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

An Object-oriented Database for IVHS

Pravin Varaiya
University of California, Berkeley

California PATH Working Paper
UCB-ITS-PWP-98-2

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 Transportation, Federal Highway Administration.

Report for MOU 217

January 1998

ISSN 1055-1417

An object-oriented database for IVHS
Final Report MOUs 169, 217

Pravin Varaiya
Department of Electrical Engineering and Computer Science
University of California, Berkeley CA 94720
Tel: (510) 642-5270 email: **varaiya@eecs.berkeley.edu**

Contents

Abstract

The objective of this proposal was to develop a database that integrates various elements—traffic simulation packages, data sets, and computational tools. The difficulty in integration was felt to be the incompatibility of data structures, formats, and software.

It was proposed to design, build and test a software environment with an open architecture that would facilitate a synergistic use of these software elements. The core of this environment was to be an object-oriented commercial database system that would be sufficiently general to support many software elements, and which would guide new software development.

The deliverables of this project were:

- A specification of an object hierarchy implemented in C++, and a graphical user interface to facilitate object specification;
- An interface to SmartPath so that simulation configurations and simulation runs can be stored in the database;
- An interface to the 1-880 database, developed in the Freeway Service Patrol project;
- Additional facilities to be determined after consultation with other PATH researchers.

Work done under this project resulted in the system called SmartDB, implemented on top of the commercial object-oriented database management system, Versant. SmartDB includes a graphical user interface. SmartDB is documented in the PhD thesis included as Appendix, and in several papers.

A basic version of SmartPath is implemented in SmartDB. However, implementation of the complete version was abandoned, since SmartPath itself was being reimplemented as SmartAHS, built on top of the hybrid system simulation language SHIFT.

The 1-880 data set meanwhile was stored as flat files, with many additional data processing features, and made accessible through the World Wide Web. Since that facility was being widely used, and did not require users to install Versant, it was futile to provide an interface to it from SmartDB.

Testing with SmartDB revealed major limitations in Versant, which the manufacturer was unable to overcome. These limitations severely weakened the performance of SmartDB, and further development of SmartDB was stopped. Instead, the experience gained with SmartDB was used in the design of the highly successful simulation language SHIFT.

1 Executive Summary

The period of performance of MOUs 169, 217 is November 1994—December 1997. This project initially was assigned the number MOU 169; its continuation was assigned the number MOU 217. The originally approved funding was subsequently significantly reduced so the project cost was lower than planned. In addition, part of the research was supported by a grant from the U.S. Army Research Office.

Problem statement Empirical research in transportion is hampered by the inability to integrate traffic simulation packages, empirical data sets, and computational tools. It is difficult, for example, to link or effectively compare different simulation packages, to combine say loop detector data with geographical information systems, or to use standard statistical data analysis tools on loop detector data sets. This difficulty in part is due to an incompatibility of data structures, formats, software. At the bottom of this incompatibility is the lack of a common semantic model.

The objective of this project was to overcome this difficulty. We proposed to design, build and test a software environment with an open architecture that would facilitate a synergistic use of these software elements. The core of this environment was to be an object-oriented commercial database system that would be sufficiently general to support many software elements, and which would guide new software development.

The deliverables of this project were:

1. A specification of an object hierarchy implemented in C++, and a graphical user interface to facilitate object specification;
2. An interface to SmartPath so that simulation configurations and simulation runs can be stored in the database:
3. An interface to the 1-880 database, developed in the Freeway Service Patrol project.
4. Additional facilities to be determined after consultation with other PATH researchers.

Work done under this project resulted in the following products:

- Design and building of SmartDB, implemented on top of the commercial object-oriented database management system, Versant, and including a graphical user interface.

This addresses the first deliverable. SmartDB is fully documented in the thesis [1], "Object Management Systems," which is also available as Path Research Report UCB-ITS-PRR-95-19.

- Implementation of a basic version of SmartPath in SmartDB. This implementation is called SmartAHS.

This addresses the second deliverable. Appendix 1 provides a detailed discussion of this implementation [3]. The full implementation of SmartPath was abandoned, because SmartPath itself was being reimplemented as a library of the SHIFT simulation language, also called SmartAHS.

- Development of a new approach for developing large-scale object-oriented software systems, called Object Management Systems (OMS).

This indirectly addresses the third deliverable. **A** detailed discussion is provided in Appendix 2 [2]. During the course of this project, the 1-880 database was implemented as flat files, with many additional data processing features, and made accessible through the World Wide Web. Since that facility was being widely used, and did not require users to install Versant, it was futile to provide an interface to it from SmartDB.

- Design guidelines for SHIFT

Testing with SmartDB revealed major limitations in Versant, which the manufacturer was unable to overcome. These limitations weakened the performance of SmartDB. Discussions with other PATH researchers in terms of their simulation and data needs revealed a weakness in the design of the SmartDB object hierarchy: the objects were too closely tied to SmartPath and other vehicle-oriented simulations and data sets. In other words, the object hierarchy in SmartDB was not sufficiently abstract. These limitations motivated the development of (1) Object Management Systems — these are model-based applications used to simulate, evaluate, and control large-scale physical environments such as highway networks; and (2) SHIFT, an object-oriented simulation language for dynamically varying networks of hybrid systems.

2 Discussion

This project resulted from a response to the PATH RFP calling for development of a "system for acquiring, storing, recovering, communicating with, and processing IVHS-related databases in California," that would accommodate databases of different structure and type, that would be easy to use, and would ensure security. As we pointed out in our proposal the RFP posed a very stiff requirement, and it was difficult to imagine how such a system could be developed even with PATH's *entire* budget. (For purposes of comparison, the first phase of the Sequoia project, which had a similar goal for weather-related databases in California, cost $10 million.) Of course, "accommodation" of different databases could mean something very modest: a shell around different databases that make them appear "similar," but this is not much help.

The goal of the proposal which led to MOUs 169 and 217, was more modest. It was to demostrate that some of the objectives of the RFP could be met by a well-designed object-oriented environment that can integrate various traffic related data sets, highway configurations, simulation models, and computational tools.

These goals have been realized through the development of an approach, called Object Mangement Systems, an implementation of that approach, called SmartDB, and its elaboration to the IVHS context called SmartAHS.

SmartDB has two main elements. The first is a an object hierarchy model for highways, vehicles, sensors and communicating devices. The second is an implementation of that hierarchy in the Versant database management system. Another element is a graphical user interface implemented in TKL-TK.

In this section we provide a critical assessment of this work. Detailed description of the work is provided in Appendix 1, 2 and in [1].

2.1 Object management systems (OMS)

In our formulation, OMS are object-oriented software systems for simulating, evaluating and controlling large-scale physical environments like transportation networks and power distribution systems. The usefulness of OMS depends on three factors:

- Data and process models used to describe the physical environment;
- Software tools used to implement these models; and
- Software engineering process followed to realize OMS.

Data models are the data structures used to represent the instantaneous description or the 'state' of the physical environment or system. Process models represent changes in the state or the system 'behavior' over time.

Software tools associated with data and process models range from a programminglanguage, database engine, modeling tools, and application development utilities.

Finally, the software engineering process determines the life cycle of an OMS: the generation, modification, documentation, and maintenance of the system. We will not discuss this factor any further.

The most popular data model is provided by relational databases. Because it has been followed for a long time, approaches based on the relational model can make use of a mature set of tools. The relational model has well-defined mathematical semantics which give the foundation for database operations (join, projection, etc.) and storage algorithms. The simplicity of the relational model permits the implementation of relational databases as flat, fixed-format tables and a powerful Structured Query Language (SQL) for database operations.

The relational model does not support complex process models such as finite state machines or differential equations that may be the natural way to describe the evolution of the physical environment. The process model in such cases will be described by some standard programming language.

Object-oriented approaches are prompted by this disjunction between data and process models forced by relational models. These approaches permit a much richer set of abstract data structures whose semantics simultaneously support data and process models. However, the generality of the model itself has precluded the development of a standardized set of widely applicable tools, and the databases themselves have failed to converge to a standard query language like SQL. Object-oriented databases are of two types: those tied closely to the relational model (e.g., Postgres) and those tied to programming languages such as C++. The former provide enhanced relational databases with an object interface, while the latter provide programming languages with persistent objects.

The OMS approach is motivated by applications involving management systems used to control the behavior of heterogeneous, dynamic and distributed physical environments (remember that systems of automated vehicles and highways is one important application). The OMS Object Model is the chief theoretical construct underlying OMS, and SmartDB is an implementation of this model.

The OMS object is based on object-oriented modeling and dynamical systems theory. The principal features of the OMS object are: state, methods, transitions and constraints.

An object's attributes are its state, inputs and outputs. These terms are have a meaning similar to that in dynamical system theory. The system as a whole is a collection of objects, and the state of the system (in the strict sense of dynamical systems theory) is the state of its objects and the state of their input-output interconnections.

The methods of an object are memoryless (reentrant) functions from its state and input to a new state and output. Thus methods can encode or describe the system dynamics: finite state machines or differential equations, deterministic or stochastic. A method may create or delete objects, and it can change input-output connections. Finally, a method specifies its triggering inputs and outputs, i.e., the conditions under which a method is executed.

State transitions are activated by triggering a subset of inputs. **All** object methods triggered by these inputs are executed. (Remember that the triggering conditions are specified within each object.) The resulting outputs which may become inputs because of input-output interconnections, may trigger other transitions. The executions sequence of transitions may not be unique and it may not terminate. If an execution sequence satisfies system constraints, then the system at the end of the sequence is committed, otherwise the triggering input is discarded.

Constraints may be placed on: values of the state, inputs and outputs of individual objects; establishment of connections between objects; relationships between objects; and pre- and post-execution state constraints of method execution.

The OMS object provides a very powerful construct that can be used to model all the physical systems in which we are interested.

2.2 SmartDB

SmartDB is a software implementation of the OMS Object model. It provides several additional features:

- It implements a relationship object and provides specific relationships such as input-output, containment, views, agent-manager, client-server, and process layers. A process is a collection of objects that are executed at a common time- or event-granularity. (For instance, time-driven objects, used to model difference or differential equations, may be collected in one process.)
- It implements commonly used objects such as events, sensors, actuators] schedulers.
- It implements the OMS engine which executes the model dynamics. The engine provides an interface for creating and deleting objects, executing transitions, etc.
- It provides system architecture tools for data distribution, process distribution, object migration, packaging objects into process layers, etc.

The SmartDB implementation uses the C++ programming language, Versant Object Database, TCL/TK user interface toolkit, and the UNIX operating system.

2.3 SmartAHS

SmartAHS is an extension of SmartDB, with a family of objects and processes that are used to describe automated vehicles and highway systems. SmartAHS is described in detail in Appendix 1. SmartAHS allows the configuration of different highway, traffic, and control schemes. All possible highway and traffic configurations can be represented using a set of building blocks. Vehicles are specified in a similar manner as a collection of building blocks such as engines, brakes, throttle and steering mechanisms.

The SmartAHS semantic model is inherited from the SmartDB semantics. Each SmartAHS object is described using the following characteristics: static properties such as length; dynamic properties such as speed; inputs and outputs; state evolution behavior; monitors to record behavior; sensors; communication devices.

SmartAHS has an open architecture which simplifies integration with other simulation environments. The simulation objects such as vehicles] highways, sensors, etc. are placed in the object-oriented database Versant. The database provides an open interface for integration with other simulation packages. The physical system model, the control layers, and the monitor agents are organized in different process layers based on their frequency of access to this database. The process coordinator schedules the different layers for execution.

2.4 Conclusion

The products of this MOU in some ways went beyond what we had proposed and in some ways fell short of our objectives.

The implementation of SmartDB is a strong validation of our object-oriented approach. The rapid implementation of SmartAHS as an extension of SmartDB proves that SmartDB provides a very powerful modeling framework. The graphical user interface helps in the design of applications and in running simulations.

Two shortcomings have become evident. First, the implementation of SmartDB objects as C++ objects permits the full flexibility of C++. This flexibility is, however, also a shortcoming in that it enforces no discipline on the applications designer. For expert programmers this is not a problem. But in the hands of non-experts, there will be a lack of structure which would prevent re-usability of objects. To some extent this is evident in SmartAHS. This lesson was absorbed in the design of SHIFT.

Unlike SmartDB, SHIFT is a programming language, with a strict syntax, and a clear semantics based on the mathematical framework of hybrid systems. As reimplemented on top of SHIFT, SmartAHS comprises a set of types (SHIFT notion of objects) such as highways, vehicles, etc. Because it is a full-fledged language, the compiler provides various error checks that assist the programmer. The success of SHIFT derives in no small part from the success and limitations of SmartDB.

The second and more limiting weakness comes from Versant, the object-oriented commercialdatabase system used by SmartDB for persistent data storage. We had selected Versant after a careful search of all commercial systems. (Versant was also the choice of JHK in its recommendation for the Caltrans District 4 TMC.) However, Versant proved unable to support the large scale updating of objects that a simulation requires. **A** promised distributed version of Versant did not materialize] making it impossible to migrate objects and processes.

Ultimately] our hope of developing a system that would combine both storage of data (eg., from loop detectors) and simulation runs remains unmet.

References

[1] A. Gollu. Object management systems. PhD thesis, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, CA., 1995. Also, PATH Research Report UCB-ITS-PRR-95-19.

[2] A. Gollu and A. Deshpande. Object Management Systems: A summary. In Proceedings *IFAC World* Congress, San Francisco, CA, 1996.

[3] A. Gollu and P. Varaiya. SmartAHS: A simulation framework for Automated Vehicles and Highway Systems. Mathematical and Computer Modelling, 1997. To Appear.

Appendix 1

SmartAHS: A simulation framework for Automated vehicles and Highway systems

SmartAHS: A Simulation Framework for Automated Vehicles and Highway Systems *

Aleks Gollu and Pravin Varaiya
UC-Berkeley
{gollu,varaiya}Qeclair.eecs.berkeley.edu
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley
Corresponding Author: Aleks Gollu
2271 Cedar St.
Berkeley CA 94709
gollu@eecs.berkeley.edu
510 548 1315 (work); 510 231 9466 (work); 510 848 2301 (fax)

Abstract

SmartAHS is a software framework for the uniform specification, simulation, and objective evaluation of Intelligent Vehicle Highway System (IVHS) alternatives. This paper illustrates the use of the object-oriented paradigm in its design, implementation and use. Objective comparison of proposed highway automation alternatives is achieved, when all control architectures are specified in SmartAHS.

The framework implementation is decomposed into three layers. A core set of entities, called SmartDb, implement the base classes of the framework, the scheduling mechanism, and a special syntax for a state machine formalism. The second layer, SmartAHS customizes these classes and implements entities specific to highway automation. The control and communication engineers are the users of SmartAHS; they further customize it to implement specific simulation applications such as SmartPATH. The users of the applications are system analysts and system planners.

We discuss the design and implementation of SmartDb and SmartAHS and give a use-case example of a subset of SmartPATH.

keywords: simulation] evaluation, highway automation, object-oriented, framework.

1 Introduction

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 strategic plan [1] 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.

It is important to distinguish the actual control and communication design of an automation strategy from its simulation and evaluation. This paper describes SmartAHS, a software framework for the uniform specification, simulation, and objective evaluation of Intelligent Vehicle Highway System (IVHS) alternatives and illustrates how the object-oriented paradigm is used in its design, implementation and use. Objective comparison of proposed alternatives is achieved, when all control architectures are specified in this framework.

The concepts for SmartAHS have emerged from the SmartPath project at the California PATH Laboratory at the University of California] Berkeley [2]. SmartAHS implements SmartPath concepts in an object oriented, distributed] and open architecture.

"**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.**

The salient concepts in SmartAHS are: 1) layered control architecture, 2) combined discrete and continuous dynamical systems, known as hybrid systems, **3)** object oriented simulation of hybrid systems, and 4) distributed and open architecture of the simulation framework.

SmartAHS uses the object oriented approach **[3]** to construct a logical model of the physical components and their control agents. The objects in the logical model have semantic content corresponding to their characteristics, inter-relationships, constraints, and behaviors.

SmartAHS's object oriented approach allows the configuration of different highway, traffic, and control schemes. All possible highway and traffic configurations can be represented using a set of building blocks: zones, segments, junctions, sources, sinks, sections, entries, and exits. The rules for containment and connection of these building blocks are specified using a context-free grammar.

Traffic patterns can be generated internally in SmartAHS, or SmartAHS can interface with traffic simulation packages such as MitSim [4] and NetSim [5].

The vehicles are specified in a similar manner as a collection of building blocks—physical components such as engines, brakes, and steering, control components such as sensors and actuators, and communications components such as transmitters and receivers.

In the semantic model each object is described uniformly using the following characteristics: static properties such as length, width, and weight; dynamic properties such as vehicle speed and lane density; inputs and outputs; state evolution behavior, control behavior, monitor behavior for observing state evolution, e.g., a gas tank agent that monitors the amount of carbon-monoxide produced; sensors for providing information about the environment, e.g., distance to vehicle in front; and transmitters and receivers for communicating with neighboring objects.

This description can be summarized by noting that each object represents a dynamical system with state, state evolution, interface, inputs, and outputs. Such a summary representation is shown in Figure 1.

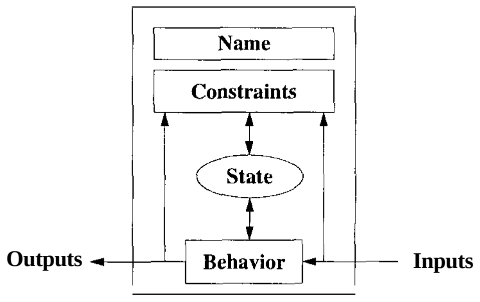

Figure 1: Simplified IVHS Object

SmartAHS has an open architecture which simplifies integration with other simulation environments. The simulation objects such as vehicles, highways, sensors, and others are placed in an object oriented database (OODB). The database provides an open interface for integration with other simulation packages. The physical system model, the control layers, and the monitor agents are organized in different "scheduling layers" based on their frequency of access to this database. The process coordinator schedules the different layers for execution.

Section 2 provides a background in the object-oriented paradigm. Section **3** summarizes the requirements of the framework. Section 4 discusses the design and implementation of the framework. Section 5 summarizes SmartAHS functional modules. Section 6 gives a use-case example. Section 7 summarizes performance results. Conclusions are provided in Section 8.

2 Background

The SmartAHS framework is developed as an Object Management System (OMS) [6, 7].

OMS focus on an important class of applications, namely management systems that are used to control the behavior of heterogeneous, dynamic, and distributed physical environments.

OMS start with basic software tools such as programming languages and databases and develop an abstract object model.

This model is then customized for specific application domains. Domain customization starts with the object model and uses it to specify the relevant components of the physical environment: objects, their interrelationships, constraints, behavior, observation and control channels, event propagation, control strategies, and user interfaces. System architecture determines the optimal partitioning of the deployed system with respect to distributed processing, distributed databasing, process and object migration strategies, concurrency control, and versioning.

Finally, application programmers fill in the details of object behaviors and control and coordination strategies keeping in mind the system constraints and implement end-user applications [1].

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

OMS 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 the object-oriented methodology and programming language characteristics that make our design viable.

2.1 Semantic Modeling

Wegner [8] 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.

2.2 Object-Oriented Paradigm

Extensive treatments of the OO paradigm can be found in several books, we discuss the main characteristics.

2.2.1 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.

[1]Naturally, these stages are followed by the system test, release, and maintenance stages.

2.2.2 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.*

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.

2.2.3 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.

2.3 Existing Formalisms

Many formalisms exist for specifying hybrid systems [9, 10]. These formalisms are not suitable for large scale simulation. Many other notational techniques exist [11, 12, 13]. These formalisms are better suited for systems with static object relationships. In our framework relationships such as "front vehicle" are dynamic, i.e they refer to different vehicles based on the configuration of vehicles on the highway.

Software frameworks [14] have gained popularity in the last decade since they greatly simplify implementation of applications for specific domains. Many simulation frameworks exist [15, 16].

3 Framework Requirements

In this Section we list the functional, modeling] and software system requirements that drove the design and implementation of SmartAHS.

3.1 Problem Summary

We first provide a summary of one proposed control architecture.

In the layered control architecture proposed by Varaiya and Shladover [17, 18], 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 2.

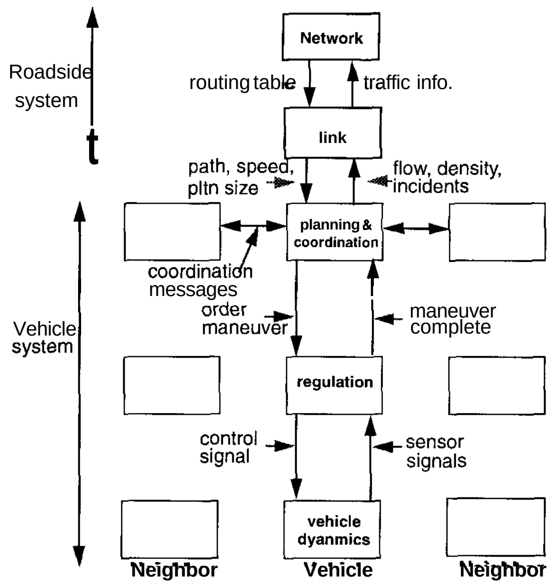

Figure 2: Layered Control Architecture

The physical layer represents the dynamics of vehicles. 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 must allows the specification of this and other layered control architectures.

3.2 Targeted Users

The simulation framework must address the needs of several categories of users. These are control and communication engineers who will design, implement, and test individual control and communication components; system analysts who will test and evaluate automation strategies; and system planners who will select the automation strategy for deployment based on evaluation results.

3.3 Functional Requirements

The framework has to provide constructs to represent the following entities *or* operations.

- Ability to represent arbitrary highways;
- Ability to represent incoming and outgoing traffic patterns;
- Ability to create vehicles consisting of many components;
- Ability to create roadside controllers consisting of many components;
- Ability to have different types of vehicles on the highway;
- Ability to represent inter-vehicle and vehicle-to-roadside communication;

- Ability to detect and to create accidents;
- Ability to collect arbitrary statistics;

3.4 Software Engineering Requirements

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 he added to the framework as work progresses. The incorporation of such new components should require minimal rewrite of any other existing software.

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

  As part of evaluation support, the framework should be able to *truce* the behavior of any subset of objects for future review and replay. The framework should provide an open interface to interact with other statistics and data processing 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. The simulation framework, however, does impose a lower bound on the simulation performance due to time taken to perform bookkeeping operations.

  - 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.

4 SmartAHS Design and Implementation

No existing simulation tools satisfy all the criteria summarized in Section **3.** After some deliberation SunSparc stations were selected 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 framework implementation.

The framework implementation is decomposed into three layers. **A** core set of entities, called SmartDb, implement the base classes of the framework, the scheduling mechanism, and a special syntax for a state machine formalism. The second layer, SmartAHS customizes these classes and implements entities specific to highway automation. The control and communication engineers are the users of SmartAHS; they further customize it to implement specific simulation applications such as SmartPATH. The users of the simulations are system analysts and system planners.

In this section we present the classes in SmartDb and SmartAHS.

4.1 SmartDb and SmartAHS Data Model

We first discuss the class and containment hierarchies of the entities. For each class we discuss the basic functionality and omit most details for simplicity. For a more detailed discussion of the classes the user is referred to [19].

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.

4.1.1 Base Classes

The inheritance hierarchy for base classes is described in Figure 3.

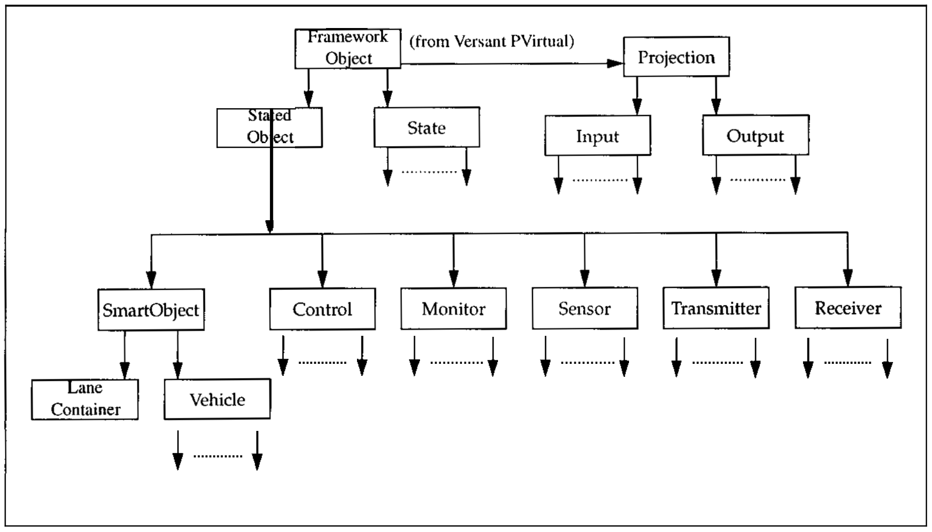

Figure 3: Base Class Inheritance Hierarchy

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.

FrameworkObject is the base class of the simulation framework. It defines some basic methods that apply to all framework classes. Most subclasses are expected to specialize these methods and implement specific behaviors.

Statedobject captures the relation between an object and its State. The static parts of an object are given by its specialization of Statedobject. The dynamic part, i.e. state information that evolves during the simulation is given in a separate class State.

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. This separation facilitates the recording of state history.

The use of the base classes in the time and event driven scheduling is illustrated in the next Section.

4.1.2 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.

The highway entities are organized in the inheritance hierarchy described in Figure 4. Note that Lanecontainer, Generator, Absorber, and Junction are introduced as abstract base classes.

The Smartobject class is discussed in Section 4.1.5.

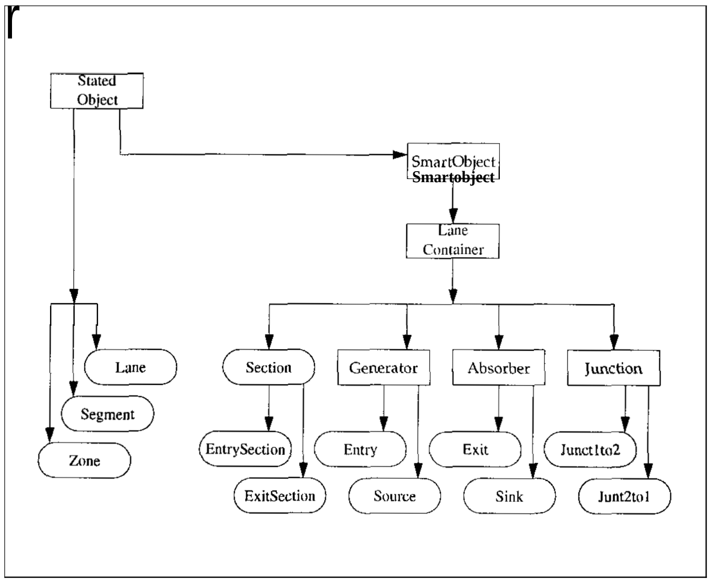

Figure 4: Class Hierarchy for Highway Entities

The containment hierarchy for these entities is given in Figure 5. Since the containment hierarchy is a one to many relationship, cardinality is indicated on the containee's side only.

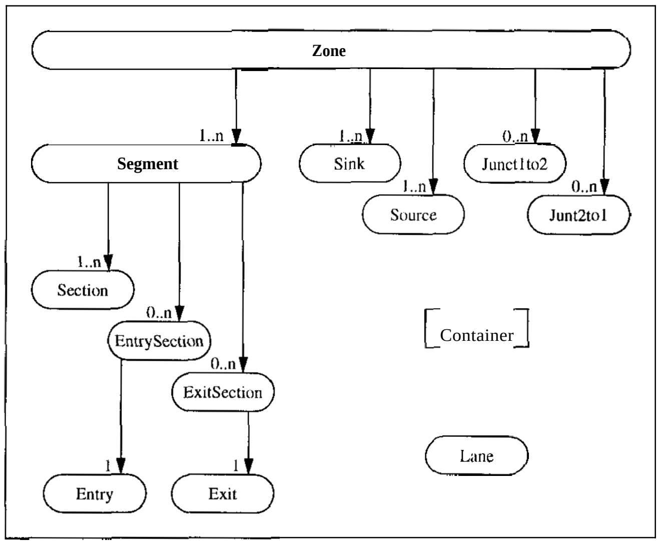

Figure 5: Containment Hierarchy for Highway Entities

Finally a number of binary relationships specify how instances of highway classes can be connected to create highway networks. We illustrate these relationships with examples and omit the details of the relationship rules. **A** Lanecontainer may be related one or zero `LaneContainers` 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. Similarly, a Lane may be related to one or zero Lanes through its prevLane relationship. Again the reciprocal of this particular relationship is nextLane. A Junct1to2 has two next relationships nextLC1 and nextLC2.

A highway network consists of multiple Zones; the Zones provide the basis for distributed simulation. However, for simplicity in this paper, 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.

A number of highway entity methods are used by scheduling objects to create and move the Vehicles along the highway. These methods are discussed in Section 4.2.

4.1.3 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.

Vehicle has a number of static state attributes such as length, width, origin and destination. Vehiclestate inherits from State. It contains basic continuous and discrete state information about Vehicles. 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 Vehicles access is limited to Sensors only.

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 Vehicles. 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 SmartAHSas discussed in Section 4.2.2. In particular the Lane method MoveVehiclesInLane() sets the absDist, currLane, and cellId attributes of the Vehicle.

4.1.4 Automation Devices

Sensors, Controllers, Receivers, Transmitters, and Monitors are added to vehicles and to the roadside for automation and evaluation. These five entities 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.

Longitudinal and lateral Sensors provide information about the environment such as distance and speed to the Vehicle in front, or to the left. Transmitters and Receivers are used for communicating with other objects. Example Control objects are regulators that determine speed, and coordinators that select maneuvers to perform. Monitors are read-only entities that collect statistics.

SmartAHS provides a number of default sensor, transmitter, and receiver implementations such as VehicleSensor, ReceiverProxy, InVehTransmitter and OutVehTransmitter. These classes model ideal devices.

4.1.5 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.

The automation devices are specialized based on the automation strategy and based on the type of Smartobject they are contained in. Application developers specify the particular relations among automation devices contained in a given Smartobject.

4.1.6 Traffic Entities

Entities used to create traffic and vehicles are called traffic entities. SmartAHS provides them only as abstract base classes.

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.

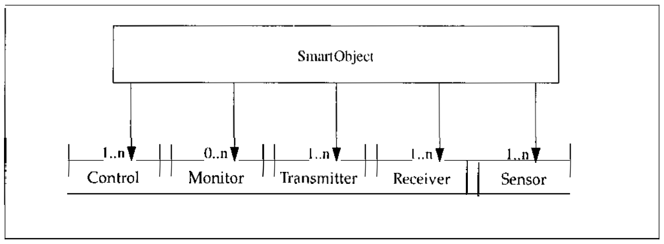

Figure 6: The **Smartobject** and its containees.

Vehicles 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.

InTraffics are responsible for generating incoming traffic patterns, **OutTraffics** are responsible for generating outgoing traffic patterns. These objtcts 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 **Vehicles.** The location of these entities in the inheritance and containment hierarchy is given in Figure 7.

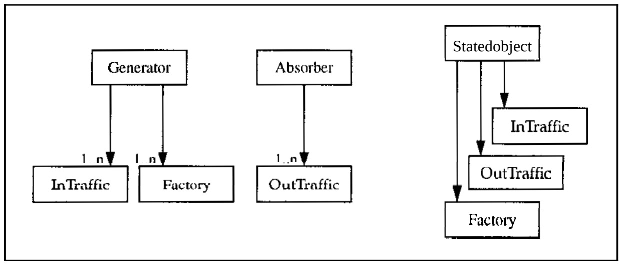

Figure 7: Inheritance and Containment Hierarchy for Traffic Entities

4.2 SmartDb Process Model

In this section we discuss the dynamic aspects of the simulation and the scheduling objects that govern evolution.

4.2.1 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 **Vehicles** 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. **Vehicles** leave the highway at **Absorber** objects based on outgoing traffic patterns.

4.2.2 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 **Vehicles** are dynamic. **Vehicle** position and the containment relationship between a **Vehicle** and its **Lane** are maintained by SmartAHS. The **Vehicle Engine** computes the movement of the **Vehicle** given its speed, acceleration, and jerk. The **Lane** object is responsible for updating the position of a **Vehicle** in the Lane. In particular a **cellArray** is used in each lane to create a partitioning with 5 meter long cells. This partitioning serves as a hash table when computing the ranges of sensor and communication devices.

All other inter-vehicle relations are derived using Sensors. Sensors are also used to identify the receiver ids of objects in the sensing range.

Packets are used for event driven communication. SmartAHS maintains the relationships of Packets, i.e., it implements Packet delivery.

4.2.3 Time Scale of Evolution

In SmartAHS, object state evolution is driven by passage of time or by occurrence of events. SmartAHS provides 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 and implements a method that updates its state by one time 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. Events and Messages 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 simulation is implemented in three tears.

The top tear contains a ProcessCoordinator which is in charge of managing the execution of the the process Layers and the global clock BigBen (objects in the middle tier). 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. The bottom tier consists of the highway Zone and all of its containees.

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.

The process layers simulates collections of objects that evolve at the same time step or that respond to the same collection of events. The process layers themselves can be time or event driven. 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. 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.

The inheritance hierarchy of scheduling entities is given in Figure 8. Their containment hierarchy is given in Figure 9.

Example

The process architecture that would implement the layered architecture proposed by Varaiya [17] is shown in Figure 10.

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.

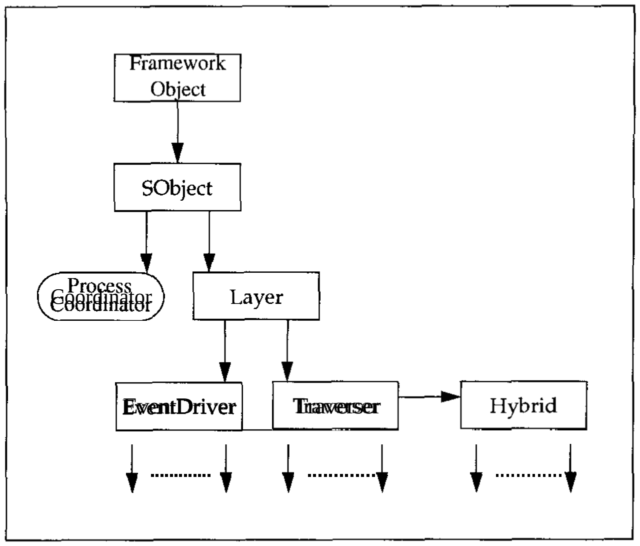

Figure 8: Class Hierarchy for Scheduling Entities

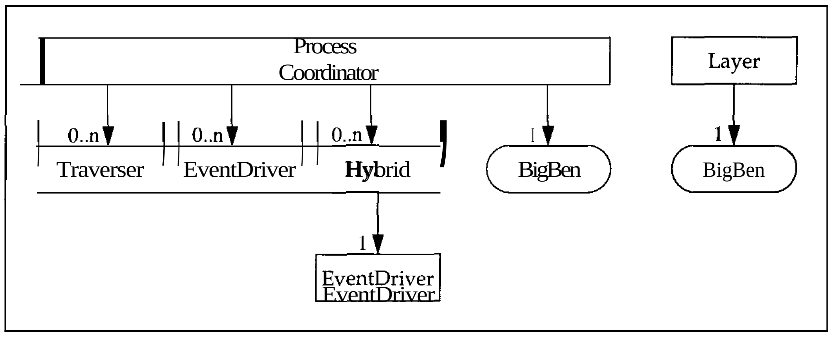

Figure 9: Containment Hierarchy for Scheduling Entities

4.2.4 Behavior Descriptions

The **FrameworkObject** class provides a number of virtual methods that are used to abstract object behavior.

The **WakeUp(Message* message)** method is used to deliver a timeout identified by **message** to an object.

The **ProcessEvent(int event)** method is used to deliver the **event** to this object.

The **ProcessMsg(Message* message)** method is used to deliver **message** to the object.

The **Run()** method is used to advance an object's state by one time step. Objects with time driven behavior must provide two copies of their state variables. At a given time step the "current" (output) values of all instances are used to compute the "next" (state) values. Once the "next" values are computed in all instances they are copied into "current" values.

FrameworkObject subclasses must specialize these methods to implement the appropriate time or event driven behavior.

SmartAHS also provides a special notation for specifying a dynamic network of hybrid automata. A discussion of this formalism can be found in [20]. Automata specified in this notation are translated into C++ code. The framework provides a library of classes for their run-time simulation.

4.2.5 Implementation of Scheduling Objects

The scheduling objects rely on inheritance and polymorphism since they invoke methods on base classes only.

In SmartAHS the **ProcessCoordinator** manages the execution of process **Layer** objects. Arbitrary number of process **Layers** register with the **ProcessCoordinator** and specify the period with which they

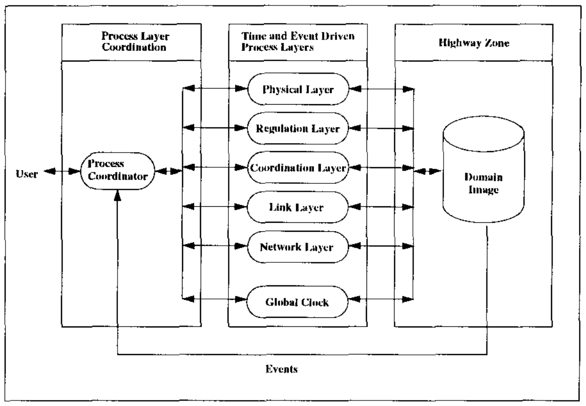

Figure 10: SmartPATH Process Architecture.

want to be scheduled. If a process Layer wants to be scheduled on an event driven basis, it sets its period to infinity'.

The ProcessCoordinator loops through its Layers and executes their `Go()` method according to their period. After every loop, it checks if there are any Events requesting the scheduling of event driven Layers.

SmartAHS provides three abstract classes Traverser, EventDriver, and Hybrid that subclass Layer.

All Layers contain a BigBen. They increment the BigBen when scheduled and ask it to deliver all outstanding timeouts.

In its specialized `Go()` method, the Traverser traverses the Zone from Sinks to Sources. The traverser defines a number of virtual methods that abstract operations on `LaneContainers`. For each Lanecontainer along the traversal, it invokes these virtual methods in a particular sequence. These methods are intended for specialization by the Traverser subclasses to invoke the time driven evolution methods (Run()) of the objects which the subclass is responsible for simulating and are within the Lanecontainer.

The `EventDrivers` contain a PacketBox and deliver all Packets in the PacketBox when their specialized `Go()` method is invoked. Note that the delivery of a Packet may result in more Packets being placed into the PacketBox in response to the delivered Packets.

The Hybrids exercise both time and event driven behavior.

In a simulation, objects communicate with Events or Messages. The base class Packet captures the common features of Events and Messages.

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.

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.

The BigBen class provides a timer that delivers Packets 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 LO deliver all Packets that correspond to timeout requests that expire at that time.

The class hierarchy for these classes is given in Figure 11.

5 SmartAHS Modules and Their Use

In this section we describe the various functional modules of SmartAHS and their use in the specification and evaluation of control hierarchies.

[2] In implementation, infinity is **just** a very large number.

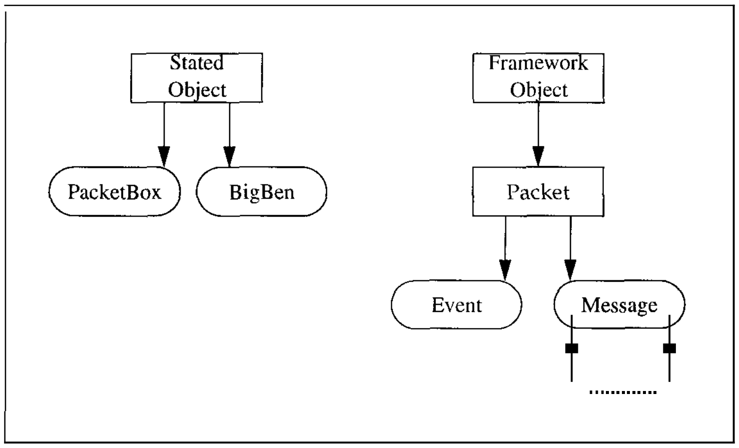

Figure 11: Class Hierarchy for Event Management Entities

5.1 SmartAHS functional Modules

The steps for simulation setup are summarized in Figure 12. These steps are:

1. 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.

   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.

2. 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.

3. 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.

4. Vehicle Automation Device Specification

   Different automation strategies will choose different configurations of automation devices in the Vehicles. Application developer are expected to provide **`Factory`** entities that have the capability of creating vehicles with the appropriate configuration.

   The use of **`Factorys`** is discussed in Section 6.

5. 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.

6. 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 Customizing SmartAHS

Figure 13 describes how application developers extend SmartAHS objects and how specialized classes become part of the simulation setup. The details of application development are discussed in section 6.

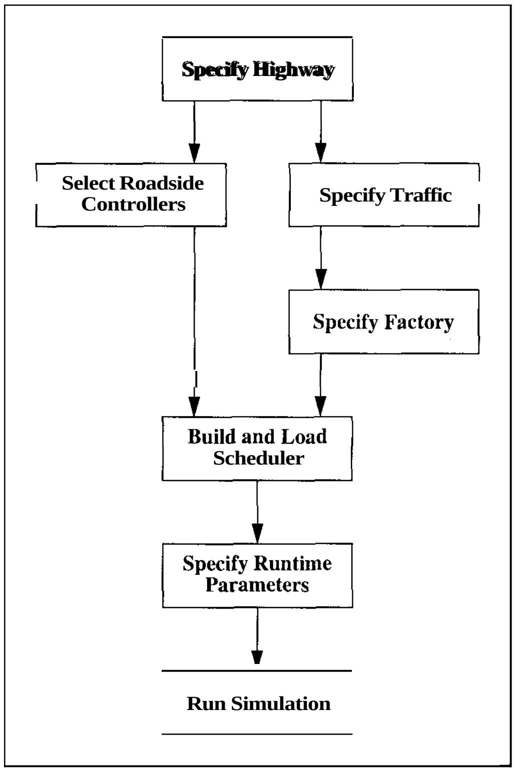

Figure 12: SmartAHS Specification Sequence for Simulation Setup

5.3 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.

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. 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 a.t 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.

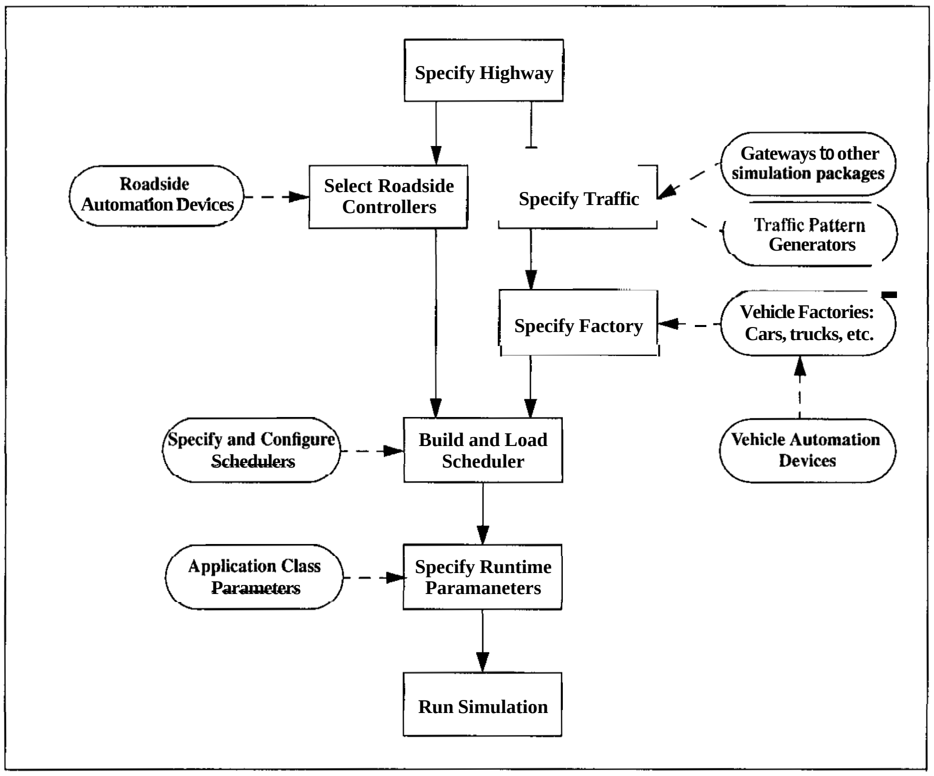
Specify Highway
Roadside Automation Devices
Select Roadside Controllers
Specify Traffic
Gateways to other simulation packages
Traffic Pattern Generators
Specify Factory
Vehicle Factories: Cars, trucks, etc.
Vehicle Automation Devices
Specify and Configure Schedulers
Build and Load Scheduler
Application Class Parameters
Specify Runtime Paramaneters
Run Simulation

Figure 13: Integrating Specialized Classes During Simulation Setup

5.4 Graphical Debugger

SmartAHS provides a graphical debugger which enables the user to view the evolution of the system. 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 overall architecture of the graphical debugger is illustrated in Figure 14.

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.

The `GUI` layer recognizes SmartDb objects only. The methods it invokes on objects are restricted to `SetErrorLevel()`, `SetLogLevel()`, and `Pack()`. Each class implements a specialized `Pack` method[3]. Poly-

[3]This method is code-generated from the class header file.

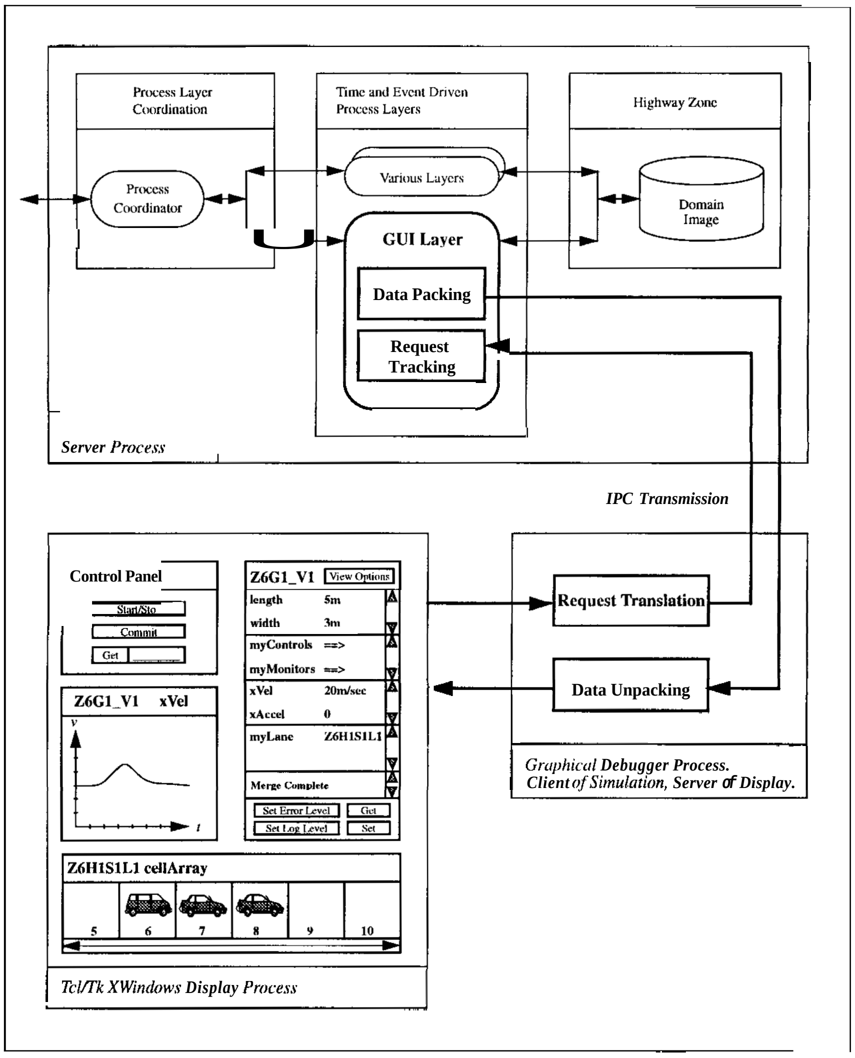

Figure 14: Graphical Debugger Architecture

morphism ensures that proper SmartAHS or SmartPATH object methods are invoked.

6 Use Case Example of SmartAHS

In this section we give examples of **`Control`**, **`Monitor`**, and **`Traverser`** subclasses. In particular we create the **`Myvehicle`** that is capable of performing *merge*, *lead*, and *follow* maneuvers.

6.1 A Specialized Vehicle

The **`Myvehicle`** class is composed of many components.

The class hierarchy for **`Myvehicle`** components is given in Figure 15. The **`Myvehicle`** containment hierarchy is given in Figure 16.

The Vehicle constructor is given in Table 1. Here `Link<type>` is a C++ template for an overloaded pointer. The keyword **`Persistent`** in the instantiation indicates that the instances are to be stored in the database.

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

```
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 1: The Myvehicle Constructor

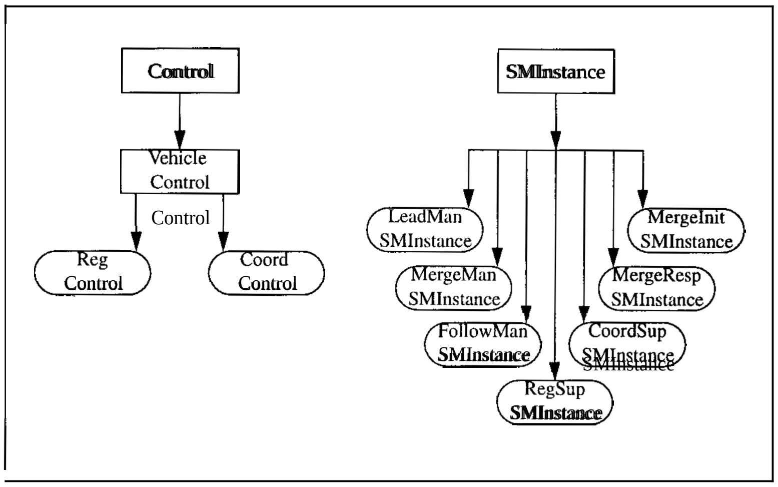

Figure 15: Class Hierarchy for **Myvehicle** components

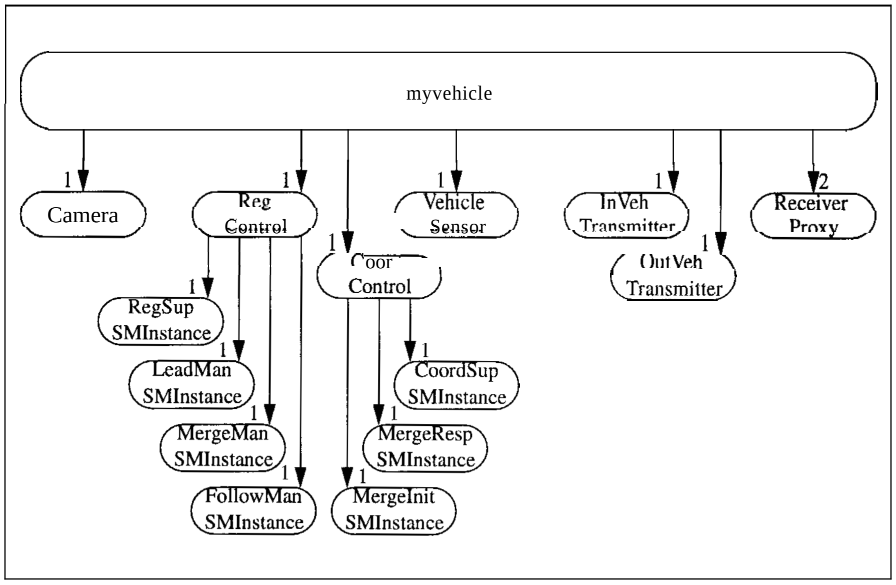

Figure 16: Containment Hierarchy for **Myvehicle** components

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** methods set the container-containment relationship. These objects' constructors have the responsibility of recursively instantiating their containees.

After all containees are instantiated, the **Myvehicle** constructor **Connects** its containees. The **Connect** method of each object has the responsibility of establishing its relationships, and the relationships of its containees.

The **Myvehicle** uses the default **Sensor,Transmitter** and **Receivers** provided by SmartAHS.

The **CoordControl** and **Regcontrol** classes consists of state machines and specialize the **Control** class. The constructors of these classes instantiate the respective state machines and establish their initial relationships. The state machines are specified in the special state machine syntax and code generated into C++ classes. The user need not perform any customizations for them since they react to events, messages and timeouts, a behavior that is implemented by base classes.

The **CoordControl** class is event driven, and all of its behavior is given by its state machines that determine what maneuver to execute. The **Regcontrol** has a number of state machines each capable of executing a different maneuver. This layer is time and event driven. For time driven behavior, it specializes the **Run** method to deliver a **click Event** to the active maneuver state machine.

Finally, the **Camera** object models a camera mounted on the front hood of a vehicle. It captures information about the **Vehicles** 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.

6.2 Constructing a Scheduler

The simulation uses four **Layers.** The **Physical** layer is used to update the position of a **Vehicle** on the highway. 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.

The **Physical** layer is used as provided by SmartAHS.

The **Regulation** layer subclasses the **Hybrid** class. It contains the **PacketBox** for the **Regcontrol** state machines. The **Regulation** layer also specializes **Traverser** methods, to invoke the **Run** method of each **Regcontrol** object.

The **Coordination** class does not provide any specialized methods. It only sets the name of the **EventDriver** to "Coordination".

The **GUI** class directly inherits from the **Layer** class and specializes the **Go** method to execute the appropriate graphical debugger code.

6.3 Creating A Simulation

We follow the steps outlined in Section 5

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.

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.

No roadside automation devices are used for this simulation.

The **Vehicle** and scheduling object specializations were discussed above. A separate executable creates the scheduler objects and commits them to the simulation database.

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 0.05 seconds. This information is specified in the simulation parameters file.

The GUI debugger does not require any particular configuration.

The executable that runs the simulation establishes a connection to the simulation database and retrieves the **ProcessCoordinator.** It takes in an argument that specifies how many times the **Go()** method of the **ProcessCoordinator** should be invoked. It **Commits** the simulation state to the database at specified intervals.

7 Performance Results

In this section we provide performance results for several simulation runs. The simulations were run on a Sparc10 workstation with 64Meg of memory.

Figures 17, 18, and 19 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 17, 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.

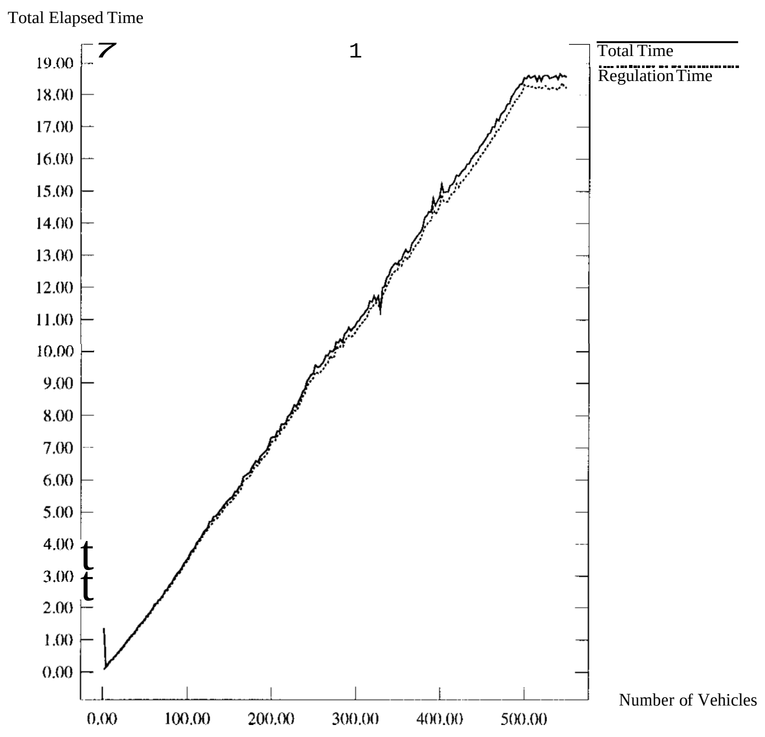

Figure 17: Single Agent Vehicles Moving On Highway

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.

The second scenario, Figure 18, 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 19, 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 hard-coded 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 20 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.

8 Conclusions

We have discussed the use of the object-oriented paradigm in the implementation of a simulation framework for the uniform specification, simulation, and evaluation of highway automation architectures.

The framework was developed in C++. Recently we developed a new language called SHIFT that is

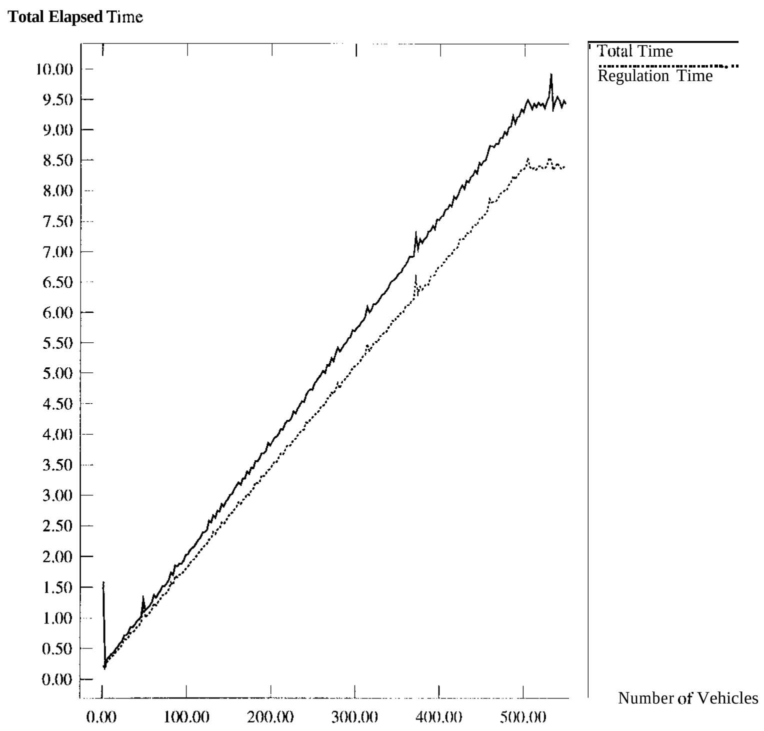

Figure 18: 2-Vehicle Platoons Moving On Highway

more suitable for hybrid system specification and simulation. Currently we are reimpelmenting SmartAHS in SHIFT.

References

[1] IVHS America. *Strategic Plan for Intelligent Vehicle-Highway Systems in the United States.* Report No IVHS-AMER-92-3. 20 May 1992.

[2] F. Eskafi and P. Varaiya. "SmartPath: Automatic Highway Simulator" *PATH Technical Memorandum,* UC Berkeley. June 1992.

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

[4] Q. Yang. "A microscopic traffic simulation model for IVHS applications". Master's thesis, Department of Civil and Environmental Engineering, MIT, Cambridge, MA, 1993

[5] KLD Associates. NetSim and WatSim Reference Manuals.

[6] V. Swaminathan, J. Storey. "A Framework for Telecommunications Operations Systems" *Proceedings of IFAC.* SF, 1996.

[7] A Gollii, A. Deshpande. "Object Management Systems". *Proceedings* of *IFAC.* SF, 1996.

[8] Peter Wegner. "Concepts and Paradigms of Object-Oriented Programming", *ACM SIGPLAN OOPS Messenger,* 1(1), Aug 1990.

[9] 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.

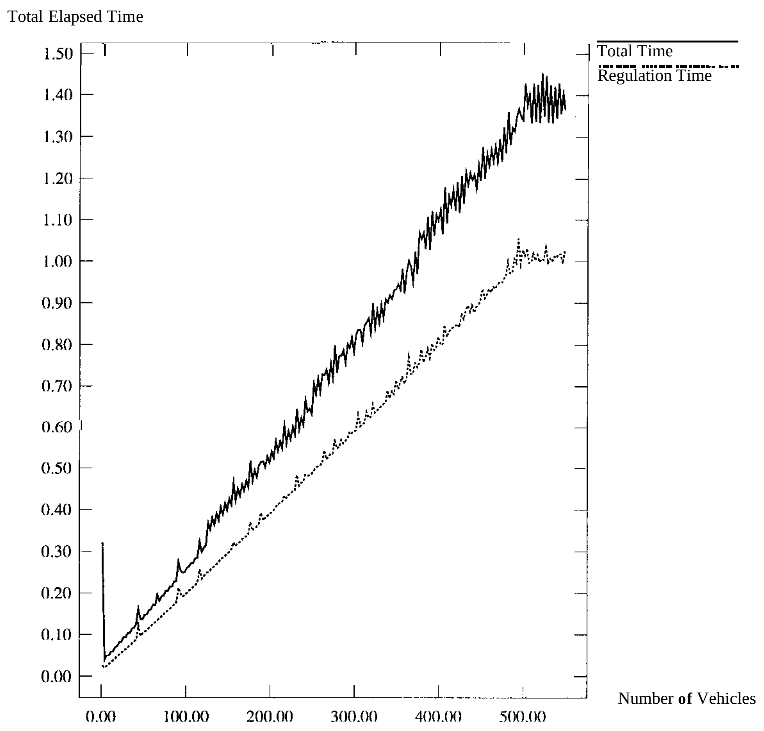

Figure 19: Single Agent Vehicles Moving On Highway with Hard-Coded Displacement

[10] J. McManis and P. Varaiya. "Suspension Automata: A Decidable Class of Hybrid Automata", *Proceedings 6th Workshop Computer-Aided Verification,* Stanford CA 1994.

[11] "Estelle – A Formal Description Technique Based on Extended State Transition Model" ISO9074, 1988

[12] "LOTOS – A Formal Description Technique Based on the Temporal Ordering of Observational Behavior" ISO8807, 1989

[13] "Specification and Description Language SDL", *International Telecommunications Union-T Rec.Z.100* 1988.

[14] Ralph E. Jonson. "How to Develop Frameworks", *OOPSLA Tutorial Notes,* ACM Press. 1993.

[15] *The Almagest.* Ptolemy Manual Vol 1-4, Version 0.5.2, College of Engineering, UC Berkeley, 1995

[16] H. Schwetman. *CSIM Reference Manual (Revision13),* Microelectronics and Computer Technology Corporation, 3500 West Balcones Center Drive, Austin, TX 78759, 1989.

[17] Pravin Varaiya. "Smart Cars on Smart Roads: Problems of Control", *IEEE Trans. Automatic Control* Vol. 38. No 2. Feb. 1993.

[18] Steve Shladover *et.al.* "Automated Vehicle Control Developments in the PATH program", *IEEE Trans. Vehicular Tech.* Vol. 40. pp. 114-130. Feb. 1991.

[19] A. Gollii. "Object Management Systems" PhD. thesis, UC Berkeley 1995.

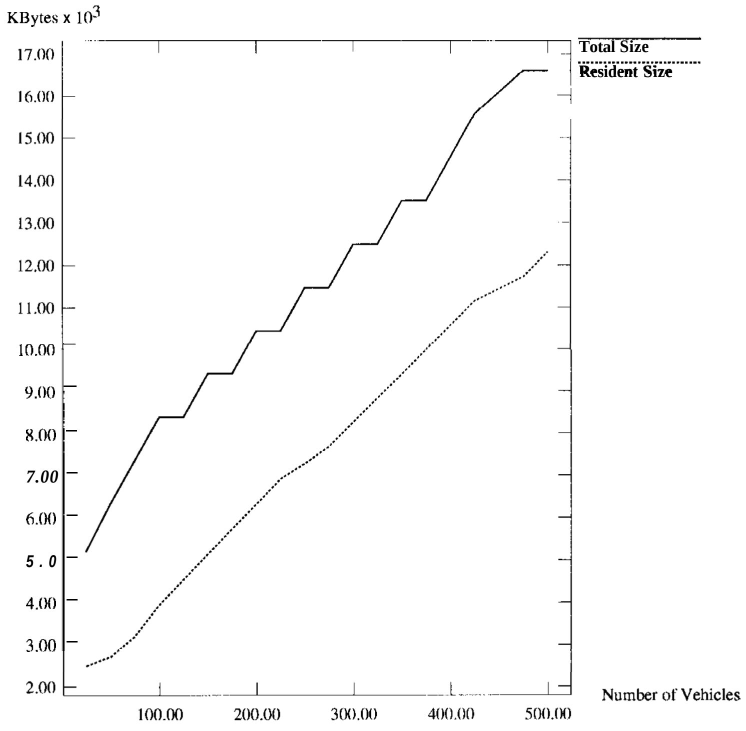

Figure 20: Memory Use of Simulation

[20] A. Gollii, P. Varaiya. "A Dynamic Network of Hybrid Automata". *Sixth Annual Conference on AI, Simulation, and Planning in High Autonomy Systems,* San Diego, CA 1996.

Appendix 2

Object Management Systems

Object Management Systems *

Aleks Gollii and Akash Deshpande
UC-Berkeley
golluQeclair.eecs.berkeley.edu
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley, CA 94720

Abstract

We describe a new approach for developing large-scale object-oriented software systems, which we call Object Management Systems (OMS). OMS are model-based distributed applications used for managing complex physical environments. The management functions supported by OMS are configuration, fault, performance, accounting, access and security, resource, and planning management.

The OMS Tool Set consists of a semantic data and process model called the OMS Object Model; the SmartDB software platform that implements this object model and provides formalisms, tools, and interfaces for modeling, synthesizing, optimizing, and verifying application specifications and implementing them in an open and distributed architecture; customized extensions of SmartDB — SmartNet, Smartpower, and SmartAHS—for telecommunications, power distribution, and automated highway systems application domains; and the OMS Software Engineering process.

Use of the OMS Tool Set significantly alters the software engineering life-cycle of large-scale projects, reducing their risk, budget, and schedule. The OMS-based development process consists of three stages: domain customization, system architecture, and application programming. Each stage produces software specifications and implementations integrated around the OMS Tool Set.

The OMS approach achieves integrated models, tools, and processes by focusing on a specific, but large, class of applications and by following a model-based approach to system development. We believe that the OMS approach meets an important need of today's software industry by providing standardized application-level development tools for object-oriented databases and software systems.

1 Introduction

Object Management Systems (OMS) are object-oriented software systems for simulating, evaluating, and controlling 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.'

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

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;

*Thisresearch 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.

'These functions are based on the OSI NM/Forum functional categories for network management[12].

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

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

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

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

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

Three technological factors have a profound impact on the success and usefulness of such management systems for complex physical environments:

- The data and process models used for describing 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 which includes the core database engine, modeling tools such as form and report generators, and application development utilities. Further, the relational model has a powerful Structured Query Language (SQL) with a sound mathematical basis [6]. 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 [16, 1]. The emerging object databases are of two types: those tied closely to the relational model and those tied closely to programming languages. 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

[2]Refer to the object-oriented methodologies described by Booch [3], Coad [5], Rumbaugh [13], and Shlaer and Mellor [15].

between objects on the 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.

With respect to software engineering processes, today's typical project life-cycles consist of the following stages: requirements analysis, functional specification, system architecture, prototype development, system design, implementation, integration, test, release, and maintenance. The stages up to prototype development are often treated as the first phase of the project and the subsequent stages up to system test are treated as its second phase. 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 some 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 slipping schedules during the project's second phase.

In section 2 we show how our OMS-based approach addresses these project technology and management concerns. In section 3 we apply the OMS approach to the automated highway systems transportation project.

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. Naturally, 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, SmartNet for network management, and Smartpower for power distribution management. Figure 1 shows these components, collectively known as the OMS Tool Set.

The OMS software engineering process consists of the following stages:

- Domain Customization;
- System Architecture; and
- Application Programming.

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, observation and control channels, event propagation, 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

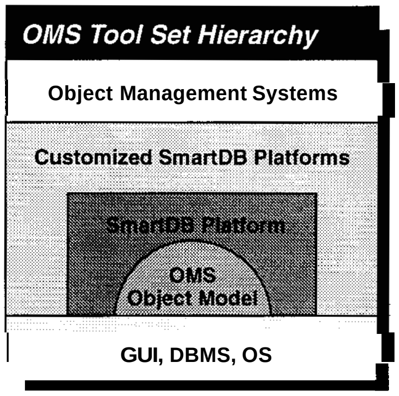

Figure 1: The OMS Tool Set

behaviors and control and coordination 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, and the risk of a "disconnect)' between domain experts and system experts is absent. 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 systems development.

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.

2.1.1 State

An object's attributes describe its state, inputs, and outputs. The system is a 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.

2.1.2 Methods

The methods of an object are memoryless (or reentrant) maps from its state and input to a new state and new outputs. Methods can encode object behaviors using different kinds of deterministic, nondeterministic, or stochastic dynamical models such as finite state machines or differential equations. 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.

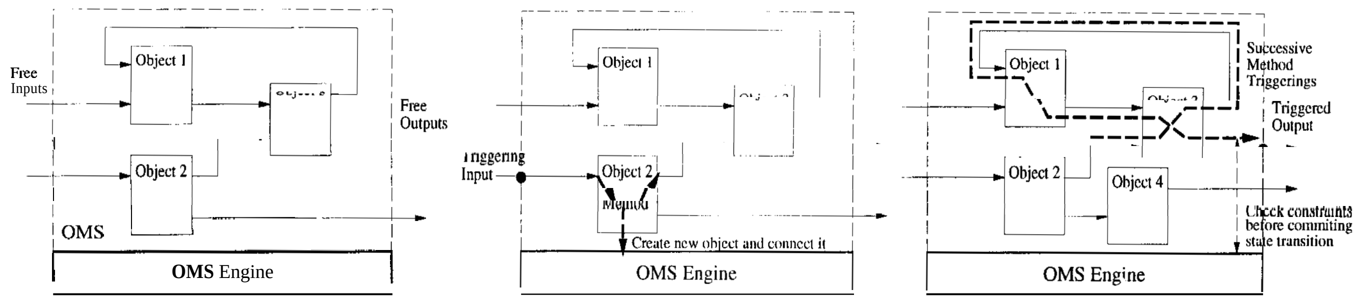

Figure 2: (a) A Sample OMS. (b) Triggering Input. (c) State Transition.

2.1.3 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 or indeterminacies) and that it is finite (i.e., 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.

2.1.4 Constraints

Constraints of four types are defined:

State construints—
constraints on the values of the state, inputs, and outputs of an individual object;

Connection construints—
constraints on establishment of input-output connections between objects;

Relationship constraints—
state constraints expressed over several objects that are related through relationships; and

Behavior construints—
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.

Note the resemblance of this OMS data and process model to integrated circuit diagrams, output feedback control systems, Petri nets, and neural networks.

2.1.5 Distribution

Because of the restrictions imposed on the state transitions, the expressive power of a stand-alone OMS is less than that of recursively enumerable languages. This is so by design: formal tools are available for synthesis, verification, simulation, and testing in the case of regular languages and some context free and context sensitive languages [8, 11]. The restricted expressive power of a stand-alone OMS enables us to exploit such tools.

However, a combination of two or more interacting OMS can be used to obtain the full expressive power of recursively enumerable languages. Such a combination can be used to realize distributed and hierarchical system architectures. For example, we use a hierarchy of three OMS to implement time- and event-driven scheduling (see [10]). In addition to providing full expressive power, distributed and hierarchical systems can be used to enhance modularity and control complexity of the implemented systems.

2.2 SmartDB

SmartDB is a software implementation of the OMS Object Model. In addition to the implementation of this 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. Process layers are used to build layered or hierarchical system architectures.)
- It implements commonly used 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.

In addition to these modeling tools, SmartDB provides database interfaces and programming mechanisms that simplify OMS implementation tasks.

2.2.1 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, commit and rollback, and backup and restore.

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: backup and restore, utilities for forms and reports, and on-line schema evolution.

If the desired or useful features are available in the database management system, SmartDB functionality can be enhanced accordingly.

2.2.2 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 telecommunications networks and power distribution management. 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. (See section **3** for further details.)

SmartNet—
Links: channels, facilities, circuits, packets, and services. Network elements: equipment, functions, modules, multiplexors, buffers, switches, terminals, and users. Process layers (following the OSI reference model): physical, data link, network, transport, session, presentation, and application; Data and processing distribution based on geographical regions. Sensors and actuators based on SNMP and OSI NM/FORUM protocols [12, 4].

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

3 The AHS Application

The concepts and tools for OMS have emerged from research on the Automated Highway Systems project at the University of California at Berkeley [17, 14]. The AHS project at UC-Berkeley is part of a comprehensive program initiated by the U.S. government under the Intermodal Surface Transportation Efficiency Act of 1991 to improve safety and reduce congestion in the surface transportation system. UC-Berkeley's PATH program is a partner in a nine-member consortium along with General Motors, Bechtel, Parsons Brinckerhoff, Martin Marietta, Delco, Hughes, Caltrans, and Carnegie Mellon University. The consortium is funded in part by the U.S. Department of Transportation and it is responsible for designing, evaluating, and demonstrating **a** prototype AHS. The SmartAHS platform and the associated SmartPath simulator are important tools used by the consortium for designing, evaluating, and deploying AHS.

There is also substantial related activity in Europe under the PROMETHEUS[3] and the DRIVE[4] projects, and in Japan under the RACS[5], AMTICS[6] and VICS[7] projects.

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 — join, split, lane change, entry, and exit — that are coordinated.

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

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;

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

Coordination Layer—
communication protocols that vehicles and highway segments follow to coordinate their maneuvers 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.

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 [9, 7].

SmartAHS provides generic objects for modeling highway configuration, vehicles, control and communication agents, and performance monitors. SmartAHS also provides a scheduler 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 100ms and roadside controllers every 15s.

SmartPath consists of specialized objects and their behaviors 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 seven specifications: highway configuration, travel demand, weather conditions, highway automation devices, vehicle automation devices, and the simulation scheduling policy. (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 5–50ms 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. (The integration step-size is dynamically adjusted by SmartAHS based on vehicle displacement.) Simulation profiles indicate that 80% of the simulation time is spent on time-driven simulation of the differential equations that model vehicle dynamics. Implementing these on a vector parallel processing system is expected to speed up the simulation significantly.

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

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 distributed operating system for command and control of the deployed AHS.

4 Conclusion

Object Management Systems are a new approach for developing large-scale object-oriented software systems. OMS are model-based distributed applications used for managing complex physical environments. The management functions supported by OMS are configuration management, fault management, performance management, accounting management, access and security management, resource management, and planning management.

The OMS Tool Set consists of a semantic data and process model called the OMS object model, the SmartDB software platform, customized extensions of SmartDB—SmartNet, Smartpower, and SmartAHS—for specific application domains, and the OMS Software Engineering process.

The OMS-based development process consists of three stages: domain customization, system architecture, and application programming. Each stage produces specifications and implementations in software integrated around the OMS Tool Set. This software development life-cycle reduces project risk, budget, and schedule.

The current implementation of SmartDB uses the cSS programming language, Versant Object Database, TCL/TK user interface tool kit, and the UNIX operating system. In future releases of SmartDB, we expect to encapsulate platform dependencies and provide independence from specific database or graphical user interface platforms. The SmartDB and SmartAHS platforms together represent about fifteen person-years of effort, and the SmartPath platform represents another fifteen person-years of effort. The SmartAHS platform is being used for highway systems evaluation by the nine-member National Automated Highway Systems Consortium including General Motors, Hughes, Martin Marietta, Delco, and other industry, university, and government partners.

References

[1] M. Atkinson *et. al.* "The Object-Oriented Database System Manifesto." *Proc. of 1st Intn. Conf. on Deductive and Object-Oriented Databases.* Kyoto, Japan. Dec. 1989.

[2] K. Beck. *et. al.* "Can Structured Methods Be Objectified?" Panel discussion in *Proceedings of OOPSLA '91.*

[3] G. Booch. *Object-oriented analysis and design with applications.* Benjamin/Cummings. 1994.

[4] J. Case, K. McCloghrie, and M.T. Rose. "Introduction to Version 2 of the Simple Network Management Protocol." Internet Society. July 1993.

[5] P. Coad and E. Yourdon. *Object-Oriented Analysis*. Yourdon Press, Englewood Cliffs, NJ. 1991.

[6] Chris J. Date. *An Introduction to Database Systems*. Addison-Wesley, Reading, Massachusetts. 1985.

[7] F. Eskafi, Delnaz Khorramabadi, and P. Varaiya, "An Automatic Highway SystemSimulator." *Transpn. Res.-C* Vol. 3, No. 1, pp. 1-17, 1995.

[8] J-C Fernandez, *et al.* "A Toolbox for the Verification of LOTOS Programs" in Proceedings of the 14th International Conference on Software Engineering ICSE'14 (Melbourne, Australia). 1992.

[9] A. Gollii, 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*. Gainesville, Florida. 1994.

[IO] A. Gollii. "Object Management Systems." *Ph.D. Thesis,* University of California at Berkeley. 1995.

[11] R. P. Kurshan. *Formal Verification of Coordinating Processes: The Automata-Theoretic Approach.* Princeton University Press. 1994.

[12] OSI/Network Management Forum. "OSI Basic Reference Model." ISO 7498-1. 1992.

[13] James Rumbaugh *et. al. Object Oriented Modeling and Design.* Prentice-Hall, Englewood Cliffs, N.J. 1991.

[14] Steve Shladover *et al.* "Automated Vehicle Control Developments in the PATH program." *IEEE Trans. Vehicular Tech.* Vol. 40. pp. 114-130. Feb. 1991.

[15] Sally Shlaer and Stephen Mellor. *Object-Oriented Systems Analysis: Modeling the World in Data.* Yourdon Press, Englewood Cliffs, N.J. 1988.

[16] M. Stonebraker *et al.* "Third Generation Data Base System Manifesto." *Proc. of IFIP DS-4 Workshop on Object-Oriented Databases.* Windermere, England, July 1990.

[17] P. Varaiya. "Smart Cars on Smart Roads: Problems of Control." *IEEE Trans. Automatic Control.* Vol. 38, No 2. Feb. 1993.

Akash Deshpande received his Ph.D. in EECS from the University of California, Berkeley (1994), M.S. in EE from the University of Florida, Gainesville (1987), and B.Tech. in EE from the Indian Institute of Technology, Bombay (1985).

He is a lecturer in the EECS Department of the University of California, Berkeley. He is one of the founders of OMS Technologies. He was a Senior Engineer at Teknekron Communications Systems and an Engineer at AT&T.

He has a decade of industry experience in the areas of telecommunications, control, and information technologies, project management and technical leadership, and international marketing of technology products and services.

Aleks Gollu received his Ph.D. in EECS at the University of California, Berkeley (1995), M.S. in EECS at UC-Berkeley (1989), and B.S. in EE at Massachusetts Institute of Technology (1987).

He is one of the founders of OMS Technologies. He was a Systems Engineer at Teknekron Communications Systems and a Software Engineer at Oracle.

He has a decade of industry experience in large-scale software development and database management systems ranging from system development to project management. His domain expertise includes telecommunications, power distribution, highway automation, and control technologies.