SHIFT and SMART-AHS: A Language For Hybrid System Engineering Modeling and Simulation

Marco Antoniotti
A. Göllü
University of California, Berkeley

SHIFT and SMART-AHS: A Language For Hybrid System Engineering Modeling and Simulation

Marco Antoniotti Aleks Göllü
California PATH
University of California at Berkeley
{marcoxa,gollu}@path.berkeley.edu

Abstract

SHIFT *is a new programming language, whose aim is to facilitate the implementation of* reusable simulation frameworks *by teams of engineers.* SHIFT *incorporates system theoretic concepts emerging from the field of* Hybrid Systems *analysis and modeling. The SMART AHS framework is a collection of* SHIFT *libraries devoted to the construction of* Hybrid System *based simulation of* Automated Highways Systems*. In this paper we describe how the* SHIFT *simulation environment and language have impacted on the development of the SMART AHS framework. Our claim is that* SHIFT *provides the proper level of abstraction for engineers who face complex modeling and simulation tasks, where phase changes and continuous variables interact in subtle ways.*

1 Introduction

SHIFT is a new programming language, whose aim is to facilitate the implementation of *reusable simulation frameworks* by teams of engineers. SHIFT incorporates system theoretic concepts emerging from the field of *Hybrid Systems* analysis and modeling (e.g. see [1]) into an object-oriented language environment that offers the proper level of abstraction for describing complex applications such as automated highway systems, air traffic control systems, robotic systems, shop floors, coordinated submarines and other systems whose operation cannot be captured easily by conventional models. These application domains share the following key characteristics.

- The behavior of objects in the system have both continuous and discrete event components;
- The systems consist of heterogeneous set of interacting objects where models of individual objects are known and the goal is the study of the emergent behavior resulting from their interaction; and
- A static block diagram representation is not sufficient to specify all data dependencies among objects since the sets interacting objects vary over time.

Moreover the practitioners involved in these projects are, most of the time, engineers who like to work with their own "tools of the trade". SHIFT offers them with two of them – automata and differential equations – well integrated into a single environment and language.

Our work on simulation frameworks was driven by application needs in highway automation. In the early 90s, PATH (Partners for Advanced Transit and Highways) proposed a specific control hierarchy for the highway automation project. The need for a simulation environment was obvious, since we were dealing with a complex and very large system, which gave no hope for a closed form mathematical analysis. Following an unsuccessful market study of the available tools, a decision was made to internally develop a system to simulate the PATH control architecture for highway automation. A C based simulation system, SmartPath, was created [8]. The project evolved and gained national attention.

Following the original proposal, the National Automated Highway System Consortium (NAHSC) was funded and other highway automation architectures were developed. It became evident that a generalized simulation framework was needed that could facilitate the specification, simulation, and evaluation of different highway automation architectures.

As a result SMART AHS$_{C++}$, a C++ based persistent simulation framework was developed [8].

SMART AHS$_{C++}$ used a set of class libraries developed in C++, SmartDB, as its object model, and delivered a framework for further customization by application developers. However, SMART AHS$_{C++}$ had it shortcomings. It did not allow its users to program in their "own" domain which was differential equations and state machine representations. It introduced artificial specification rules and syntax resulting from the use of a general-purpose programming language. We feel that these shortcomings are shared by most simulation libraries embedded in a host language which does not support language extensions (like Simula, C or C++ or Java).

In parallel to our AHS work, we were involved with several other projects, such as air traffic management, power transmission and distribution systems, and network management systems. In system engineering, we have observed a general shift towards hierarchical control of large systems that combined classical continuous feedback systems, with more recent discrete event based control algorithms and protocol specifications. This hybrid systems paradigm has proven ideal for the specification, control, and verification of such complex, large, dynamical systems.

Our experience with a multitude of such systems resulted in a set of requirements for frameworks for the design, specification, control, simulation, and evaluation of large dynamical systems. No language, product, or tool in the market nor in academia satisfied all requirements.

The design and implementation of a language that addressed all the requirements required expertise from several disciplines including computer science, electrical engineering, and mechanical engineering. Such a multi-disciplinary team was assembled at PATH/UC Berkeley and a new programming language, SHIFT, was born.

This paper provides an overview of the concepts and constructs of the SHIFT language, and a discussion of the impact on the development cycle of simulation model illustrated through the SMART AHS case study. The SHIFT mathematical model is beyond the scope of this paper: we refer the interested reader to [7] for a comprehensive exposition.

1.1 Related Work

SHIFT is used to describe models with switched differential equations (such as a vehicle with automatic gear shift) and coordinated behaviors (such as communicating controllers). Standard math and simulation tools such as Matlab^{tm}, Mathematica^{tm}, Maple^{tm} or Matrix$_X{}^{tm}$, while suitable for numerical or symbolic integration of fixed sets of differential equations, are difficult to use in applications with rapidly changing sets of differential equations (due to the evolution of relationships among components), complex event-triggering conditions (such as existential queries on the state of the world), and complex program logic (such as synchronous compositions of state machines). More traditional *discrete event* simulation packages like GPSS^{tm}, while offering a tried and tested base, lack the facilities for writing concise hybrid systems models.

The hybrid systems approach [1] satisfies our needs for component modeling but it lacks the capacity to model dynamically reconfigurable interactions between components.

The Omola/Omsim [11] language has a very similar approach to hybrid system modeling as SHIFT. Both systems provide a modeling language with simulation semantics; both support discrete event and continuous time behavior representation; both have the necessary constructs for hierarchical modeling and specification reuse. However, Omola is designed to represent statically interconnected objects. Furthermore, it does not provide the means to manipulate sets and arrays of components. In SHIFT, these manipulations are used to express and compute the evolution of the interconnections among components as the world evolves.

Statecharts [9] and Argos [10], based on Statecharts, are approaches for synchronous discrete event modeling. Their focus is on hierarchical specification of finite state machines. SHIFT does not provide explicit facilities for hierarchical behavior specification; instead, it provides a sub-typing mechanism wherein a subtype (presumably more detailed) must present the same interface as its super-type. SHIFT adds continuous time semantics and dynamic reconfigurability of the synchronization structure. Subtyping and other constructs may be used to organize components hierarchically.

Recent extensions to the DEVS [16] formalism have introduced notions of dynamic reconfiguration [4, 13]. However, the DEVS formalism is primarily aimed at discrete event simulation and the extensions for continuous evolution laws are limited. Model specification in DEVS is done with C++, SmallTalk or Common Lisp classes that implement the mathematical model at hand, requiring the user to work at the host language level.

1.2 Preliminary Discussion

Our mathematical model for the discrete event semantics is similar to Milner's π-calculus [12]. Both models achieve reconfiguration by a renaming of event labels used in synchronization. The finite state machine part of SHIFT implements this model. The differential equation part of SHIFT allows systems of first order ODE's.

The abstraction facilities in general-purpose programming languages such as the original Simula or C/C++, although powerful enough to encode our models, would not allow us to write simple, concise descriptions of our designs. The best that could be hoped for, would be an integration at the level of "embedded interpreters" *à la* Tcl/Tk or SQL[1].

SHIFT provides both high-level system abstractions and the flexibility of a programming language. However, all the features that SHIFT offers are carefully designed to constrain the programming style and to conform to the underlying mathematical model, while avoiding frustration for the user.

As a first statement about the impact of SHIFT in the programming of complex simulations, when we will discuss the reimplementation of SMART AHS in SHIFT in Section 3, we will see that the size of the resulting "libraries" and "projects" decreased by almost 50%, while the code could be more easily reused.

Users of SHIFT within the PATH project, NAHSC and UCB reported favourably on the ease with which their engineering models were readily translated into working simulations.

Moreover, since a SHIFT program is a direct implementation of a hybrid system specification (even though an extended one), the resulting code can be easily manipulated and fed into the new breed of *automated verification systems* like KRONOS [6].

Though not substantiated by a direct comparative study, but only by an "a posteriori" examination of the evolution of SHIFT, SmartPATH, SMART AHS$_{\mathrm{C++}}$ and SMART AHS, we claim that these results justify the considerable research and implementation effort that went into the develoment of these new tools.

[1]The SHIFT systems provides a C API for this style of programming.

2 SHIFT Language Overview

A SHIFT program describes a set of interacting objects called *components* and grouped into *component types*[2]. The SHIFT *type declaration* construct specifies the prototypical behavior of all components of a given type. SHIFT supports a single inheritance scheme which has proven sufficient for our needs.

The set of components types and their instances in a SHIFT program, directly describe a *hybrid system* with comprises synchronizing finite states machines and differential equations.

SHIFT additionally supports a small set of basic data types (number and symbol) – and of constructed types (array and set). The set of built-in types has the following characteristics:

- Objects of type number have piecewise constant or piecewise continuous real-valued time traces. The latter variables have type continuous number.
- Objects of type symbol have piecewise constant symbol-valued time traces. In SHIFT symbols are similar to C enumeration tags. However they do not require a declaration.
- An object of type set(T), where T is a native or user-defined type, contains a set of elements of type T.
- An object of type array(T) contains a one-dimensional array of elements of type T, whose dimension is determined at creation time.

A component prototype is defined by the SHIFT type declaration. The structure of a type roughly consists of

- *inputs*
  instance variables (or simply "variables") which can be read but not changed by the behavior of the component and which are visible outside the scope of the type definition.
- *outputs*
  instance variables which can be read and changed by the behavior of the component and

[2]Our terminology abuses words like *type*. Using more standard Object Oriented terminology, we would speak of *instances* and *classes*. We use the term "component" since the control theory application domain imposes a natural *part-of* metaphor on the software architecture.

which are visible outside the scope of the type definition.

- *states*
  instance variables which can be read and changed by the behavior of the component but that are not visible outside the scope of the type definition.

- *discrete modes and transitions*
  i.e. the definition of the finite state machine behavior of the type.

- *differential and algebraic equations*
  i.e. the definition of the continuous behavior of the type.

The terminology is taken from the standard Control Theory practice and it roughly translates into the well know concepts of *private* and *public* slots in a class. Also, the notions of *inputs* and *outputs* are supported by the language in order to promote a "black box" software development style.

As an example, here is a first SHIFT code fragment:

```
type car
{
        input
                continuous number throttle;
        output
                continuous number position, velocity;
                continuous number acceleration;
        state
                continuous number fuel_level;
                car car_in_front;
                controller controller;
                . . .
}
```

The *discrete finite state* behavior and the *continuous behavior* of a type are specified in different "clauses" of a user definition.

The continuous behavior is specified by ordinary differential equations and algebraic definitions which are grouped under the flow clause. Each instance variable can be used in these equations and their behavior is computed accordingly[3]. Each equation group (appropriately called a *flow*) can be labeled with a meaningful name. The default flow contains the equations which are to be used whenever there are no special provision for computing the value of the variables involved.

[3]Of course, only *continuous* number variables make sense in a differential definition.

The discrete clause defines the possible values for the type's mode (i.e. the finite state "current state") and associates to each of them a set of differential equations and algebraic definitions or one of the flows defined in the flow clause.

The differential equations are specified by systems of first order ODE of the form $x' = f(x, u)$, where x is a single variable and u is a vector of "other" variables. The algebraic definitions cannot contain circular dependencies. Such dependencies are detected at run-time, and an error is signaled by the run-time system.

As an example (continuing the "car" example):

```
type car
{
        . . .
        flow
                default {
                        position' = velocity;
                        velocity' = acceleration;
                }
        discrete
                accelerating        { acceleration = 3; },
                cruising            { velocity = 30; },
                brake               { acceleration = -5; };
}
```

Notice that the *cruising* state redefines *velocity*, which becomes algebraically defined (as a constant in this case) instead of differentially defined (as the integral of the acceleration).

Transitions between discrete modes are defined in the transition clause, as in the following example.

```
type car
{
        . . .
        transition
                accelerating -> cruising {}
                when velocity >= 30,
                cruising -> braking {}
                when position(car_in_front) - position < 5;
}
```

The example uses the state variable *car_in_front* containing a reference to another *car*, whose relative position is used in deciding when to apply brakes.

Transitions are labeled by a (possibly empty) set of event labels. These labels allow transitions to synchronize with each other. Moreover, transitions may be *guarded* by boolean expressions – introduced by the when keyword – and may trigger a set of actions grouped in a do clause. These actions *reset*[4] (i.e. assign) the values of variables, may create new components and may reconnect their inputs and outputs.

Suppose that we wish the car to brake when a roadside controller signals an emergency. This can be specified with the transition

```
type car
{
    ...
        cruising -> braking {controller:emergency}
        when position(car_in_front) - position < 5;
}
```

The definition of the *controller* type includes an *exported event*, *emergency*, and a transition that triggers it.

```
type controller
{
    export emergency, ...;
    discrete normal, panic_mode, ...;
    transition
        normal -> panic_mode {emergency}
        when some critical condition;
    ...
}
```

SHIFT allows the system modeler to specify very complex patterns of *synchronous composition* of finite state automata. The transition guards may contain existential quantifiers that query the state of sets of components (possibly all existing components). For example, let *cars* be the set of all the components of the *car* type and let the road consist of a single lane. Then, the variable *car_in_front* is updated as follows.

```
type car
{
    ...
    transition
        cruising -> cruising {}
        when exists c in cars :
            position(c) > position
            and position(car_in_front) > position(c)
        do {
            car_in_front := c;
        },
        ...
}
```

[4]The terminology, once again is borrowed from the field of hybrid system studies.

The initializations of a newly-created component of some type are defined in the setup clause. For example, each component of type *car* may add itself to the set *cars* when it is created.

```
type car
{
    ...
    setup
        do {
            cars := cars + { self };
        };
}
```

In practice a SHIFT program would not use this exact code unless *cars* were a small set. A more efficient mechanism requires maintaining multiple sets of cars associated with lanes and highway segments.

2.1 SHIFT Support Environment

SHIFT has many more features which we do not discuss here in further detail since they are outside the scope of the paper[5]. We now briefly comment on the SHIFT support environment and implementation.

SHIFT programs are translated directly into C by the `shic` compiler. The resulting C file is then linked with the SHIFT runtime library in order to produce an executable (a file conventionally ending with `.sim`). The runtime library takes care of the implementation of the high level data structures used by SHIFT (e.g. sets) and makes provisions to integrate the differential equations via standard Runge-Kutta algorithms. There are no special optimizations that are done by the compiler[6]. The only requirement imposed on the system is that the behavior of the run-time which essentially interprets the set of finite state machines and differential equations complies with underlying mathematical model.

[5]Among them: single inheritance, complex set and array formers à la SETL [14], garbage collection (using Boehm's conservative GC [5]), a foreign function interface facility and a C API which allows an experienced programmer to control SHIFT simulations from C and C++ programs.

[6]As a matter of fact, many constructs and compilation policies could be optimized away by some rather simple data flow analysis. However, this has not been so far the emphasis of our work.

There are several subtleties involved in the interaction between the RK integration routine and the guard evaluation code. Languages like Omola/Omsim that support only a static set of differential equations can perform compile-time optimizations to select integration step sizes. In SHIFT since the dependencies among the components can change at run-time, it is not possible to optimize integration step-sizes with respect to guard-crossings. SHIFT applications so far have been primarily in non-stiff systems, hence a fixed-step RK integration algorithm had the best performance. We recognize that this is still an open research field.

The executable file can be run is two ways: by starting a command line monitor or by connecting the simulation with a Tcl/Tk graphical user interface in a client/server fashion[7] (see Figure 1 for a screen shot).

Both the command line monitor and the graphical environment allow the user to control the running simulation. Typical operations include

- data inspection
- stepping by time click and by "simulated time"
- stopping and resumption of execution in correspondence of discrete transitions.

These operations support the simulation modeler "at the right level" of abstraction and allow her/him to quickly determine whether there are logical problems in the code.

3 Developing Simulation Frameworks in SHIFT: The SMART AHS Case

We used SHIFT to develop a specialized framework (SMART AHS) for the construction of simulation models of highways. The overall design principles were first described in [15]. The objectives of the SMART AHS framework are listed hereafter.

[7]The choice to use a client server architecture for the GUI was based on two considerations. (1) The development of the GUI could progress rather independently of the development on the SHIFT compiler and runtime. (2) It was recognized since the early design stages that SHIFT simulations could require huge memory spaces: hence the necessity to being able to run simulation remotely on powerful workstation while interacting with them on a local – and less powerful – machine.

1. To provide researchers with a standardized tool which can be used for evaluation of simulation results under different policies.
2. To allow the quick construction of alternative simulation models.
3. To allow the simulation of models at different granularity levels.
4. To be able to handle medium to large scale simulations.

In Sections 3.1 and 3.2 we describe the SMART AHS architecture and discuss the lessons learned in its deployment as one of the standard tools used by the National Automated Highway System Consortium (NAHSC).

3.1 SMART AHS Architecture

The SMART AHS framework is roughly divided into two parts. The first is a "static" part which contains highway type definitions used to compose different highway layouts. The second part contains different vehicle models used for diverse simulations.

The *highway* types comprise *Lane*, *Section*, *Segment*, *Barrier*, *Block* and *Weather*. These types and their structure constitute a *data description language* for highways.

The *vehicle* types are centered around a container type called *AutomatedVehicle*. Its most immediate sub-components are *Controller*, *VehicleModel* and *VREP* (*Vehicle Roadway Environment Processor*). Figure 2 contains a schematic representation of the *AutomatedVehicle* SHIFT code.

This architecture meets the requirements by allowing the system modeler to plug in different controllers and vehicle dynamic models. Researchers at PATH have successfully developed two classes of models for highly detailed vehicle dynamics simulation and for high volume highway simulations with complex vehicle maneuvers.

The detailed simulation model describes a vehicle at the level of gear shifting and engine dynamics. The model is realistic and based on real data collected by General Motor researchers.

The "high volume" simulation model uses a simplified vehicle dynamics model (there is no need to simulate the engine dynamics when computing flows over stretches of highway) and a controller

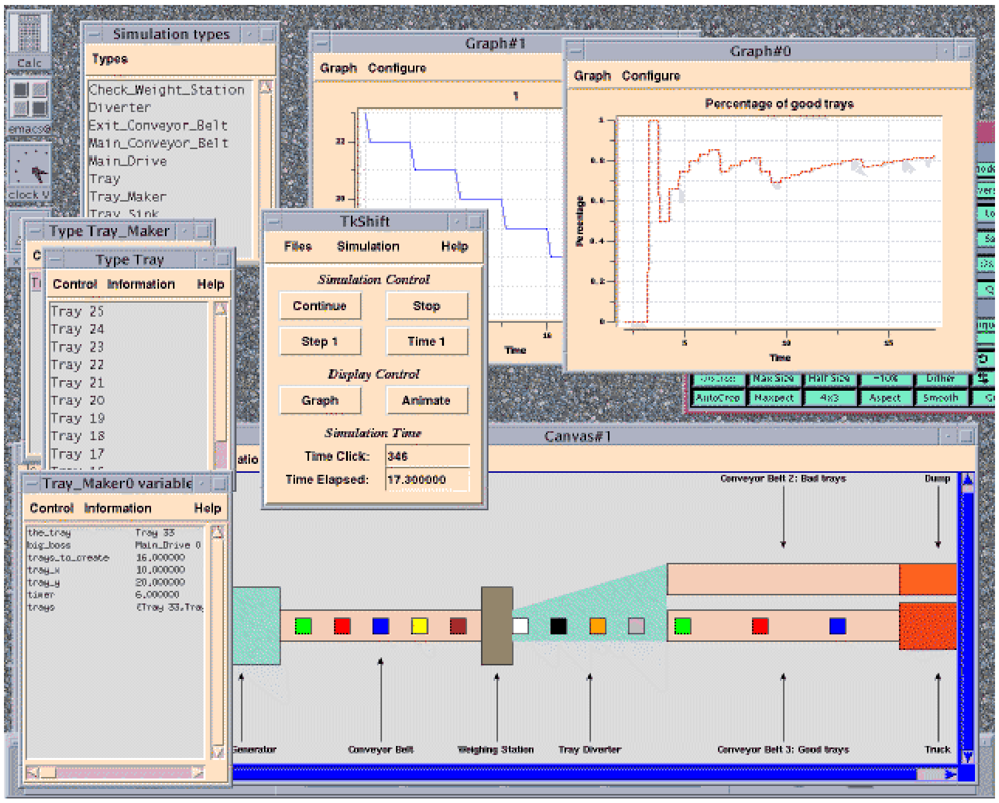

Figure 1: A screen shot of the Tcl/Tk SHIFT environment. The simulation being run models a food processing manufacturing line.

which is devoted to maintain *safety* parameters (e.g. distance from the vehicle in front) and to perform *merge* maneuvers on entrance and exit ramps.

3.1.1 Micro Simulation of Houston Metro Katy Corridor

The "high volume model" was developed to gather data for a project sponsored by the Houston Metro Transportation Authority. The project asked evaluation data for a preliminary design of a stretch of highway with three entry ramps and three exit ramps. The main objective of the study was to evaluate the congestion build up at the three entry ramps under different demand volumes with *autonomous* and highly automated vehicles[8]. Other parameters which were under study include the required length of the entry ramps to ensure completion of merge maneuvers. A more detailed description of the experiment and of its results is contained in [2, 3].

Most of the work done to develop the simulation went into the construction of a *Controller* module which would obey a distributed protocol for cruising and merging into existing traffic. A code fragment for the *Controller* is shown in Figure 3.

The GUI environment was used to visualize the results of the simulation in order to spot problematic areas of the protocol.

We tested four cases on a matrix given by *high* and *low* traffic demand and by enforcing two different *vehicle tracking* policies. The traffic demands levels can be summarized as follows:

low ca. 2000 vehicles/hour injected in the highway system.

high ca. 4000 vehicles/hour injected in the highway system.

In each case each of the vehicles was simulated by instantiating a full blown *AutomatedVechicle* con-

[8]The term "autonomous" is here intended in the following sense: a vehicle/driver which takes decisions based only on its sensor input and on certain assumptions on the behavior of nearby vehicles. The high automation characteristic can be thought of as modern *Adaptive Cruise Control* technology.

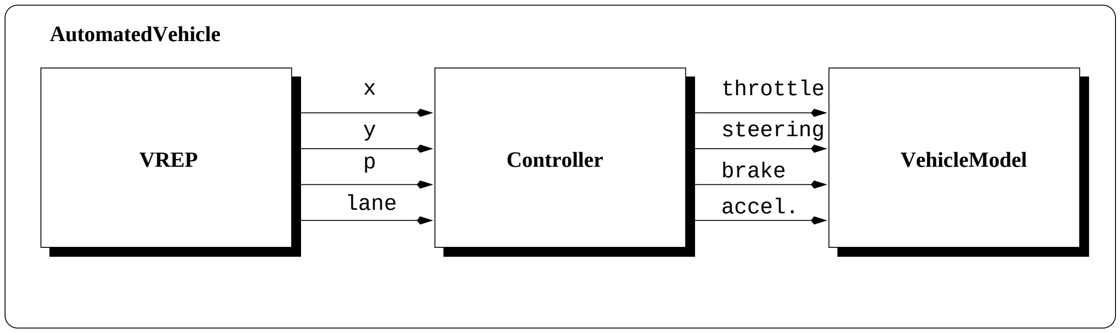

Figure 2: Schematic Block Diagram of *AutomatedVehicle* SHIFT code.

tainer, and each instance obeyed the protocol implemented in the *Controller*. The overall speed of the simulation, depending on the size of the input data was always between 4 and 9 times of *simulated physical time*[9].

3.2 Discussion and Evaluation of the SMART AHS Framework

The construction of the SMART AHS framework and of applications based on it, has given us an insight on how to conduct the development of an harmonious set of libraries. This was a requirement of the overall project and our aim. We consider our experience so far successful and we credit this success to two main characteristics of the SHIFT and SMART AHS framework.

- SHIFT provides a sound and restricted mathematical model (Hybrid Systems) which can be successfully mastered by an engineer in a rather short period of time. The tools provided (differential equations and finite state automata) are the "right" ones for the kind of models we were targeting.
- SHIFT and SMART AHS code is significantly more compact. The SHIFT implementation of SMART AHS consists of 3,300 lines of SHIFT code and 2,000 lines of C legacy code[10], plus about 5,000 lines for the full SHIFT runtime[11]. The Houston-Metro case study consisted of 3,800 additional lines of SHIFT code. The predecessor of SMART AHS (SMART AHS$_{C++}$) consisted over 20,000 lines of C++ code, without the code for the distributed merging controller and the highway building
- The SMART AHS framework has a very small set of "how-to-use" rules that a programmer needs to know about. As a result new users can immediately become more productive in application development. Based on our experience with the previous incarnations of SMART AHS, frameworks based on C/C++ usually have too many "how-to-use" rules that are not enforced by any compiler and result in unpredictable run-time errors if not followed properly.

Of course these remarks can be taken as a simple recipe for "good engineering practice", yet we claim that the overall design of the tools did pay off considerably.

The entire simulation study of the Houston Katy Corridor was built from scratch (i.e. the highway layout, the vehicle dynamics models, the merge and cruise protocol and the actual simulation runs) in less that three work weeks.

The merge simulation case study is being followed by more complex studies regarding

- emissions evaluation,
- coordination protocols involving radio communications,
- "platooning" of cars on automated highways, and
- detailed physical simulation of crash and "near-miss" situations for safety analysis.

[9]However, the slow down is due mainly to the current implementation of sets in SHIFT. New tests will be performed with a new implementation which will improve on the memory usage of the internal data structures.

[10]Mostly, I/O routines for the uploading of the engine model data.

[11]Which includes all the necessary development hooks, GUI hooks, C API and Foreign Function Interface.

```
type Controller
{
    output
        continuous number acceleration;
    ...
    state
        Vehicle the_vehicle, side_lane_vehicle;
        continuous number same_lane_accel;
        number nominal_speed;
    ...
    discrete
        cruise{cruise_law};
        yield {yield_law};
    ...
    flow
        default {
            same_lane_accel =
                track_acceleration(same_lane_rel_speed, xDot(the_vehicle), nominal_speed);
        };
        cruise_law {
            acceleration = same_lane_accel;
        };
        yield_law {
            acceleration = min(same_lane_accel, side_lane_accel)
        };
    ...
    transition
        cruise -> yield
            when rear_gxp(the_vehicle) >= L_gap_visible_range(junction)
                and exists mp in merging_vehicles(junction) :
                    ((rear_gxp(mp) >= front_gxp(the_vehicle))
                    and (rear_gxp(mp) <= front_gxp(the_vehicle) + lateral_sensor_range)
                    and (xDot(the_vehicle) <= xDot(mp) + yield_rel_speed_threshold))
            do {
                side_lane_vehicle := mp;
            },
    ...
}
```

Figure 3: A fragment of the *Controller* code. The fragment shows the transition that takes the (instance of) the *Controller* of *the_vehicle* from the *cruise* to the *yield* state. The condition upon which this transition is allowed is expressed in the when clause. The variables suffixed by *_gxp* and *_gyp* represent positions. The accessor *xDot(the_vehicle)* is integrated to the current speed of the vehicle.
When the transition has taken place, the integration will use a modified set of equations to produce the value for the *acceleration* parameter, whose computation eventually relies on a C function (*track_acceleration*).

These projects simply extend the framework or change the simulation granularity, confirming our claim that the level of abstraction provided by SHIFT and SMART AHS is the proper one. Other SHIFT applications developed within UCB Mechanical Engineering and Electrical Engineering departments also confirm that the model/simulate/analyze cycle improves considerably when compared to more traditional approaches applied to similar problems.

3.2.1 Preliminary Cost Analysis

The overall SHIFT design and development took about 18 months for a core group, averaging six people (though the complete list of people who actually contributed is much longer). The first version of SHIFT became available in September of 1996 and it did not include many of the features that were introduced later, during the winter of 1996/97. Currently there are five projects directly funded by either PATH or the NAHSC that are using SHIFT and SMART AHS. These projects directly involve about 20 people for the development and the interpretation of the simulation results. New projects will be added to this list as the FY 98, as the NAHSC expands and redirects its efforts.

The cost for the Houston case study turned out to be in the order of 3 men/month. Subsequent projects (emission control simulation, platooning and coordination) reused much of the overall structure developed in the first place for the Houston Case Study and showed a faster turnaround of the simulation results.

SHIFT is used in other application domains such as autonomous underwater vehicles, and air traffic management simulations. However, our group has not undertaken a formal project tracking effort in order to evaluate the overall impact of the technology outside California PATH.

4 Conclusion

In this paper we presented SHIFT: a new programming language based on theoretic concepts emerging from the field of *hybrid systems*. We have claimed that SHIFT offers the proper level of abstraction for describing complex applications such as automated highway systems, air traffic control systems, robotic shop floors, coordinated submarines and other systems whose operation cannot be captured easily by conventional models.

To support our claim we have described our experience with the SMART AHS framework for the simulation of complex highway systems. Our experience indicates that SHIFT and SMART AHS do achieve the objectives that were at the base of its design.

In particular, SHIFT is currently enjoying a growing popularity and is being used as a teaching tool in various courses in the Electrical Engineer Department of UC Berkeley.

Future work on SHIFT will include the following items:

- further research on the interaction between the integration and guard crossing algorithms;
- parallelization and distribution of the run-time system;
- integration with automated verification systems such as KRONOS [6].

As already mentioned, at this point we cannot provide a direct comparative study of the "simulation development costs" for SHIFT and SMART AHS with respect to a more traditional approach based on standardized libraries. Setting up such a study would require a considerable effort in itself and the identification of a proper set of tools to compare SHIFT and SMART AHS against. However, the feedback we gathered from the users of SHIFT makes us very confident that the results would tip the balance in its direction.

5 Acknowledgments

We wish to thank all the people at California PATH and elsewhere, especially A. Deshpande, F. Eskafi, A. Girault, M. Kourjanski, J. Misener, V. Murgier, L. Semenzato, J. Sousa, P. Varaiya, D. Weismann, S. Yovine, and the National Automated Highway System Consortium

6 Availability

Of course, SHIFT and SMART AHS can be downloaded for free under a UCB-style license from our home pages

- `http://www.path.berkeley.edu/`
- `http://www.path.berkeley.edu/shift`
- `http://www.path.berkeley.edu/smart-ahs`

and our ftp site

- `ftp.path.berkeley.edu:pub/PATH/SHIFT`
- `ftp.path.berkeley.edu:pub/PATH/SMART-AHS`

References

[1] R. Alur, C. Courcoubetis, T. A. Henzinger, and P. Ho. Hybrid Automata: An Algorithmic Approach to the Specification and Verification of Hybrid Systems. In R. L. Grossman, A. Nerode, A. P.

Ravn, and H. Rischel, editors, *Hybrid Systems*, volume 736 of *Lecture Notes in Computer Science*, pages 209–229. Springer-Verlag, 1993.

[2] M. Antoniotti, A. Deshpande, and A. Girault. Microsimulation analysis of automated vehicles on multiple merge junction highways. In *Proceedings of the IEEE Conference on Systems, Man, and Cybernetics (SMC97)*. IEEE, October 1997.

[3] M. Antoniotti, A. Deshpande, and A. Girault. Microsimulation analysis of multiple merge junctions under autonomous ahs operation. In *Proceedings of the IEEE Conference on Intelligent Transportation Systems (ITSC97)*. IEEE, November 1997.

[4] F. Barros. Dynamic Structure Discrete Event System Specification Formalism. *Transactions of the Society for Computer Simulation*, 1:35–46, 1996.

[5] H. Boehm and M. Weiser. Garbage Collection in an Uncooperative Environment. *Software Practice and Experience*, pages 807–820, September 1988.

[6] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In *Hybrid Systems III, Verification and Control*, volume 1066 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[7] A. Deshpande, A. Göllü, and P. Varaiya. A Formalism and a Programming Language for Dynamic Networks of Hybrid Automata. In *Hybrid Systems IV*. Springer-Verlag, 1997.

[8] F. Eskafi, D. Khorramabadi, and P. Varaiya. An Automated Highway System Simulator. *Transportation Research Journal, part C*, 3(1), 1995.

[9] D. Harel. Statecharts: A Visual Approach To Complex Systems. *Science of Computer Programming*, 8(3):231–275, 1987.

[10] F. Maraninchi. The Argos Language: Graphical Representation of Automata and Description of Reactive Systems. In *Proceedings of the IEEE International Conference on Visual Languages*. IEEE, 1991.

[11] S. E. Mattson and M. Anderson. The Ideas Behind Omola. In *Proceedings of the IEEE Symposium on Computer Aided Control System Design, CADCS 1992*. IEEE, March 1992.

[12] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I and II. *Information and Computation*, 100(1):1–77, September 1992.

[13] H. Praehofer, F. Auernig, and G. Resinger. An Environment for DEVS-based Multiformalisms Simulation in Common Lisp/CLOS. *Discrete Event Dynamic Systems: Theory and Application*, 3(2):119–149, 1993.

[14] J. T. Schwartz, R. B. K. Dewar, E. Dubinsky, and E. Schonberg. *Programming with Sets. An Introduction to SETL*. Springer-Verlag, 1986.

[15] P. Varaiya. Smart Cars on Smart Roads: Problems of Control. *IEEE Transactions on Automatic Control*, 38(2):195–207, February 1993.

[16] B. Zeigler. *Multifaceted Modeling and Discrete Event Simulation*. Academic Press, London, Orlando, 1984.