object Model. A more likely scenario is the use of several component tools that address various aspects of the application needs.

A good system architect avoids tools that meet only part of the system requirements, but do not provide any obvious means of extensions to meet the rest. With today's rapid rate of technology evolution, using tools and platforms just because one is familiar with them is bad engineering practice. Such tools provide a quick start to the project, resulting in false confidence, and are a recipe for failure.

The system architect and domain customizer bootstrap from the Object Model and implement the Customized OM based on the domain customizer's entity specifications.

The system architect is responsible for delivering the Customized OM implementation that meets all software requirements. In particular the system architect specifies and implements 1) the data and process distribution architecture, 2) the time and event driven scheduling constructs, and 3) other constructs specified in 4.2.4 and 4.3 that are not part of the Object Model but are needed for the application at hand.

We now discuss some of the decisions facing the system architect.

Distribution Support

With the help of the domain customizer the system architect organizes objects into domains. The domain boundaries should be selected such that the interaction between domains is minimized. The domains provide natural boundaries for process distribution. However, sometimes many or all domains may reside in the same process. Alternatively, if execution sequences can be parallelized, a single domain may be simulated in several processes.

In case of a distributed architecture, the system architect designs the proper mechanisms that support the interaction among OMSEngines. These mechanisms include 1) data dictionary services to locate OMSEngines and objects; 2) output propagation; 3) migration support for objects.

If a domain is simulated in several processes the system architect must ensure that each process has a consistent image of the domain.

Time and Event Driven Scheduling

With the help of the domain customizer, the system architect specifies the execution and transition sequences of the system. The system architect makes sure that the execution sequences are valid and specifies sequencer entities that implement time driven, event driven, or hybrid evolution of objects.

Some applications may require a configurable simulation granularity. In this case some of the sequencer specification must be delegated to later stages of OMS. The system architect implements the necessary mechanisms to specialize sequencer behavior. These mechanisms may vary from parameter specification to specialization through inheritance.

Analysis Support and System History

The state of an object is given by its state, input, and output attribute values. The state of a domain is given by the list of the entities and instances it contains, the particular input-output connections, and the state of these instances.

Most applications require the ability to record the state history of a single object, a collection of objects, or an entire domain. Clearly, saving the entire state of a domain after every transition provides sufficient information, but this approach is infeasible in most applications due to system size. The system architect has to provide constructs to save and restore state at the right level of granularity.

These constructs must provide a labeling mechanism that imposes an order to the recorded history. In some applications a global clock is used to represent time passage. The time stamp of the global clock may be used to label recorded history. However, if an object can participate in several transitions within the same time stamp a different labeling mechanism is needed. For example, event driven objects may take several transitions at the same time stamp.

In some applications it suffices to save the state history of individual objects. In this case a mechanism is needed to specify the objects whose state should be saved. In particular one should be able to specify an object that may be created in the future. For time driven objects, state history should be recorded at appropriate time intervals. For event driven objects, state history should be recorded only if events occur.

The system architect must decide what to do with the recorded history of an object if that object is deleted. In most applications the recorded history of an object has to be saved even after the object ceases to exist. For example, in a simulation of vehicles on the highway, the state history of the vehicle contains important statistical data that is needed even after the vehicle leaves the highway. In fact, if one wants to replay the vehicle's state trajectory along the highway the vehicle instance itself must be saved after it leaves the highway.

The system architect must decide what to do with the recorded history of an object if that object is migrated to another domain. The decision is based on how the recorded history is used. If the state history is considered to be part of the object ,it should be migrated with the object, but, if it is considered to be part of the domain in which it was recorded, it should stay behind.

Specialized Maps

It is unlikely that one Object Model can provide the perfect specification syntax for all application domains. Even within a given application, there usually are components with very different evolution models. For example, in hybrid systems, one finds both state machine and differential equation representations for object behavior. Other applications may require rule-based input-output maps.

Based on the application requirements, the system architect designs and implements extended specification languages and the necessary translators that convert specifications in these languages to the Object Model or the Customized OM constructs.

Verification

OMS are used to simulate the complete behavior of the system at hand. A simulation run starts at a specific initial condition and runs for finite duration. Even after extensive simulations, only a subset of all possible OMS states are reached.

In general, absolute statements about OMS behavior are undecidable. For highway automation, e.g., one usually cannot prove that a given architecture does not result in any accidents.

However, it is very desirable to get such results and to verify system behavior whenever possible. Such results can usually be obtained with simplified or partial object models.

As discussed in Section 2.9.1, decidability results exist for discrete event systems; but, such results are very limited for hybrid systems.

In the previous subsection we stated that the system architect designs and implements extended specification languages to meet the needs of an application. These languages should be designed to facilitate partial system verification.

Consider a hybrid system representation where the continuous behavior results in events that effect the discrete evolution. If the specification syntax has clear boundaries between its discrete and continuous components, partial verification may be possible. The effects of the continuous behavior can be replaced with a nondeterministic discrete automaton and the resulting system can be translated into a syntax, such as COSPAN, for verification.

An example of such a language is given in Section 6.3.

4.4.3 Application Development

The application developers are the users of the Customized OM. In most OMS applications, application developers include communication and control engineers who design proper local control actions to ensure that the overall system evolves without violating any constraints. The application developers bridge the gap between computer, communications, and control technologies. A detailed discussion of relevant communication and control technologies is beyond the scope of this thesis.

Based on application needs, the application developers specialize the entities of the Customized OM by adding state, input, and output attributes, methods, and constraints. Following the guidelines of the domain customizer and the system architect, application developers also implement the appropriate interfaces for the OMS users.

The OMS approach greatly reduces application development time and increases software reliability.

The application developers are shielded from system level decisions since the skeleton of an application is provided by the Customized OM. The application developers can focus on a small set of objects, their inputs and outputs, and design the proper input-output behavior.

The specialized specification maps of the Customized OM enable application developers to design in a language appropriate for their expertise, without loosing time to the peculiarities of a programming language. Code-generators are then used to reliably translate these specifications into Object Model or Customized OM constructs.

Since the Customized OM is an actual running system, it provides the necessary harness for unit-testing. Application developers do not need to duplicate effort to create test-beds to test their objects, but rather, they integrate them directly into the Customized OM for unit-testing.

4.4.4 OMS Users

The users of the OMS Application can be classified into several groups. These groups are discussed briefly.

Support Staff

Once the OMS is deployed the support staff provides day-to-day support to other OMS users. They explain the use of the OMS interfaces and identify possible shortcomings of the deployed system. These shortcomings are addressed in subsequent releases of the OMS.

OMS Operator

The OMS Operators are the daily users of the OMS. Based on their responsibilities they use specific subsets of OMS functionality.

If the OMS is targeted to simulation they run extensive simulations and evaluate the system behavior.

If the OMS is targeted to managing a deployed system, they monitor the behavior of the physical system through the OMS interface and take microscopic control actions based on the advice provided by the OMS, their own intuition, and their level of authority.

In many applications they interact with Physical System Operators and coordinate their activities.

System Supervisor and Manager

System Supervisors and Managers are the occasional users of the OMS, but, have higher level of authority in terms of macroscopic decisions.

They exercise a wide range of OMS functionality, collect high level performance information about the system, and are in charge of planning management.

Physical System Operator

In the case of a deployed system, not all operations can be performed through the OMS interfaces. A burnt transformer along a transmission line, for example, must be replaced physically.

The Physical System Operators have the responsibility to perform physical maintenance activities. Their efforts are coordinated by the OMS Operators.

4.5 Satisfying Functional Category Requirements

In Section 2.8 we have discussed seven functional categories common to all Object Management Systems and the constructs required to satisfy them. This Section discusses how the OMS approach satisfies the requirements of these functional categories.

4.5.1 Configuration management

The Object Model provides the ability to specify the logical model of a physical system.

The OMSEngine provides the necessary constructs to create, delete, and interconnect instances. The Customized OM provides specialized tools and interfaces for these operations.

The application developers design and implement the entity methods that ensure that the system evolves without violating the constraints specified by the domain customizer.

The event and control flow is specified by the domain customizer and the system architect as part of the execution sequence and the time and event driven scheduling design.

If sensors and actuators are needed to interface with a physical system, these are specified as application layer entities. The methods of these entities encapsulate and abstract the details of the physical system interfaces.

4.5.2 Fault and event management

The different modes of operation are part of system specification in the Customized OM and the OMS Application.

The Object Model provides the necessary constructs for event creation and propagation. The domain customization and system architecture stages customize these constructs based on application needs and design entities responsible for the creation, propagation, and correlation of events. The event set used for fault identification and recovery may be specified by the domain customizer, by the application developers or by both.

Fault correlation and identification are hard problems. Application developers specialize the entities responsible for event creation, propagation, and correlation to provide graceful fault recovery.

4.5.3 Performance management

Monitor and aggregation entities are specified to measure the performance of the system. These entities usually measure many different performance parameters and delegate the specification of the exact evaluation criteria as a function of these parameters to the OMS Application users.

Design and implementation of a system with good performance is a harder problem. The domain customization stage decomposes the performance management requirements and specifies a number of control entities that address localized performance criteria. Application developers design and implement the methods of these entities. The OMS Application is used to simulate and evaluate the performance of the design. Based on simulation results the designs are refined until the required level of performance is reached.

4.5.4 Access and security management

The Object Model provides constructs to satisfy access and security management requirements.

An object can restrict access to its inputs and outputs by allowing certain kinds of interconnections only. An object can choose to provide restricted access by its views. Alternatively, a user relation can be created that is capable of connecting to specific entities or instances only. The user relation can be used to represent actual human users or other software entities.

The specific access and security management structure is designed and implemented in the domain customization and application development stages.

4.5.5 Financial management

The ability to record history facilitates financial management. Since users can be represented as entities their recorded state history can be converted into billing statements according to the billing policy.

4.5.6 Resource management

Since the OMS Application provides an open interface, resource management functions can easily be integrated into the overall system. Maintenance scheduling and inventory tracking can be implemented by a set of entities that perform the required functions.

4.5.7 Planning and design management

Object Management Systems are designed for system specification, simulation, and evaluation. The configuration, fault, and performance management functionalities facilitate planning and design management.

4.6 OMS Evaluation Criteria

An OMS has to address many conflicting requirements. Each application developer and each user has different needs and there is no single metric that can assign a grade to the quality of a Customized OM or an OMS Application.

So far we have discussed constructs that should be part of the Object Model and the Customized OM. The expressive power is only the beginning of a good design, its ease-of-use is what makes the system succeed or fail. The ease-of-use is in direct conflict with the flexibility and the expressive power of the system. A system that has a single function can be operated by a single button. The more operations it supports the more "buttons" one has to add.

The popularity of an application is usually a good measure of its overall quality. However, there are many factors that contribute to an application's popularity that are beyond our scope. These factors include its price, the availability of technical support, the vendor's reputation, the absence of competition, etc.

This section addresses key criteria that determine the ease-of-use of a system and discusses potential trade-offs for conflicting software system requirements.

4.6.1 Ease-of-Use

Relevance of Entities

The entity definitions are expected to abstract and encapsulate the characteristics and behavior of physical objects. The simplicity of a system is crucial for its ease-of-use. However, abstraction is aimed at hiding *irrelevant* information. Oversimplification may lead to a neat, yet practically useless system. Three questions should be asked of a Customized OM or an OMS:

- Does the model provide "real" concepts or are the constructs too far removed from the actual system;
- Is there a clear translation mechanism between the modeled entities and the actual physical world objects;
- Is there a match between the operations provided by the OMS and the operations possible in the physical world?

Locality of Reference and Learning Curve

Application developers should be able to add entities and to specify and change behavior. The following questions should be asked of a Customized OM:

- How many entities/modules does one need to modify to perform these operations;
- How much does one need to know about other objects and the process model of the system;
- How many steps and how much time are required to integrate the changes?

The OMS Application users perform different operations based on the application; however, similar questions apply to the OMS Application. In particular one should ask:

- What's the minimum a user has to learn to start using the system;
- Do simplicity of operations match their frequency of use?

Ease of specification

A Customized OM and an OMS Application should provide tools to simplify the specification tasks of application developers and system users. Such tools may include finite state machine representations, protocol specification languages, differential equation solvers, and graphical editors.

The following questions should be asked of these tools:

- Is the tool suitable for the problem domain;
- Is it possible to specify an incorrect behavior/configuration with the tool;
- What happens at run-time if an incorrect behavior/configuration is specified?

Ease of Evaluation

Both the application developer and the OMS Application users need services to facilitate system evaluation. These services should include:

- Ability to save a snapshot of system state;
- Ability to trace individual objects;
- Ability to "replay" traces;
- Ability to collect aggregate state information.

System Level Issues

The seven functional categories of Section 2.8 apply to the Customized OM and to the development of an OMS Application.

- Configuration management;

  Ability to mix and match Customized OM constructs to create OMS Applications, ability to track and maintain releases and various versions of the Customized OM are example configuration management functions.

- Fault management;

  The ability to prevent and detect faulty specifications and the ability to identify specification errors at run time are fault management functions. "Segmentation Fault" and "Bus Error" are, unfortunately, the most common software error events.

- Performance management;

  For off-line simulation the OMS does not need to meet real-time performance requirements. In fact, no architecture can guarantee an upper bound on the simulation time of a given object, since this time greatly depends on the application layer code. On the other hand, the lower bound imposed by the Customized OM overhead has to be acceptable.

  For on-line management, the OMS Application has to meet real-time performance requirements. Events in the physical world must be communicated to the OMS Application; these events have to be delivered to the appropriate modules in the OMS Application which determine the control actions; and finally the control actions must be communicated back to the physical system, all within an acceptable delay. Furthermore, the OMS Application must be able to handle worst-case event rates.

  Memory and disk usage can also be categorized under performance management. Although memory and disk space are getting cheaper, they still impose a constraint as complex applications penetrate the lap-top computer market.

- Access and security management;

  Not all application developers nor all users should be allowed to modify all parts of an application.

- Financial management;

  Software licensing is a topic of its own and is beyond our scope.

- Resource Management;

  The maintenance of a Customized OM and OMS Application includes the tracking of bugs and the upgrading of the software with evolving hardware platforms.

- Planning and Design Management;

  Planning management includes the evolution of the Object Model and the various Customized OM frameworks.

4.6.2 Conflicting Requirements

In this section we list some key conflicting requirements in software systems.

- Object Model versus other requirements;

  Adding more constructs to the Object Model usually results in sacrificing performance and increasing memory and disk usage. Complex constructs can also make an application error-prone. For example, if the meta-data is dynamic, one can change class definitions on-line. However, in this case it is not clear how much one can modify a class without having to remove all existing instances.

- Modularity versus other requirements;

  If everything were implemented in one function using machine code, we would get fastest performance. The more components one introduces, the more interfaces one needs to maintain and the larger the overall system gets.

- Performance versus other requirements;

  One can usually trade-off memory for performance. Scalability and distribution support require bookkeeping mechanisms that add overhead. Robustness and openness require that there is minimal or no transient data; however, clearly recording system state at every step reduces performance.

Chapter 5

Automated Highway Systems

In this chapter we discuss the domain customization and system architecture stages of an OMS that is targeted to Automated Highway Systems. The Object Model specification and implementation is called SmartDb, the Customized OM is called SmartAHS, the final OMS Application, used to implement the Varaiya/Shladover architecture, is called SmartPATH.

SmartPATH decomposes the IVHS modeling and simulation problem into the following stages:

1. Parametrized modeling of the physical system and the control agents in an object oriented semantic data model; Delivery of SmartAHS;
2. Design and implementation of various control and communication strategies;
3. Simulation of the discrete and continuous behavior of all the objects in the model, and optimization of the model parameters; Delivery of SmartPATH;
4. Evaluation of system performance according to specified criteria;
5. Model validation and implementation of selected control strategies for deployment.

These stages are shown in Figure 5.1.

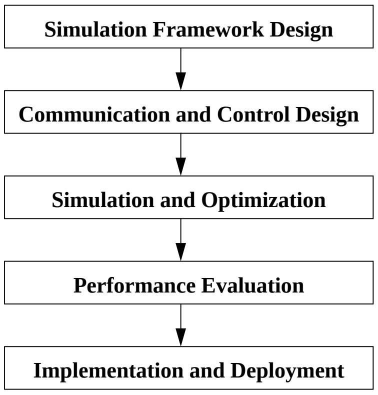
Simulation Framework Design
Communication and Control Design
Simulation and Optimization
Performance Evaluation
Implementation and Deployment

Figure 5.1: SmartPATH Stages

This thesis discusses the first step, namely the specification and implementation of SmartAHS simulation framework. The actual design and implementation of control and communication strategies are beyond our scope. However, parts of SmartPATH are discussed in Chapter 7 to illustrate the use of SmartAHS.

The domain customizers and the system architects are responsible for delivering SmartDb and SmartAHS. These roles are assumed by the author. In Section 3.1 we discussed the multi-layer control strategy for highway automation. This design constitutes significant existing domain customization effort. However, SmartAHS has to meet the needs of this or any other automation strategy.

The application developers design and implement individual control and communication components. They provide parametric interfaces for adapting their control algorithms to evaluation criteria. The application development team consists of control and communication engineers and delivers SmartPATH.

Detailed evaluation is to be performed by system analysts. They select evaluation criteria and run extensive simulations in SmartPATH to measure the performance of selected automation strategies.

Based on the performance results and other social and political considerations an automation strategy is selected for deployment by system planners.

This chapter discusses the early stages of the domain customization and system architecture for SmartDb and SmartAHS, describes the implementation environment for SmartDb, and provides a high level discussion of the data and process model for SmartAHS.

The implementation of SmartAHS is discussed in the next chapter.

5.1 SmartDb Implementation Platform

This subsection describes the selection process for the implementation platform components of SmartDb and SmartAHS. Most of the decisions are guided by concerns of practical implementation.

5.1.1 Tool Selection

No existing simulation tool satisfies all the criteria summarized in Section 3.1.3. After a review of SmartPATH requirements the system architect and domain customizer decided on SunSparc stations as the hardware platform, Unix as the operating system, C++ as the programming language, Versant as the OODB, and Tcl/Tk as the graphics package for the SmartDb implementation.

The reasons for these decisions are discussed briefly.

The need for an open architecture requires that the system state be fully and easily accessible. The need to collect arbitrary statistics during simulation requires that state be saved to a persistent storage medium during simulation. Since SmartDb is based on the object-oriented paradigm, the system architect decided to use an OODB to implement persistence. An OODB has a well-defined interface, makes the simulation state visible to the user, and provides a default mechanism for any other simulation package to interface with SmartAHS.

A survey of the available object oriented databases suggested Versant as the OODB of choice. Versant's advantages over other current systems are discussed in Appendix 9.2. Since Versant provides persistence through inheritance, making the base class of SmartAHS inherit from Versant's `PVirtual` class achieves persistence for all entities. The `PVirtual` class provides the necessary methods to make an object persistent.

Versant supports two object-oriented programming languages: C++ and SmallTalk. C++ is a compiled language and supports only static entity definitions at run time. However, since SmartDb assumes static entity definitions, this limitation of C++ does not preclude its use. Since SmallTalk is a proprietary language, and since C++ tools are more readily available in the commercial market, C++ was chosen as the implementation language for SmartDb.

Versant and C++ are only available on the Unix operating system and on SunSparc and Silicon Graphics (SGI) workstations. These restrictions are acceptable since these are good platforms for development. They provide computational power and operating system support for the implementation of large multi-user multi-process distributed applications.

Unix is selected as the operating system. Since SunSparc has better support for software tools, it is selected as the hardware platform. This choice does not preclude the option of porting the system to SGI in the future, if needed.

Finally the system architect selected the public domain graphical package Tcl/Tk for user interface implementation. Tcl/Tk is an extensible, interpretive environment that supports interprocess communication. It runs on Unix/Sparc and Unix/SGI and supports a C/C++ interface.

These components implement only a subset of the constructs discussed in the previous chapter. The domain customizer and system architect bootstrap from SmartDb and provide the remaining constructs within SmartAHS.

The following subsections describe how these component tools are used to provide the Object Model constructs.

5.1.2 Data Model

As a programming language C++[1] provides the constructs to define `class`es, `attribute`s, and `methods`. C++ distinguishes between `private`, `protected`, and `public` class components and supports multiple inheritance, polymorphism, and dynamic binding. C++ does not provide full monotonic inheritance support. In particular, there is no syntax to restrict the domain of an attribute as part of specialization[2]. SmartAHS specification depends on attribute specialization and its implementation provides work-arounds for this C++ deficiency, as discussed in Section 5.3.6.

Versant libraries are summarized in appendix 9.2. The system architect bootstraps from C++ constructs and Versant libraries and implements entities as C++ classes. Entity components are specified as follows:

- Distinguished Name;

  The `Link` construct provided by Versant, a persistent pointer, is used as the instance identifier at runtime. The scoping rules of C++ provide all other constructs to identify individual components of instances.

  Links do not carry any semantic meaning. Furthermore, the value of a Link is defined at runtime only. An independent naming convention is developed to name all instances in the system. The naming convention is discussed in Section 6.7.

- State, input, and output attributes;

  A distinction is made between static and dynamic state attributes.

  The length of a car, the width of a lane, etc., constitute static state. Their values are initialized at instance creation time and not modified during a simulation run. All static relationships and static state characteristics are implemented as class attributes.

  The speed of a car, average density of a lane, etc., constitute dynamic state. The values of these attributes change during a simulation run. Dynamic state, inputs, and outputs are implemented as independent classes, these classes also contain dynamic relationships among objects.

- Methods;

  C++ methods are used to implement behavior. These methods do not support the input-output propagation described in the previous chapter. This issue is discussed in the process model.

- Constraints;

  C++ is a strongly typed language. Each type defines the value domain of an attribute. For example, one can define an `enum`erated data type, which is a finite list of integers. Semantic constraints and participation constraints are implemented by proper attribute type definitions. These constraints are enforced by the C++ compiler.

  Other constraints are implemented as regular class methods.

[1] See Appendix 9.1 for a quick overview of the C++ programming language.

[2] According to monotonic inheritance, discussed in Section 2.4.2, a subclass can restrict the domain of a base class attribute. An attribute declared as `float`, for example, may be restricted to `integer` values only in the subclass.

5.1.3 Process Model

Execution Sequences and Transitions

C++ provides synchronous method calls only[3]. Input-output propagation among the base classes of SmartAHS is achieved by these synchronous method calls.

SmartAHS classes bootstrap from C++ methods and implement an asynchronous event delivery mechanism. SmartPATH classes are required to define their inputs and outputs as separate classes. Their public methods do not use arguments.

The OMSEngine

OMSEngine constructs are distributed to a collection of C++ classes.

Constructor and destructor methods are used in entities to specify create and delete methods. The C++ language provides "new" and "delete" operators that are accessible to all objects. SmartAHS limits the use of these operators to specific entities only.

Input-output connections are maintained by SmartAHS classes. The Link construct of Versant provides direct access to the instance it identifies. As such access to the Link of an instance provides access to all its public components. SmartAHS defines input and output interfaces for entities and expects SmartPATH objects to use these interfaces rather then using Links for direct access to objects.

Execution sequences and transitions for SmartPATH are designed and implemented as part of SmartAHS. SmartAHS implements these constructs using the C++ methods.

Sequencing Objects

In C++, a method can call other methods. This gives sequencing power to all objects. SmartAHS delivers a framework where only scheduling objects act as sequencers.

5.1.4 Extended Object Model Constructs

Virtual attributes and associations are not used. Binary relations are implemented by `Link`, `VVList` and `VVarray` constructs of Versant. More complex relationships are represented by C++ classes.

The static relationships in the application are initialized in independent modules. These modules guarantee relationship integrity. Dynamic relationships are maintained by SmartAHS classes.

Time and event driven evolution constructs are part of SmartAHS.

5.2 Domain Customization

5.2.1 Data Model

After several iterations and continued discussions with the system architect, domain customizer categorized SmartAHS entities and drew the line between SmartAHS and SmartPATH as follows:

SmartDb Entities

Two abstract base classes `FrameworkObject` and `StatedObject` are used as the base class of all SmartAHS classes. The base classes `State`, `Input`, and `Output` encapsulate the corresponding attributes of objects. These base classes are discussed in Section 6.1.1.

[3] Recall from Section 2.6.2 that a synchronous method call finishes all computation before it delivers a return value, whereas an asynchronous method call may continue computation after it delivers a return value.

Highway Entities

The description of highway networks is decomposed into smaller building blocks. The highway network is divided into `Zones`; each zone contains multiple highway `Segments` interconnected using `Junctions`. The highway segments are terminated using traffic `Sources` and `Sinks`. The highway segments consist of `Sections`, `Entrys`, and `Exits`. Junctions and sections are divided into `Lanes`. Lanes can have curvature. The full implementation and specification of highway entities are part of SmartAHS. These classes are discussed in Section 6.1.2.

Vehicles

SmartAHS provides `Vehicle` as an abstract base class. It defines the input and output attributes of a Vehicle and maintains its position within the highway. Application developers are expected to inherit from Vehicle and specialize it according to automation strategy.

Vehicles are discussed in Section 6.1.5

Automation Devices

Entities used to achieve automation and performance evaluation are called automation devices. SmartAHS provides them only as abstract base classes. Application developers are expected to inherit from these classes and to specialize them. The specializations of these classes interact with other objects through their inputs and outputs only. Their evolution is managed by the time and event driven scheduling objects.

These classes are discussed in Section 6.1.3.

Traffic Entities

Entities used to create traffic and vehicles are called traffic entities. SmartAHS provides them only as abstract base classes. These classes are discussed in Section 6.1.6.

SmartAHS defines the input and output attributes of Traffic Entities and their methods. Application developers are expected to inherit from them and specialize these methods.

Scheduling Entities

These are sequencing entities used to implement time and event driven behavior. SmartAHS provides configurable scheduler base classes. Based on simulation granularity application developers configure them with the help of a system architect.

These classes are discussed in Section 6.2.

State Machine Entities

A special language is used to implement event driven behavior of entities. SmartAHS provides full specification and implementation of this language.

The state machine language is discussed in Section 6.3.

5.2.2 SmartAHS Modules

The domain customizer also drew a line between the simulation setup and the simulation run. The steps for simulation setup are summarized in Figure 5.2. These steps are:

Highway specification

The highway network is created as part of the simulation setup and remains static during a simulation run. The highway creation mechanism of SmartAHS is designed as an independent module. A graphical object editor (GOE) is used to create highways. This object editor is discussed in 6.4.

The GOE provides a meta-data definition language for the specification of instantiable classes, their possible relationships, and their attributes. The GOE interprets the meta-data and

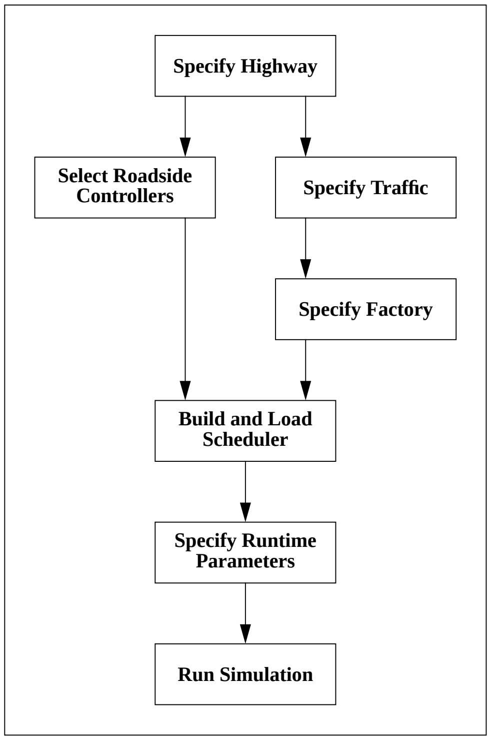

Figure 5.2: SmartAHS Specification Sequence for Simulation Setup

allows the user to create and connect instances and to set attributes, i.e., define the data, according to the meta-data specification.

Traffic Pattern Specification

Traffic entities are used to specify incoming and outgoing traffic patterns. An independent module is used to select specific traffic entities for each simulation run.

Roadside Automation Device Specification

Different automation strategies will choose different configurations of automation devices on the highway. An independent module is used to configure a highway network with automation devices.

Vehicle Automation Device Specification

Different automation strategies will choose different configurations of automation devices in the Vehicles. Application developers are expected to provide `Factory` entities that have the capability of creating vehicles with the appropriate configuration.

`Factory`s are discussed in Section 6.1.6.

Specification of Simulation Granularity

The time step of each simulation, the degree of monitoring and evaluation, and the set of objects simulated in detail may vary for each simulation. A special module is used to configure the scheduling objects and to specify the simulation granularity.

Specification of Simulation Parameters

Traffic entities and automation devices may provide parametric interfaces. A special module is used to set these parameters before each simulation run.

5.2.3 Process Model

For the simulation run the domain customizer makes the following process model decisions:

Creation and Deletion

The highway network, the highway automation devices, and the traffic patterns are created as part of the simulation setup and remain static during a simulation run.

For `Vehicle`s each control strategy is required to provide its own `Factory` that knows how to create the proper `Vehicle` types. During the simulation run, various types of Vehicles are created by `Factory` entities in `Generator` objects based on incoming traffic patterns. `Vehicle`s leave the highway at `Absorber` objects based on outgoing traffic patterns. Outgoing vehicles are not deleted but flushed to the database. Vehicle instantiation is further discussed in Sections 6.1.6 and 7.1.1.

Relationship Evolution

In SmartAHS all relations among highway entities and all relations among the objects within a Vehicle are set at creation time and remain static during a simulation run. The `Vehicle` has a static relationship with the Entry/Source where it enters the highway and with its Exit/Sink destination. These relationships are initialized at creation time.

All relationships based on the location of `Vehicle`s are dynamic.

`Vehicle` position and the relationship between a `Vehicle` and its `Lane` are maintained by SmartAHS as discussed in Sections 6.1.2 and 7.2.1.

All other inter-vehicle relations are derived using `Sensor`s and are discussed in Section 6.1.3. `Sensor`s are also used to identify the receiver ids of objects in the sensing range.

`Packet`s are used for event driven communication. SmartAHS maintains the relationships of `Packet`s, i.e., it implements `Packet` delivery.

Time Scale of Evolution

In SmartAHS, object state evolution is driven by passage of time or by occurrence of events. SmartAHS provides distributed scheduling for time and event driven objects.

A global clock is used to define the simulation time.

Each time driven object specifies the time step for its state evolution as a multiple of the global clock step. Rapidly evolving objects, such as engines, change their state more frequently, while more passive objects, such as roadside link controllers, change their state at larger time steps. Time driven objects are capable of creating events.

Event driven objects exercise their behavior only when events are delivered to them. Events are generated by objects as output messages and are communicated to the addressed objects as input messages. SmartAHS objects communicate using their transmitters and receivers. SmartAHS does not simulate a communication channel; it assumes reliable message transmission. Events are delivered at increments of the global clock time step. As such, event delivery is not instantaneous, but happens within a finite time interval.

Time and event driven evolution is implemented in a three layer hierarchy by the System Architect discussed in Section 6.2.

Constraint Specification

C++ is a strongly typed language, and it is used to restrict the attribute domains.

SmartAHS does not specify other constraints. If needed constraints can be implemented as special monitor entities that observe system state and state transitions and determine if any constraints are violated. These objects then become part of SmartAHS simulation.

Identification of Operation Modes

The different modes of operation are part of the semantics of individual automation strategies. As such the domain customizer identifies a single undesired behavior: an `Accident`.

If two Vehicles occupy the same space, or they fall outside the highway boundaries, we have an `Accident`. Detailed accident simulation is not part of SmartAHS. SmartAHS detects accidents, and creates corresponding notification events.

User Needs

The users of SmartAHS are control and communication engineers who will design and implement the automation devices. They will use SmartAHS as a simulation tool to debug and improve their designs. To this end they need graphical tools that simplify the display of the system state during simulation.

The users of SmartPATH will run extensive simulations, collect statistics, save and replay state trajectories of objects. They, too, need a graphical tool for these tasks.

The graphical debugger that addresses these requirements is discussed in Section 6.5. Conceptually, the graphical debugger is another sequencing object that enables the user to execute a sequence of transitions. In particular, it provides the following functionality:

- provides access to objects by name;
- displays object attributes in numerical or graphical format;
- allows users to set error levels of objects. Objects with higher error level print more detailed information;
- allows users to set the logging level of objects. Turning the logging on results in recording the state history of an object;
- allows users to stop and start the simulation;
- replays the recorded history of objects.

The replayer has five buttons: 1) Play; 2) Stop; 3) Step forward; 4) Step backward; and 5) Go to time. The last button takes an argument specifying the global clock time stamp.

Finally the domain customizer determines how application developers extend SmartAHS objects. Figure 5.3 describes how specialized classes become part of the simulation setup. The details of application development are discussed in Chapter 7.

5.3 System Architecture

So far the domain customizer has addressed most modeling, configuration, fault, and performance management requirements of SmartAHS. The system architect addresses the software and system requirements.

5.3.1 Process Structure

The system requirements dictate that the simulation be distributed. The system architect uses the highway Zones as the basis for distribution. Distribution is discussed in Section 5.3.3.

Within a Zone all objects are simulated in a single process. This decision is based on performance evaluation results of early prototype implementations where separation of the simulation of the various automation devices into independent processes resulted in poor performance. The prototypes attempted to use the database as a blackboard that would always reflect the state of a zone. The prototypes revealed that read/write database access duration for a zone with 2000 vehicles is in the order of seconds. Since the simulation granularity requires that the global clock time step be about 0.1 seconds, this observation made it infeasible to save the simulation state to the database after every transition.

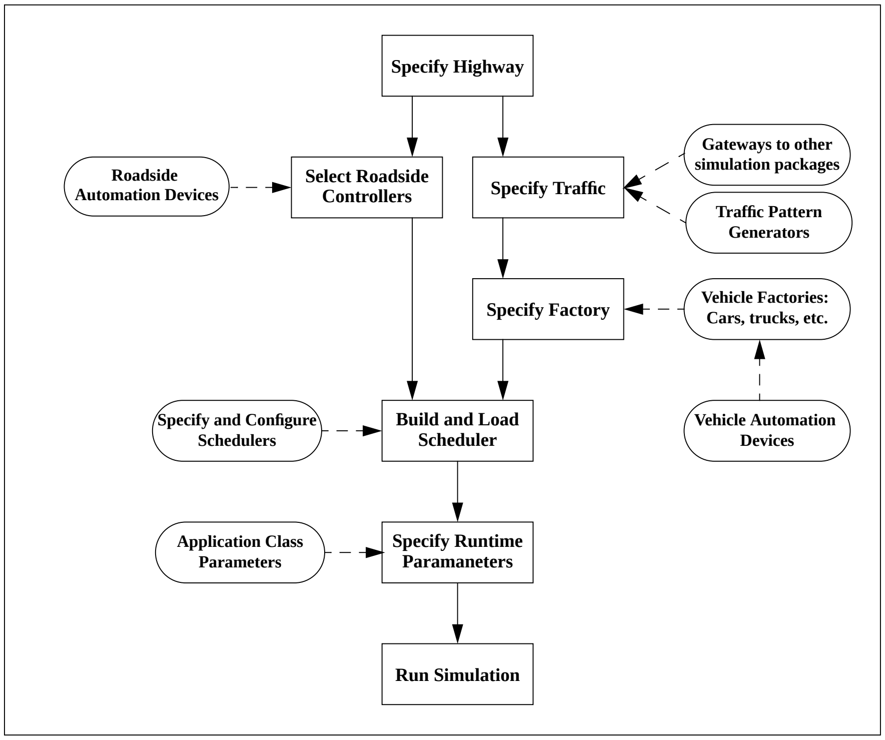
Specify Highway
Roadside Automation Devices
Select Roadside Controllers
Specify Traffic
Gateways to other simulation packages
Traffic Pattern Generators
Specify Factory
Vehicle Factories: Cars, trucks, etc.
Specify and Configure Schedulers
Build and Load Scheduler
Vehicle Automation Devices
Application Class Parameters
Specify Runtime Paramaneters
Run Simulation

Figure 5.3: Integrating Specialized Classes During Simulation Setup

Modularity of simulation requires that the scheduling architecture itself be configurable. A given simulation should be able to select its own granularity and decide time scale of evolution for objects within the simulation.

After these observations the system architects started specifying the time and event driven scheduling and event delivery mechanisms.

5.3.2 Time and Event Driven Simulation

This section describes how SmartAHS guarantees the timely evolution of each object and the timely communication of events to objects. Distribution is deferred to the next section.

Time and Event driven simulation is achieved in a three layer architecture similar to the example given in Section 6.2.

The bottom layer domain is defined by the highway `Zone` and all of its containees.

The middle layer scheduling entities are called *Process Layers*. They are used to execute the simulation of collections of objects that evolve at the same time steps or respond to the same collection of events. The process layers themselves can be time or event driven.

The top layer domain contains a *Process Coordinator*. The process coordinator is in charge of managing the execution of the middle layer scheduling objects, the process layers, and the *Global Clock*.

The process coordinator schedules the execution of time driven process layers based on their time step and schedules the execution of event driven process layers if any events are raised against them by an object.

The global clock represents the passage of time and defines the smallest time step of the system. All evolution takes place at discrete advancements of this clock. The clock value is accessible to all objects in the system.

The clock also provides a timer service. Objects can send an event to the clock to register a timeout request. This request specifies the number of time clicks after which the timer expires and a message to be delivered when it does. When scheduled for execution, the clock delivers timeout events as part of its behavior.

A process layer can do the following:

- simulate the time driven evolution of all instances of an object type;
- for all instances of an object type with an outstanding event, deliver the event and simulate the event driven evolution.

If event driven objects are put in a time driven process layer, event delivery for these objects takes place only when the corresponding process layer is executed.

In a simulation run, the process coordinator executes the process layers according to their time step, which in turn execute the simulation of the objects they are responsible for.

Example

The process architecture that would implement the layered architecture proposed by Varaiya [41] is shown in Figure 5.4.

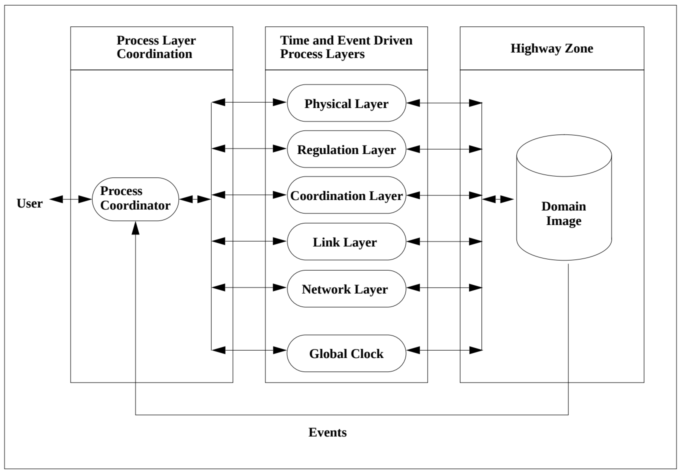

Figure 5.4: SmartPATH Process Architecture.

The physical layer is time driven and maintains the position of the vehicles on the highway.

The regulation layer is time driven. It contains event driven regulation supervisors and time driven maneuver objects. The supervisors switch between maneuvers based on incoming messages from the coordination layer; the maneuvers control the behavior of the throttle, braking, and steering actions and generate the vehicle displacement.

The coordination layer is time driven. It contains event driven coordination objects. Coordination objects in different vehicles exchange messages to determine the maneuver a vehicle should execute. These decisions are communicated to regulation layer supervisors through messages.

The link layer is time driven. It contains time driven link objects that set traffic parameters such as target speed and average platoon size in highway sections.

The network layer is event driven. It is executed only if an accident occurs. Upon an accident it reconfigures the routing tables.

5.3.3 Distribution Architecture

The need for a distributed architecture was identified in the problem specification. The distribution can be vehicle or highway based, i.e., a domain either consists of a given set of vehicles or a given subset of the highway network for simulation. Since locality of reference is better represented by highway network subsets the system architect chose to use the zone object as the basis for distribution.

SmartAHS supports distributed simulation. Zones serve as the unit of distribution: different zones can be distributed to different processors; they have their own database and their own clock. Since a vehicle can communicate with and sense other vehicles in its neighborhood only, communication between the distributed processors is restricted to objects in adjacent highway segments. This locality of reference enables efficient distributed processing.

Distributed simulation is depicted in Figure 5.5. The boundary object between the last section of the previous zone and the the first section of the next zone coordinates communication between the processors, ensures synchronization of simulation clocks on different processors, and controls object migration between databases.

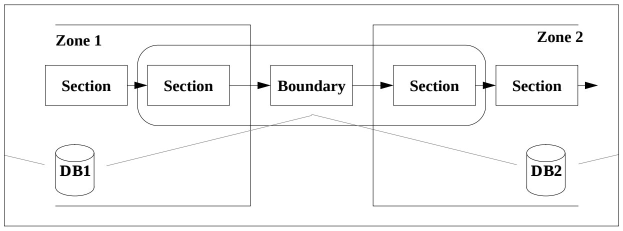

Figure 5.5: Distributed Simulation

The locality of reference assumption only holds for vehicles and roadside control objects. Traffic management centers, or network routing optimizers need a global view of the highway network and hence need access to all zones. SmartAHS provides for time and event driven scheduling of such External Global Processes. Figure 5.6 summarizes the distributed scheduling of time and event driven layers.

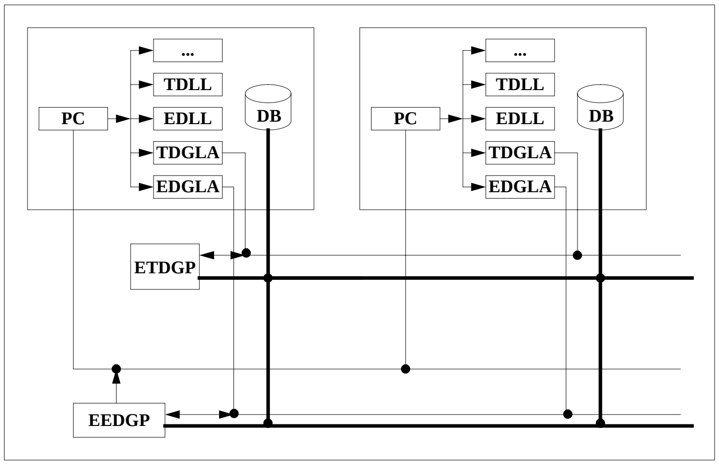

Figure 5.6: Distributed Simulation

Time and Event Driven Global Layer Agents (TDGLA and EDGLA) are used within each zone to coordinate the scheduling of the external processes.

An External Time Driven Global Process (ETDGP) exercises its behavior periodically. The TDGLA's in each zone are scheduled with this period and send a “Go” message to their ETDGP when scheduled. The ETDGP exercises its behavior after receiving the “Go” message from all TDGLA's, and returns a “Done” message to all of them.

An External Event Driven Global Process (EEDGP) exercises its behavior upon an event. The EEDGP should exercise its behavior when all EDGLAs are scheduled. Since a local event in a zone may require the scheduling of an EEDGP, its scheduling requires more coordination effort. First the EDGLA with the event notifies the EEDGP. The EEDGP in turn sends an event to all process coordinators, requesting the scheduling of the EDGLA layer at every time click, until otherwise noted. The process coordinator acknowledges this request upon receiving it. The EDGLAs send a “Go” message to the EEDGP when scheduled. The EEDGP responds to these messages with a “Don't Block” message until it receives the acknowledgements of all process coordinators. An EDGLA returns control to its process coordinator if it receives the “Don't Block” message. After receiving all acknowledgements, the EEDGP collects all the "Go" messages, exercises its behavior, and returns a "Done" message. At that point the EDGLA's cancel their scheduling request.

The details of distributed simulation are not discussed in this thesis. They can be found in [34].

5.3.4 Collecting Statistics

The evaluation of system performance is achieved by monitor objects that collect statistical data. In most cases monitor objects need to record state histories for future statistical processing.

Versant provides a versioning mechanism that saves versions of objects. However, Versant does not support the migration of versioned objects. Since the state trajectories of vehicles need to be recorded, and since vehicles do migrate from one domain to the other, the system architect designed a state history recording mechanism as part of SmartAHS.

During a SmartAHS simulation the evolution of an object results in a change in its state, input, and output attributes only. Since state, input, and output attributes are implemented as independent classes, an object's history can be recorded by versioning the state, input, and output instances. The object itself then can provide the necessary bookkeeping constructs to version, save, and restore their history. These bookkeeping constructs are implemented within base classes and further discussed in Section 6.1.1. Here we summarize the key features:

- The global clock is used to index the state history of objects;
- If an object takes several transitions at the same time stamp, a secondary index is used to label them;
- State history of objects is recorded only if their logging is turned on;
- Logging can be turned on through the graphical interface. The parametric interface can be used to turn logging on at object instantiation time;
- Objects keep track of the intervals during which their logging was turned on, i.e, their state was recorded;
- Time driven objects record their state at every time click at which their state changes;
- Event driven entities record their state at every transition;
- Monitor objects have the ability to specify the frequency with which to save their history.

The recorded history is saved within the OODB and is accessible to any application.

5.3.5 Verification Support

It is envisioned that the automation strategies will result in a hybrid control framework. Automation devices, such as coordination layer controllers, will use Discrete Event Controllers to specify symbolic control actions. These controllers use events to interact with each other and to observe the continuous evolution of the system.

The system architect and the domain customizer designed a special language, called State Machine Language, to specify event driven behavior. This language distinguishes the events exchanged among event driven controllers from events resulting from continuous behavior.

As such, it becomes possible to partially verify the discrete event behavior of the automation devices. In the simulated environment, the events resulting from continuous behavior are deterministic and result from actual simulation. In the verification environment, these events are treated as the nondeterministic component of state machines.

The state machine language is discussed in Section 6.3.

5.3.6 Monotonic Inheritance

C++ does not provide full monotonic inheritance support. This becomes a problem when referring to state, input, and output classes. The SmartAHS base class `StatedObject` defines a relation with the base class `State`. This relation must be defined at this level since base class methods need to access it.

However, this relation needs to be specialized for each subclass. The `Vehicle` class must have a relation with a `VehicleState` and not a `LaneState`. Furthermore, `Vehicle`'s methods need access to the attributes defined in the `VehicleState`. The base class `State` can not have all these attributes.

SmartAHS provides a work-around for this problem by introducing a new relation for each subclass. The `StatedObject` has a relation named `theState`, the Vehicle introduces a relation named `vehicleState`, a `Truck` introduces a relation named `truckState`. The same `TruckState` instance then participates in all three relations.

Chapter 6

SmartAHS Implementation

In this chapter we discuss the implementation of SmartAHS. SmartAHS uses the programming constructs[1] provided by C++ and Versant to create a Customized OM. SmartAHS provides the Object Model constructs described in Chapter 4 and the entity specifications for automated highways.

The reader should understand that SmartAHS is a "live" system that will evolve over time. The entity specifications in this thesis serve as examples of the concepts discussed in earlier Sections. For up-to-date documentation, the reader should always refer to the latest version of the reference manual and the user's guide of SmartAHS.

To maintain readability many details of entities are omitted. In particular:

- The distinctions between public, protected, and private components are not discussed;
- For all relations, SmartAHS entities provides specific `Get` and `Set` methods. If the relationship is represented by a list, further methods such as `Insert`, `Delete`, `Replace`, and `GetNumOf` are provided. These methods are not discussed;
- Most public methods implement their behavior with a number of private methods to maintain modularity. In most cases the private methods are omitted.

A relationship between two entities is implemented by a `Link` attribute in each object. If the object enters the same relationship with multiple objects a `VVList` attribute is used. In some cases explicit definition of the relationship attribute type is omitted.

For each object we describe the following components as appropriate:

- Static state relationship attributes;
- Static state value attributes;
- Dynamic state relationship attributes;
- Dynamic state value attributes;
- Inputs and Outputs; and
- Methods.

In the figures instantiable classes are represented by ovals, abstract classes are represented by rectangles. In the class hierarchy diagrams, inheritance is indicated by an arrow from parent to child class. Since the containment hierarchy is a one-to-many relationship, in the containment hierarchy diagrams, cardinality is indicated on the containee's side only.

We make an explicit distinction between static and dynamic attributes. The static attributes are part of the class, the dynamic attributes are in a separate class[2]. This separation facilitates the recording of state history.

[1] The C++ and Versant constructs are summarized in Sections 9.1 and 9.2 respectively.

[2] A pair of `Links` are used to establish the relationship between an object and its state.

SmartAHS base classes use C++ methods to communicate. For these methods we list their return value, their name, their list of arguments, and a brief description of their functionality. Unless otherwise noted, return values of methods that are of type `int` correspond to an error code. In the text we refer to methods by their name, e.g. `Run()`. If it is clear from context that `Run` is a method, the parentheses may be omitted.

Many methods are defined in a base class and specialized in all subclasses in some specific way. These methods are discussed only in the base class with a description of how they are specialized.

SmartAHS base classes develop an input-output formalism for control devices. Example SmartPATH control objects are discussed in the next chapter.

6.1 Domain Customization: Entities and Relationships

SmartAHS specifies abstract base classes for all objects and implements the leaf classes for highway objects.

In this section we discuss the SmartDb, highway, vehicle, automation, and traffic entities of SmartAHS, their class hierarchy, their containment hierarchy, their relationships, their state, input, and output attributes, and their methods.

6.1.1 SmartDb Classes

We define five abstract base classes: `FrameworkObject`, `StatedObject`, `State`, `Input`, and `Output`. The SmartAHS inheritance hierarchy for base classes is described in Figure 6.1.

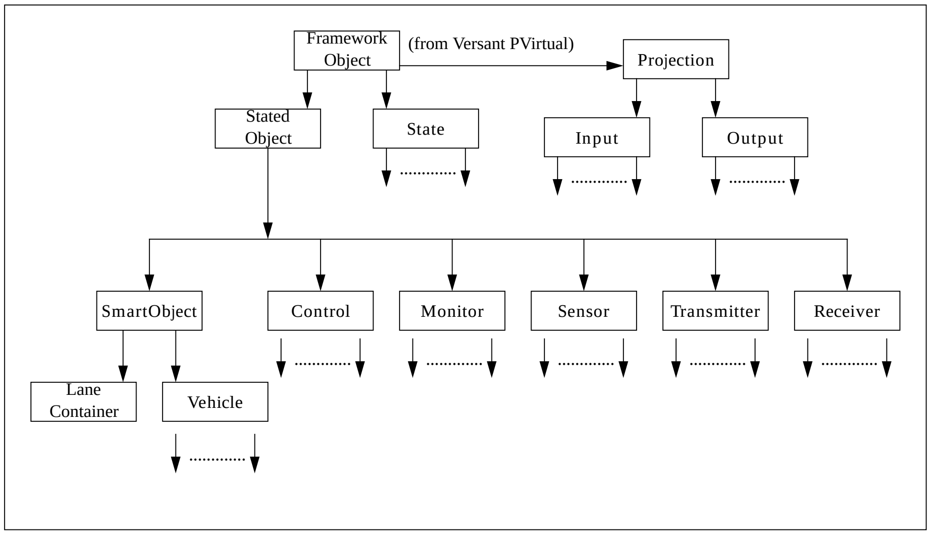

Figure 6.1: Base Class Inheritance Hierarchy

FrameworkObject

`FrameworkObject` is the base class of the simulation framework. It defines some basic methods that apply to all framework classes. These methods and their arguments are:

`int`	`SetName`	`(const char* newName);`
This method sets the name of the object to `newName`.		

`const char*`	`GetName`	`();`
This method returns the name of an object.		

```
virtual int        SetAttribute          (const char *attrName
                                          const char *attrValue);
```
This method sets the `attrName` attribute's value to `attrValue`. All subclasses should implement this method for their attributes and conclude the method with an explicit call to their parent's `SetAttribute` method.

```
virtual int        SetRelation           (Link<PVirtual> otherObject,
                                          const char *relation,
                                          const char *inputName,
                                          const char *outputName);
```
This method establishes the `relation` between `this` and the `otherObject`. The `inputName` and `outputName` specify the input and output to be connected. All subclasses should implement this method for their relations and conclude the method with an explicit call to their parent's `SetRelation` method.

```
virtual int        GetContainees         (LinkVstrAny & myContainees)
```
This method sets the contents of `myContainees` to all the containees of the object. All subclasses should implement this method for their containees and conclude the method with an explicit call to their parent's `GetContainees` method.

```
virtual int        WakeUp                (o_u4b handle,
                                          Link<Message> state);
```
This method is used to deliver a timeout to `this` object by `BigBen`. The request is identified by `handle`; `state` is passed to `BigBen` while registering the timeout request. This method is revisited in Sections 6.3 and 6.2.2.

```
virtual int        ProcessEvent          (o_u4b packetType);
```
This method is used to deliver events identified by `packetType`. It is revisited in Sections 6.3 and 6.2.2.

```
virtual int        ProcessMsg            (Link<Message> newMsg);
```
This method is used to deliver `newMsg` to the object. It is revisited in Sections 6.3 and 6.2.2.

```
virtual void       Pack                  (char* request);
```
This method is used by the graphical debugger to display the static and dynamic state of the object. It encodes all or part of the object in a standard format, based on `request` type. This method is revisited in Section 6.5.

```
virtual LinkAny    Duplicate             ();
```
This method creates a deep copy of the object.

```
virtual int        SetErrorLevel         (o_u1b newLevel);
```
This method sets the `errorLevel` attribute of the object to `newLevel`.

The attributes of `FrameworkObject` are `errorLevel` and `name`.

StatedObject

`StatedObject` captures the relation between an object and its `State`. This static relationship is defined by the `currState` and `myObj` attributes respectively. As discussed in Section 5.3.6 C++ does not provide full monotonic inheritance support. Hence, subclasses of `StatedObject` may define attribtues other than `currState` to identify the relationship between a `StatedObject` and its `State`. Examples, such as `VehicleState, SMInstanceState`, and `LaneState`, are discussed in subsequent sections.

The static attributes of `StatedObject` are:

`StatedObject Static Attributes`		
Values		
`VEArray<o_u4b>`	`logIntervals`	Used to maintain intervals of recorded history.
`o_u1b`	`logLevel`	If it is set to greater than zero, logging is active.
Relations		
`Link<State>`	`currState`	Identifies the State of the object.

`Link<BigBen>`	`myBigBen`	The clock used to register timeouts. This attribute is set by the instantiable leaf class.
`VVArray<State>`	`stateHistory`	Used to record history. Index corresponds to time stamp.

We said that the dynamic attributes of an object are stored in its `State` class. The `logIntervals` and `stateHistory` attributes do change during a simulation when the `logLevel` is turned on and off. However, these attributes are not part of the dynamic `State` of an object since they do not effect the actual evolution of objects. Furthermore, if `stateHistory` were part of `State` it would have to record its own history. Future releases of SmartAHS may move the `logLevel`, `logIntervals`, and `stateHistory` attributes into an independent `StateHistory` class.

Key `StatedObject` methods are:

```
virtual int        SaveState              ();
```
This method duplicates the `currState` of the object if `logLevel` is greater than zero. It uses the global clock to get the time stamp and inserts the duplicated state at the time stamp index of `stateHistory`. If state has already been saved at this time stamp, the `History` object is used to implement the necessary bookkeeping.

```
virtual int        SetToTime              (o_u4b time);
```
This method restores `currState` to the `State` from the `time` index of the `stateHistory`.

```
virtual int        SetLogLevel            (o_u1b newLevel);
```
This method sets the `logLevel` attribute to the value of `newLevel`. It implements the necessary bookkeeping to maintain `logIntervals` and to turn logging on or off.

```
virtual o_bool     InLogIntervals         (o_u4b time);
```
This method returns true if `State` has been recorded at the specified `time`.

```
virtual int        Run                    ();
```
This method should be specialized by subclasses to implement time driven behavior.

State

This abstract base class has two static relationship attributes identifying its `StatedObject` and its `History`. We document them below:

`State Static Attributes`

Relations		
`Link<History>`	`myHist`	Identifies its `History`.
`Link<StatedObject>`	`myObj`	Identifies the `StatedObject`.

The `History` object is used to implement a linked list structure for recording several state transitions at the same time stamp. The first transition is inserted into the `stateHistory` array of the `StatedObject` at the index given by the time stamp. Subsequent transitions are saved in a linked list and chained along the `sameClickState` attribute of the `myHist` of the `State`.

The `History` class has the following static attributes:

`History Static Attributes`

Relations		
`Link<State>`	`myState`	Used to identify the `State` owning the `History`.
`Link<State>`	`sameClickState`	Next transition recorded at the given time click.
`eventId`	`o_u4b`	Secondary index identifying transition within the given time stamp.

Inputs and Outputs

As illustrated in Figure 6.1 these objects share a base class called `Projection`. Input and Output `Projections` are used by the State Machines only. Their discussion is deferred to Section 6.3.5.

6.1.2 Highway Entities

Highway network description is decomposed into a number of highway entities. These are:

`Lane`:	A portion of the road wide enough for one vehicle.
`Section`:	A portion of the road consisting of several lanes.
`Junct2to1`:	A portion of the road, where two sections merge into one.
`Junct1to2`:	A portion of the road, where one section splits into two.
`EntrySection`:	A particular type of section where vehicles are allowed to enter the highway.
`ExitSection`:	A particular type of section where vehicles are allowed to leave the highway.
`Segment`:	A consecutive set of sections that do not contain any junctions.
`Sink`:	An ending point for highways.
`Source`:	A beginning point for highways.
`Entry`	An on-ramp for entering the highway.
`Exit`	An off-ramp for exiting the highway.
`Zone`:	A part of the highway network, containing sinks, sources, segments, and junctions.

The highway entities are organized in the inheritance hierarchy described in Figure 6.2. Note that `LaneContainer`, `Generator`, `Absorber`, and `Junction` are introduced as abstract base classes. The `SmartObject` class is discussed in Section 6.1.4.

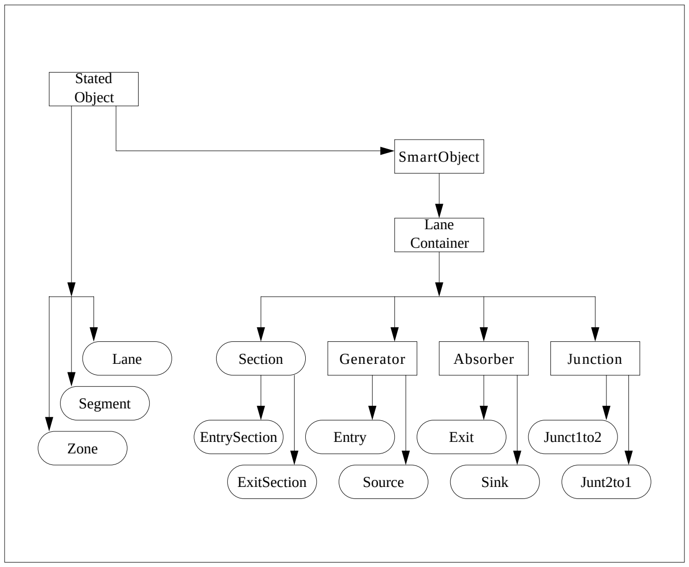

Figure 6.2: Class Hierarchy for Highway Entities

The containment hierarchy for these entities is given in Figure 6.3. Since the containment hierarchy is a one-to-many relationship, cardinality is indicated on the containee's side only.

Finally a number of binary relationships among highway entities are summarized in Table 6.1. These relationships specify how instances of highway classes can be connected to create highway networks. The first column of the table has the attribute name that identifies the relationship in a given class. Relationships are implemented by `Link` and `VVList` attributes; Table 6.1 omits the attribute type definitions. The second column identifies the second class participating (or classes that can participate) in the relationship. The third column of the table has the attribute name that identifies the relationship in the second participant. If a relationship is between two instances of the same class, the relationship is listed only once. The fourth column describes the cardinality of the relationship.

Cardinality is given as $a : b$ where $a, b \in \{0, 1, 0|1, n, m\}$. A cardinality of the form $a : b$ denotes that the first class can participate in a such relationships, whereas the second in b. A `LaneContainer` (LC) may be related zero or one `LaneContainer`s through its `prevLC1` relationship. The reciprocal of this particular relationship is `nextLC1`. A `LaneContainer` may be related to one or zero `LaneContainer`'s through its `nextLC1` relationship.

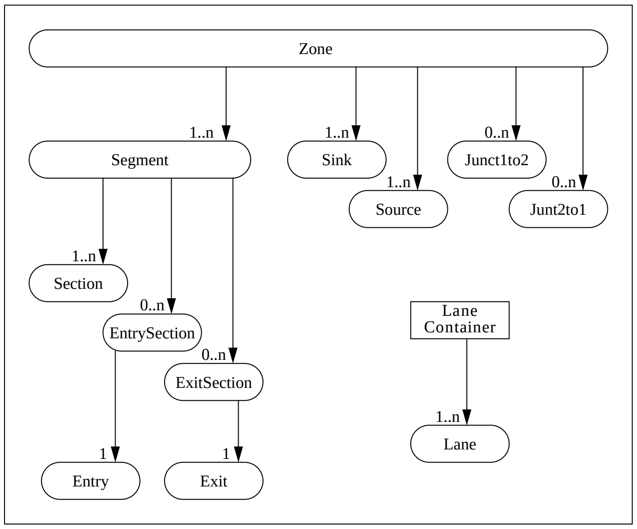

Figure 6.3: Containment Hierarchy for Highway Entities

Some relationships are specified in a base class and then specialized in subclasses. A `Generator`, for example, further restricts the `prevLC1` relationship and states that one `Generator` can participate in zero such relationships.

A highway network consists of multiple `Zone`s; the `Zone`s provide the basis for distribution and define domain boundaries. However, for simplicity, in what follows, we assume that the highway network consists of a single `Zone`.

Most highway entities have little behavior of their own and are used to create highway networks only. Their behavior is derived from the automation devices they are configured with.

We now discuss the highway entity methods that are used by the physical layer scheduling objects to create and move the `Vehicle`s along the highway. These methods are revisited in Section 7.2.1.

Lane

The static relationships of `Lane` were discussed above. The `Lane`'s dynamic state attribtues are defined in the `LaneState` class. The `Lane` specializes the `currState` relationship of `StatedObject` by its `laneState` attribute as discussed in Section 5.3.6. The `LaneState` attributes are:

`LaneState Attributes`

Relations		
`VVList<Vehicle>`	`myVehicles`	List of Vehicles in the Lane.
`VVArray<Vehicle>`	`cellArray`	List of Vehicles in the Lane indexed by their position along the Lane.

The methods that are used to move the Vehicles along the highway are:

```
int                MoveVehiclesInLane     ();
```
This method updates the `cellArray` based on the current `xP` and `yP` values of the `Vehicle`, and sets the `Vehicle`'s `absDist`, `currLane`, and `cellId` attributes. It moves `Vehicles` to the next `LaneContainer` if they have moved beyond the end of this `Lane`.

The `Lane`s have curvature. The `Vehicle` movement methods encapsulate the curvature of the `Lane`, so methods and attributes regarding `Lane` curvature are not discussed.

Entity	Relationship	With Entity	Reciprocal	Cardinality
Lane	prevLane	Lane	nextLane	0|1: 0|1
Zone	prevZones	Zone	nextZones	n:m
Segment	prevLC	Junction | Source	nextSegment1 | nextSegment2	1:1
	nextLC	Junction | Sink	prevSegment1 | prevSegment2	1:1
Lane Container	prevLC1	LaneContainer	nextLC1	0|1:0|1
Junction	prevSegment1	Segment	nextLC	1:1
	nextSegment1	Segment	prevLC	1:1
	prevLC1	Section	nextLC1	1:1
	nextLC1	Section	prevLC1	1:1
Generator	prevLC1	NONE		1:0
Absorber	nextLC1	NONE		1:0
Section	prevLC1	Junction | Section	nextLC1	1:1
Section	nextLC1	Junction | Section	prevLC1	1:1
EntrySection	prevLC2	Entry	nextLC1	1:1
ExitSection	nextLC2	Exit	prevLC1	1:1
Junct2to1	prevSegment2	Segment	nextLC	1:1
	prevLC2	Section	nextLC1	1:1
Junct1to2	nextSegment2	Segment	prevLC	1:1
	nextLC2	Section	prevLC1	1:1
Sink	prevSegment1	Segment	nextLC	1:1
	prevLC1	Section	nextLC1	1:1
Source	nextSegment1	Segment	prevLC	1:1
	nextLC1	Section	prevLC1	1:1
Exit	nextLC1	NONE		1:0
	prevLC1	ExitSection	nextLC2	1:1
Entry	nextLC1	ExitSection	prevLC2	1:1
	prevLC1	NONE		1:0

Table 6.1: Relationships Among Highway Entities

LaneContainer

`LaneContainer` provides one method for Vehicle movement `int MoveVehicles()`. This method calls `MoveVehiclesInLane()` on each of the `Lane`s of the `LaneContainer`.

Generator

`Generator`s are the creation source of `Vehicle`s. They contain `Factory` and `InTraffic` objects. (See Figure 6.6.) The `Generator`'s state depends on its `InTraffic`'s `State`.

`Generator` provides one method: `int GenerateVehicles()`. When invoked this method may create a new `Vehicle`. The `InTraffic`'s `AnyVehicle()` method decides whether to create a `Vehicle` and if so what type of `Vehicle` to create. The actual `Vehicle` creation is performed by the `Factory`.

Absorber

All `Vehicle`s eventually reach an `Absorber` and leave the highway. The `Absorber` contains an `OutTraffic` object which models the road conditions outside the highway. The `Absorber`'s `State` depends on its `OutTraffic`'s `State`.

The `Absorber` provides one method: `int AbsorbVehicles()`. This method implements the necessary bookkeeping for deactivating `Vehicle`s, and invokes the `OutTraffic`'s `TakeVehicle` method.

`Vehicle`s that leave the highway are "deactivated" and written to database. Deactivation `Terminate`s all state machines (See Section 6.3.4.) and sets the `logLevel` and `errorLevel` of automation devices to zero.

6.1.3 Automation Devices

`Sensor`s, `Control`lers, `Receiver`s, `Transmitter`s, and `Monitor`s are added to vehicles and to the roadside for automation and evaluation. These five entities are called automation devices.

Longitudinal and lateral `Sensor`s provide information about the environment such as distance and speed to the `Vehicle` in front, or to the left. `Transmitter`s and `Receiver`s are used for communicating with other objects. Example `Control` objects are regulators that determine speed, and coordinators that select maneuvers to perform. `Monitor`s are read-only entities that collect statistics.

SmartAHS provides only abstract base classes for these objects and expects the application developers to fully develop their behavior in subclasses.

Control

Control objects use the `Input` and `Output` classes to define their input and output attributes. None of their public methods take any arguments.

An `Engine` object in the `Vehicle`, for example, accepts `xJerk` and `yJerk` as inputs, and when enabled generates `xP`, `xVel`, `xAccel`, `yP`, `yVel`, and `yAccel` as outputs.

Control objects can be time driven, event driven, or both. The event driven controllers utilize state machines and their discussion is deferred to Section 6.3. The time driven controllers specialize the `Run()` method. This method is invoked by the process layer in charge of scheduling the extent of a given `Control` class. An example `Run()` method is discussed in Section 7.2.2.

Transmitter and Receiver

In SmartAHS transmitters and receivers are modeled as zero-delay devices that deliver messages with full reliability. Application developers have the ability to implement more detailed transmitter and receiver models by specializing their methods,

The communication channel between objects is not part of SmartAHS simulation. Objects establish a communication link by using their `Sensor`s. An object that wants to send a `Message` to another object locates and identifies its target using its `Sensor`. In particular the `Sensor` returns a `Link` to the `ReceiverProxy` of the target object.

Since transmitters and receivers are mainly used for event delivery we defer their discussion to Section 6.2.3.

Sensor

In SmartAHS `Sensor`s are used to establish communication channels between objects. To send a `Message` to another object, the sender uses its `Sensor` to get the `Receiver`'s identifier.

SmartAHS provides a `VehicleSensor` that can locate the `Vehicle`s to the `front, back, left`, and `right` of a given `Vehicle`. The `VehicleSensor` models a sensor with full accuracy within a given sensing distance. The sensing distance of the `VehicleSensor` is a runtime parameter.

Application developers can specialize the SmartAHS `VehicleSensor` and its methods to implement more accurate sensor models.

The `VehicleSensor` has the following static attributes that set its sensing distance in each direction:

VehicleSensor Static Attributes	
`static o_float`	frontDistance.
`static o_float`	leftDistance.
`static o_float`	rightDistance.
`static o_float`	backDistance.

The `VehicleSensor` provides the following Method:

```
Link<Vehicle>      GetVehicle             (eSENSOR_SIDE side,
                                           eSENSOR_LANE lane,
                                           o_float &deltaX,
                                           o_float &deltaVel) const;
```

This method determines the `deltaX` and `deltaVel` to the first `Vehicle` in the `side` direction within the sensing range. `eSENSOR_SIDE` and `eSENSOR_LANE` are enumerated data types. `side` can be one of { `eFront, eLeft, eBack, eRight` }. `lane` can be one of { `eThisLane, eAdjacentLane, eNextToAdjacentLane` }.

Subclasses of `Sensor` are implemented by application developers and specialize the methods based on the accuracy of the particular sensor being modeled.

Monitor

`Monitor`s are read only objects and their behavior does not effect the simulation. Their execution is scheduled on an as needed basis. The behavior of a `Monitor` is executed by invoking its `Run()` method.

Like with any other object, a process layer can be used to schedule the execution of all `Monitor`s of a given type.

Alternatively, a single `Monitor` instance can be activated during simulation. In this case the `Monitor` uses its `myBigBen`[3] to periodically schedule its execution. A `Monitor` can be activated through its parametric interface on instantiation or by using the graphical debugger.

The history of a `Monitor` is recorded like any other object. Turning the logging on for a `Monitor` automatically activates it.

The `Monitor` class implements the following methods:

```
virtual int        SetErrorLevel          (o_u1b newLevel);
```

A `Monitor` is activated if `newLevel` is one or higher. A `Monitor` is deactivated if `newLevel` is zero and logging is off. When activated the `Monitor` registers a timeout request with its `myBigBen`.

[3]The `BigBen` class is described in Section 6.2.1

```
virtual int        SetLogLevel            (o_u1b newLevel);
```
The logging of a `Monitor` is turned on if `newLevel` is one or higher. Turning logging on automatically activates the `Monitor`. If logging is turned on, the `Monitor` registers a timeout request with its `myBigBen`. Logging is turned off if `newLevel` is zero.

```
virtual int        WakeUp                 (o_u4b handle,
                                           Link<Message> state);
```
This method is invoked by `myBigBen` when a timeout request expires. The method invokes the `Run()` method. If logging is on, it invokes the `SaveState()` method. Finally, it registers a new timeout request with the `myBigBen`.

```
virtual int        Run                    ();
```
This method has no behavior in the base class and must be specialized by subclasses.

Subclasses of `Monitor` define the particular `State` attributes for a `Monitor` and implement the `Run()` method that sets the `State` values. The `Monitor` objects have free access to all objects' inputs, outputs, and state.

6.1.4 SmartObject

Automation devices are either on the roadside or in a `Vehicle`. We formalize this relationship with the abstract entity `SmartObject`. `SmartObject` and its containees are depicted in Figure 6.4.

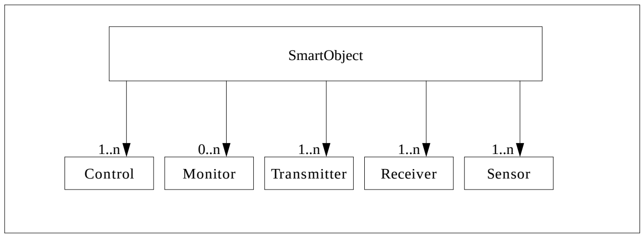

Figure 6.4: The `SmartObject` and its containees.

The automation devices are specialized based on the automation strategy and based on the type of `SmartObject` they are contained in. The particular relations among automation devices in a given `SmartObject` are specified by application developers.

6.1.5 Vehicle

The `Vehicle` is a composite object, i.e., it is composed of many other objects. Its state and behavior is derived from its components. State attributes that are shared by its automation devices are specified in the `VehicleState` class. The `Vehicle` specializes its `currState` relation by its `vehicleState` attribute. The `Vehicle` does not distinguish between input, output, and state attributes. Automation devices specify which attributes they treat as inputs and which ones as outputs.

The `Vehicle` moves along the highway. All relationships based on the location of `Vehicle`s are dynamic. At a given time a `Vehicle` is in a given cell of the `cellArray` of a given `Lane`. A `Vehicle`'s position in a `Lane` is given by a pair of coordinates `xP` and `yP`. These coordinates define the location of its right front bumper. The `Vehicle`'s position on the highway is maintained by SmartAHS classes. Although at a given time a `Vehicle` is in one cell of one `Lane`, its body can cover up to four cells, four `Lane`s, and two `Section`s. (See Fig 6.5).

Subclasses of `Vehicle` are expected to provide constructors that configure the `Vehicle` with automation devices as discussed in Section 7.1.1.

`Vehicle` has a number of static state attributes these are:

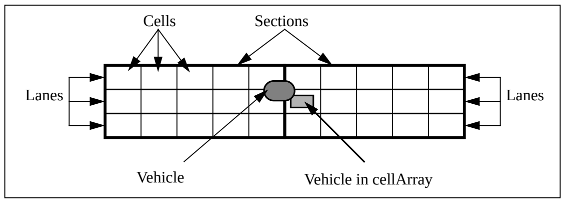

Figure 6.5: Logical Representation of Vehicle in Lane

`Vehicle Static Attributes`

Value		
`o_float`	`length`	Vehicle length.
`o_float`	`width`	Vehicle width.
Relations		
`PString`	`origin`	The name of its generator.
`PString`	`destination`	The name of its intended Absorber.
`Link<VehicleState>`	`vehicleState`	A Link to its State.

`VehicleState` inherits from State. It contains basic continuous and discrete state information about `Vehicle`s. `Vehicle` subclasses may use this class directly or subclass it further to add other attributes. Within a vehicle, automation devices can read and write `VehicleState` values. Between two `Vehicle`s access is limited to Sensors only. `VehicleState` attributes are:

`VehicleState Attributes`

Values		
`o_float`	`xP`	x position in a cell array, used as input.
`o_float`	`yP`	y position in a cell array, used as input.
`o_float`	`xVel`	x Velocity, used as input.
`o_float`	`yVel`	y Velocity, used as input.
`o_float`	`xAccel`	x Acceleration, used as input.
`o_float`	`yAccel`	y Acceleration, used as input.
`o_float`	`xJerk`	x Jerk, used as input.
`o_float`	`yJerk`	y Jerk, used as input.
`o_float`	`absDist`	x position in a `Lane`.
`o_float`	`oldXP`	x position, used as output.
`o_float`	`oldXP`	y position, used as output.
`o_float`	`oldXVel`	x Velocity, used as output.
`o_float`	`oldYVel`	y Velocity, used as output.
`o_float`	`oldXAccel`	x Acceleration, used as output.
`o_float`	`oldYAccel`	y Acceleration, used as output.
`o_float`	`oldXJerk`	x Jerk, used as output.
`o_float`	`oldYJerk`	y Jerk, used as output.
`o_float`	`oldAbsDist`	x position in a `Lane`, used as output.
Relations.		
`Link<Lane>`	`currLane`	The `Lane` it is contained in.
`o_u4b`	`cellId`	The `cellArray` index of the `currLane`.

The `VehicleState` maintains the current and the last continuous state attributes. The current values are set by appropriate control devices within the `Vehicle`. The last values are used as outputs to other automation devices within the `Vehicle` or to `Sensors` in other `Vehicle`s. The current values are copied to last values at the end of each physical layer transition.

A `Vehicle`'s position on the highway is maintained by SmartAHS. In particular the `Lane` method `MoveVehiclesInLane()` sets the `absDist`, `currLane`, and `cellId` attributes of the `Vehicle`.

In the Varaiya/Shladover automation strategy a `Vehicle` is either a *single agent* or it is in a *platoon*. If a `Vehicle` is in a platoon, it can be a *leader* or a *follower*. At all times a `Vehicle` is performing a *maneuver*. Sample maneuvers are lead, follow, split, merge, and lane change.

The following attributes are added to `VehicleState` to support these concepts.

`VehicleState Attributes for SmartPATH`

Values		
PString	`currManeuver`	Type of current maneuver, identified by state machine name
o_bool	`amILeader`	True if `Vehicle` is a single agent or it is a platoon leader.
Relations		
`VVList<ReceiverProxy>`	`myPlatoon`	Identifies the followers if `Vehicle` is a platoon leader.
`Link<ReceiverProxy>`	`myLeader`	Identifies the platoon leader if `Vehicle` is a follower.

The automation devices are expected to maintain these attributes.

6.1.6 Traffic Entities

`Vehicle`s are created and deleted in the `Generator` and `Absorber` objects based on incoming and outgoing traffic patterns. The traffic entities `Factory`, `InTraffic`, and `OutTraffic` are used to implement this functionality.

`InTraffic`s are responsible for generating incoming traffic patterns, `OutTraffic`s are responsible for generating outgoing traffic patterns. These objects may in fact provide gateways to other urban traffic simulation packages for more detailed traffic modeling. `Factory` objects have the knowledge to create various types of `Vehicle`s. The location of these entities in the inheritance and containment hierarchy is given in Figure 6.6.

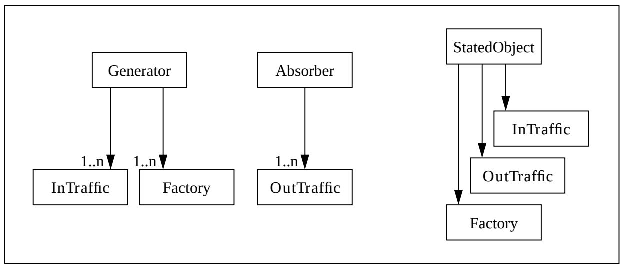

Figure 6.6: Inheritance and Containment Hierarchy for Traffic Entities

Factory

The abstract base class `Factory` provides one method:

`virtual Link<Vehicle>`	`MakeVehicle`	`(VEH_TYPE vehType);`
This method creates a subclass of `Vehicle` identified by `vehType`.		

The `vehType` is an enumerated integer identifying the `Vehicle` subclass; the user of a given subclass of Factory must know what `Vehicle` subclass a given `vehType` corresponds to.

Subclasses of `Factory` specialize the `MakeVehicle()` method and make an explicit call to the constructor of a particular `Vehicle` subclass.

OutTraffic

The `OutTraffic` absorbs `Vehicle`s leaving the highway. Its `State` reflects the conditions off the highway. The `Absorber`'s `State` depends on its `OutTraffic`'s `State`.

It is the responsibility of the application developer to design control algorithms that would ensure that the departing `Vehicle`s meet the `OutTraffic`'s requirements such as speed, spacing, maneuver etc.

The `Absorber` "deactivates" a `Vehicle` that leaves the highway and writes it to the database.

An `OutTraffic` can model multiple lanes. It has the following virtual method:

```
virtual int        TakeVehicle            (Link<Vehicle> vehicle,
                                           int laneId,
                                           Link<Lane> containingLane);
```

This method moves `vehicle` from `containingLane` to the `OutTraffic` lane identified by `laneId`.

InTraffic

The `InTraffic` generates `Vehicle`s entering the highway, it determines the arrival rate and type of `Vehicle`s. Its `State` reflects the conditions off the highway.

It is the responsibility of the application developer to design control algorithms and check-in protocols that would ensure that the incoming `Vehicle`s meet the highway requirements, such as speed, spacing, and maneuver or the presence of certain automation devices.

An `InTraffic` can model multiple lanes. It has the following virtual method:

```
virtual Link<Vehicle>        AnyVehicle      (int laneId,
                                              o_u4b numOfLanes,
                                              Link<LaneState> receiving);
```

Based on the `receiving` `LaneState`, this method may create a `Vehicle` in lane `numOfLanes`. Its return value is `NULL` if no `Vehicle` is created.

6.2 System Architecture, Entity Specifications

In this section we discuss the time and event driven scheduling mechanism implemented by SmartAHS.

6.2.1 Packets, Events, and Messages

This section describes the building blocks of inter-object communication. SmartPATH objects communicate with `Event`s or `Message`s. The base class `Packet` captures the common features of `Event`s and `Message`s.

The creator of a `Packet` has the responsibility to set the `Receiver` of the `Packet`. It passes the `Packet` to its `Transmitter`, which in turn puts it in an appropriate `PacketBox`.

All `Packets` are channeled through a `PacketBox`. Currently `PacketBox` is one big table of `Packet`s with little other behavior. In future releases, this object may support filter processing for event notification.

Each `PacketBox` is in a process layer. When the process layer is scheduled for execution each `Packet` is delivered to the specified `Receiver`. This ensures that within an event driven process layer only objects with an outstanding `Packet` exercise any behavior.

Objects can send `Packet`s to themselves.

The `BigBen` class provides a timer that delivers `Packet`s after a specified time interval. To register a timeout request with a `BigBen`, an object specifies a `Packet` and the number of time clicks after which the `Packet` should be delivered.

Each `BigBen` is contained in a process layer or in the process coordinator. When a `BigBen` is advanced by a time step by its container, it is also asked to deliver all `Packet`s that correspond to timeout requests that expire at that time.

We now discuss the classes involved in event delivery. The class hierarchy for these classes is given in Figure 6.7.

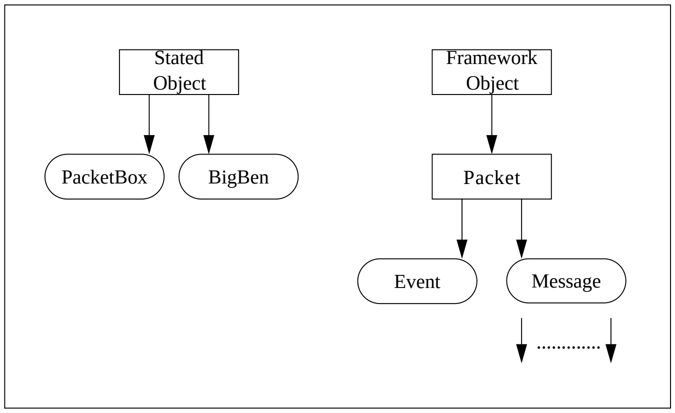

Figure 6.7: Class Hierarchy for Event Management Entities

Packet

The abstract base class `Packet` has the following relation attributes.

Packet Static Attributes		
Relations		
`Link<Receiver>`	`msgTo`	`Packet` to be sent to the `Receiver` identified by `msgTo` .
`Link<Receiver>`	`rspTo`	Receiver responds to the `rspTo` field of `Packet`

The `fullType` of a `Packet` is given by three parts `Type, Subtype` and `Id`. These fields are used for the categorization of `Packets`. `Packet` does not impose any semantics to them.

Currently `Type` is used to identify whether the `Packet` is a `Message` or an `Event`. `Id` is used to identify the particular `Message` or `Event` type. `Subtype` is not used.

The following static methods are provided to set and get the type of a packet:

```
static o_u1b       GetType                (o_u4b packetType);
```
This method returns the `Type` of a `Packet` given its full type.

```
static o_u1b       GetSubtype             (o_u4b packetType);
```
This method returns the `Subtype` of a `Packet` given its full type.

```
static o_u4b       GetId                  (o_u4b packetType);
```
This method returns the `Id` of a `Packet` given its full type.

```
static o_u4b       MakePacketType         (o_u1b type, o_u1b subtype, o_u4b id);
```
This method constructs a full type given `type, subtype`, and `id`.

The `Packet` constructor takes the packet type as an argument.

Event

The `Event` class extends the `Packet` class by an `o_bool status` attribute. This attribute can be used to set the event on or off.

Message

The `Message` does not specialize the `Packet` class in any way. The subclasses of `Message` specialize it by adding attributes.

Message and its subclasses are instantiated by `SMInstance` objects as discussed in Section 6.3.4. The subclass constructor is overloaded, the `PLMessage`, for example, is given below:

```
                   PLMessage              (o_u4b fullType
                                          Link<SMInstance> csi);
```
The constructor uses the `Inputs`, `Outputs`, and the `State` of `csi` to derive the attribute values of the `PLMessage`.

BigBen

The `BigBen` class implements clocks and provides timer services. Among other bookkeeping constructs, the `BigBenState` has the attribute `o_u4b currTimeClick` corresponding to the current time.

`BigBen` has the following methods:

```
o_u4b              WakeMeUp               (o_u2b clicks,
                                          Link<FrameworkObject> owner,
                                          Link<Message> clientData);
```
The `owner` of the request, wants to be woken up in `clicks` time clicks with the `clientData` `Message`. The return value is an identifier for this request.

```
int                ClearWakeUp            (o_u4b handle);
```
This method cancels the wake-up request identified by `handle`.

```
o_u4b              GetTime                ();
```
This method returns the value of the `currTimeClick`, i.e., the current time.

PacketBox

The `PacketBox` class stores and delivers `Packets`. The details of the bookkeeping constructs are omitted.

6.2.2 Scheduling Mechanism

In SmartAHS the `ProcessCoordinator` manages the execution of process `Layer` objects. Arbitrary number of process `Layer`s register with the `ProcessCoordinator` and specify the period with which they want to be scheduled. If a process `Layer` wants to be scheduled on an event driven basis, it sets its period to infinity[4].

The `ProcessCoordinator` loops through its `Layer`s and executes them according to their period. After every loop, it checks if there are any `Event`s requesting the scheduling of event driven `Layer`s.

SmartAHS provides three abstract classes `Traverser`, `EventDriver`, and `Hybrid` that subclass `Layer`. Any `Layer` can request to be scheduled on a time or event driven basis.

All `Layer`s contain a `BigBen`. They increment the `BigBen` when scheduled and ask it to deliver all outstanding timeouts.

The `Traverser` traverses the `Zone` from `Sink` to `Source`. For each `LaneContainer` along the traversal, it invokes a sequence of virtual methods. These methods are specialized by the `Traverser` subclasses and invoke the time driven evolution methods of the proper objects within the `LaneContainer`.

The `EventDriver`s contain a `PacketBox` and deliver all `Packet`s in the `PacketBox`. Note that the delivery of a `Packet` may result in more `Packet`s being placed into the `PacketBox` in response to the delivered `Packet`s.

The `Hybrid`s exercise both time and event driven behavior.

The `ProcessCoordinator` and the `Layer`s are Sequencers, as discussed in Section 4.2.3. The delivery of each timeout request in a `Layer` and the delivery of each `Packet` in an `EventDriver` is considered to be a transition. Each virtual method on a `LaneContainer` in the `Traverser` corresponds to at least one transition. Subclass specializations of these methods may choose to execute a number of transitions in these methods.

We now discuss the scheduling objects in more detail. Their inheritance hierarchy is given in Figure 6.8. Their containment hierarchy is given in Figure 6.9.

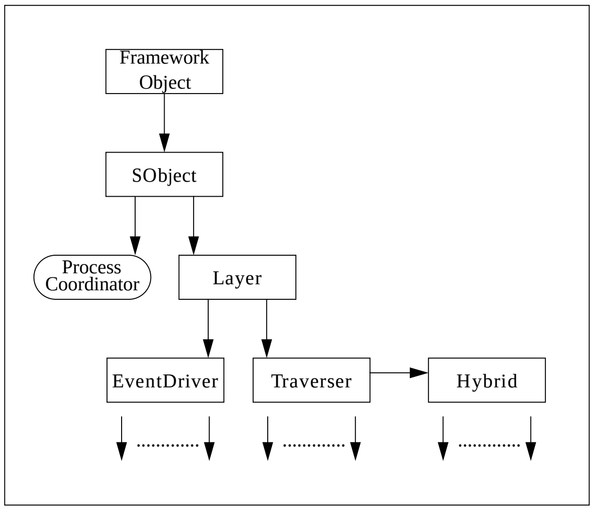

Figure 6.8: Class Hierarchy for Scheduling Entities

SObject

This abstract base class contains a `Link<Zone> myZone` attribute identifying the `Zone` that a given scheduling object is managing. It provides the necessary private methods for proper initialization of this attribute.

[4] In implementation, infinity is just a very large number.

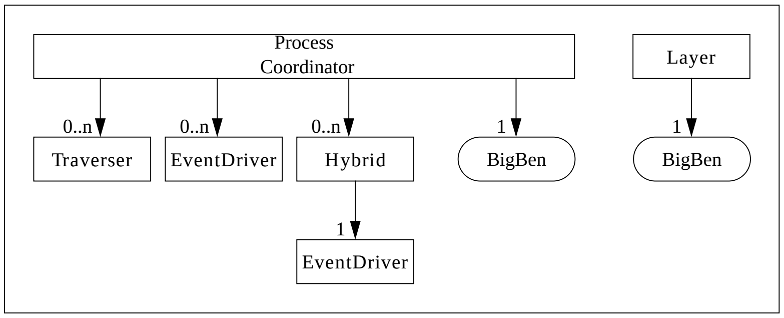

Figure 6.9: Containment Hierarchy for Scheduling Entities

ProcessCoordinator

The `ProcessCoordinator` implements the main control loop. It allows a number of `Layer` objects to register with it at a given period and manages the execution of these scheduling objects. The `ProcessCoordinator` also manages the global clock.

The `ProcessCoordinator` derives a `duration` from the periods of its `Layers`. When scheduled for execution, the `ProcessCoordinator` runs for `duration` clicks of the global clock.

The internal data structures that keep track of `Layers`, their periods, and execution times are not discussed.

The `ProcessCoordinator` provides the following methods.

```
int            Register            (Layer* aLayer,
                                    char* name,
                                    o_4b period);
```
This method registers the `named` `aLayer` for execution with the specified `period`.

```
int            DeRegister          (Layer* aLayer);
int            DeRegister          (char* layerName);
```
These methods stop the execution of the Layer identified by `aLayer` or `layerName`.

```
int            WakeUp              (char* layerName,
                                    int clicks);
```
This method schedules the `layerName` for execution in `clicks` time clicks. It is used for event driven layer scheduling.

```
int            Ping                (char* layerName);
```
This method schedules the `layerName` for immediate execution. This means that the `layerName` execution is scheduled and completed before this method returns. `Ping` is currently not used.

```
int            virtual Go          ();
```
This method schedules the `ProcessCoordinator` for execution. It runs for `duration` time clicks of the global clock, scheduling the execution of its `Layers` at each click, as appropriate.

Layer

The `Layer` abstracts the `Go()` method of its subclasses. It also holds common attributes such as `o_4b period`, `PString layername`, and `Link<BigBen> myBigBen`.

The `Layer` subclasses have the responsibility of scheduling the execution of their `myBigBen` in their `Go()` method.

Traverser

The `Go()` method of the `Traverser` traverses the `Zone` from `Sinks` to `Sources`.[5] For each `LaneContainer`, it invokes three virtual methods: prepare, execute, and finish.

The traversal guarantees the following conditions:

[5] Current implementation of the `Traverser` assumes that there are no loops within a `Zone`.

1. When a `LaneContainer` is executed its `prevLC`s will have already been prepared.

2. When a `LaneContainer` is executed its `nextLC`s will have already been executed, but not finished.

For `LaneContainer`s of type `EntrySection` and `ExitSection`, the `Traverser` also invokes prepare, execute, and finish methods on the corresponding `Entry` and `Exit`s.

The virtual methods of the `Traverser` are:

virtual	Start	()
virtual	End	()
virtual	ELC	(Link<LaneContainer> LC)
virtual	PLC	(Link<LaneContainer> LC)
virtual	FLC	(Link<LaneContainer> LC)
virtual	EAbsorber	(Link<Absorber> LC)
virtual	PAbsorber	(Link<Absorber> LC)
virtual	FAbsorber	(Link<Absorber> LC)
virtual	EGenerator	(Link<Generator> LC)
virtual	PGenerator	(Link<Generator> LC)
virtual	FGenerator	(Link<Generator> LC)

`Start()` is invoked before the `Zone` traversal is started, `End()` is invoked after the `Zone` traversal is finished. `ELC,PLC` and `FLC` are used for all `LaneContainers` that are not `Generators` and `Absorbers`. Special methods are used for these `LaneContainer` subtypes.[6]

These methods have no behavior in the `Traverser`, and its subclasses are expected to specialize them as needed. The `Physical` layer, for example, uses the `ELC()` method to move each `Vehicle` in a `LaneContainer` based on the displacement generated in the `Regulation` layer. The `Physical` layer, a subclass of `Traverser`, is discussed in Section 7.2.1.

The exact traversal algorithm is not discussed here. The `Traverser` class code can be found in Appendix 9.3.

The execution of the `myBigBen` precedes the `Zone` traversal in the `Go()` method.

EventDriver

The `EventDriver` contains a `PacketBox`. Its `Go()` method delivers all `Packet`s in the `PacketBox` until the `PacketBox` is empty. The delivery of a `Packet` may result in more `Packets` being placed in the `PacketBox`. Current `EventDriver` implementation assumes that the application development will guarantee that `Packet` generation will terminate.

The execution of the `myBigBen` precedes the `Packet` delivery in the `Go()` method.

Hybrid

A `Hybrid` displays both `Traverser` and `EventDriver` behavior. It specializes the `Start` method to schedule the execution of the contained `EventDriver`.

Note that, rather than multiply inheriting from `Traverser` and `EventDriver`, this class only uses the `EventDriver`.

6.2.3 Transmitters and Receivers Revisited

The class hierarchy for communication entities is given in Figure 6.10.

We now discuss these classes.

[6]Note that the overloading implementation in C++ does not provide overloading based on subclass (signature resolution takes place at compile time, not at run time.). As such the `ELC` method, for example, can not be overloaded to invoke the method of correct signature based on the `LaneContainer` subclass with which it is invoked. We have to resort to using a different method name.

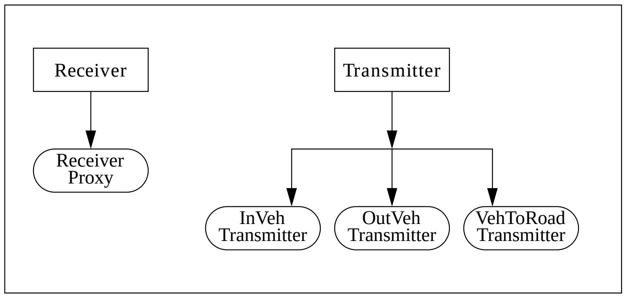

Figure 6.10: Class Hierarchy for Communication Entities

Transmitter

Every `Transmitter` has a primary `PacketBox` identified by the `myPacketBox` attribute. Unless otherwise specified `Packet`s are delivered to this `PacketBox`.

It provides three delivery methods

```
virtual int        SendPacket             (Link<Packet> packet);
```
This method puts the `Packet` in the `myPacketBox`.

```
virtual int        SendPacket             (Link<Packet> packet,
                                           int destination)
```
This method is expected to be specialized by subclasses.

```
virtual int        SendPacket             (Link<Packet> packet,
                                           char* layerName,
                                           int destination)
```
This method is expected to be specialized by subclasses.

InVehTransmitter

This `Transmitter` specializes the third `SendPacket` method to properly channel the `Message`s within a `Vehicle`. It uses the `Coordination` layer and `Regulation` layer `PacketBox`es.

OutVehTransmitter

This `Transmitter` is specialized to use the `Coordination` layer `PacketBox` for `Packet` delivery.

Receiver

The abstract class `Receiver` provides one virtual method `ReceivePacket(Link<Packet>)` which is expected to be specialized by its subclasses.

ReceiverProxy

The `ReceiverProxy` implements a `Receiver` with a buffer. Arbitrary number of classes can share a `ReceiverProxy`.

The `ReceiverProxy` provides two service modes: *waiting mode* and *listening mode*. Objects register with the `ReceiverProxy` by specifying the service mode and the type of Packets they want to listen to/wait for.

If an object is waiting for a `Packet` and the `Packet` arrives, the `ReceiverProxy` notifies the object.

If an object is listening to a `Packet` and the `Packet` arrives, the `ReceiverProxy` buffers the `Packet`. If the object starts waiting for a `Packet` that has been buffered, it is immediately delivered the buffered `Packet`. If the object stops listening to a `Packet`, any such buffered `Packet`s are discarded.

Between the two services, waiting has priority. If the same object is both listening to and waiting for a `Packet` and the `Packet` arrives, the `Packet` is delivered to the object without buffering.

If several objects are waiting for the same `Packet` and the `Packet` arrives, each object is delivered a duplicated copy of the `Packet`.

The methods `Listen`, `ClearListen`, `Wait` and `ClearWait` allow an object to use the services of the `ReceiverProxy`. The bookkeeping methods of `ReceiverProxy` are not discussed.

6.3 The State Machine Language

Finite state machines are based on quintuples that define state, alphabet, transitions, begin state, and end states. The quintuples define a single automaton or state machine. Algebraic operators are then used to build more complex automata from these basic quintuples. One such operator is the synchronous composition operator which requires that two automata take transitions jointly if the transition is defined for a common symbol of their alphabet. Semantically this operator can also be interpreted as an input-output connection between two automata.

Extended automaton descriptions sometimes partition the alphabet into input, output, and internal symbols to represent input-output behavior. Again an operator is defined that synchronizes output generation and input consumption of two automata.

Recall the discussion in Section 2.2. Automata descriptions are object based. An automaton is rarely thought of as an instantiable class. System descriptions consist of a number of automata that are combined with algebraic operators. Usually, the automata capture the behavior of objects, the algebraic operators represent their relationships. Such a static description has limitations in those real life applications where the relationships themselves have to evolve.

Consider a number of vehicles moving on a highway. Assume each vehicle's behavior is given by an automaton with inputs and outputs. Assume each vehicle can interact with its front, back, left, and right vehicle. These other vehicles may be input providers or output consumers. At any given time the list of vehicles with which a given vehicle interacts depends on the physical position of vehicles on the highway; clearly, this list is not static.

In this example, assume the alphabet of the automaton describing a vehicle has a number of input and output symbols. These symbols are independent of the other objects with which they are communicated. To model such interaction among objects with traditional automata, one must define a new element of the alphabet for every pair of objects in the system. This results in an explosion of alphabet size for large systems.

Hybrid automata are used to model systems with both continuous and discrete behavior. Whereas analysis and verification tools are readily available for finite automata, most questions concerning hybrid automata are proving to be undecidable. Most analysis tools rely on proper abstraction of the continuous components of the model and provide only partial system verification. Simulation is used to explore the full behavior of the hybrid automaton.

Unfortunately, the interoperation between the verification and the simulation tools is limited. A language used to represent the full behavior of a hybrid automaton is rarely compatible with the languages used to represent verifiable abstractions. Manual translations from one representation syntax into the other reduce the reliability of the verification and simulation process.

In this section we define an input-output based *State Machine Language* which distinguishes a) the input-output representation of an automaton from the mechanism that establishes the input-output interconnections; and b) the continuous behavior from the discrete behavior. Furthermore, these automata are class based, i.e., the state machine descriptions correspond to instantiable entities.

Automata specified in this language can be compiled to become SmartAHS entities for simulation. Since in simulation both continuous and discrete behavior are fully exercised, automata behavior in SmartAHS is deterministic.

Alternatively, these automata can be translated into COSPAN representations.[7] Since the verification tools exercise discrete behavior only, events and input-output reconfigurations resulting from continuous behavior become nondeterministic. Nonetheless, partial verification of the discrete behavior of the original automaton is streamlined.

In the following, we describe the basic syntax for this language and its associated semantics. We then discuss its implementation and use in SmartAHS. The SmartAHS implementation of the SML language introduces syntactic sugar to facilitate specification ease.

6.3.1 Basic Syntax

A State Machine (SM) class consists of the following components:

[7] Translation into COSPAN has not yet been carried out.

Machine Name	The name of this instance.
States	The list of states.
Input Events	The inputs resulting from continuous behavior.
Input Messages	The inputs resulting from other SMs.
Output Messages	The outputs sent to other SMs.
Active Partners	List of variables to keep track of other SMs.
Passive Partners	List of variables to keep track of other SMs.
Begin State	Begin state for this SM.
Transitions	A set of transitions for this SM.

A Packet is defined as a triple, *(Machine Name $\times$ Message $\times$ Machine Name)*. Consider the packet: (RspTo, message, MsgTo). It represents a message sent from the SM with machine name RspTo to the SM with machine name MsgTo; message is an output message in RspTo, and is expected to be an input message in MsgTo.

An active partner is a state machine that initiates an input-output connection with this machine by sending the first packet. A passive partner is a SM with which this machine initiates a connection by sending the first packet.

The passive partners are set by a mechanism external to the SM.

A transition is defined as a quadruple,
(FromState $\times$ Input Action $\times$ Output Actions $\times$ ToState).
FromState and ToState are states. A transition is executed if its Input Action takes place. As part of the transition, the SM moves from the FromState to the ToState and generates the Output Actions.

An input action can have several forms. The specification syntax and the associated semantics are described below:

ActivePartnerName:MessageName	Transition is executed if a packet containing named message arrives. The named active partner variable is set to the sender of the packet.
ActivePartnerName=MessageName	Transition is executed if a packet containing named message arrives and it is sent by the SM identified by the current value of the named active partner variable.
PassivePartnerName=MessageName	Transition is executed if a packet containing named message arrives and it is sent by the SM identified by the current value of the named passive partner variable.
PassivePartnerName=0	Transition is executed if the named passive partner variable is not set (connected).
PassivePartnerName!=0	Transition is executed if the named passive partner variable is set (connected).
InputEventName	Transition is executed for named event.

An output action consists of a set of packets specified with the syntax:

PartnerName:OutMessageName	PartnerName is an active partner or passive partner variable name and OutMessageName is an output message.

The existence of a reliable channel is assumed. For each output action generated by a SM instance, the channel delivers the packet to its recipient as an input action.

The semantics of a SM is simple. It is initialized to its Begin State. Based on incoming events, incoming messages, or availability of connections it takes state transitions. As part of these transitions it generates output messages.

Two SM instances interact if one machine sends a packet to the other. Two definitions are possible to specify what happens if a packet arrives at a SM for which no transition is defined at the current state: 1) the packet is ignored, 2) the packet is queued until it can be used by a transition. Either definition is acceptable, however, the latter introduces a "buffer state" and complicates the semantics.[8]

The constructs used to create or delete SM instances, to set PassivePartner variables, and to generate Events are external to the SM. The specification of a State Machine is independent of the nature of these constructs.

SM specifications can be verified if passive and active partner value assignments are static and event generation is assumed to be arbitrary and nondeterministic. However, in general, a more accurate model of event generation is needed to restrict the possible behavior of the SMs.

If SM specifications are simulated, passive partner value assignments and event generation are derived from simulated behavior.

Example

Above we did not introduce the complete specification notation of a state machine, but, assuming that the notation is evident, we define the "Hello Friend" state machine.

States	Asleep, Awake, Chat, Tired;		
InputEvents	WakeUp, FallAsleep;		
InputMessages	Hello, Bye;		
OutputMessages	Hello, Bye;		
ActivePartner	AFriend;		
PassivePartner	TheFriend;		
BeginState	Asleep;		
Transitions			
Asleep	WakeUp	()	Awake;
Awake	TheFriend!=0	TheFriend:Hello	Chat;
Awake	AFriend:Hello	AFriend:Bye	Awake;
Chat	TheFriend=Bye	()	Tired ;
Chat	AFriend:Hello	AFriend:Bye	Chat;
Tired	AFriend:Hello	AFriend:Bye	Tired;
Tired	FallAsleep	()	Asleep;

The external components of this machine wake it up, put it to sleep, and determine if TheFriend is available to accept an input.

The messages exchanged with other sate machines keep it awake until this machine gets to chat with the TheFriend. Meanwhile, the machine reponds to other machines by saying Bye. Note that any state machine is capable of sending the Hello message, not only the "Hello Friend" machine instances.

In this specification, a machine may not fall asleep before it gets a response from TheFriend.

The State Machines do not use the packets to synchronize transitions. Sending a packet does not guarantee an immediate transition at the recipient. As a result, a SM may be blocked while waiting for incoming packets. In the above example, if TheFriend is Asleep when the Hello packet is sent, the SM will not be able to leave the Chat state –assuming packets are not buffered.

6.3.2 Extended Syntax

The SmartAHS version of the state machine consists of more components. We restate the State Machine definition and give its SmartAHS interpretation.

MachineName	The name of the State Machine.
Children	Any child SMs used by this machine.
SMStates	The list of state machine states.
InputEvents	The list of input events resulting from continuous behavior.
InputMessages	The list of input messages and their types.
OutputMessages	The list of output messages and their types.

[8] In SmartAHS the `ReceiverProxy` class is used to implement a buffer.

Parent	The Parent SM. Message is sent to Parent's proxy by `InVehTransmitter`.
Child	The Child SMs. Message is sent to Child's proxy by `InVehTransmitter`.
REG	Regulation Controller. Message is sent to Regulation proxy by `InVehTransmitter`.
COORD	Coordination Controller. Message is sent to Coordination proxy by `InVehTransmitter`.
LINK	The Link Layer Controller. Message is sent to the current Link's proxy by `VehToRoadTransmitter`.
PL	The Platoon Leader. Message is sent to the Platoon Leader's Coordination proxy by `OutVehTransmitter`.
PLATOON	All followers of a platoon. Message is sent to all following `Vehicle`s' Coordination proxies by `OutVehTransmitter`.

Table 6.2: Special Correspondents

Correspondents	List of variables to keep track of other SMs.
BeginState	Begin state for this SM.
EndState	End state for this SM.
Timers	List of timers used by this SM.
Inputs	Name of class inheriting from `Input`. Used for inputs to this SM.
Outputs	Name of class inheriting from `Output`. Used for outputs from this SM.
Variables	List of variables and their types used by this SM.
Transitions	The transitions for this SM.

Children are introduced to allow modularity. A SM can activate and deactivate a child SM.

A SMState[9] consists of a name, an `Enter()` method, and an `Exit()` method. The methods provide the link between the automaton and the continuous evolution. These methods have access to the inputs, outputs, variables, and the last incoming message. In particular the `Enter` methods are responsible for generating the InputEvents.

In SmartAHS `Message`s are typed. A Message can be an instance of the `Message` class or one of its subclasses. A `Message` is identified by its `Message Id`[10]. State machines use symbolic names that correspond to numeric `Packet Id`s.

Active and Passive Partners are collectively called Correspondents. All correspondents are identified by their `ReceiverProxy`. Some Correspondent Names have predefined semantics. These correspondents, summarized in Table 6.2, are used without being declared.

Timers are used to register timeouts with the `BigBen`. Usually, if a `Message` is sent to another object, a timer is registered to be able to take the necessary action in case of a delayed response.

When *activated* the state machine moves to its BeginState. Every state machine has an EndState. When *deactivated* the state machine moves to the EndState. There are no transitions defined out of the EndState. A state machine is activated either at instantiation or by its parent, if it has one.

Before we give the full specification syntax of the SML we first define some terms. Below * is used as Kleene Closure. Each occurring of "Name" refers to a unique name. Angle brackets refer to optional arguments. The bar refers to alternate options.

MessageClass	Name	Name can be "Message" or the name of any other class inheriting from Message.
MessageId	Name	Symbolic name corresponding to a `Message Id`.
VariableType	Name	This is any valid variable type in the programming environment.
InitialCondition	Name	This is any valid value for the variable type it is used with.

[9] The word State is overloaded. State machine states are referred to as SMStates for clarity.

[10] Recall from Section 6.2.1 that `Message` inherits from `Packet` and `Packet` has a `Type`, `Subtype` and `Id`.

FromState	Name	Transitions originate in these states. Name should be a state.
ToState	Name	Transitions end in these states. Name should be a state.
InAction	See Below	These are the inputs for which a transition is defined.
OutAction	See Below	One of the Outputs generated during a transition.
StartChild	Name	Name of Child SM activated during this transition.
StopChild	Name	Name of Child SM deactivated during this transition.

A transition is defined as follows:

FromState InAction OutActions ToState [StartChild [StopChild]]

Recall that in SmartAHS a `Message` has an `Id` and a `RspTo` field identifying its sender. An InAction can have one of the following forms:

CorrName:MessageId	Transition defined for `Message Id`. The named correspondent is set to the `RspTo` field of the `Message`.
CorrName=MessageId	Transition defined for `Message Id` if `Message` is sent by the current value of the named Correspondent.
EventName	Transition defined for internally generated named `Event`. These `Events` are usually generated by the `Enter` method of a `SMState`.
TimerName	Transition defined if named Timer expires. Timeouts are delivered by a `BigBen` instance.

At every transition several OutActions are generated. Each OutAction can have one of the following forms:

CorrName:MessageId	The `Message` is sent to the named Correspondent.
CorrName:Forward:[MessageId]	The last incoming `Message` is forwarded to the named Correspondent. Optionally the `Id` of the `Message` is changed to MessageId.
TimerName:Value	A timeout is registered with the `BigBen`, which will expire in Value clicks. Named timer is used to identify the request.
TimerName:Cancel	The named timer is cancelled with the `BigBen` instance of the SM.

OutActions is defined as "(OutAction (, OutAction)*)" or as "()" if no messages are generated.

The InAction and OutAction definitions are also used for their specification syntax. The overall SM specification syntax is given below:

MachineName:	Name;
[Children:	Name (, Name)* ;]
States:	Name (, Name)* ;
InputEvents:	Name (, Name)* ;
InputMessages:	MessageClass:MessageId (,MessageClass:MessageId)* ;
OutputMessages:	MessageClass:MessageId (,MessageId:Name)* ;
[Correspondents:	Name (, Name)* ;]
BeginState:	Name;
EndState:	Name;
[Timers:	Name (, Name)* ;]
Inputs:	Name;
Outputs:	Name;
[Variables:	VariableType Name [InitialCondition] , (VariableType Name [InitialCondition])*;]
Transitions:	(cr Transition)*

In the last line "cr" refers to carriage return.

6.3.3 The Dynamics

Each state machine is associated with a number of `Transmitter`s and one `ReceiverProxy`.

Messages are created as part of the output actions and passed to a `Transmitter`. The `Transmitter`s used for special correspondents are defined in Table 6.2. All other `Messages` are sent to the `OutVehTransmitter`.

Message delivery is asynchronous. The `Transmitter`s place the `Message` in a `PacketBox`. `Message`s are delivered only when the `PacketBox` is scheduled for execution in a process layer.

Each SM and all its children share a `ReceiverProxy`[11]. They all pass the list of input messages they are *listening* to and/or *waiting* for to their `ReceiverProxy`. If a SM is waiting for a message and the message comes in, the proxy passes it to the SM. If the message is only being listened to, the proxy will hold the message until the SM starts waiting for it or stops listening to it.

When activated a state machine goes to its begin state. It starts listening to all its input messages and starts waiting for the messages for which the begin state has transitions.

A transition is executed if one of these messages comes in; if an input event is generated; or if a timeout happens.

The execution of a transition has several stages:

1. If there is a child SM to stop, it is deactivated;

2. If there is a child SM to start, it is activated;

3. All OutActions are generated;

4. The `Exit` method of the FromState is called;

5. The `Enter` method of the ToState is called.

The `Enter` and `Exit` methods provide access to external continuous behavior through the inputs, outputs, and variables of the State Machine. In particular the `Enter` method is responsible for creating `InputEvent`s.

At this stage one of several things can happen:

- If an InputEvent is set by the `Enter` method that enables an immediate transition, the FromState clears the messages it was waiting for, and the new transition is executed;

- Else, if the From State and ToState are the same, nothing else happens;

- Else the FromState clears the messages it was waiting for and the ToState starts waiting for its messages. In this case, if a message is already present in the proxy, the corresponding transition is executed.

  If several `Message`s are present that the ToState waits for, the message with highest priority determines the transition. The order of transitions in the state machine specification assigns `Message`s priority.

When a SM is deactivated, it clears all the messages it was listening to, and goes to its stop state.

6.3.4 State Machine Implementation

We have already discussed that State Machines are class based, i.e., a SM specification corresponds to an instantiable class. The SM implementation reflects this characteristic. The abstract base classes `SMState` and `SMAlgorithm` implement the SM classes. The abstract base class `SMInstance` implements SM instances.[12]

A State Machine specification is implemented by the following C++ classes:

[11] `ReceiverProxy` was discussed in Section 6.2.3.

[12] The reader should not be confused by the fact that, as far as C++ is concerned, `SMState`, `SMAlgorithm`, and `SMInstance` are all classes.

- An `SMAlgorithm` subclass is created that represents the SM Class;
- A `SMState` subclass is created for each `SMState` in the SML specification;
- An `SMInstance` subclass is created that represents the SM instances;
- An `SMInstanceState` subclass is created that represents the SM instance state.

The class `TransitionNode` is used to represent transitions. This class is not specialized, and its instances are used to represent the transitions for each state.

The inheritance and containment hierarchy of these classes are given in Figure 6.11 and 6.12 respectively.

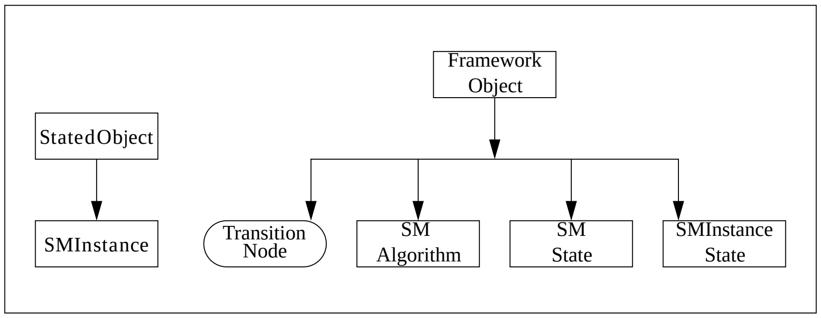

Figure 6.11: Class Hierarchy for State Machine Entities

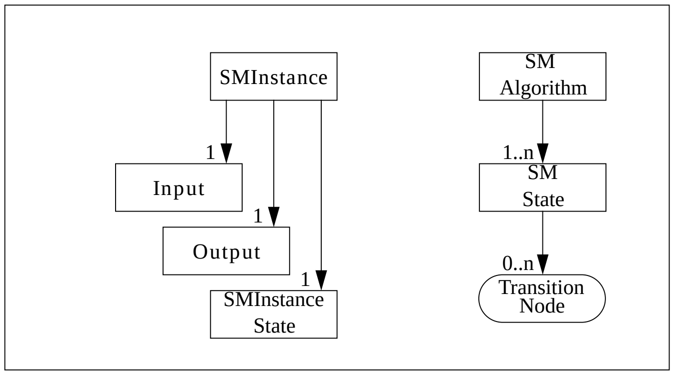

Figure 6.12: Containment Hierarchy for State Machine Entities

An `SMAlgorithm` contains several `SMState`s.

Each `SMState` contains a list of `TransitionNode`s defining the `Event`s, `Messages`, and timeouts (`WakeUp` calls) it responds to. In a given simulation process, only one instance of these classes exist, and they have no dynamic state.

Each `TransitionNode` instance is defined for a specific `Packet` (`Event` or `Message`). It has a field identifying the next `SMState` and the child SMs that need to be activated or deactivated as part of the transition.

Each `SMInstance` has a reference to its `SMAlgorithm` and maintains a reference to its `currSMState` in its own `SMInstanceState`.

When a `Packet` arrives at the `SMInstance`, the `currSMState` is used to locate the correct transition, to validate the `Packet` (InAction), and to generate new `Packets` (OutActions).

The details of these classes are discussed below.

TransitionNode

A `TransitionNode` has the following attributes.

`TransitionNode Attributes`

`o_u4b`	InEvent	`Packet Id`, or timeout identifier for which the transition is defined
`o_u4b`	OutEvent	If there are any OutAction to be generated this field is set to one.
`Link<SMState>`	NextState	The `SMState` this transition leads to.
`PString`	stopSM	Name of child SM to stop.
`PString`	startSM	Name of child SM to start.

Instances of `TransitionNode` are created with the appropriate values for each transition of a `SMState`.

SMState

Each `SMState` is in charge of managing all its transitions. It contains a list of transitions that are expected to be specified by its subclasses. These are:

`SMState Attributes`

`VVList<TransitionNode>`	`MsgList`	Transition list for `Messages`.
`VVList<TransitionNode>`	`EventList`	Transition list for `Events` and timeouts.

The following methods are implemented by this class.

```
Link<TransitionNode>        AcceptEvent     (o_u4b inEvent);
```
Given an input `inEvent` returns the corresponding `TransitionNode`.

```
Link<TransitionNode>        AcceptMessage   (o_u4b inMessage);
```
Given an input `inMessage` returns te corresponding `TransitionNode`.

```
Link<Message>               StartWait       (Link<SMInstanceState> theState);
```
This method invokes `GetProxy()` on `theState` to get the `ReceiverProxy`. `MsgList` is registered with the `ReceiverProxy` in waiting mode.

The following virtual methods are expected to be implemented by the base classes. The `SMState` class provides no implementation.

```
virtual int        Enter                  (Link<SMInstanceState> theState,
                                           Link<InputProjection> inputs,
                                           Link<OutputProjection> outputs);
```
Using `inputs, outputs`, and `theState` this method implements arbitrary behavior. It might generate an `Event`. This method is expected to be written by the user.

```
virtual int        Exit                   (Link<SMInstanceState> theState,
                                           Link<InputProjection> inputs,
                                           Link<OutputProjection> outputs);
```
Using `inputs, outputs`, and `theState` this method implements arbitrary behavior. This method is expected to be written by the user.

```
virtual int        ProcessMessage         (Link<SMInstanceState> theState);
```
This method processes the `lastMessage` in `theState`. Among other things it checks the `RspTo` field of the `lastMessage` against appropriate correspondents. The code of this method is auto-generated from the InAction specification.

```
virtual int        ProcessTimeout         (o_u4b handle,
                                           Link<SMInstanceState> theState);
```
This method validates the timeout using the data in `theState`. The code of this method is auto-generated from the InAction specification.

```
virtual int        PostMessage           (Link<SMInstanceState> theState);
```
This method posts `Messages` as appropriate using the data in `theState`. The code of this method is auto-generated from the OutAction specification.

SMAlgorithm

The `SMAlgorithm` is primarily a data structure. The `SMAlgorithm` subclass code corresponding to a State Machine is auto-generated. An `SMAlgorithm` contains the following attributes:

SMAlgorithm Attributes		
`VVList<SMState>`	`stateList`	The list of `SMStates`, set by subclasses.
`VEList<o_u4b>`	`eventList`	The list of `Events`, set by subclasses.
`VEList<o_u4b>`	`messageList`	The list of `Messages`, set by subclasses.
`Link<SMState>`	`startState`	The StartState of the SM, set by subclasses.
`Link<SMState>`	`stopState`	The StopState of the SM, set by subclasses.

The `SMAlgorithm` implements one method:

```
int                StartListen           (Link<SMInstanceState> inst);
```
This method invokes `GetProxy` on `inst` to get the `ReceiverProxy`. `MsgList` is registered with the `ReceiverProxy` in listening mode.

SMInstanceState

The `SMInstanceState` is a specialization of the `State` class. Each State Machine further specializes this class to add its own attributes corresponding to the Variables field of the SM specification. The `SMInstanceState` has the following attributes:

SMInstanceState Attributes		
`Link<SMInstance>`	`theInst`	The corresponding `SMInstance`.
`Link<SMState>`	`currSMState`	The current `SMState` of this `SMInstance`.
`Link<SMInstance>`	`curChild`	The current active child SM of this `SMInstance`.
`Link<SMInstance>`	`theParent`	The parent SM of this `SMInstance` if any.
`Link<Message>`	`lastMessage`	The most recent `Message` received by this `SMInstance`.
`o_u4b`	`directEvent`	The `Enter` methods of a `SMState` set this field to a non-zero value corresponding to an `Event id`.

The `SMInstanceState` class provides a number of methods that provide access to the `Receiver` and `Transmitters` in a `Vehicle`. These are: `GetProxy()`, `GetInVT()`, `GetOutVT()`, `GetLT()`.

Finally the `SMInstanceState` class has a virtual method, `UpdateCorr()`, that needs to be implemented in the subclasses. This method is not auto-generated.

```
virtual int        UpdateCorr            ();
```
This method should use the `Sensors` in the `Vehicle` to set the PassivePartner Correspondent `ReceiverProxys`.

SMInstance

The `SMInstance` class provides the methods that govern the behavior of a State Machine instance. `Packets` arrive through `ProcessEvent()`, `ProcessMsg()`, or `WakeUp()` methods. These methods specialize the corresponding `FrameworkObject` methods and invoke the `AcceptEvent()` or the `AcceptMessage()` method of the `currSMState` to locate the correct `TransitionNode`.
The `TransitionNode` is then passed to the `AcceptTransition` method for proper execution.

A `SMInstance` has the following attributes.

SMInstance Static Attributes		
Link<SMAlgorithm>	theAlg	A reference to the State Machine class.
Link<SMInstanceState>	theState	A reference to the State.
Link<Input>	inputs	A reference to the Input.
Link<Output>	outputs	A reference to the Output.
VVList<SMInstance>	myChildren	The list of child SMInstances. Note that this list is maintained in the SMInstance and not in the SMAlgorithm.

Each subclass has a proper specialization of Input and Output classes.

The SMInstance class implements the following methods:

virtual int	AcceptTransition	(Link<TransitionNode> myT);
This method implements the control loop described in Section 6.3.3.		
virtual int	Initialize	();
This method forces a transition to the StartState when the SM is activated.		
virtual int	Terminate	();
This method forces a transition to the StopState when the SM is deactivated.		

The following methods are expected to be specialized by SMInstance subclasses. These methods are all auto-generated.

virtual int	ProcessEvent	(o_u4b newEvent);
This method processes Events and is auto-generated.		
virtual int	ProcessMsg	(Link<Message> myMsg);
This method processes Messages and is auto-generated.		
virtual int	WakeUp	(o_u4b handle, Link<Message> state);
This method processes timeouts and is auto-generated.		

6.3.5 Inputs and Outputs

Finally we discuss the Input and Output classes that are used by the State Machines. The Input class encapsulates all external information available to a SMInstance. In a Vehicle, this information may include a Vehicle's internal state or other information accessible through Sensors. The Output class encapsulates all outputs a SMInstance generates.

These classes have the same base class: Projection. The Projection has one attribute, Link<FSMInstance> myFSM, identifying the State Machine of this Projection. It has one virtual method Update() which is expected to be specialized by its subclasses. Subclasses of Projection are expected to add attributes that specify the input and output attributes. For an Input, the specialized Update() method refreshes all input attribute values. For an Output, the specialized Update() method propagates the outputs.

In the current implementation, it is the application developer's responsibility to invoke the Update method when needed in the specialized Enter() and Exit() methods of an SMState.

6.3.6 Usage

Code Generation takes care of the SM construction and transition execution. The code generator also assigns numeric Ids to symbolic message and event names used by the state machines.

Users are expected to write the following components:

- Actual implementation of Inputs, Outputs and specialized Messages;
- The Updateing of the Inputs and Outputs;
- Initialization of declared correspondents in the UpdateCorr() method.
- The implementation of the Enter() and Exit() methods for each SM state.

For an example of a state machine specified in State Machine Language and its SmartAHS implementation the user is referred to Appendix 9.4.

6.4 Graphical Object Editor

In Chapter 4 we discussed how the Object Model provides the constructs for specifying the *possible* relationships among objects at the entity level. The Graphic Object Editor (GOE) provides an environment for specifying the *actual* relationships among instances.

The GOE provides a meta-data definition language for the specification of instantiable classes, their possible relationships, and their attributes. The GOE interprets the meta-data and allows the user to create and connect instances and to set attributes, i.e., define the data, according to the meta-data specification.

The GOE is class based, i.e, it does not support inheritance. Its classes correspond to instantiable classes in the Object Management System. The GOE is used to create initial configurations of objects and is best suited for the specification of the static relationships, the static attributes, and the initial conditions of the dynamic relationships and attributes.

In the GOE, objects are represented by icons; each object icon has a number of input and output ports. Directed arcs are used to represent ordered binary relationships. A binary relationship is established by connecting output and input ports. (More complex relationships can be represented by objects.)

The GOE consists of two columns: the class column and the instance column. Instantiation takes place by clicking on a class icon in the class column and dragging and dropping it to the instance column. A relationship is established by clicking on the corresponding output and input ports of the two participating instances.

The containment relationship has special semantics. Double-clicking on an instance icon opens a window with the containees of that instance. The containment hierarchy results in viewing *plane*s. In the highway grammar, for example, zones contain segments and junctions, segments contain sections, sections contain lanes. As such, at the highway plane, the editor displays the zones and their interconnections. Opening a zone, places the user in the zone-contents plane, displaying the segments and junctions of that zone. Opening a segment, places the user in the segment-contents plane etc. This idea is illustrated in Figure 6.13.

The GOE supports relationships among instances that are part of different containment hierarchies. Consider two instances A and B, containing the instances $A.1$ and $B.1$ respectively. It is possible to connect $A.1$ and $B.1$. As illustrated in Figure 6.13, the graphical editor allows connections between the output of an object and an output of its container. If the output of $A.1$ is connected to the output of A and if the input of $B.1$ is connected the input of B the connection between $A.1$ and $B.1$ is channeled via the connection between A and B. In other words, the relationship (connection) between A and B nests the relationship between $A.1$ and $B.1$. This convention imposes a restriction on connecting instances. Two instances can be connected only if they are in the same plane, or if their containers are connected.

In the graphical editor, double-clicking on a relationship displays its nested elements. This idea is illustrated in Figure 6.14.

Object attributes are displayed in independent windows. Clicking on a class icon displays the attributes and default values of that class. Clicking on an instance displays the attribute values of that instance.

6.4.1 Object Editor Grammar

So far we have discussed the main concepts of the object editor. We now introduce the meta-data grammar. A sample specification file that describes the highway editor can be found in Appendix 9.5. Table 6.3 gives a simple example. The top layer consists of boxes. A box can contain arbitrary number of circles. Boxes are connected to boxes, circles are connected to circles. The BoxOut-BoxIn relationship nests the relationship between the circles. The circle-to-circle relationship is intentionally named BoxOut-BoxIn to illustrate the need for a relationship nesting type. A sensible design would use the names CircleOut and CircleIn for these ports. The bus construct is discussed below.

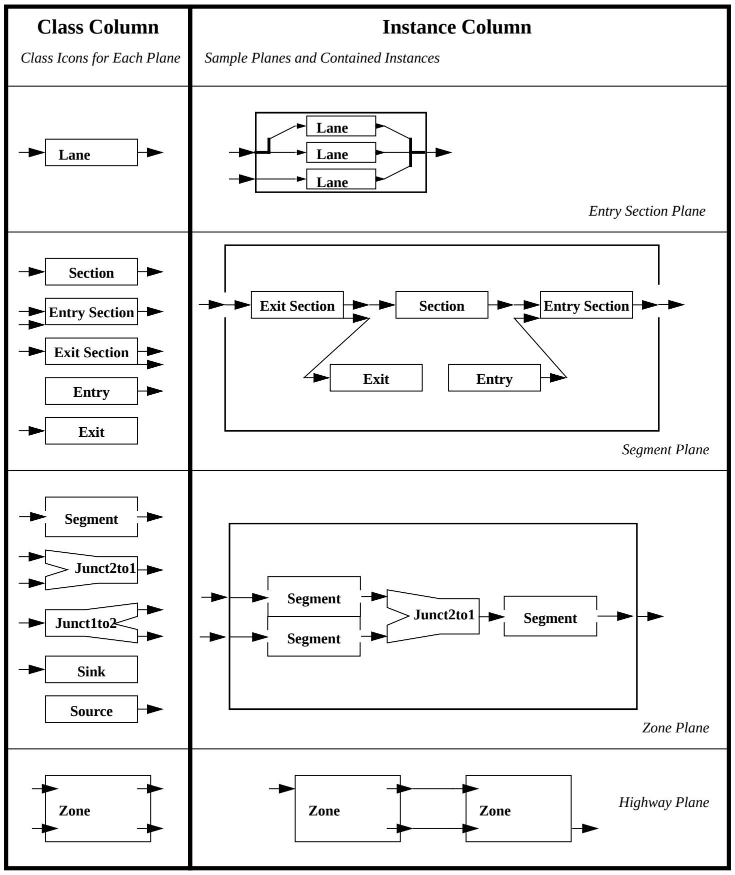

Figure 6.13: Editor Planes

The Configuration File

A configuration file consists of four elements:
(Top Plane Classes, Classes, Relationships, Buses)
The top plane classes list the classes that are at the top viewing plane. Classes list the class definitions, relationships list relationship definitions, and buses list bus definitions.

Class

A class is defined by the following components:
(Class Name, Instance Prefix, Bitmap Name, Inputs, Outputs, Attributes, Containees).

Class name is the name of the class.

Instance prefix is used to name instances. Each instance name consists of the name of its container concatenated with its instance prefix concatenated with an ordinal, e.g. the third circle in box “B5” is called “B5C3”.

Bitmap name is used to identify an icon file.

An input consists of the following components:
(Input Name, C++ Information, Graphical Information, Cardinality).
The C++ information is used to translate input and output names to C++ relationship attribute names. In the above example, the BoxIn input is mapped to an attribute called `boxIn`. The graphical information specifies the location of the arc on the icon. The BoxOut output, for example, is on the right side (R = Right, L = Left, T = Top, B = Bottom) half way (0.5) from the top. Cardinality specifies the number of such inputs this object can have.

An output consists of the following components:
(Output Name, C++ Information, Graphical Information, Cardinality, Input Names).

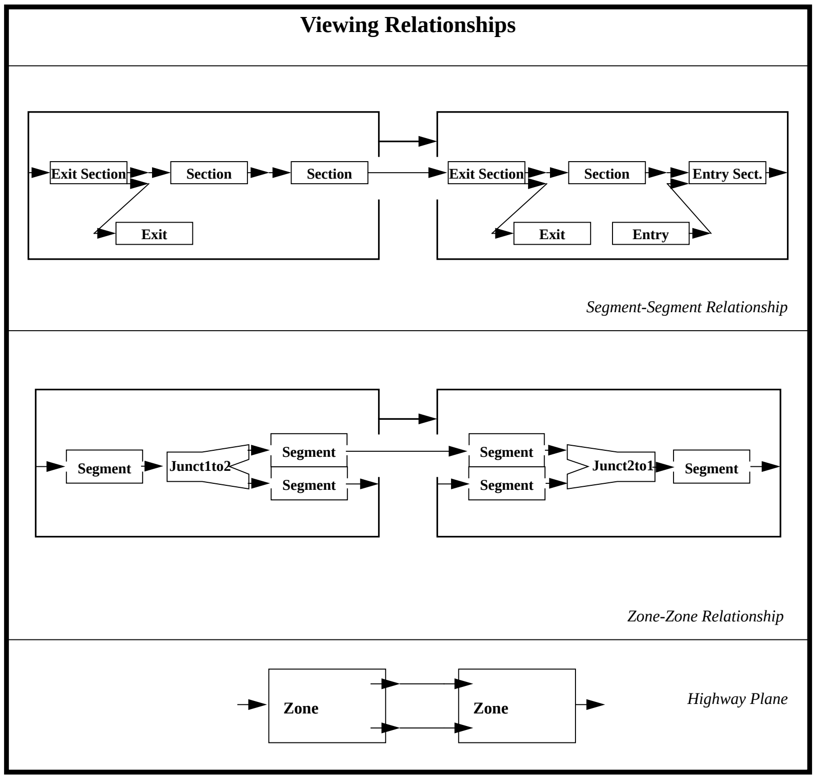

Figure 6.14: Relationship Nesting

The input names field contains the names of the inputs this output is allowed to connect to. These names are independent of class names.

An attribute consist of two components:
(Attribute Name, Initial Value).
The current grammar does not provide attribute types. Attribute types can easily be introduced in future versions.

A containee is given by a pair:
(Class Name, Cardinality).
Containees specify the types of objects a given class can contain and how many of them it can/must contain.

Possible cardinality values are defined by the following set:
{0, 0|1, 1, 0..n, 1..n, K..n, 0..N, 1..N, K..N}
where capital letters correspond to fixed integers. A box can contain arbitrary number of circles.

Relationship

A relationship is defined by the following components:
(Output Name, Input Name, Nested Elements).

The input-output name pair defines the relationship. These names correspond to input and output names.

A nested element is given by the following components:
(Output Name, Input Name, Nesting Type)
The nested element is identified by its input-output name pair. The nesting type is needed to avoid ambiguities.

Consider Figure 6.15. The connection between the boxes on the left side, may contain either of the three relationships on the right. To make this representation unique a type is associated with each nested relationship. In Figure 6.15, the box-box relationship would contain the box-circle relationship as type one, the circle-box relationship as type two, and the circle-circle relationship as

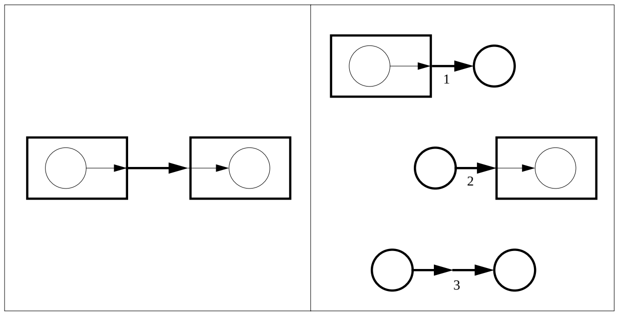

Figure 6.15: Relationship Nesting Types

type three. Note that both the circle-box and the box-circle relationships can contain the circle-circle relationship (as type two and type one respectively). In the example of Table 6.3, the relationships between circles are nested as type 3.

The input-output name values of the nested elements of a relationship must be unique, i.e., the same relationship can not be declared as a nested element twice. Furthermore, a relationship is allowed to nest at most one instance of a relationship. Consider two sections on the highway containing three lanes each. If all three lane-to-lane connections were nested by the section-to-section relationship, as depicted in Figure 6.16, it would be unclear which lane is connected to which.

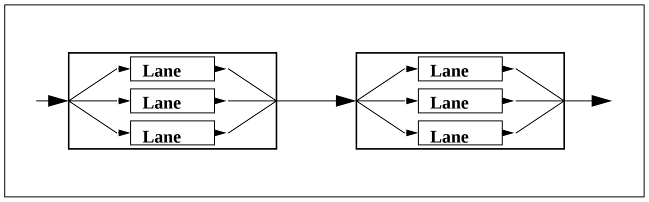

Figure 6.16: Ambiguous Relationship Containment

However, it is desirable to channel all lane-to-lane relationships through one section-to-section connection. Which leads us to our final construct, the Bus, illustrated in Figure 6.17.

Bus

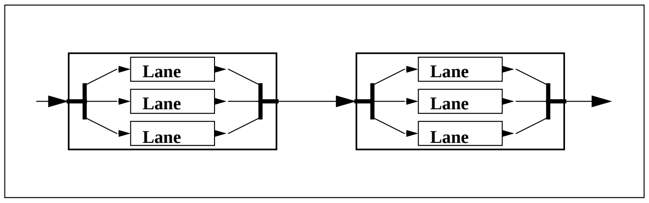

Figure 6.17: The BUS construct

Buses come in pairs. The 1ToN bus has outputs that can connect to inputs; the NTo1 bus has inputs that outputs can connect to. A bus pair is given by two components:
(Bus Name, Terminals)

The bus name identifies the particular bus class.

Each terminal is given by a triple:
(Input Name, Output Name, Cardinality)

The input and output names identify the outputs and inputs that can be connected to this bus. Cardinality identifies the number of such relationships a bus can group.

```
// Top Plane
{Box}
// Classes
{
        {Box B bitmaps/box.bm
                {{BoxIn boxIn L .5 0..n}}
                {{BoxOut boxOut R .5 1 {BoxIn}}}
                {{height 5}{width 8}}
                {{Circle 0..n}}
        }
        {Circle C bitmaps/circle.bm
                {{BoxIn circleIn L .5 1 1}}
                {{BoxOut circleOut R .5 1 1 {BoxIn}}}
                {{radius 2}}
                {{}}
        }
}
// Relationships
{
        {BoxOut-BoxIn { {BoxOut-BoxIn 3} }}
}
// Buses
{
        {CircleBus {BoxOut-BoxIn n}}
}
```

Table 6.3: Boxes and Circles

The output of an NTo1 bus of a given type can only connect to the input of a bus of the same type. Therefore, if a bus connection is nested in a relationship this nesting can only be of type three.

Bus-to-bus connections are not declared as nested relationships. A relationship nests a bus-to-bus connection if it can nest all the terminal connections of the bus.

6.4.2 Editor Implementation

The editor is implemented using the Tcl/Tk graphics package. The current implementation does not support cardinality constraints. It implements inputs and outputs with cardinality one and buses with fixed number of ports. The editor implementation is not discussed here. A sample specification file that describes the highway editor can be found in Appendix 9.5.

6.5 Graphical Debugger

Conceptually, the graphical debugger is another sequencing object that enables the user to collect outputs and to execute transitions. The graphical debugger enables the user to view the evolution of the system at instance level. In particular, it provides the following functionality:

- provides access to objects by name;
- displays object attributes in numerical or graphical format;
- allows users to set error levels of objects. Objects with higher error level print more detailed information;
- allows users to set the logging level of objects. Turning the logging on results in recording the state history of an object;
- allows users to stop and start the simulation;
- replays the recorded history of objects.

The replayer has five buttons: 1) Play; 2) Stop; 3) Step forward one time click; 4) Step backward one time click; and 5) Go to time. The last button takes an argument specifying the global clock time stamp.

The following functionality is planned; however, it is not available in the current implementation:

- provide write access to object attributes;
- provide an event generator to send events to objects;
- provide a view definition language to create composite objects within the graphical environment;
- provide hooks in SmartDb classes to support instance level breakpoints in key object methods.

In this section we discuss the overall design of the graphical debugger. Its Tcl/Tk implementation is not discussed.

6.5.1 Overall Architecture

The overall architecture of the graphical debugger is illustrated in Figure 6.18.

The `GUI` layer acts as a proxy to the graphical debugger process (GDP). It maintains the list of objects currently displayed by the GDP. When the `GUI` layer is scheduled, it "packs" all information regarding the objects currently displayed and sends it to the GDP. The GDP has the responsibility to "unpack" this information and to display it on the screen. As the user specifies new requests through the display, such as retrieving an object by name, stopping the simulation etc, the GDP propagates these requests to the `GUI` layer. The `GUI` layer processes these requests when scheduled.

An object can "pack" itself or send other textual information to the GDP.

6.5.2 The GUI layer

When a stop request is propagated to the `GUI` layer, it stops the simulation by simply going into a loop that reads further requests from the GDP. Simulation is continued when a start request is received.

When a get request is propagated to the `GUI` layer, it sets the `errorLevel` of that instance to one. For time driven objects the `GUI` layer packs and sends the state information of these objects when scheduled. Event driven objects are expected to assume responsibility for packing themselves and for sending state information only as their state changes. The `AcceptTransition()` method of the `SMInstance` class, for example, checks the `errorLevel` of the `SMInstance` and packs and sends

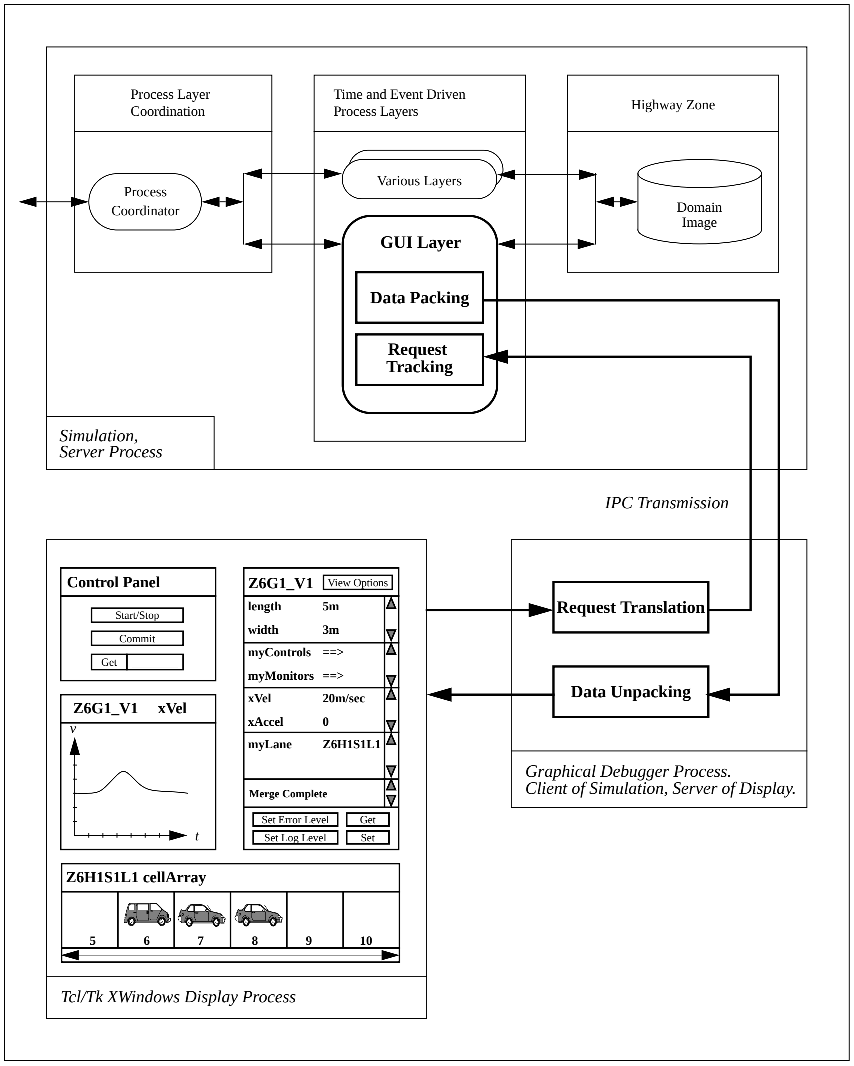

Figure 6.18: Graphical Debugger Architecture

its state if necessary. Objects use the return value of the `SetErrorLevel()` method to indicate to the `GUI` layer whether they are time or event driven.

The `GUI` layer recognizes SmartDb objects only. The methods it invokes on objects are restricted to `SetErrorLevel()`, `SetLogLevel()`, and `Pack()`. Polymorphism ensures that proper SmartAHS or SmartPATH object methods are invoked.

The `Pack` method must be specialized by each class. A special code-generator is implemented that can generate the `Pack` method of a class from its header file specification. The code-generator also generates a template file that is interpreted at run time by the GDP.

6.5.3 The Pack Code-Generator

The code-generator works for arbitrary classes; however, the set of attributes it recognizes are fixed. It recognizes all elemental data types, arbitrary `Link`s, `VVList`s, `VEList`s, `VVArray`s, and `VEArrray`s. It knows to distinguish between the static and dynamic attributes of objects and between the relation and value attributes. `Link`s, `VVList`s, `VEList`s, `VVArray`s, and `VEArrray`s are interpreted as relations. An object's attributes are treated as static, its `State` is treated as dynamic. Static attributes are packed only once when the get request is received; subsequent pack operations pack the state attributes only.

New entities can be integrated into the graphical debugger environment simply by running the code-generator on the header file of the class.

6.5.4 The Graphical Debugger Process

The graphical debugger is implemented in Tcl/Tk. It recognizes the same set of attributes as the pack code-generator. It interprets the packed data using the template file generated by the code-generator.

It displays attributes in tabular or graphical format.

In the tabular format all attributes of an object are displayed in one window. In Figure 6.18, "Z6G1_V1" is a `Vehicle` displayed in tabular format. The window is divided into six subwindows. The first subwindow contains the instance's name and a menu button to select viewing options for the attributes. The second through fourth subwindows display static value, static relationship, dynamic value, and dynamic relationship attribtues. The fifth subwindow displays events and messages generated by the instance. The sixth subwindow displays operation buttons. The second through fifth subwindows can be scrolled independently; they can be closed and reopened.

Double-clicking on a relationship attribute opens a new window. In the case of a `Link` attribute, an instance window is opened. In the case of a `VVList` attribute, the window displays a list of `Link`s, denoted by icons. In the case of a `VVArray` attribute, the window displays an array, where each index may contain a `Link`. In Figure 6.18, the `cellArray` is a `VVArray<Vehicle>` attribute.

The user can choose to display numerical value attributes in graphical format. In this case, a new window is opened. In Figure 6.18, the `xVel` of the vehicle is displayed graphically. The template file can be used to specify the labels of the graph axes.

Data Unpacking

A packed instance consists of a header identifying the class and the instance and an ordered list of attribute values. The attribute order for each class is specified in the template file.

The template file contains the class definitions. The template file class definitions are similar to the corresponding C++ class definitions. Methods are omitted, graphical information is added. Template file class definitions support inheritance. Each class consists of its parent and an ordered list of attributes. An attribute is given by its name, type, and default display driver.

Each attribute type can have several display drivers. Currently only text and x-t[13] graph drivers are supported. More complex drivers for `VEArray`, `VEList`, and complex attributes are under implementation.

The template file is interpreted by the GDP class manager which maintains a class definition structure for each class.

Each instance window has its own manager. Upon initialization, the instance manager retrieves its class definition information from the class manager.

The instance manager maintains an independent list of display drivers for each attribute. Once the window is opened, the user may change the display format of attributes.

When a packet arrives at the GDP, based on its header, it is forwarded to the corresponding instance manager. The instance manager forwards each attribute value to the proper display driver.

[13] x-t graphs are used for numerical attributes only. The attribute value is displayed over time.

6.6 Parametric Interfaces

Parametric interfaces are used to specify initial values of attributes at instantiation time of an object.

SmartAHS provides a tool to simplify the specification of parametric interfaces for elemental data types.

Each class is expected to declare its parameters in an independent "parameters" class. Parameter class definitions consist of elemental data type attributes. Each attribute must also specify its global default value.

A code-generator is provided that can read in parameter class definitions and produce the following:

- A `Read()` function that can read in and interpret a parameter value file;
- For each parameter class, say `VehParams`, a function of the form `VehParams* GetVehParams(char* instanceName)`.

A parameter file has entries of the form:
ClassName[InstanceName]*AttributeName*AttributeValue.*

Here, ClassName identifies the parameter class name, InstanceName identifies an instance. If InstanceName is omitted, the entry specifies a class level attribute default value.

The `Read` function parses the parameters file and creates data structures with class and instance level default attribute values.

The `Get()` method looks up the `instanceName` in the data structure. If any instance level attribute values are provided, it uses these. Otherwise, if class level attribute values are provided, it uses these. Otherwise it uses the global defaults. It returns these values in a parameter class instance, e.g. a `VehParams` instance.

The constructors of classes are expected to `Get` the default attribute values based on the specific instance's name.

6.7 Naming Convention

The `name` of an object is an attribute in the `FrameworkObject`. The `SetName()` method can be used to set the `name` of an instance. Class constructors are overloaded; instance names can be passed in as an argument, or a default naming mechanism may be used by a class to name its instances.

The naming convention is based on the static containment hierarchy of objects. Each instance's name consists of the concatenation of its container's name, a string identifying its class name, and an ordinal.

The containment hierarchy of highway and traffic instances is static, as such, their names are derived from the concatenation of their container's name, their class identifier, and an ordinal. The `Zone` object provides the root for highway object names.

`Vehicle`s are named by the `Factory` they are created in.

The containment hierarchy within a `SmartObject` is static, as such, the names of the automation devices and their containees are derived from the concatenation of their container's name, their class identifier, and an ordinal. The `Vehicle` or the particular highway object provide the root for automation device names.

Since scheduling objects are created for each `Zone`, the `Zone` name acts as the root of scheduling objects.

`Event`, `Message` and `Message` subclass instances are not assigned unique names. These instances usually take a name based on their class.

`State, Input`, and `Output` subclass instances are named after their `StatedObject` instance name.

The following subsections discuss the naming convention in more detail and present examples.

6.7.1 Highway Entities

The highway and traffic entity classes are identified by short strings. These strings and example instance names are given below:

Class	Id	Examples
Zone	Z	Z1, Z3, Z98
Segment	H	Z1H1, Z3H12
Sink	AS	Z1AS5, Z3AS2
Source	GS	Z1GS12, Z13GS1
Junct1to2	O	Z1O3, Z2O5
Junct2to1	T	Z1T5
Section	S	Z1H1S4, Z12H3S78
EntrySection	ES	Z1H1ES4, Z12H3ES78
ExitSection	XS	Z1H1XS4, Z12H3XS78
Entry	GE	Z1H1GE12, Z13H12GE1
Exit	AE	Z1H12AE5, Z3H1AE2
Lane	L	Z1O3L1, Z1H1S1L1

The "G" of the `Entry` and `Source` class ids is a mnemonic for `Generator`, the "A" in `Sink` and `Exit`, a mnemonic for Absorber.

The traffic entity names are derived from the concatenation of the containing `Absorber` or `Generator` name and the particular traffic entity identifier. Example classes and their identifiers are given below:

myInTraffic	I	Z6GS1I1
myOutTraffic	U	Z6GS1U1
myFactory	F	Z6GS1F1, Z6H1GE1F2

The application developers are responsible for selecting proper identifiers for their traffic object specializations.

6.7.2 Vehicles

`Vehicle`s are named by the `Factory`s they are created in. A `Vehicle` name consists of a `Factory` name and an ordinal, e.g. "Z6H1GE1F2_7" or "Z3GS2F1_12".

6.7.3 Automation Devices

Automation device names are created by concatenating the `SmartObject` name with a class identifier for the automation device and an ordinal. Automation devices are responsible for naming their containees.

Some automation device class identifiers and example instance names are given below:

InVehTransmitter	INTX	Z1GS1F1_1_INTX1
OutVehTransmitter	OUTTX	Z1GS1F1_1_OUTTX1
ReceiverProxy	RCVR	Z1GS1F1_1_RCVRReg, Z1GS1F1_1_RCVRCoord
VehicleSensor	SENS	Z1GS1F1_1_SENSBasic
Camera	CAMERA	Z1GS1F1_1_CAMERA1
CoordControl	CC	Z1GS1F1_1_CC1
RegControl	RC	Z1GS1F1_1_RC1

The automation devices have the responsibility to name their containees. For example, state machine instance names are derived by concatenating the containing controller's name with the particular `SMInstance` name.

6.7.4 Scheduling Objects

There is a single instance of the `ProcessCoordinator` in a `Zone`; its name is a concatenation of the `Zone` name and the class name, e.g. "Z1_ProcessCoordinator". `Layer`s use a similar convention. Their instance names are derived by concatenating the `Zone` name with the class name, e.g., "Z6_Physical".

The names of `PacketBox` and `BigBen` instances are created by concatenating the containing scheduling instance's name with the corresponding class name, e.g., :"Z6_Regulation_BigBen", "Z1_Coordination_PacketBox".

6.7.5 States

The names of `State`, `Input`, and `Output` instance are derived by concatenating their `StatedObject` instance name with the corresponding subclass name.

6.8 Other Modules

Currently, the GOE does not support meta-data evolution, i.e., highway layouts created with different highway editor specifications can not be shared. As a result the first three steps of the simulation setup must be combined.

Using the GOE meta-data language, the application developers are expected to extend the highway editor specification to include the newly created automation devices. The GOE is then used to create the highway layout along with the automation devices and the traffic entities.

Chapter 7

SmartPATH Implementation

7.1 Implementing SmartPATH Entities

In this section we give examples of State Machines, and `Control`, `Monitor`, `and Traverser` subclasses. In particular we create the `MyVehicle` that is capable of performing *merge*, *lead*, and *follow* maneuvers.

First we summarize how SmartAHS objects are extended.

- `Control` objects;

  SmartAHS provides minimal definition of `Control` object interfaces. `Control` objects can be arbitrarily complex composite objects.

  The virtual `Run` method is used to implement time driven behavior. `Control` objects are expected to use state machines to implement event driven behavior.

- `Monitor` objects;

  Each `Monitor` subclass has a specialized `State` consisting of attributes. Each `Monitor` subclass specializes the `Run` method for time driven scheduling and implements the necessary functionality to collect the data for the `Monitor`'s `State`.

  The `Monitor` subclasses can schedule themselves using the `BigBen` timers. This functionality is implemented in the `Monitor` class.

  Alternatively `Monitor` subclasses can receive `Packet`s from other objects. In this case, the other objects must be implemented to send the `Packet`s.

- `Sensor`, `Transmitter`, and `Receiver` objects;

  SmartAHS defines the virtual method interfaces of these objects. Subclasses of these classes have to provide the proper method implementations. Alternatively new virtual method interfaces can be added. However, these interfaces would be used by SmartPATH objects only.

  Currently `Sensor`s are used to set the `Input`s and `Output`s of `Control` objects, and to collect the data for `Monitors`. `Transmitter`s and `Receiver`s are used by state machines for `Message` delivery.

- State machines;

  State machines are implemented in a special syntax and translated into C++ code. The application developer is expected to implement the `Enter()` and `Exit()` methods of the `SMState` subclasses and the `UpdateCorr()` method of the SMInstanceState subclass.

- `Input`, `Output` and `Message` objects;

  `Input` and `Output` subclasses specialize the base class by adding attributes and by specializing the `Update()` method. The relationship attributes specify the relationships from which the data is derived, the value attributes specify the actual input and output values. The `Update()` method is responsible for propagating the inputs and outputs.

Entity	Relationship	With Entity	Cardinality
VehicleControl	myProxy	ReceiverProxy	1
	myInVT	InVehTransmitter	1
	myOutVT	OutVehTransmitter	1
CoordControl	regProxy	ReceiverProxy	1
RegControl	coordProxy	ReceiverProxy	1

Table 7.1: Relationships within a Vehicle

- `Vehicles`;

  The `Vehicle` is a composite object. Application developers are expected to implement a constructor that instantiates proper automation devices and initializes their relationships.

- Highway objects;

  Application developers are not expected to specialize these classes. Any extesions to highway classes should become part of SmartAHS.

- `InTraffic`, `OutTraffic`, and `Factory`;

  SmartAHS defines the virtual method interfaces of these objects. Subclasses of these classes have to provide the proper method implementations. Alternatively new virtual method interfaces can be added. However, these interfaces would be used by SmartPATH objects only.

- Scheduling objects.

  SmartAHS defines the virtual method interfaces of three scheduling base classes. Specific sublasses specialize these methods.

7.1.1 Creating Vehicles

The `MyVehicle` class is composed of many components.

The class hierarchy for `MyVehicle` components is given in Figure 7.1. The figure omits the `SMState` and `SMAlgorithm` subclasses that are generated, as well as the `Input` and `Output` classes used by each state machine. The `MyVehicle` containment hierarchy is given in Figure 7.2. The static relationships within a `MyVehicle` are given in Table 7.1. These methods use the `L_AS` macro, discussed in Appendix 9.2, for safe type-casting.

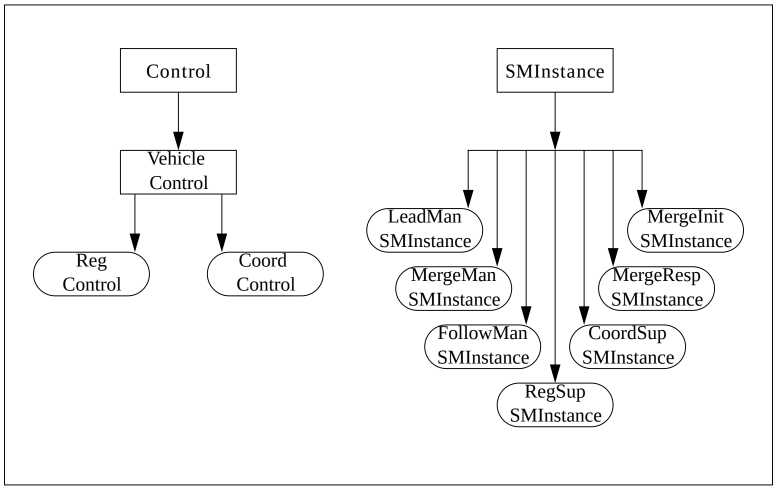

Figure 7.1: Class Hierarchy for `MyVehicle` components

```
MyVehicle::MyVehicle(char* name)
{
Link<VehicleSensor> vehSensor = new Persistent VehicleSensor();
InsertSensor(vehSensor, 0, TRUE);

Link<InVehTransmitter> inXmitter = new Persistent InVehTransmitter();
InsertTransmitter(inXmitter, 0, TRUE);
Link<OutVehTransmitter> outXmitter = new Persistent OutVehTransmitter();
InsertTransmitter(outXmitter, 1, TRUE);

ReceiverProxy* coordRcvr = new Persistent ReceiverProxy();
InsertReceiver(coordRcvr, 0, TRUE);
ReceiverProxy* regRcvr = new Persistent ReceiverProxy();
InsertReceiver(regRcvr, 1, TRUE);

CoordControl* coordControl = new Persistent CoordControl();
InsertControl(coordControl, 0, TRUE);
RegControl* regControl = new Persistent RegControl();
InsertControl(regControl, 0, TRUE);

Camera* camera = new Persistent Camera();
InsertMonitor(camera, 0, TRUE);

myBigBen = ProcessCoordinator::GetBigBen("GLOBALBIGBEN");

coordControl->Connect();
regControl->Connect();
camera->Connect();
}
```

Table 7.2: The `MyVehicle` Constructor

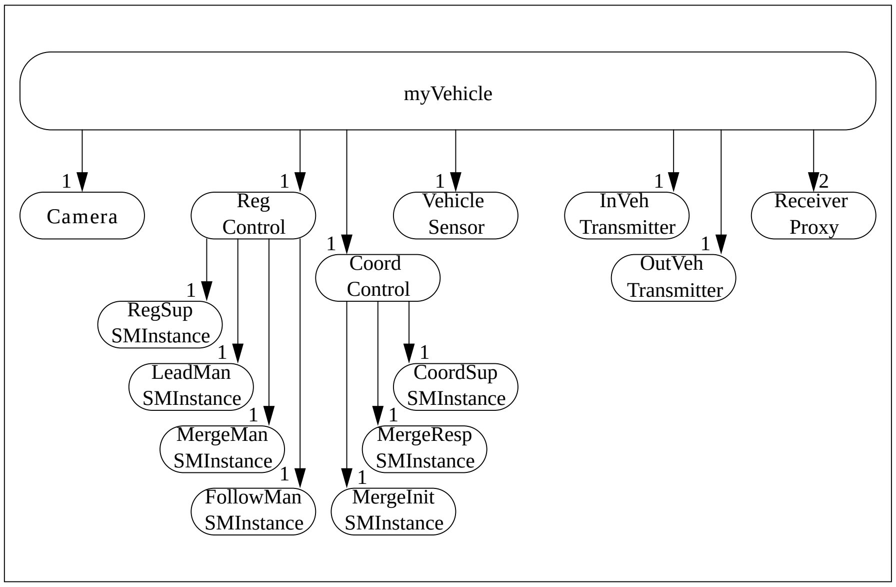

Figure 7.2: Containment Hierarchy for `MyVehicle` components

The `Vehicle` creation takes place in two stages. First all objects in the containment hierarchy are instantiated and their container-containee relationship is set. Then all other relationships of these objects are established. This two-stage creation is necessary, since a relationship between two objects can only be established after they both have been instantiated.

The `MyVehicle` constructor instantiates and properly inserts automation devices into the `MyVehicle`. The `Insert` method sets the container-containment relationship. These objects' constructors have the responsibility of recursively instantiating their containees. The `VehicleSensor, InVehTransmitter, OutVehTransmitter` and `ReceiverProxy` classes were discussed in the last chapter. `CoordControl, RegControl`, and `Camera` are the specific `Control` and `Monitor` devices used by `MyVehicle`. These classes are discussed below.

After all containees are instantiated, the `MyVehicle` constructor `Connect`s its containees. The `Connect` method of each object has the responsibility of establishing its relationships, and the relationships of its containees.

The Vehicle constructor is given in Table 7.2.

7.1.2 Creating Control Objects

CoordControl

The `CoordControl` class consists of three state machines. These are the CoordSup, the MergeInit, and the MergeResp state machines. The CoordSup is the parent of the other two state machines and has a supervisory role. It determines if and when to activate/deactivate the MergeInit and MergeResp state machines. The CoordSup is activated at instantiation and remains active until the `MyVehicle` leaves the highway.

The `CoordControl` constructor instantiates the state machines; the `Connect` method establishes all initial relationships. The `CoordControl` object is strictly event driven; as such, it does not specialize the `Run` method.

The constructor of `CoordControl` is given as in Table 7.3; its `Connect` method is given in Table 7.4.

RegControl

The `RegControl` class consists of four state machines. These are the RegSup, the MergeMan, the FollowMan, and the LeadMan state machines. The RegSup is the parent of the other three state machines and has a supervisory role. Based on the requests sent by the coordination layer

```
CoordControl::CoordControl()
{
CoordSupSMInstance* cSup1 = new Persistent CoordSupSMInstance();
InsertSM(cSup1);
MergeInitSMIInstance* MI1 = new Persistent MergeInitSMIInstance();
InsertSM(MI1);
MergeRespSMInstance* MR1 = new Persistent MergeRespSMInstance();
InsertSM(MR1);

/* Establish the parent child relationship among the state machines */
cSup1->InsertChild(MI1);
cSup1->InsertChild(MR1);

layerName = ''COORD'';
regProxy = NULL_LINK;
}
```

Table 7.3: The `CoordControl` Constructor

state machines, it determines which maneuver to activate. The RegSup is activated at instantiation and remains active until the `MyVehicle` leaves the highway. Upon activation, RegSup activates the LeadMan.

The constructor and the `Connect` methods of the `RegControl` class are similar to the methods of `CoordControl` and are omitted. However, since `RegControl` is both time and event driven `RegControl` specializes the `Run` method. The `Run` method delivers a `click Event` to the active maneuver state machine. The `RegControl` `Run` method is given in Table 7.5.

7.1.3 Creating State Machines

Seven `SMInstances`, their `Inputs`, `Outputs`, and specialized `Messages` need to be defined. We discuss the `MergeInitSMInstance` only.

The MergeInit state machine definition is given in Table 7.6. The state machine code generator is used to translate this specification into the `MergeInitSMInstance` class.

First, we introduce the specialized `Messages`, `Inputs`, and `Outputs` of the MergeInit state machine, then we discuss its behavior.

MRMessage and MDMessage

The only specialized `Message` classes used by the MergeInit state machine are `MRMessage`, and `MDMessage`. They contain the following attributes:

`MRMessage Attributes`		
`o_u1b`	`platoonSize`	Size of platoon, one if single agent.

`MDMessage Attributes`		
`VVList<Receiver>`	`myFollowers`	The new followers.

Assume platoon A initiates a merge with platoon B in front of it. A's leader sends the `MRMessage` containing the size of platoon A. (If a single agent initiates the maneuver, platoon size is one.) Using this size and other information the leader of platoon B determines if the merge should

```
CoordControl::Connect
{
/* Call Parents Connect First */
VehicleControl::Connect();

/* Establish the myOutVT relationship */
Link<Transmitter> myT = inSmartObject->GetTransmitter(``OUTTX1'');
if (myT == NULL_LINK) { /* Error Code OMITTED */ }
SetOutVT(myT);

/* Establish the myProxy relationship */
Link<Receiver> myR = inSmartObject->GetReceiver(``RCVRCoord'') ;
if (myR == NULL_LINK) { /* Error Code OMITTED */ }
SetProxy( L_AS(ReceiverProxy, myR ) );

/* Establish the regProxy relationship */
myR = inSmartObject->GetReceiver(``RCVRReg'') ;
if (myR == NULL_LINK) { /* Error Code OMITTED */ }
SetRegProxy( L_AS(ReceiverProxy, myR ) );

myBigBen = Zone::GetBigBen(``Coordination_BigBen'');

/* Set the relationships of the State Machines */
o_u4b s = myInstances.size();
for (o_u4b i =0; i < s; i++){
   if (myInstances[i].inputs != NULL_LINK){
     myInstances[i].inputs->myVehicle = L_AS(Vehicle, inSmartObject);
   }
   if (myInstances[i].outputs != NULL_LINK){
     myInstances[i].outputs->myVehicle = L_AS(Vehicle, inSmartObject);
   }
}

/* Activate the CoordSupSMInstance and register a timeout for CoordSup */
Link<FSMInstance> temp = & myInstances[0];
Link<CoordSupSMInstance> cSup = L_AS(CoordSupSMInstance, temp);
cSup->Initialize();
cSup->myState->mergeTimer = cSup->RegisterTimer(3, MERGETIMEROUT);

return TRUE;
}
```

Table 7.4: The `CoordControl` `Connect` Method

```
RegControl::Run() {
Link<FSMInstance> mySup = & myInstances[0];
mySup->theState->curChild->ProcessEvent(TICK);
return TRUE;
}
```

Table 7.5: The `RegControl` `Run` Method

take place. When A completes the merge, it sends an `MDMessage` containing a list of the `Vehicle`s in platoon A. The leader of platoon A adds these `Vehicle`s to its list of `myFollowers`.

CoordSupIn and MIOut

The specialized `Input` and `Output` classes used by MergeInit are `CoordSupIn` and `MIOut`. These classes contain the required information for the `Enter` and `Exit` methods to generate the necessary internal `Event`s.

The attributes of `CoordSupIn` are:

`CoordSupIn Attributes`		
Relations		
`Link<Vehicle>`	`sourceVehicle`	Inputs are derived from the `sourceVehicle`'s `State`.
Values		
`o_bool`	`amILeader`	True if platoon leader.
`o_u1b`	`platoonSize`	Size of platoon, one if single agent.
`o_u1b`	`targetPlatoonSize`	Preferred platoon size.
`Link<Receiver>`	`myLeadersProxy`	If follower, leaders proxy.
`VVList<Receiver>`	`myFollowers`	If leader, followers' proxys.

The attributes of `MIOut` are:

`MIOut Attributes`		
Relations		
`Link<Vehicle>`	`targetVehicle`	Outputs are propagated to `targetVehicle`'s `State`.
Values		
`o_bool`	`amILeader`	True if platoon leader.
`Link<Receiver>`	`myLeadersProxy`	If follower, leaders proxy.

Both objects specialize the `Update` method. The `Update` method of `CoordSupIn` derives the values of the attributes, the `Update` method of `MIOut` propagates the values of the attributes to the `VehicleState`. The `Update` methods are invoked in the specialized `Enter` and `Exit` methods of the `SMState` subclasses.

MergeInitSMInstance Behavior

The MergeInit state machine is activated by its parent, the CoordSup state machine. As part of activation, the parent sends the `START` `Message`.

The `Initialize` method of a `SMInstance` invokes the virtual `UpdateCorr` method of its `SMInstanceState`. `MergeInitSMInstanceState` specializes this method and sets the correspondent variable `F` for the front `Vehicle` using its `Sensor`. The `UpdateCorr` method is given in Table 7.7.

machineName:	MergeInit;
children:	NULL;
states:	Idle, Check, ReqWait, Accel, Set;
InputEvents:	YES, NO;
inputMessages:	Message:START, Message:OK_MERGE, Message:DONT_MERGE, Message:BUSY, Message:MERGE_M_DONE, Message:MERGE_M_FAILED;
outputMessages:	MRMessage:MERGE_REQUEST, Message:MERGE_MAN, Message:LEAD_MAN, Message:MERGE_I_DONE, Message:MERGE_I_FAILED, MDMessage:MERGE_DONE, Message:MERGE_FAILED;
beginState:	Idle;
correspondents:	F, FLead;
timers:	miTimer;
inputs:	CoordSupIn;
outputs:	MIOut;

transitions:			
Idle	Parent:START	()	Check
Check	YES	(F:MERGE_REQUEST, miTimer:20)	ReqWait
Check	NO	(Parent:MERGE_I_FAILED)	Idle
ReqWait	FLead=OK_MERGE	(REG:MERGE_MAN, miTimer:Cancel, miTimer:50)	Accel
ReqWait	FLead=DONT_MERGE	(Parent:MERGE_I_FAILED, miTimer:Cancel)	Idle
ReqWait	FLead=BUSY	(Parent:MERGE_I_FAILED, miTimer:Cancel)	Idle
ReqWait	miTimer	(Parent:MERGE_I_FAILED)	Idle
Accel	REG:MERGE_M_DONE	(miTimer:Cancel, FLead:MERGE_DONE, Parent:MERGE_I_DONE)	Set
Accel	REG:MERGE_M_FAILED	(FLead:MERGE_FAILED, Parent:MERGE_I_FAILED, miTimer:Cancel)	Idle
Accel	miTimer	(FLead:MERGE_FAILED, Parent:MERGE_I_FAILED, REG:LEAD_MAN)	Idle
Set	YES	()	Idle

Table 7.6: The MergeInit State Machine Definition

```
MergeInitSMInstanseState::UpdateCorr() {
Link<VehicleSensor> myS = L_AS(VehicleSensor,
   theInst->myController->GetSmartObject()->GetSensor("SENSBasic") );
double frontDist, frontVel;
Link<Vehicle> frontVeh = myS->GetVehicle(eFront, eThisLane, frontDist,
                        frontVel);
if (frontVeh == NULL_LINK){ F = NULL_LINK; }
else { F = frontVeh->GetReceiver("RCVRCoord"); }
return TRUE;
}
```

Table 7.7: The `MergeInitSMInstanceState` `UpdateCorr` Method

```
CheckSMState::Enter(Link<SMInstanceState> theState, Link<Input> i,
Link<Output> o)
{
Link <MergeInitInstanceState> myState =
            L_AS(MergeInitInstanceState, theState);
Link <CoordSupIn> myIn = L_AS(CoordSupIn, i);
Link <MIOut> myOut = L_AS(MIOut, o);

 /* User specified code of the Enter method consists of the next two lines.
 The wrapper is auto-generated. */

if (myState->F == NULL_LINK) { myState->directEvent = NO; }
else { myState->directEvent = YES; }

return TRUE;
}
```

Table 7.8: The `CheckSMState` `Enter` Method

The `Enter` method of the `CheckSMState` is specialized to determine if there is a `Vehicle` ahead. Its `Enter` method is given in Table 7.8.

If there is a `Vehicle` within the sensing distance a `MERGE_REQUEST` `Message` is sent to that `Vehicle`, `F`. The response comes from the leader of `F`, or just `F` if it is a single agent.

If `FLead`, the leader of the front platoon, gives permission for the merge maneuver the regulation controller is asked to execute the merge maneuver.

The `Enter` and `Exit` methods of the `AccelSMState` are specialized to `Update` the outputs of the `MergeInitSMInstance`. If the merge maneuver is successful, the regulation layer automatically switches to the follow maneuver which depends on the `myLeadersProxy` relationship. Therefore, it is necessary to set the `State` of the `Vehicle` to reflect the new `myLeadersProxy`. The `Enter` method of `SMAccelState` updates the `State` of the `Vehicle` and sets its `myLeadersProxy` relationship. The `Exit` method reverses this operation in case the maneuver fails. The corresponding code fragments are given in Table 7.9 and 7.10.

If the maneuver is successful, the `SetSMState`'s `Enter` method is called to set the `Vehicle`'s `State`. The code fragment for this method is given in Table 7.11

```
myOut->myLeadersProxy = myState->FLead;
myOut->amILeader = FALSE;
myOut->Update();
```

Table 7.9: The `AccelSMState` `Enter` Method, Code Fragment

```
myOut->myLeadersProxy = NULL_LINK;
myOut->amILeader = TRUE;
myOut->Update();
```

Table 7.10: The `AccelSMState` `Exit` Method, Code Fragment

```
myState->directEvent = YES;

myOut->amILeader = FALSE;
myOut->myLeadersProxy = myState->FLead;
myOut->Update();
```

Table 7.11: The `SetSMState` `Enter` Method, Code Fragment

```
Camera::Camera()
{
/* Create the CameraState and establish the myCameraState relationship */

/* The state constructor establishes the myObj relationship */
myCameraState = new Persistent CameraState(this);

/* Establish the currState relationship of StatedObject */
currState = (State*)myState;

/* Hardcode sensing distance to 15 cellArray cells */
frontDist = 15 + 1;
myBigBen = Zone::GetBigBen("Regulation_BigBen");
}
```

Table 7.12: The `Camera` Constructor

7.1.4 Creating Monitors

Camera

The `Camera` object models a camera mounted on the front hood of a vehicle. It captures information about the `Vehicle`s ahead in its `CameraState`.

Most of `Camera` behavior is implemented by its parent, `Monitor`. `Camera` only specializes the `Run` method to invoke the `Update` method of `CameraState`.

The `Update` method of `CameraState` uses the `Camera`'s static relationships to derive the current `CameraState` values.

The `Camera` has the following static attributes and relations:

Camera Static Attributes		
Relations		
`Link<VehicleState>`	`itsState`	The containing `Vehicle`'s `State`.
`Link<Sensor>`	`itsSensor`	The containing `Vehicle`'s `Sensor`.
Values		
`o_u2b`	`frontDist`	Distance within the vision of Camera.

The `CameraState` consists of the following attributes:

CameraState Attributes		
Values		
`o_float`	`dxFront`	Distance differential with Vehicle ahead.
`o_float`	`dyFront`	Speed differential with Vehicle ahead.
Relations		
`VVArray<Vehicle>`	`frontVehicles`	Vehicles ahead within the next `frontDist` meters.

The constructor and the `Connect` methods of the `Camera` are given in Table 7.12 and 7.13 respectively. The `Camera` is turned on or off through the graphical debugger.

7.1.5 Specializing Sensors, Transmitters, and Receivers

In this example `myVehicle` uses the SmartAHS communication and sensor devices. However, one could specialize these objects simply by inheriting from them and specializing their virtual

```
Camera::Connect()
{
if (inSmartObject == NULL_LINK) { /* Error code OMITTED */ }
Link<Vehicle> myVehicle = L_AS( Vehicle, inSmartObject );
Link<State> temp = myVehicle->GetState();
if (temp == NULL_LINK) { /* Error code OMITTED */ }
itsState = L_AS(VehicleState, temp);
itsSensor = L_AS(VehicleSensor, myVehicle->GetSensor("SENSBasic"));
return TRUE;
}
```

Table 7.13: The `Camera` `Connect` Method

methods.

7.2 Creating the Process Layers

Four special process layers are created.

The `Physical` layer is used to update the position of a `Vehicle` on the highway. The `Physical` layer is actually part of SmartAHS.

The `Regulation` layer is used to generate the displacement of a `Vehicle`.

The `Coordination` layer is used to implement the coordination control layer.

The `GUI` layer is used to communicate with the graphical debugger.

7.2.1 Physical Layer

The `Physical` Layer subclasses the `Traverser` and specializes the methods `EAbsorber()`, `EGenerator()`, and `ELC()`. As shown in in Table 7.14, these methods invoke the `LaneContainer` methods that implement `Vehicle` movement on the highway, `Vehicle` creation, and `Vehicle` removal.

7.2.2 Regulation Layer

The `Regulation` layer subclasses the `Hybrid` class. It contains the `PacketBox` for the `RegControl` state machines.

The `Regulation` layer also specializes the `ELC` method, to invoke the `Run` method of each `RegControl` object.

The corresponding methods are in Table 7.15.

7.2.3 Coordination Layer

The `Coordination` class does not provide any specialized methods. It only sets the name of the `EventDriver` to “Coordination”.

7.2.4 GUI Layer

The `GUI` class directly inherits from the `Layer` class and specializes the `Go` method to execute the appropriate graphical debugger code.

```
Physical::ELC(Link<LaneContainer> LC){
LC->MoveVehicles();
return 0;
}


Physical::EAbsorber(Link<Absorber> LC){
LC->AbsorbVehicles();
return 0;
}


Physical::EGenerator(Link<Generator> LC){
LC->GenerateVehicles();
return 0;
}
```

Table 7.14: Specialized `Physical` Layer Methods

7.3 Simulation Setup

We follow the steps outlined in Section 5.2.2

7.3.1 Highway Layout

A highway layout is created using the GOE.

This layout is saved to a C++ file and compiled to executable format. The executable is run to place the highway into the database.

7.3.2 Traffic Patterns

An `InTraffic` subclass is created that generates a `MyVehicle` every n time clicks, where n is a configurable parameter. A `Factory` subclass is created that instantiates a `myVehicle`. These objects are inserted into the highway.

7.3.3 Roadside Automation

No roadside automation devices are used for this simulation.

7.3.4 Vehicle Automation

The implementation of `MyVehicle` was discussed above.

7.3.5 Simulation Granularity

The implementation of the scheduling objects were discussed above. The executable that instantiates the scheduling objects is given in Table 7.16.

The `ConfigFile` object provides methods to get simulation parameters. It reads in a simulation parameters file and returns the required information. Using the `ConfigFile` object, this executable creates the scheduling object instances and names them.

Other simulation granularity is specified through the simulation parameters.

```
Hybrid::Start(){
myEventDriver->Go();
return 0;
}



Regulator::ELC(Link<LaneContainer> LC){
VVList<Lane> theLanes = LC->GetLanes();
VVList<Vehicle> vehicleList;
Link<Vehicle> aVehicle;
Link<Control> aControl;
o_u4b i, j;

i = 0;
while (i < myLanes.size()) {
   vehicleList = &myLanes[pos].GetVehicles();
   j = 0;
   while (j < vehicleList.size()) {
     aVehicle = &vehicleList[j];
     aControl = aVehicle->GetControl("RC1");
     if (aControl == NULL_LINK) { /* Error Code OMITTED */ }
     aControl->Run();
   j++;
   }
i++;
}
return 0;
}
```

Table 7.15: Specialized `Regulation` Layer Methods

```
extern ConfigFile* CFO;
main(int argc, char **argv)
{

dom = new PDOM();
dom->beginsession(CFO->getMyDbName(), "session1");

InternalSynchronizer* myIS =
      new Persistent InternalSynchronizer(CFO->getMyISName());

Physical* myP = new Persistent Physical();
myIS->Register(myP, myP->getName(), myP->getPeriod());

Regulator* myR = new Persistent Regulator();
myIS->Register(myR, myR->getName(), myR->getPeriod());

Coordination* myE = new Persistent Coordination();
myIS->Register(myE, myE->getName(), myE->getPeriod());

GUI* myGUI = new Persistent GUI(dom);
myIS->Register(myGUI, myGUI->getName(), myGUI->getPeriod());

dom->Commit();

}
```

Table 7.16: The Executable Creating the Scheduler Objects

```
extern ConfigFile* CFO;
main(int argc, char **argv) {
int steps = atoi(argv[1]);
dom = new PDOM();
dom->beginsession(CFO->getMyDbName(), "session1");

PPredicate aPredicate = PAttribute("FrameworkObject::name") ==
      CFO->getMyISName();
LinkVstr<InternalSynchronizer> IS =
      PClassObj<InternalSynchronizer>.select(NULL, FALSE, aPredicate);
Link<InternalSynchronizer> myIS = IS[0];

for (int i = 1; i <= steps; i++){
  myIS->Go();
}

dom->Commit();

}
```

Table 7.17: The Executable to Run the Simulation

7.3.6 Simulation Parameters

The controllers used in this simulation do not use any parameters.

The `Physical` and `Regulation` layers are run at every time click. The `Coordination` and `GUI` layers are run every fourth time click. The time click step size is set to 0.1 seconds. The integration time step is set to 0.005 seconds. This information is specified in the simulation parameters file.

7.3.7 The GUI Debugger

The GUI debugger does not require any particular configuration.

7.3.8 Running the Simulation

The executable that runs the simulation is in Table 7.17. Two inputs are read in from the simulation parameters file: the name of the `InternalSynchronizer` to be simulated and the name of the database the simulation is stored in. Using this information the executable retrieves the corresponding `InternalSynchronizer` instance from the database.

The `ProcessCoordinator` runs for `duration` time clicks every time its `Go()` method is invoked. The main executable takes in an argument that specifies how many times the `Go()` method of the `ProcessCoordinator` should be invoked.

This particular executable chooses to `Commit` to database only at the end of simulation.

7.4 Implementation Critique

Evaluation criteria for a Customized OM and an OMS Application were discussed in Section 4.6. This section provides a critique of SmartAHS design and implementation. In particular, the SmartAHS must satisfy the requirements identified in Section 3.1.2.

7.4.1 Ease-of-use

We discuss the requirements of Section 4.6 in the context of SmartAHS.

- Relevance of Entities;

  SmartAHS attempts to provide abstractions for microsimulation. Most SmartAHS entities are "planned" objects; a process needs to be developed to convert software specifications into actual physical components.

- Locality of reference and learning curve;

  The developer of automation devices needs minimal understanding of the process model. The required information is limited to the inputs, outputs and the time scale of evolution of the object.

  The coordination between various automation devices requires more design effort.

  The developer of scheduling objects, usually the system architect, needs a good understanding of the process model; however, in general the system architect is shielded from the details of other objects.

  Adding entities requires the writing and compiling of C++ code. The steps to integrate a new entity were summarized in Figure 5.3. The steps required to set up a simulation are more streamlined. In particular, the parameter file and the graphical debugger provide quick specification mechanisms.

- Ease of specification;

  A number of tools are provided to simplify system specification. These tools have room for improvement. In particular, better syntax checking is needed both for the state machine code generator and the GOE grammar parser. A new tool is needed to automatically convert continuous time differential equations into C++ programming constructs. Currently this conversion is manual.

- Ease-of-evaluation;

  Monitor specification syntax is C++. However, once a monitor entity is created its use is straightforward. The use of the database simplifies the recording and the replaying of system state.

- System level issues;

  - Configuration management;

    The GOE is a tool for creating instance configurations. As this tool matures, it will be used to create automation strategies by mixing and matching existing entities. A configuration management tool is needed to track simulation runs. Other configuration management problems are addressed by system utilities such as Revision Control System (RCS).

  - Fault management;

    All C++ development tools can be used by SmartAHS. The graphical debugger aids in fault management. Better syntax checking is needed both for the state machine code generator and the GOE grammar parser.

  - Performance management;

    SmartAHS allows time driven objects to evolve at different time scales. Event driven schedulers execute the behavior of objects with outstanding events only. The users are allowed to set simulation granularity.

    The client-server architecture makes it possible to run the graphical debugger and the simulation on different workstations.

    Performance and memory use statistics are discussed in Section 7.4.4.

  - Access and security management;

    Application developers do not need write access to the SmartAHS classes. Proper implementation of SmartPATH will ensure that system users do not need write access to the SmartPATH classes.

  - Financial management;

    Financial management is beyond the scope of this thesis.

  - Resource Management;

    SmartAHS is designed to run on SunSparc and SGI platforms. Future releases should eliminate the dependence on the Versant database.

  - Planning and Design Management;

    Both SmartDb and SmartAHS are expected to evolve over time.

7.4.2 AHS Requirements

This section summarizes requirements that are not fully addressed by SmartAHS.

- All user requirements are addressed;

- Configuration requirements;

  Actual specification and implementation of automation devices and traffic entities are left to the application developer. The state machine language is designed to streamline protocol specifications.

  Currently, no special language is provided to define traffic rules. As these rules are better understood a particular specification language may emerge.

- Fault management requirements;

  Accident detection is part of SmartAHS. Current version of the graphical debugger has a read only interface. As the read-write interface is implemented, inducing component failures will be streamlined.

- Performance and planning requirements;

  Gateways need to be designed to interface with other packages.

- Modeling requirements;

  Better geometric highway representation support is needed. Automation device specification and implementation is left to application development.

- Validation and deployment;

  A new automated process is needed to meet these requirements. The modularity of SmartAHS is expected to provide a head-start.

  The state machine language provides verification support.

7.4.3 Software System Requirements

A number of desirable software system characteristics were discussed in Section 2.1 and revisited in Section 3.1.2. Here, these characteristics are discussed in the context of SmartAHS.

- Ability to associate physical and logical representations: Modularity;

  The object-oriented approach and the OMS Object Model make SmartAHS a modular system. Actual and planned physical objects are well encapsulated in classes.

- Ability to add new components to the system with minimal code rewrite: Openness, Modularity, Robustness;

  Application developers are able to add new objects by subclassing SmartAHS classes.

  The application developer and the system user are both capable of composing complex objects from simpler components.

  Current SmartAHS architecture does not provide sufficient software fault isolation. Since all objects are simulated in the same process, it may not always be possible to pinpoint the exact source of a software failure. A more robust architecture would simulate each control layer in a different process. However, this approach results in poor performance.

- Ability to collect arbitrary statistics during simulation: Openness, Modularity;

  SmartAHS enables monitor objects to collect and store arbitrary statistics at run-time.

  The entire simulation state can be saved in the OODB for later use. However, it is not practical to save the entire state after every transition since the commit time of a zone with thousand vehicles is in the order of seconds.

- Ability to run simulations with acceptable performance: Performance;

  Performance statistics are presented in the next section.

- Ability to adjust simulation granularity: Modularity, Openness;

  The scheduling objects are configurable like any other object.

- Ability to simulate up to 100.000 vehicles: Performance;

  A distributed version of SmartAHS is forthcoming. Current implementation introduces significant overhead for distribution support. Distribution is discussed in more detail in [34].

7.4.4 Performance Statistics

In this section we provide performance results for several simulation runs. The simulations were run on a Sparc10 workstation with 64Meg of memory.

Figures 7.3, 7.4, and 7.5 plot the amount of time taken for one second of highway simulation versus the number of vehicles simulated. The graphs have two curves; the first plots the total elapsed time, the second plots the time taken up by the regulation layer.

The simulation is started with a 2km long empty highway and a new vehicle is created every 2 seconds. The 2km highway accommodates 500 vehicles. Hence, the curves level off after 500 vehicles.

Vehicle positions on the highway are updated every 0.1 seconds. The integration time step is set at 0.005 seconds. The elapsed time is measured every 4 seconds of simulation time.

The first scenario, in Figure 7.3, uses the physical and regulation layers only. The vehicles enter the highway, remain single agents, traverse the highway, and eventually leave the simulation. The regulation layer determines the vehicle displacements through integration, the physical layer moves the vehicles on the highway based on these displacements.

About 98.5% of simulation time is taken up by the integration routines in the regulation layer and the two curves are barely distinguishable. The plot indicates that about 32 vehicles can be simulated in real time.

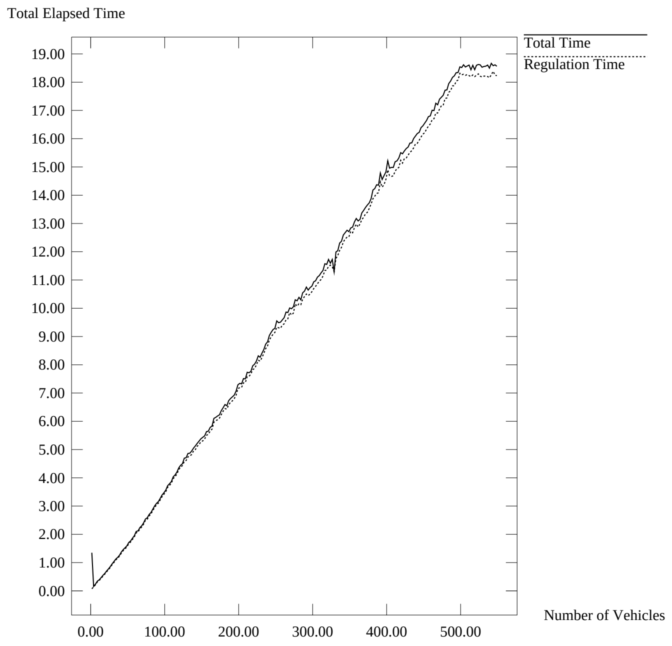

Figure 7.3: Single Agent Vehicles Moving On Highway

The second scenario, Figure 7.4, introduces the coordination layer. The coordination layer is scheduled twice a second. Vehicles enter the highway, try to merge with other vehicles, traverse the highway in platoons, and eventually leave the simulation. About 88% of elapsed time is taken up by the regulation layer. Platoons of size 2 are created. Since the displacement for a follower vehicle in a platoon is based on the leader vehicle's displacement, less time is taken up in integration routines. As a result, about 55 vehicles are simulated in real time.

The third scenario, Figure 7.5, provides a measure for the framework simulation overhead. Only the physical and regulation layers are used. Instead of calculating the vehicle displacement through integration, the displacement is hardcoded to 2m. Still, the regulation layer takes up 75% of the total elapsed time. Framework bookkeeping, such as, creating vehicles, removing vehicles, traversing objects, maintaining the vehicle positions within a lane, etc. is limited to 25%.

Finally, Figure 7.6 displays the memory use of the simulation. The resident and total sizes of the program are plotted against the number of vehicles in the simulation.

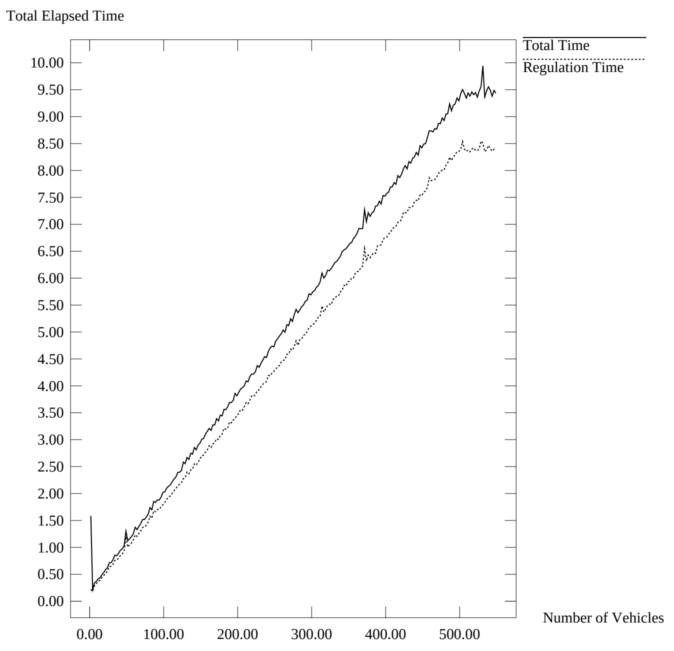

Figure 7.4: 2-Vehicle Platoons Moving On Highway

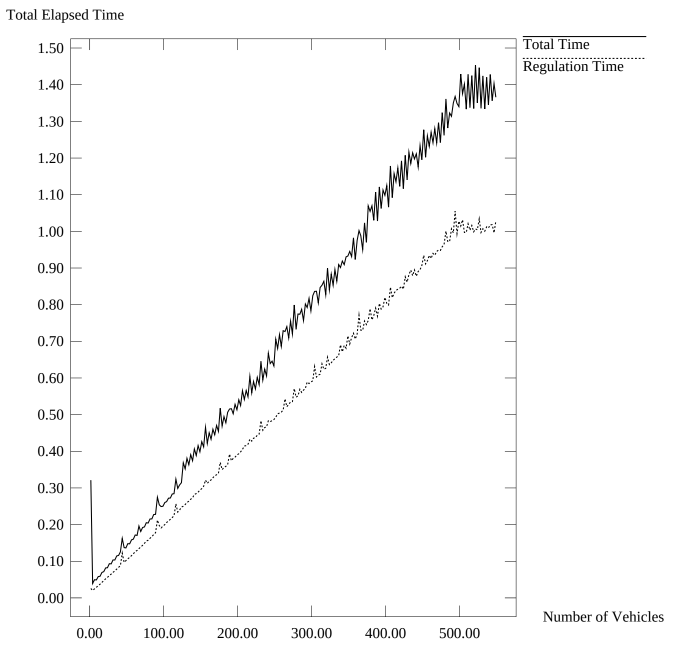

Figure 7.5: Single Agent Vehicles Moving On Highway with Hard-Coded Displacement

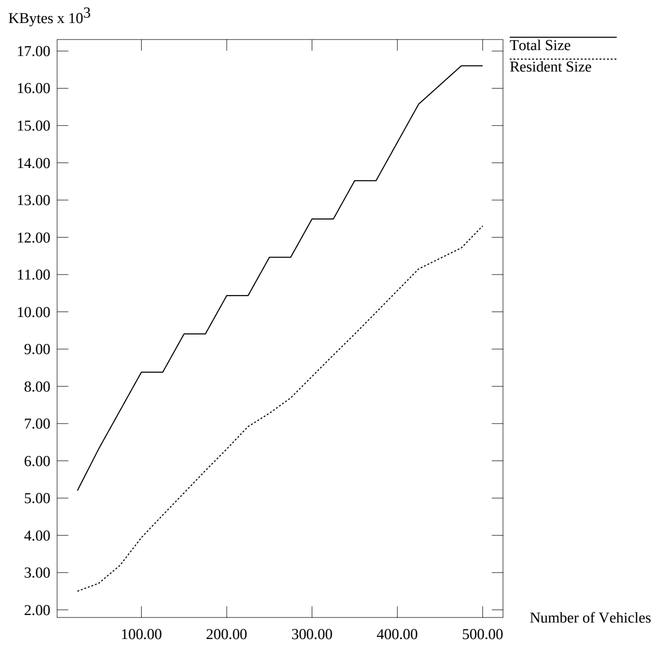

Figure 7.6: Memory Use of Simulation

Chapter 8

Conclusions

A summary of the thesis is provided in the Introduction.

As we stated earlier SmartDb and SmartAHS are "live" systems that will evolve over time. In this chapter we summarize some of the future evolution directions for the OMS Object Model, its SmartDb implementation, and the SmartAHS simulation framework.

8.1 The Object Model and SmartDb

The current SmartDb implementation supports only a subset of the Object Model constructs. Most of the Object Model constructs are actually implemented as part of SmartAHS.

A number of modules that were discussed in Chapter 6 as part of SmartAHS should become application independent SmartDb constructs. Some of these modules are already quite generic, others require some design work to eliminate their dependency on the highway automation application. These modules are:

- The graphical object editor;

  As the reader may have already noticed this module is generic and does not depend on the highway application.

- The graphical debugger;

  The graphical debugger is able to display any entity, so long its attributes are from a specific set. However, some work is required to be able to define application domain specific attributes and attribute icons.

- State machine language; and

  The code generator for the state machine language uses SmartAHS entities, such as `Receiver`s and `Transmitter`s, as target C++ classes. A more abstract SML syntax and a configurable code generator is needed to make the SML part of SmartDb. Each Customized OM can then provide customizations of the SML.

- Time and event driven scheduling objects;

  The `Traverser` traverses highways and is application specific. However, the concept of traversing a system for the extent of a class is generic. Similarly, `Packet`s and `PacketBox`es used by the `EventDriver`s can be used by any application domain. Application independent time and event driven scheduling mechanisms should become part of SmartDb.

As other specialized specification languages emerge, such as differential equation solvers and rule-based systems, their generic components should become part of SmartDb.

The current implementation of SmartDb does not have a clear interface with Versant and depends on Versant data types. Although the use of Versant data types has initially speeded up the implementation process, eventually, these data types should be replaced with vendor independent implementations. Eventually SmartDb should eliminate its dependence on Versant entirely and work with any persistent storage system.

8.2 SmartAHS Expansion Directions

An implementation critique of SmartAHS is provided in Section 7.4. Most suggested improvements are in the direction of "productizing" the SmartAHS simulation platform.

The current development efforts for SmartAHS are:

- More domain specific objects;

  More accurate sensor, transmitter, and receiver models, vehicle dynamics models and eventually some standard automation devices should become part of SmartAHS. More highway entities, such as bridges and tunnels, and more traffic entities are being developed.

- Other specialized specification languages;

  These specification languages may include differential equation solvers, rule-based specification languages, or even traffic rule specification languages.

- Streamlining simulation setup;

  Better graphical and command line interfaces are planned for simulation setup and monitor specification.

- Interfaces to other simulation and verification platforms; and

  SmartAHS has an open architecture that facilitates integration with other tools. However, such integration requires that the other tools be open as well. A SML-to-COSPAN translator is under development. Integration with urban traffic simulators is planned.

- SmartAHS Distribution.

  A distributed version of SmartAHS is under development. Current implementation introduces significant overhead for distribution support. Distribution is discussed in more detail in [34]

Bibliography

[1] "The Common Object Request Broker (CORBA): Architecture and Specification", Object Management Group (OMG), Framingham, MA, Dec. 1991.

Software Engineering

[2] K. Beck, *et.al.*"Can Structured Methods Be Objectified?", Panel discussion in *Proceedings of OOPSLA '91.*

[3] B. Boehm. *Software Engineering Economics*, Prentice-Hall, Engelwood Cliffs, NJ, 1981.

[4] G. Myers. *Composite/Structured Design*, New York Ny: Van Nostrand Reinhold, 1978, p21.

[5] Edward Yourdon and Larry Constantine. *Structured Design*, Yourdon Press, Englewood Cliffs, N.J., 1979.

[6] Edward Yourdon. *Modern Structured Analysis*, Yourdon Press, Englewood Cliffs, N.J., 1989.

Object Oriented Paradigm

[7] L. Cardelli and P. Wegner. "On Understanding Types, Data Abstraction, and Polymorphism", *Computing Surveys*, vol 17, no.4, 1985 pp. 471-522.

[8] Ralph E. Jonson. "How to Develop Frameworks", *OOPSLA Tutorial Notes*, ACM Press. 1993.

[9] T. Rentsch. "Object-Oriented Programming", *SIGPLAN Notices* vol 17(12), p 51.

[10] Peter Wegner. "Concepts and Paradigms of Object-Oriented Programming", *ACM SIGPLAN OOPS Messenger*, 1(1), Aug 1990.

Object Oriented Methodologies

[11] G. Booch. *Object Oriented Design with Applications*, Benjamin/Cummings, Redwood City, CA. 1991.

[12] [11] p 27.

[13] R.J.A. Buhr. *Practical Visual Techniques in System Design: With Applications to Ada* Prentice-Hall, Englewood Cliffs, N.J., 1991.

[14] P. Coad and E. Yourdon. *Object-Oriented Analysis*, Yourdon Press, Englewood Cliffs, NJ. 1991.

[15] Ivar Jacobson *et.al. Object-Oriented Software Engineering*,
Addison-Wesley, ACM Press, 1992.

[16] James Rumbaugh *et.al. Object Oriented Modeling and Design*,
Prentice-Hall, Englewood Cliffs, N.J., 1991.

[17] Ron Schultz. "A Game Plan for OOD Developers",
in *Open Systems Today*, Sept. 21 1992.

[18] Sally Shlaer and Stephen Mellor. *Object-Oriented Systems Analysis: Modeling the World in Data*,
Yourdon Press, Englewood Cliffs, N.J., 1988.

[19] Sally Shlaer and Stephen Mellor. *Object-Oriented Systems Analysis: Modeling the World in States*,
Yourdon Press, Englewood Cliffs, N.J., 1988.

[20] Rebecca Wirfs-Brock, B. Wilkerson, and L. Weiner. *Designing Object-Oriented Software*,
Prentice Hall, Englewood Cliffs, N.J. 1990.

Object Oriented Programming Languages

[21] Margaret Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*,
Addison-Wesley, Reading MA, 1990.

[22] Adele Goldberg and D. Robson. *Smalltalk 80: The language and its Implementation*,
Addison-Wesley, Reading, Massachusetts, 1983.

[23] Stanley B. Lippman. *C++ Primer*,
Addison-Wesley, Reading MA, 1991

Databases

[24] S. Ahmed, A. Wong, D. Sriram, and R. Logcher. "A Comparison of Object-Oriented Database Management Systems for Engineering Applications",
Research Report R91-12, Massachusetts Institute of Technology, Intelligent Engineering Systems Laboratory.

[25] M. Atkinson *et.al.* "The Object-Oriented Database System Manifesto",
Proceedings of 1st Intn. Conf. on Deductive and Object-Oriented Databases Kyoto, Japan, Dec. 1989

[26] Douglas K. Barry. "ODBMS Feature Listing",
Object Magazine, January-February 1993, pp. 48-53.

[27] Edgar F. Codd. "A Relational Model for Large Shared Data Banks",
Communications of the ACM, 13(60, pp. 377-387, Jun 1970.

[28] Edgar F. Codd. "Relational Completeness of Database Sublanguages",
Data Base Systems, R. Rustin ed.,Prentice Hall, Englewood Cliffs, N.J. 1972.

[29] Chris J. Date. *An Introduction to Database Systems*,
Addison-Wesley, Reading, Massachusetts, 1985.

[30] A.R. Hurson, S.H. Pakzad, and J. Cheng. "OODBMS: Evolution and Performance Issues",
IEEE Computer Magazine, February 1993, pp. 48-60.

[31] Larry Lai and Leon Guzenda. "How to Benchmark an OODBMS",
JOOP 1991.

[32] Bindu Rao. *Object-Oriented Databases*,
McGraw Hill, 1994.

[33] M. Stonebreaker *et.al.* “Third Generation Data Base System Manifesto”,
Proceedings of IFIP DS-4 Workshop on Object-Oriented Databases Windermere, England, July 1990.

Automated Highway Systems

[34] F. Eskafi. Work in progress
PhD thesis, UC Berkeley. June 1996.

[35] F. Eskafi and P. Varaiya. “SmartPath: Automatic Highway Simulator”
PATH Technical Memorandum, UC Berkeley. June 1992.

[36] F. Eskafi, Delnaz Khorramabadi, and P. Varaiya, “An Automatic Highway System Simulator”
Transportation Research -C Vol. 3, No, 1, pp. 1-17, 1995.

[37] A. Göllü, A. Deshpande, P. Hingorani, P. Varaiya. “SmartDb: An Object Oriented Simulation Framework for Highway Systems.”,
Fifth Annual Conference on AI, Simulation, and Planning in High Autonomy Systems, pp. 244-251, Gainesville, Florida. 1994.

[38] Ivy Hsu and Jean Walrand. “Communication Requirements and Network Design for IVHS”
PATH Technical Report, UCB-ITS-PWP-93-18.

[39] Steve Shladover *et.al.* “Automated Vehicle Control Developments in the PATH program”,
IEEE Trans. Vehicular Tech. Vol. 40. pp. 114-130. Feb. 1991.

[40] W.B. Stevens. “The Automated Highway System (AHS) Concept Analysis”,
MITRE Research Report MTR-93W0000123, (draft.), August 1993, McLean, Viriginia.

[41] Pravin Varaiya. “Smart Cars on Smart Roads: Problems of Control”,
IEEE Trans. Automatic Control Vol. 38, No 2. Feb. 1993.

[42] IVHS America. *Strategic Plan for Intelligent Vehicle-Highway Systems in the United States.*
Report No IVHS-AMER-92-3. 20 May 1992.

Power Distribution Systems

[43] Arthur R. Bergen. *Power System Analysis*,
Prentice-Hall, Englewood Cliffs, N.J, pp. 34-35.

[44] Matthew Brundjar. *Power Distribution Management*,
Work in progress, Masters’ thesis, UC Berkeley.

[45] William R. Cassel. “Distribution Management Systems: Functions and Payback”,
IEEE/PES 1992 Summer Meeting, Seattle, WA, July 12-16, 1992, 92 SM 450-7 PWRS.

[46] D. Chiang and M.E. Baran. “On the Existence and Uniqueness of Load Flow Solution for Radial Distribution Power Networks”,
IEEE Trans. on Circuits and Systems, Vol 37, No 3, pp. 410-416, March 1990

[47] Aleks Göllü and Felix F. Wu. “New Directions for Network Management”,
Proceedings of Distribution 2000. Melbourne, Australia. Electricity Supply Association of Australia.

[48] Y. Sekine, K.Takahashi, T.Sakaguchi. “Real-Time Simulation of Power System Dynamics”
Proceedings of 11th Power Systems Computation Conference, Avignon France, pp. 3, August 1993.

[49] Felix F. Wu and R. D. Masiello eds.;
Special Issue on Computers in Power System Operation, Proceedings of the IEEE, Vol 75, pp. 1553-1712, Dec 1987.

[50] EPRI Report. *Guidelines for Evaluating Distribution Automation*,
Technical Report EL-3728, November 1984.

[51] IEEE Tutorial Course,*Distribution Automation*,
IEEE Power Engineering Society 88EH0280-8-PWR 1988.

Network Management Systems

[52] Subodh Bapat. *Object-Oriented Networks*,
Prentice Hall, Englewood Cliffs, N.J.

[53] Uyless D. Black. *Network management standards : the OSI, SNMP, and CMOL protocols*
McGraw-Hill, New York, 1992

[54] Uyless D. Black. *Network management standards : SNMP, CMIP, TMN, MIBs, and object libraries*,
McGraw-Hill, New York, 1994.

[55] J. Case, M. Fedor, M. Schoffstall, and J. Davin. "A Simple Network Management Protocol",
Request for Comments 1098, DDN Network Information Center, SRI International, April, 1989

[56] Douglas E. Comer. *Internetworking with TCP/IP Volume 1: Principles, Protocols, and Architecture*,
Prentice-Hall, Englewood Cliffs, N.J.,1991.

[57] Douglas E. Comer and David L. Stevens. *Internetworking with TCP/IP Volume 2: Design, Implementation, and Internals*,
Prentice-Hall, Englewood Cliffs, N.J.,1991.

[58] Richard J. Edell, Nick McKeown and Pravin Varaiya. "Billing Users and Pricing for TCP Traffic",
To appear in *IEEE Journal of Selected Areas in Communications, Special Issue on "Advances in the Fundamentals of Networking."*

[59] Aleks Göllü and D. Moen. "Building Large Network Management Systems with C++"
Proceedings of OOPSLA. 1991.

[60] Aleks Göllü and W. Caplinger. "Performance Issues in Network Management Systems",
Proceedings of Wescon 1992.

[61] Aleks Göllü, W. Caplinger, and D. Moen. "IMS a Network Management Integrator Compliant with OSI/NM Forum",
Proceedings Silicon Valley Networking Conference, April 1991

[62] Aleks Göllü and G. Caravias. "Network Management in a Heterogeneous Environment",
Proceedings Wescon 91, San Francisco, California.

[63] Shau-Ming Lun, Felix Wu, Ning Xiao, and Pravin Varaiya. "NetPlan: An Integrated Network Planning Environment",
State of the Art in Performance Modeling and Simulation: Computer and Communication Networks, Vol.1, Kluwer Academic Publisher, 1993.

[64] J. B. Postel. "User Datagram Protocol",
Request for Comments 768, DDN Network Information Center, SRI International, August 1980.

[65] M. Rose. *The Open Book, A Practical Perspective*,
Prentice Hall.

[66] OSI/Network Management Forum; "Basic Reference Model", International Electrotechnical Committee 1984.

[67] OSI/Network Management Forum; "Application Services", Forum 002; Issue 1; January 1989

[68] OSI/Network Management Forum; "Object Specification Framework", Forum 003; Issue 1.0; September 1989

[69] "OSI Basic Reference Model", ISO 7498-1, 1992.

[70] "Common Management Information Service (CMIS) Definition", ISO 9596-1, 1992.

[71] "Common Management Information Protocol, Part 1:Specification", ISO 9596-1, 1992.

[72] "International Standardized Profiles — OSI Management — Management Functions", Parts 1-5, ISO ISP 12060, 1993.

[73] "Management Information Services — Structure of Management Information —" Parts 1-7, ISO 10165-1 to ISO 10165-7.

Formal Methods

[74] R. Alur, C. Courcoubetis, and D. Dill. "Model-Checking for Real Time Systems", *Proceedings 5th IEEE Symp. on Logic in Computer Science*, IEEE Computer Society Press, 1990.

[75] F.J. Barros, M.T.Mendes, and B.P. Zeigler. "Variable DEVS — Variable Structure Modeling Formalism: An Adaptive Computer Architecture Application", *Fifth Annual Conference on AI, Simulation, and Planning in High Autonomy Systems*, pp. 185-192, Gainesville, Florida. 1994.

[76] J. R. Büchi. "On a decision method in restricted second order arithmetic", *Proceedings of the International Congress on Logic, Methodology and Philosophy of Science*, pages 1–11. Stanford University Press, 1962.

[77] Akash Deshpande. *Control of Hybrid Systems*, PhD. thesis, UC Berkeley 1994.

[78] Aleks Göllü and Pravin Varaiya. "Hybrid Dynamical Systems", *Proceedings 28th Conf. on Decision and Control.* Vol. 3, pp. 2708-2712, Tampa FL, December 1989.

[79] C.A.R. Hoare. *Communicating Sequential Processes*, Prentice/Hall International, 1985

[80] G.P. Hong and T.G. Kim. "The DEVS Formalism: A Framework for Logical Analysis and Performance", *Fifth Annual Conference on AI, Simulation, and Planning in High Autonomy Systems*, pp. 170-178, Gainesville, Florida. 1994.

[81] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computations* Addison-Wesley, Reading Massachusetts, 1979

[82] Kemal Inan and Pravin Varaiya. "Finitely Recursive Process Models for Discrete Event Systems,"
IEEE Trans. Auto. Control, vol. AC-33, no. 7, pp. 626-639, July 1988.

[83] Kemal Inan and Pravin Varaiya. "Algebras of Discrete Event Models,"
Proceedings of the IEEE, vol. 77, no. 1, pp. 24-38, January 1989.

[84] J. McManis and P. Varaiya. "Suspension Automata: A Decidable Class of Hybrid Automata",
Proceedings 6th Workshop Computer-Aided Verification, Stanford CA 1994.

[85] R.Milner. *A Calculus of Communicating Systems*,
Springer-Verlag, 1980

[86] D.E. Muller. Infinite sequences and finite machines.
In *Proceedings of the 4th Annual Symposium on Switching Circuit Theory and Logical Design*, pages 3–16, IEEE, October 1963.

[87] C. M. Özveren. *Analysis and Control of Discrete Event Dynamic Systems: A State Space Approach.*
PhD thesis, MIT, 1989.

[88] H. Praehofer, F. Auernig, adn G. Resinger. "An Environment for DEVS-based multiformalisms simulation in Common Lisl/CLOS",
Discrete Event Dynamic Systems: Theory and Application, 3(2):119-149, 1993.

[89] P. Ramadge and W. Wonham. Supervisory control of a class of discrete event processes,
SIAM J. Control Optim., 25(1):206–230, January 1987.

[90] Bernard Zeigler. *Multifacetted modeling and discrete event simulation*,
Academic Press, London, Orlando, 1984.

[91] "Specification and Description Language SDL",
International Telecommunications Union-T Rec.Z.100 1988.

[92] "Estelle – A Formal Description Technique Based on Extended State Transition Model"
ISO9074, 1988

[93] "LOTOS – A Formal Description Technique Based on the Temporal Ordering of Observational Behavior"
ISO8807, 1989

Simulation and Verification Tools

[94] J.C. Fernandez, H. Garavel, L. Mounier, A. Rasse, C. Rodríguez, and J. Sifakis. "A Toolbox for the Verification of LOTOS Programs",
Proceedings of the 14th International Conference on Software Engineering, Melbourne, Australia, 192.

[95] Z. Har'El and R. P. Kurshan. *COSPAN User's Guide*,
AT&T Bell Laboratories, Murray Hill, NJ, 1987.

[96] H. Schwetman. *CSIM Reference Manual (Revision13)*,
Microelectronics and Computer Technology Corporation, 3500 West Balcones Center Drive, Austin, TX 78759, 1989

[97] *The Almagest.* Ptolemy Manual Vol 1-4,
Version 0.5.2, College of Engineering, UC Berkeley, 1995

Chapter 9

Appendix

9.1 C++ Overview

This section provides a quick overview of the C++ object model and its implementation of object-oriented constructs. The reader is recommended to read either [21] or [23] for a complete treatment.

C++ is everything C is and more; in particular it provides the "class" construct. A sample class definition can be found in Table 9.1.

C++ has many built-in data types. It enables the user to define new types and new classes. Each class is a data type. (Clearly the converse is false.). Some C++ data types are summarized below:

`int`	integer.
`enum`	an enumerated set of integers, a name is assigned to each integer in the set.
`char`	a single character.
`float`	a floating point number.
`void`	any type.

C++ also provides some built-in operators. These are summarized below:

`*`		The pointer operator.
	`int* foo`	Declares `foo` as a pointer to an integer.
	`*foo`	Assuming that `foo` is a pointer, it dereferences the pointer to get the value.
	`int* f(char*)`	A function that accepts a pointer to a character and returns a pointer to an integer.
`&`		The address operator.
	`&foo`	Returns the address of `foo`.
	`void f(int&)`	A function that takes in a reference to an integer. When invoked as `f(foo)`, the argument passed to `f` is the address of (a reference to) `foo`.
`[]`		The array operator.
	`int[] foo`	Declares `foo` as an array of integers.
`new`		The instantiation operator.
	`new Car`	Instantiates a `Car`.
`delete`		The deletion operator.
	`delete foo`	Deletes the instance foo.

```
class Car: public Vehicle
{
public:

/* Two constructors. The constructor is overloaded. */
Car(char* name);
Car();
/* The destructor */
~Car();

/* Virtual Public Instance Method */
virtual int Run();

/* Public Class Method */
static int HowManyCars();

protected:

/* Protected Instance Method */
int GetLength();

private:

/* Private static attribute, used to maintain cardinality of the extent */
static int numberOfCars;

/* Private attribute */
float length;

};
```

Table 9.1: Sample C++ Class Definition

sizeof(.)		Returns the memory size of its argument in bytes.
	sizeof(int)	Returns size of integer.
`++`		Increment operator.
	`int foo; foo++`	Equivalent to `foo = foo +1`
	`int* foo; foo++`	Increments the pointer by `sizeof(int)`. Consider `int i; int[] bar;`. If `foo == &(bar[i])` then `foo++ == &(bar[i++])`

A class consists of its members, these are attributes and methods. Attributes can be of any data type. Methods take arguments and return a value. C++ methods can take arguments by value and by reference. Methods have access to the attributes of their class.

Classes can declare their members to be private, protected, or public. Public members are accessible to all other classes, protected members are accessible only to subclasses, private members are only accessible within that class.

There are two special methods: the constructor and the destructor. They both must have the same name as the class. The destructor name is preceded by $\sim$.

The constructor is invoked on the class to create instances. Declaring the constructor of a class private makes the class abstract.

An instance of a class assigns values to the attributes. In C++, an instance is identified by a pointer to it. Methods are invoked on instances. The syntax to invoke a method is
`instancePointer->methodName(arguments)`.
When a method is invoked on an instance, it uses the attribute values of that instance.

A class may contain static members. Static attributes take on values within the class and are shared by the extent of the class. If they are public, other objects can access them with the syntax:
`ClassName::attributeName`.
Static methods are invoked on the class. If they are public other objects can invoke them with the syntax:
`ClassName::methodName(arguments)`.
In the above example any class can invoke `Car::HowManyCars()`, however, `Car::NumberOfCars` is not accessible to other classes.

C++ supports single and multiple inheritance. A subclass can add new attributes and methods or specialize existing methods. Attribute specialization[1], method argument specialization, and method return value specialization are not supported.

In the above example `Car` inherits all public and protected members of `Vehicle` and adds a number of methods and attributes of its own.

C++ provides signature overloading. Signature identification takes place at compile time. However, two methods $f(a)$ and $f(b)$ have the same signature if b is a subclass of a. In the above example the constructor is overloaded.

C++ provides dynamic binding and parametric polymorphism. If a method is declared to be virtual it can be overwritten (specialized) in a subclass. At run-time C++ determines the proper method to invoke based on the specific type of an instance. In the above example the `Run` method is virtual.

[1]Here specialization refers to restricting the domain of an attribute

9.2 The Versant OODB

Bindu Rao observes "Versant is an object-oriented database system that is particularly useful when several teams work in parallel on different parts of a project, project requirements change over time, diverse activities must be tied together, a distributed environment is necessary, or complex data types are employed" [32].

Versant was chosen for the SmartAHS implementation for the following key reasons:

- Versant provides persistence through inheritance, as such the application developer does not need to be aware of the persistent nature of the program;
- Versant provides a library of persistent data types such as lists and arrays. The use of these constructs speeds up the development process;
- Versant provides instance level locking which is essential for distributed simulation;
- Versant provides dirty reads which is essential for distributed simulation;

Other features of Versant are summarized in [32].

9.2.1 Versant Data Types

Versant provides a set of persistent data types. These are:

`o_1b, o_2b, o_4b`	n-byte signed integer, an elemental type.
`o_u1b, o_u2b, o_u4b`	n-byte unsigned integer, an elemental type.
`o_double, o_float`	floating point numbers, an elemental type.
`o_bool`	Boolean TRUE or FALSE, an elemental type.
`Vstr<type>`	dynamic array of any of above elemental types.
`Link<class>`	persistent equivalent of a pointer.
`LinkVstr<class>`	array of links to persistent objects.
`BiLink<class, returnAttr>`	bidirectional link between 2 persistent objects.
`BiLinkVstr<class, returnAttr>`	array of bidirectional links.
`PString`	persistent string, used as part of a class.
`Vstring`	persistent stand-alone string.
`VVList<class>`	list of persistent objects.
`VVArray<class>`	array of persistent objects.
`VEList<type>`	list of persistent elemental types.
`VEArray<type>`	array of persistent elemental types.

The `VVList, VEList, VVArray`, and `VEArray` objects provide a number of methods these are:

`set(const VVList<type>&)`	Copies contents of passed in list to this list and drop its own contents.
`o_u4b size() const`	Returns number of elements in the list.
`o_bool valid_key(o_u4b pos) const`	`TRUE` if `pos` is valid for this list.
`type first() const`	Returns first element of the list.
`type last() const`	Returns last element of the list.
`replace_first(type elem)`	Replaces the first element with `elem`.
`replace_last(type elem)`	Replaces the last element with `elem`.
`replace_position(o_u4b pos, type elem)`	Replaces the element at `pos` with `elem`.
`insert_first(type elem)`	Inserts `elem` at first `pos` in list.
`insert_last(type elem)`	Inserts `elem` at last `pos` in list.
`insert_position(o_u4b pos, type elem)`	Inserts `elem` before position `pos`.
`delete_first()`	Deletes first element in list.
`delete_last()`	Deletes last element in list.
`delete_position(o_u4b pos)`	Deletes the element in list at position `pos`.

Versant also provides a macro for safe type-casting. The line
`Link<A> foo = L_AS(derivedClass, baseClass);`
converts the pointer `baseClass` to a member of class `derivedClass` and returns a pointer if possible. If the conversion is not possible, it raises an exception.

9.2.2 Versant Limitations

Versant is an evolving product. If the following limitations of Versant were addressed, the implementation of SmartAHS would be improved and significantly simplified.

- Partial commits are not supported. In particular there is no construct to commit the extent of a class;
- Partial object retrieval is not possible;
- There are no distributed directory services;
- Versioned objects can't be migrated;
- Migration is not supported in shared sessions;
- The Link construct can't be subclassed to implement relationship semantics;
- The server does not provide events or triggers.

9.3 The Traverser Class

```
/*
 * Copyright (c) 1994-1996 The Regents of the University of California.
 * All rights reserved.
 *
 * Permission is hereby granted, without written agreement and without
 * license or royalty fees, to use, copy, modify, and distribute this
 * software and its documentation for any purpose, provided that the
 * above copyright notice and the following two paragraphs appear in
 * all copies of this software.
 *
 * IN NO EVENT SHALL THE UNIVERSITY OF CALIFORNIA BE LIABLE TO ANY PARTY FOR
 * DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING OUT
 * OF THE USE OF THIS SOFTWARE AND ITS DOCUMENTATION, EVEN IF THE UNIVERSITY OF
 * CALIFORNIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 *
 * THE UNIVERSITY OF CALIFORNIA SPECIFICALLY DISCLAIMS ANY WARRANTIES,
 * INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY
 * AND FITNESS FOR A PARTICULAR PURPOSE.  THE SOFTWARE PROVIDED HEREUNDER IS
 * ON AN "AS IS" BASIS, AND THE UNIVERSITY OF CALIFORNIA HAS NO OBLIGATION TO
 * PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.
 */

/*
  Traverser.C
*/

/*
This is a simplified version of the actual Traverser.
This version works for one Zone only.

Also we need to resolve how to access the other database
*/

#include <Traverser.h>
#include <LaneContainer.h>
#include <Access.h>
#include <Sink.h>
#include <Source.h>
#include <Junct1to2.h>
#include <Junct2to1.h>
#include <Zone.h>
#include <Highway.h>
#include <Section.h>
#include <EntrySection.h>
#include <ExitSection.h>
#include <Entry.h>
#include <Exit.h>
#include <Lane.h>
#include <State.h>
#include <Vehicle.h>

extern "C" {
#include <unistd.h>
}

#include <cxxcls/try.h>

Implement PClassObj<Traverser>;

// ============================================================
Traverser::Traverser(char* name):
Layer(name)
{
  // The following to be used for bookkeeping purposes
  // Get number of Junct1to2s in my zone
    myJ1to2size = (MyZone->GetJunct1to2s()).size();
  // create array of integers representing them
```

```
    myJ1to2s = new JSTATE[myJ1to2size];
  // initialize list
  for (int i = 0; i < myJ1to2size; i++){
    myJ1to2s[i] = ZERO;
  }
}

/**
Traverser::Traverser()
{
  // The following to be used for bookkeeping purposes
  // Get number of Junct1to2s in my zone
    myJ1to2size = (MyZone->GetJunct1to2s()).size();
  // create array of integers representing them
    myJ1to2s = new JSTATE[myJ1to2size];
  // initialize list
  for (int i = 0; i < myJ1to2size; i++){
    myJ1to2s[i] = ZERO;
  }
}
**/

// ===============================================================
Traverser::~Traverser()
{
}

// ===============================================================
int
Traverser::Go()

{
  o_u4b j;

  Start();

  // Get the List of Sinks
  VVList<Sink> mySinks = MyZone->GetSinks();
  o_u4b numSinks = mySinks.size();
  for (j=0; j < numSinks; j++){
    Link<Sink> currSink = &mySinks[j];
    // Deals with boundary condition and then calls ZapHighway
    ZapSink(currSink);
  }

  End();

  return 0;
}

// ===============================================================
int
Traverser::ZapHighway(Link<LaneContainer> inCurr,
                      Link<LaneContainer> inNext,
                      int alreadyExecuted){

  Link<LaneContainer> currLC = inCurr;
  Link<LaneContainer> nextLC = inNext;
  o_u4b i;

  LaneContainer *currLCPtr = inCurr;
  LaneContainer *nextLCPtr = inNext;

  Link<Highway> currHighway = (L_AS(Section, currLC))->GetHighway();
  Highway *currHighwayPtr = currHighway;

  // How many sections do I have to execute
  o_u4b  mySize = (currHighway->GetSections()).size() +
    (currHighway->GetEntrySections()).size() +
    (currHighway->GetExitSections()).size() - alreadyExecuted - 1;
  // Now iterate thru the mySize sections in this segment
  for (i=0; i < mySize; i++) {
    Link<LaneContainer> prevLC = currLC->GetPrevLaneContainer1();
    LaneContainer *prevLCPtr = prevLC;
    ZapSection( prevLC, currLC, nextLC );
    nextLC = currLC;
    currLC = prevLC;
    // prevLC set at beginning of loop

  }

  // We are about to hit the beginning of the segment
  // Get the prevLC and delegate

  Link<LaneContainer> prevLC = currLC->GetPrevLaneContainer1();
  const char* myName = prevLC->class_name();
  // Check and see if we hit a Highway boundary
  if (!strcmp(myName, "Source")){ // I am a Source

    // Deal with this elsewhere
    ZapSource(L_AS(Source, prevLC), currLC, nextLC);
  }
  else if (!strcmp(myName, "Junct2to1")){
    // We have to branch
    ZapJunct2to1(L_AS(Junct2to1, prevLC), currLC, nextLC);
  }
  else if (!strcmp(myName, "Junct1to2")){
    ZapJunct1to2(L_AS(Junct1to2, prevLC));
  }
  else {

    exit(0);
  }
}

// =================================================================
int
Traverser::ZapSink(Link<Sink> currSink)
{
  Sink *currSinkPtr = currSink;
  Link<LaneContainer> prevLC = currSink->GetPrevLaneContainer1();
  Link<LaneContainer> prevprevLC = prevLC->GetPrevLaneContainer1();
  LaneContainer *prevLCPtr = prevLC;
  LaneContainer *prevprevLCPtr = prevprevLC;

  PAbsorber(currSink);

  PrepareSection(prevLC);
  EAbsorber(currSink);

  PrepareSection(prevprevLC);
  ExecuteSection(prevLC);
  FAbsorber(currSink);

  // We have executed one section
  ZapHighway(prevprevLC, prevLC, 1);

  return 0;
}

// =================================================================
// ??????? Should prevLC had been cast as a Source
int
Traverser::ZapSource(Link<Source> mySource,
   Link<LaneContainer> currLC, Link<LaneContainer> nextLC)
{
  PGenerator(mySource);
  ExecuteSection(currLC);
  FinishSection(nextLC);
  EGenerator(mySource);
  FinishSection(currLC);

  FGenerator(mySource);

  return 0;
}

// ============================================================
int
Traverser::ZapJunct1to2(Link<Junct1to2> myJunct)
{
  // Get its index -> check error
  o_u4b myIndex = GetIndex( MyZone->GetJunct1to2s(), myJunct );
  // Get its State
  JSTATE myJState = myJ1to2s[myIndex];
  if (myJState == ZERO){ // First time around
    myJ1to2s[myIndex] = ONE; // Can't do much else now
    return 0;
  }
  else if (myJState == ONE) { // Other side is done
    myJ1to2s[myIndex] = ZERO;
    // We have just found a Junct1to2 where the other side was marked
    // Finish Both incoming Highways and move to the next one

    Link<LaneContainer> nextLC1 = myJunct->GetNextLaneContainer1();
    Link<LaneContainer> nextnextLC1 = nextLC1->GetNextLaneContainer1();
    Link<LaneContainer> nextLC2 = myJunct->GetNextLaneContainer2();
    Link<LaneContainer> nextnextLC2 = nextLC2->GetNextLaneContainer1();
    Link<LaneContainer> prevLC = myJunct->GetPrevLaneContainer1();
    Link<LaneContainer> prevprevLC = prevLC->GetPrevLaneContainer1();

    PLC(myJunct);

    ExecuteSection(nextLC1);
    FinishSection(nextnextLC1);
    ExecuteSection(nextLC2);
    FinishSection(nextnextLC2);

    PrepareSection(prevLC);
    ELC(myJunct);
    FinishSection(nextLC1);
    FinishSection(nextLC2);

    PrepareSection(prevprevLC);
    ExecuteSection(prevLC);
    FLC(myJunct);


    // We have not executed any sections of the segment
    ZapHighway(prevprevLC, prevLC, 1);
  }

  return 0;
}

// ============================================================
int
Traverser::ZapJunct2to1(Link<Junct2to1> myJunct,
Link<LaneContainer> inCurr, Link<LaneContainer> inNext){

  Link<LaneContainer> prevLC1 = myJunct->GetPrevLaneContainer1();
  Link<LaneContainer> prevprevLC1 = prevLC1->GetPrevLaneContainer1();
  Link<LaneContainer> prevLC2 = myJunct->GetPrevLaneContainer2();
  Link<LaneContainer> prevprevLC2 = prevLC2->GetPrevLaneContainer1();

  PLC(myJunct);
  ExecuteSection(inCurr);
  FinishSection(inNext);
  PrepareSection(prevLC1);
  PrepareSection(prevLC2);
  ELC(myJunct);
  FinishSection(inCurr);

  PrepareSection(prevprevLC1);
  PrepareSection(prevprevLC2);
  ExecuteSection(prevLC1);
  ExecuteSection(prevLC2);
  FLC(myJunct);

  // Now recurse on both prev Highways
  // Note that we have already executed one section
  ZapHighway(prevprevLC1, prevLC1, 1);
  ZapHighway(prevprevLC2, prevLC2, 1);

  return 0;
}

// ==============================================================
int
Traverser::ZapSection(Link<LaneContainer> prevLC,
      Link<LaneContainer> currLC, Link<LaneContainer> nextLC ){

  PrepareSection(prevLC);
  ExecuteSection(currLC);
  FinishSection(nextLC);

  return 0;
}

// ==============================================================
int
Traverser::PrepareSection(Link<LaneContainer> myLC){

  const char* myName = myLC->class_name();

  if (!strcmp(myName, "Section")){
    PLC(myLC);
  }
  else if (!strcmp(myName, "EntrySection")){
    PLC(myLC);
    PGenerator( (L_AS(EntrySection, myLC))->GetEntry() );
  }
  else if (!strcmp(myName, "ExitSection")){
    PAbsorber( (L_AS(ExitSection, myLC))->GetExit() );
    PLC(myLC);
  }
  else{ // Oooooops!

    exit(0);
  }
  return 0;
}

// ==============================================================
int
Traverser::ExecuteSection(Link<LaneContainer> myLC){

  const char* myName = myLC->class_name();

  if (!strcmp(myName, "Section")){
    ELC(myLC);
  }
  else if (!strcmp(myName, "EntrySection")){
    ELC(myLC);
    EGenerator( (L_AS(EntrySection, myLC))->GetEntry() );
  }
  else if (!strcmp(myName, "ExitSection")){
    EAbsorber( (L_AS(ExitSection, myLC))->GetExit() );
    ELC(myLC);
  }
  else{

    exit(0);
  }
  return 0;
}

// ==============================================================
int
Traverser::FinishSection(Link<LaneContainer> myLC){

  const char* myName = myLC->class_name();

  if (!strcmp(myName, "Section")){
    FLC(myLC);
  }
  else if (!strcmp(myName, "EntrySection")){
    FLC(myLC);
    FGenerator( (L_AS(EntrySection, myLC))->GetEntry() );
  }
  else if (!strcmp(myName, "ExitSection")){
    FAbsorber( (L_AS(ExitSection, myLC))->GetExit() );
    FLC(myLC);
  }
  else{ // Oooooops!

    exit(0);
  }

  return 0;
}

// ==============================================================
int
Traverser::GetIndex(VVList<Junct1to2> myList, Link<Junct1to2> myObj){

  o_u4b mySize = myList.size();
  Link<Junct1to2> currJunct;

  for (o_u4b i = 0; i < mySize; i++){
    currJunct = &myList[i];
    if (currJunct == myObj){
      return i;
    }
  }

  return -1;
}
```

9.4 Sample State Machine

9.4.1 The CoordSup state machine

```
machineName: CS;

children: CMI, CLCI, CMR, CLCR, CSR;

states: Idle, SetPL, MR, MI, AmILMR;

InputEvents: YES, NO;

inputMessages: MRMessage:MERGE_REQUEST, Message:MERGE_R_DONE,
               Message:MERGE_R_FAILED, Message:MERGE_I_DONE,
               Message:MERGE_I_FAILED, Message:SET_PL;

outputMessages: Message:BUSY, MRMessage:MERGE_REQUEST_F,
                Message:START;

correspondants: aCar, otherCar, PL;

beginState: Idle;

inputs: CoordSupIn;

outputs: CoordSupOut;

timers: mergeTimer;

transitions:
Idle   aCar=MERGE_REQUEST       ()                              AmILMR  CMR  NULL
Idle   mergeTimer               (Child:START)                   MI      CMI  NULL
Idle   PL=SET_PL                ()                              SetPL   NULL NULL
SetPL  YES                      ()                              Idle    NULL NULL
AmILMR YES                      (Child:forward:MERGE_REQUEST_F) MR      CMR  NULL
AmILMR NO                       PL:forward                      Idle    NULL NULL
MR     otherCar=MERGE_REQUEST   otherCar:BUSY                   MR      NULL NULL
MR     Child:MERGE_R_DONE       ()                              Idle    NULL CMR
MR     Child:MERGE_R_FAILED     ()                              Idle    NULL CMR
MR     mergeTimer               mergeTimer:20                   MR      NULL NULL
MI     otherCar=MERGE_REQUEST   otherCar:BUSY                   MI      NULL NULL
MI     Child:MERGE_I_DONE       ()                              Idle    NULL CMI
MI     Child:MERGE_I_FAILED     mergeTimer:10                   Idle    NULL CMI
```

9.4.2 The Generated Header File CS.h

```
#ifndef CS_H
#define CS_H

#include <iostream.h>
#include <FSM.h>
#include <FSMState.h>
#include <FSMInstance.h>

// State Classes
// ======================================================================
class CSIdleState: public FSMState {
public:
     CSIdleState();
     ~CSIdleState();
     virtual int Enter(Link<FSMInst_State> theState,
                 Link<InputProjection> i, Link<OutputProjection> o);
     virtual int Exit(Link<FSMInst_State> theState,
                 Link<InputProjection> i, Link<OutputProjection> o);
     virtual int ProcessMessage(Link<FSMInst_State> theState);
     virtual int PostMessage(Link<FSMInst_State> theState);
     virtual int ProcessTimeout(o_u4b handle, Link<FSMInst_State> theState);
 };
// ========================================================================
class CSSetPLState: public FSMState {
public:

     CSSetPLState();
     ~CSSetPLState();
     virtual int Enter(Link<FSMInst_State> theState,
                 Link<InputProjection> i, Link<OutputProjection> o);
     virtual int Exit(Link<FSMInst_State> theState,
                 Link<InputProjection> i, Link<OutputProjection> o);
     virtual int ProcessMessage(Link<FSMInst_State> theState);
     virtual int PostMessage(Link<FSMInst_State> theState);
     virtual int ProcessTimeout(o_u4b handle, Link<FSMInst_State> theState);
 };
// ========================================================================
class CSMRState: public FSMState {
public:
     CSMRState();
     ~CSMRState();
     virtual int Enter(Link<FSMInst_State> theState,
                 Link<InputProjection> i, Link<OutputProjection> o);
     virtual int Exit(Link<FSMInst_State> theState,
                 Link<InputProjection> i, Link<OutputProjection> o);
     virtual int ProcessMessage(Link<FSMInst_State> theState);
     virtual int PostMessage(Link<FSMInst_State> theState);
     virtual int ProcessTimeout(o_u4b handle, Link<FSMInst_State> theState);
 };
// ========================================================================
class CSMIState: public FSMState {
public:
     CSMIState();
     ~CSMIState();
     virtual int Enter(Link<FSMInst_State> theState,
                 Link<InputProjection> i, Link<OutputProjection> o);
     virtual int Exit(Link<FSMInst_State> theState,
                 Link<InputProjection> i, Link<OutputProjection> o);
     virtual int ProcessMessage(Link<FSMInst_State> theState);
     virtual int PostMessage(Link<FSMInst_State> theState);
     virtual int ProcessTimeout(o_u4b handle, Link<FSMInst_State> theState);
 };
// ========================================================================
class CSAmILMRState: public FSMState {
public:
     CSAmILMRState();
     ~CSAmILMRState();
     virtual int Enter(Link<FSMInst_State> theState,
                 Link<InputProjection> i, Link<OutputProjection> o);
     virtual int Exit(Link<FSMInst_State> theState,
                 Link<InputProjection> i, Link<OutputProjection> o);
     virtual int ProcessMessage(Link<FSMInst_State> theState);
     virtual int PostMessage(Link<FSMInst_State> theState);
     virtual int ProcessTimeout(o_u4b handle, Link<FSMInst_State> theState);
 };

// State Machine Classes
// ========================================================================
class CSAlgorithm: public FSMAlgorithm {
  public:
    CSAlgorithm();
    ~CSAlgorithm();
  private:

};
// ========================================================================
class CSInstance: public FSMInstance {
  public:
    CSInstance();
    ~CSInstance();
    Link<CSInst_State> myState;
    static Link<CSAlgorithm> myAlg;

    virtual int Reconnect();
};
// ======================================================================
class CSInst_State: public FSMInst_State {
public:
     CSInst_State();
     ~CSInst_State();
      Link<Receiver> aCar;
      Link<Receiver> otherCar;
      Link<Receiver> PL;
      o_u4b mergeTimer;
      o_u4b lcTimer;
     virtual int UpdateCorr();
};

#endif
```

9.4.3 The Generated Source File CS.C

```
#include <iostream.h>
#include <CS.h>
#include <Message.h>
#include <def.h>
#include <InVehTransmitter.h>
#include <Vehicle.h>
#include <MRMessage.h>
#include <Message.h>
#include <LinkMessage.h>
#include <MRMessage.h>
#include <CoordSupOut.h>
#include <CoordSupIn.h>
#include <Zone.h>
#include <ConfigFile.h>

extern ConfigFile *CFO;

Implement PClassObj<CSIdleState>;
Implement PClassObj<CSSetPLState>;
Implement PClassObj<CSMRState>;
Implement PClassObj<CSMIState>;
Implement PClassObj<CSAmILMRState>;
Implement PClassObj<CSAlgorithm>;
Implement PClassObj<CSInst_State>;
Implement PClassObj<CSInstance>;

CSIdleState::CSIdleState() {SetName("CSIdle");}
CSIdleState::~CSIdleState() {}
CSSetPLState::CSSetPLState() {SetName("CSSetPL");}
CSSetPLState::~CSSetPLState() {}
CSMRState::CSMRState() {SetName("CSMR");}
CSMRState::~CSMRState() {}
CSMIState::CSMIState() {SetName("CSMI");}
CSMIState::~CSMIState() {}
CSAmILMRState::CSAmILMRState() {SetName("CSAmILMR");}
CSAmILMRState::~CSAmILMRState() {}

// ======================================================================
CSAlgorithm::CSAlgorithm() {

CSIdleState* myIdleStateObject= new Persistent CSIdleState;
CSSetPLState* mySetPLStateObject= new Persistent CSSetPLState;
CSMRState* myMRStateObject= new Persistent CSMRState;
CSMIState* myMIStateObject= new Persistent CSMIState;
CSAmILMRState* myAmILMRStateObject= new Persistent CSAmILMRState;
```

```
StateList.insert_last(*myIdleStateObject);
StateList.insert_last(*mySetPLStateObject);
StateList.insert_last(*myMRStateObject);
StateList.insert_last(*myMIStateObject);
StateList.insert_last(*myAmILMRStateObject);

startState = myIdleStateObject;
MessageList.insert_last(MERGE_REQUEST);
MessageList.insert_last(MERGE_R_DONE);
MessageList.insert_last(MERGE_R_FAILED);
MessageList.insert_last(MERGE_I_DONE);
MessageList.insert_last(MERGE_I_FAILED);
MessageList.insert_last(SET_PL);

EventList.insert_last(YES);
EventList.insert_last(NO);
EventList.insert_last(MI);
EventList.insert_last(MERGETIMEROUT);

myIdleStateObject->InsertMTransition
        (MERGE_REQUEST, NOPOST , myAmILMRStateObject, "CMR", NULL);
myIdleStateObject->InsertETransition
        (MERGETIMEROUT, POST , myMIStateObject, "CMI", NULL);
myIdleStateObject->InsertMTransition
        (SET_PL, NOPOST , mySetPLStateObject, NULL, NULL);
mySetPLStateObject->InsertETransition
        (YES, NOPOST , myIdleStateObject, NULL, NULL);
myAmILMRStateObject->InsertETransition
        (YES, POST , myMRStateObject, "CMR", NULL);
myAmILMRStateObject->InsertETransition
        (NO, POST , myIdleStateObject, NULL, NULL);
myMRStateObject->InsertMTransition
        (MERGE_REQUEST, POST , myMRStateObject, NULL, NULL);
myMRStateObject->InsertMTransition
        (MERGE_R_DONE, NOPOST , myIdleStateObject, NULL, "CMR");
myMRStateObject->InsertMTransition
        (MERGE_R_FAILED, NOPOST , myIdleStateObject, NULL, "CMR");
myMRStateObject->InsertETransition
        (MERGETIMEROUT, POST , myMRStateObject, NULL, NULL);
myMIStateObject->InsertMTransition
        (MERGE_REQUEST, POST , myMIStateObject, NULL, NULL);
myMIStateObject->InsertMTransition
        (MERGE_I_DONE, NOPOST , myIdleStateObject, NULL, "CMI");
myMIStateObject->InsertMTransition
        (MERGE_I_FAILED, POST , myIdleStateObject, NULL, "CMI");
}

CSAlgorithm::~CSAlgorithm() {}

Link<CSAlgorithm> CSInstance::myAlg= NULL_LINK;


// ===========================================================================
CSInstance::CSInstance() {
  SetName("CS");
  if (myAlg == NULL_LINK) {
    LinkVstr<CSAlgorithm> aAlg =
      PClassObj<CSAlgorithm>.select(CFO->getMyDbName(), FALSE,
    NULL_PREDICATE);
    if (aAlg.size() > 0) {
      if (aAlg[0] != NULL_LINK)
        myAlg = aAlg[0];
    }
    else {
      myAlg = new Persistent CSAlgorithm;
      LinkVstrAny listOfObjects;
      listOfObjects.add(myAlg);
      myAlg->GetContainees(listOfObjects);
      dom->gwriteobjs(listOfObjects);
      for (o_4b i = 0; i < listOfObjects.size(); i++)
        dom->downgradelock(listOfObjects[i], NOLOCK);
    }
  }
  theAlg = (CSAlgorithm*)myAlg;
  myState = new Persistent CSInst_State;
  theState = (CSInst_State*)myState;
  currState= (CSInst_State*)myState;
  myState->theInst = this;
  theState->curState = myAlg->startState;
  outputs = new Persistent CoordSupOut;
  outputs->SetFSM(this);
  inputs = new Persistent CoordSupIn;
  inputs->SetFSM(this);
  myBigBen = Zone::GetBigBen("COORDBIGBEN");
  dirty() ;
}

CSInstance::~CSInstance() {}

// Process Message Methods
// ==========================================================================
// ==========================================================================
int
CSMIState::ProcessMessage(Link<FSMInst_State> theState)  {
    Link<CSInst_State> myState= L_AS(CSInst_State, theState);
    o_u4b msgID = myState->directEvent;
    Link<Message> myMsg = myState->lastMessage;
    o_bool retval=FALSE;
    switch (msgID) {
    case MERGE_I_FAILED:
      {
if (myMsg->rspTo == myState->curChild->GetProxy())
  retval=TRUE;
break;
      }
    case MERGE_I_DONE:
      {
if (myMsg->rspTo == myState->curChild->GetProxy())
  retval=TRUE;
break;
      }
    case MERGE_REQUEST:
      {
myState->otherCar = myMsg->rspTo;
retval=TRUE;
break;
      }
    }  // End of the Case
    return retval;
  }
// ==========================================================================
int
CSMRState::ProcessMessage(Link<FSMInst_State> theState)  {
    Link<CSInst_State> myState= L_AS(CSInst_State, theState);
    o_u4b msgID = myState->directEvent;
    Link<Message> myMsg = myState->lastMessage;
    o_bool retval=FALSE;
    switch (msgID) {
    case MERGE_R_FAILED:
      {
if (myMsg->rspTo == myState->curChild->GetProxy())
  retval=TRUE;
break;
      }
    case MERGE_R_DONE:
      {
if (myMsg->rspTo == myState->curChild->GetProxy())
  retval=TRUE;
break;
```

```
      }
    case MERGE_REQUEST:
      {
myState->otherCar = myMsg->rspTo;
retval=TRUE;
break;
      }
    }  // End of the Case
    return retval;
}
// ===========================================================================
int
CSAmILMRState::ProcessMessage(Link<FSMInst_State> theState)  {
    Link<CSInst_State> myState= L_AS(CSInst_State, theState);
    o_u4b msgID = myState->directEvent;
    Link<Message> myMsg = myState->lastMessage;
    o_bool retval=FALSE;
    return retval;
}
// ===========================================================================
int
CSSetPLState::ProcessMessage(Link<FSMInst_State> theState)  {
    Link<CSInst_State> myState= L_AS(CSInst_State, theState);
    o_u4b msgID = myState->directEvent;
    Link<Message> myMsg = myState->lastMessage;
    o_bool retval=FALSE;
    return retval;
}
// ===========================================================================
int
CSIdleState::ProcessMessage(Link<FSMInst_State> theState)  {
    Link<CSInst_State> myState= L_AS(CSInst_State, theState);

    o_u4b msgID = myState->directEvent;
    Link<Message> myMsg = myState->lastMessage;
    o_bool retval=FALSE;
    switch (msgID) {
    case SET_PL:
      {
myState->PL = myMsg->rspTo;
retval=TRUE;
break;
      }
    case SPLITF_REQUEST:
      {
if (myMsg->rspTo == myState->PL)
  retval=TRUE;
break;
      }
    case MERGE_REQUEST:
      {
myState->aCar = myMsg->rspTo;
retval=TRUE;
break;
      }
    }  // End of the Case
    return retval;
}

// Post Message Methods
// ===========================================================================
// ===========================================================================
int
CSMIState::PostMessage(Link<FSMInst_State> theState) {
  Link<CSInst_State> myState= L_AS(CSInst_State, theState);
  o_u4b msgID = myState->directEvent;
  Link<Message> myMsg;
  o_bool retval=FALSE;

  switch (msgID)    {
```

```
  case MERGE_I_FAILED:
    {
      myState->mergeTimer = myState->theInst->RegisterTimer(10, MERGETIMEROUT);
      break;
    }
  case MERGE_REQUEST:
    {
      myMsg = new Persistent Message(MSG_TYPE, 0, BUSY);
      myMsg->SetMsgTo(myState->otherCar);
      myMsg->SetRspTo(myState->theInst->GetProxy());
      myMsg->SetName("BUSY");
      myState->theInst->GetOutVT()->SendPacket(myMsg);
      retval=TRUE;
      break;
    }
  }
  return retval;
}
// ==========================================================================
int
CSMRState::PostMessage(Link<FSMInst_State> theState) {
  Link<CSInst_State> myState= L_AS(CSInst_State, theState);
  o_u4b msgID = myState->directEvent;
  Link<Message> myMsg;
  o_bool retval=FALSE;

  switch (msgID)    {
  case MERGETIMEROUT:
    {
      myState->mergeTimer = myState->theInst->RegisterTimer(20, MERGETIMEROUT);
      break;
    }
  case MERGE_REQUEST:
    {
      myMsg = new Persistent Message(MSG_TYPE, 0, BUSY);
      myMsg->SetMsgTo(myState->otherCar);
      myMsg->SetRspTo(myState->theInst->GetProxy());
      myMsg->SetName("BUSY");
      myState->theInst->GetOutVT()->SendPacket(myMsg);
      retval=TRUE;
      break;
    }
  }
  return retval;
}
// ==========================================================================
int
CSAmILMRState::PostMessage(Link<FSMInst_State> theState) {
  Link<CSInst_State> myState= L_AS(CSInst_State, theState);
  o_u4b msgID = myState->directEvent;
  Link<Message> myMsg;
  o_bool retval=FALSE;

  switch (msgID)    {
  case NO:
    {
      myMsg = myState->lastMessage;
      myMsg->SetMsgTo(myState->PL);
      myState->theInst->GetOutVT()->SendPacket(myMsg);
      myState->lastMessage = NULL_LINK;
      retval=TRUE;
      break;
    }
  case YES:
    {
      myMsg = myState->lastMessage;
      myMsg->SetMsgTo(myState->curChild->GetProxy());
      myMsg->SetPacketType(Packet::MakePacketType(MSG_TYPE, 0,MERGE_REQUEST_F));
      myState->theInst->GetInVT()->
SendPacket(myMsg, myState->theInst->GetLayerName(), CHILD);
```

```
      myState->lastMessage = NULL_LINK;
      retval=TRUE;
      break;
    }
  }
  return retval;
}
// ==========================================================================
int
CSSetPLState::PostMessage(Link<FSMInst_State> theState) {
    Link<CSInst_State> myState= L_AS(CSInst_State, theState);
    o_u4b msgID = myState->directEvent;
    Link<Message> myMsg;
    o_bool retval=FALSE;
    return retval;
}
// ==========================================================================
int
CSIdleState::PostMessage(Link<FSMInst_State> theState) {
  Link<CSInst_State> myState= L_AS(CSInst_State, theState);
  o_u4b msgID = myState->directEvent;
  Link<Message> myMsg;
  o_bool retval=FALSE;

  switch (msgID)    {
  case MERGETIMEROUT:
    {
      myMsg = new Persistent Message(MSG_TYPE, 0, START);
      myMsg->SetMsgTo(myState->curChild->GetProxy());
      myMsg->SetRspTo(myState->theInst->GetProxy());
      myMsg->SetName("START");
      myState->theInst->GetInVT()->
SendPacket(myMsg, myState->curChild->GetLayerName(), CHILD );
      retval=TRUE;
      break;
    }
  }
  return retval;
}

// Process Timeout Methods
// ==========================================================================
// ==========================================================================
// ==========================================================================
int
CSMIState::ProcessTimeout(o_u4b handle, Link<FSMInst_State> theState) {
  Link<CSInst_State> myState = L_AS(CSInst_State, theState);
  o_u4b msgID = myState->directEvent;
  o_bool retval=FALSE;
  handle = 0;
  return retval;
}
// ==========================================================================
int
CSMRState::ProcessTimeout(o_u4b handle, Link<FSMInst_State> theState) {
  Link<CSInst_State> myState = L_AS(CSInst_State, theState);
  o_u4b msgID = myState->directEvent;
  o_bool retval=FALSE;
  switch (msgID) {
  case MERGETIMEROUT:
    {
      if (myState->mergeTimer== handle) {retval=TRUE; }
      break;
    } // End of the case
  }
  return retval;
}
// ==========================================================================
int
CSAmILMRState::ProcessTimeout(o_u4b handle, Link<FSMInst_State> theState) {
  Link<CSInst_State> myState = L_AS(CSInst_State, theState);
  o_u4b msgID = myState->directEvent;
  o_bool retval=FALSE;
  handle = 0;
  return retval;
}
// ==========================================================================
int
CSSetPLState::ProcessTimeout(o_u4b handle, Link<FSMInst_State> theState) {
  Link<CSInst_State> myState = L_AS(CSInst_State, theState);
  o_u4b msgID = myState->directEvent;
  o_bool retval=FALSE;
  handle = 0;
  return retval;
}
// ==========================================================================
int
CSIdleState::ProcessTimeout(o_u4b handle, Link<FSMInst_State> theState) {
  Link<CSInst_State> myState = L_AS(CSInst_State, theState);
  o_u4b msgID = myState->directEvent;
  o_bool retval=FALSE;
  switch (msgID) {
  case MERGETIMEROUT:
    {
      if (myState->mergeTimer== handle) {retval=TRUE; }
      break;
    } // End of the case
  }
  return retval;
}
```

9.4.4 The Generated File CSEntEx.C

This file contains the user provided modifications.

```
#include <iostream.h>
#include <CS.h>
#include <Message.h>
#include <def.h>
#include <InVehTransmitter.h>
#include <Vehicle.h>
#include <MRMessage.h>
#include <LinkMessage.h>
#include <Message.h>
#include <MRMessage.h>
#include <CoordSupIn.h>
#include <CoordSupOut.h>

int
CSIdleState::Enter(Link<FSMInst_State> theState,
   Link <InputProjection> i,
   Link <OutputProjection> o)
{
   Link <CSInst_State> myState = L_AS(CSInst_State, theState);
   Link <CoordSupIn> myIn = L_AS(CoordSupIn, i);
   Link <CoordSupOut> myOut = L_AS(CoordSupOut, o);
   return FALSE;
}

int
CSIdleState::Exit(Link<FSMInst_State> theState,
  Link <InputProjection> i,
  Link <OutputProjection> o)
{
  Link<CSInst_State> myState = L_AS(CSInst_State, theState);
  Link <CoordSupIn> myIn = L_AS(CoordSupIn, i);
  Link <CoordSupOut> myOut = L_AS(CoordSupOut, o);
  return FALSE;
}

int
CSSetPLState::Enter(Link<FSMInst_State> theState,
    Link <InputProjection> i,
    Link <OutputProjection> o)
{
   Link <CSInst_State> myState = L_AS(CSInst_State, theState);
   Link <CoordSupIn> myIn = L_AS(CoordSupIn, i);
   Link <CoordSupOut> myOut = L_AS(CoordSupOut, o);
   // ===========================================
   myOut->myLeadersProxy = myState->PL;
   myOut->Update();
   myState->directEvent = YES;
   return TRUE;
   // ===========================================
}

int
CSSetPLState::Exit(Link<FSMInst_State> theState,
   Link <InputProjection> i,
   Link <OutputProjection> o)
{
   Link<CSInst_State> myState = L_AS(CSInst_State, theState);
   Link <CoordSupIn> myIn = L_AS(CoordSupIn, i);
   Link <CoordSupOut> myOut = L_AS(CoordSupOut, o);
   return FALSE;
}

int
CSMRState::Enter(Link<FSMInst_State> theState,
 Link <InputProjection> i,
 Link <OutputProjection> o)
{
   Link <CSInst_State> myState = L_AS(CSInst_State, theState);
   Link <CoordSupIn> myIn = L_AS(CoordSupIn, i);
   Link <CoordSupOut> myOut = L_AS(CoordSupOut, o);
   return FALSE;
}

int
CSMRState::Exit(Link<FSMInst_State> theState,
Link <InputProjection> i,
Link <OutputProjection> o)
{
   Link<CSInst_State> myState = L_AS(CSInst_State, theState);
   Link <CoordSupIn> myIn = L_AS(CoordSupIn, i);
   Link <CoordSupOut> myOut = L_AS(CoordSupOut, o);
   return FALSE;
 }

int
CSMIState::Enter(Link<FSMInst_State> theState,
 Link <InputProjection> i,
 Link <OutputProjection> o)
{
   Link <CSInst_State> myState = L_AS(CSInst_State, theState);
   Link <CoordSupIn> myIn = L_AS(CoordSupIn, i);
   Link <CoordSupOut> myOut = L_AS(CoordSupOut, o);
   return FALSE;
}

int
CSMIState::Exit(Link<FSMInst_State> theState,
Link <InputProjection> i,
Link <OutputProjection> o)
{
   Link<CSInst_State> myState = L_AS(CSInst_State, theState);
   Link <CoordSupIn> myIn = L_AS(CoordSupIn, i);
   Link <CoordSupOut> myOut = L_AS(CoordSupOut, o);
   // ==========================================
   myState->UpdateCorr();
   // ==========================================
   return FALSE;
}

int
CSAmILMRState::Enter(Link<FSMInst_State> theState,
     Link <InputProjection> i,
     Link <OutputProjection> o)
{
   Link <CSInst_State> myState = L_AS(CSInst_State, theState);
   Link <CoordSupIn> myIn = L_AS(CoordSupIn, i);
   Link <CoordSupOut> myOut = L_AS(CoordSupOut, o);
   // ==========================================
   myIn->Update();
   if (myIn->amILeader == TRUE){
     myState->directEvent = YES;
   }
   else {
     myState->directEvent = NO;
   }
   return TRUE;
   // ==========================================
}

int
CSAmILMRState::Exit(Link<FSMInst_State> theState,
    Link <InputProjection> i,
    Link <OutputProjection> o)
{
   Link<CSInst_State> myState = L_AS(CSInst_State, theState);
   Link <CoordSupIn> myIn = L_AS(CoordSupIn, i);
   Link <CoordSupOut> myOut = L_AS(CoordSupOut, o);
   return FALSE;
}

CSInst_State::UpdateCorr() {
   Link<InputProjection> In  = theInst->inputs;
   Link<CoordSupIn> myIn = L_AS(CoordSupIn, In);
   // ==========================================
   myIn->Update();
   PL = myIn->myLeadersProxy;
   return TRUE;
   // ==========================================
}

CSInst_State::~CSInst_State() {}
CSInst_State::CSInst_State() {
    aCar = NULL_LINK;
    otherCar = NULL_LINK;
    PL = NULL_LINK;
    mergeTimer = 0;
    lcTimer = 0;

}
```

9.5 Highway Creation Grammar

Note that the following file does not use the C++ information field of a class specification. This filed is set to "Wire".

```
{
   Zone
}
{
   {Zone Z bitmaps/highway/zone22.bm
      {{ZoneIn Wire L .25 1 1} {ZoneIn Wire L .75 1 1}}
      {{ZoneOut Wire R .25 1 1 {ZoneIn}} {ZoneOut Wire R .75 1 1 {ZoneIn}}}
      {{}}
      {{Junct2To1 I} {Junct1To2 I} {Segment I} {Sink I} {Source I}}
   }
   {Junct2To1 T bitmaps/highway/injunct.bm
      {{JunctIn Wire L .25 1 1} {JunctIn Wire L .75 1 1}}
      {{JunctOut Wire R .50 1 1 {SegmentIn SectionIn}}}
      {{length 200}}
      {{Lane I}}
   }
   {Junct1To2  O bitmaps/highway/outjunct.bm
      {{JunctIn Wire L .50 1 1}}
      {{JunctOut Wire R .25 1 1 {SegmentIn SectionIn}}
         {JunctOut Wire R .75 1 1 {SegmentIn SectionIn}}}
      {{length 200}}
      {{Lane I}}
   }
   {Lane L  bitmaps/highway/lane.bm
      {{LaneIn Wire L .50 1 1}}
      {{LaneOut Wire R .50 1 1 {LaneIn}}}
      {{width 4}}
      {}
   }
   {Segment H bitmaps/highway/segment.bm
      {{SegmentIn Wire L .50 1 1}}
      {{SegmentOut Wire R .50 1 1 {SegmentIn JunctIn SinkIn}}}
      {{}}
      {{Section I} {EntrySection I} {ExitSection} {Entry I} {Exit I}}
   }
   {Section S  bitmaps/highway/section.bm
      {{SectionIn Wire L .5 1 1}}
      {{SectionOut Wire R .5 1 1 {SectionIn JunctIn SinkIn}}}
      {{length 200}}
      {{Lane I}}
   }
   {EntrySection ES  bitmaps/highway/ensection.bm
      {{EntryIn Wire B 0.15 1 1} {SectionIn Wire L .5 1 1}}
      {{SectionOut Wire R .5 1 1 {SectionIn JunctIn SinkIn}}}
      {{length 200}}
      {{Lane I}}
   }
   {ExitSection XS  bitmaps/highway/exsection.bm
      {{SectionIn Wire L .5 1 1}}
      {{SectionOut Wire R .5 1 1 {SectionIn JunctIn SinkIn}}
         {ExitOut Wire B .85  1 1 {ExitIn}}}
      {{length 200}}
      {{Lane I}}
   }
   {Entry GE  bitmaps/highway/source.bm
      {}
      {{EntryOut Wire R .5 1 1 {EntryIn}}}
      {{length 200}}
      {{Lane I}}
   }
   {Exit AE  bitmaps/highway/sink.bm
      {{ExitIn Wire L .5 1 1}}
      {}
      {{length 200}}
      {{Lane I}}
   }
   {Source GS  bitmaps/highway/source.bm
      {}
      {{SourceOut Wire R .5 1 1 {SegmentIn SectionIn}}}
      {{length 200}}
      {{Lane I}}
   }
   {Sink AS  bitmaps/highway/sink.bm
      {{SinkIn Wire L .5 1 1}}
      {}
      {{length 200}}
      {{Lane I}}
   }
}
# Relation list
{
   {ZoneOut-ZoneIn
      {{JunctOut-SegmentIn 3} {SegmentOut-SegmentIn 3}}
   }
   {SegmentOut-SegmentIn
      {{SectionOut-SectionIn 3}}
   }
   {JunctOut-SegmentIn
      {{JunctOut-SectionIn 1}}
   }
   {SegmentOut-JunctIn
      {{SectionOut-JunctIn 2}}
   }
   {SegmentOut-SinkIn
      {{SectionOut-SinkIn 2}}
   }
   {SourceOut-SegmentIn
      {{SourceOut-SectionIn 1}}
   }
   {SourceOut-SectionIn
      {{LaneOut-LaneIn 3}}
   }
   {JunctOut-SectionIn
      {{LaneOut-LaneIn 3}}
   }
   {SectionOut-JunctIn
      {{LaneOut-LaneIn 3}}
   }
   {SectionOut-SinkIn
      {{LaneOut-LaneIn 3}}
   }
   {EntryOut-EntryIn
      {{LaneOut-LaneIn 3}}
   }
   {ExitOut-ExitIn
      {{LaneOut-LaneIn 3}}
   }
   {SectionOut-SectionIn
      {{LaneOut-LaneIn 3}}
   }
   {LaneOut-LaneIn
      {}
   }
}
# Bus List
{
    {MyBus {LaneOut-LaneIn LaneOut-LaneIn}}
}
```

Index