UC Berkeley
Research Reports

**Title**
A General Framework for Verification, Simulation and Implementation of Real-Time Control Algorithms

**Permalink**
https://escholarship.org/uc/item/38z7n5vz

**Author**
Eskafi, Farokh H.

**Publication Date**
1999-05-01

eScholarship.org Powered by the California Digital Library
University of California

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

# A General Framework for Verification, Simulation and Implementation of Real Time Control Algorithms

**Farokh H. Eskafi**

**California PATH Research Report**
**UCB-ITS-PRR-99-15**

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

The contents of this report reflect the views of the authors who are responsible for the facts and the accuracy of the data presented herein. The contents do not necessarily reflect the official views or policies of the State of California. This report does not constitute a standard, specification, or regulation.

Report for MOU 333

May 1999

ISSN 1055-1425

CALIFORNIA PARTNERS FOR ADVANCED TRANSIT AND HIGHWAYS

# Final Report
# MOU 333
# A General Framework for Verification, Simulation, and Implementation of Real Time Control Algorithms

Farokh H. Eskafi
farokh@eecs.berkeley.edu

October 13, 1998

## 1 Executive Summary

We have proposed, investigated, and implemented a general framework for the simulation, verification, and prototyping of control algorithms for intelligent vehicles and highways. The immediate use of such framework will be in fault management project (MOU 288). Prior to this project the protocols and control algorithms developed under MOU 288 should have been manually verified, translated to a simulation language for simulation, and then modified for the QNX real-time operating system for porting to the vehicle's computer. This manual translation process is error prone at every stage. Our framework performs the translations automatically, and therefore, removes the possibility of the translation errors.

The proposed framework uses a coherent set of tools that model the system at hand; it takes a control design and verifies and simulates it; it generates code that can be executed on a target real-time software platform in the physical system. The specification of the control algorithms is performed in the SHIFT specification language. An existing verification platform is used to carry out the correctness proofs of the control algorithms. The QNX real-time operating system, which is currently in use at PATH in automated vehicles, is used as the target platform for the generated code.

To create the general framework, the following three tasks has been performed. In task 1, we specified the restrictions on the SHIFT specification that allows automatic verification and code generation for the real time software. In task 2 we have generated a parser that converts the SHIFT specification to an already available verification software tool. As part of this task we selected KRONOS from VERIMAG based on its suitability and availability of support. In task 3 we have generated a parser and code-generator from SHIFT to the QNX platform.

The short term goal of this proposal was to embed the SHIFT language in a general framework that from a single specification simulates, verifies, and generates real-time codes with minimal interventions from the user. The long term goal which will be followed in another funded proposal is to do the above with *no* intervention.

In the new proposal, the fault management control algorithms developed by MOU 288 is used to validate the framework. These algorithms are simulated and verified in the framework and experiments will be run on vehicles with the generated code.

Our research and implementation effort is presented in the forms of a conference article, a research paper, users manuals, and release of the software implementations under the SmartAHS software release.

## 2 Introduction

Large engineering systems, such as automated highway systems (AHS), autonomous vehicle systems (AVS), material handling systems, and air traffic management systems (ATMS) face the challenge of providing reliable services using scarce resources. Clients of such systems demand performance, safety, comfort, and efficiency.

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

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

Complex software applications are needed to specify, simulate, evaluate, and manage the behavior of such large scale systems. Currently there are no coherent software tools that can facilitate large scale system development from concept inception to actual deployment. There is a gap between the specification and implementation constructs required to build such systems on the one hand, and the interfaces provided by software design tools and programming languages on the other hand.

In this report we describe a general framework for the modeling, design, simulation, verification , and prototyping of large scale systems. The framework uses a coherent set of tools that model the system at hand, analyze, verify and simulate a control design, and generate code that can be run in a target real-time software platform in the physical system.

The general framework as well as the tools that implement it have evolved at PATH (Partners for Advanced Transit and Highways) over the last seven years as large number of researchers have worked on Intelligent Transportation Systems, AHS and AVS.

Our simulation tools have been evolving through the years. Since the systems we were working on were inherently complex the need for simulation was obvious and we have developed C and C++ based simulation frameworks to evaluate PATH's and other organizations' proposals for highway automation [1, 2]. In parallel to our AHS work, we were involved with several other projects, such as ATMS, 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 systems. No language, product, or tool in the market nor in academia came close to satisfying all requirements. Many simulation frameworks are available that consist of a set of class libraries developed in a programming language such as C++. These frameworks impose semantic

notions such as inputs, outputs, events, differential equations etc. onto the C++ syntax and expect the user to follow framework rules for using the class libraries. This approach does not provide the user with any syntactic support for large scale system development. Several discrete event simulation tools exist. However, these tools do not provide enough support for continuous evolution. Block-diagram based simulation tools are easy to use, but do not provide the necessary expressive power to represent dynamic interaction patterns among the simulated objects.

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.

The SHIFT formalism [3, 4] which we briefly discuss in Section 4.1 is the first programming language with well defined simulation semantics that addressed our requirements. It combines system-theoretic concepts into one consistent and uniform programming language with object-oriented features. SHIFT is ideal for the design, specification, simulation, control, and evaluation of large dynamical systems that consist of multiple interacting agents whose behavior are described by state machines and ordinary differential equations. SHIFT's strength lies in the ability of instantiating agents and evolving the interaction network among them at run-time as part of the simulation.

Our work on the general framework has been more recent. As the control designs for automated vehicles matured we started in-vehicle prototype experiments. At this stage it became essential to streamline the translation of the control law specifications that had been simulated and verified into in-vehicle real-time control instructions.

This report is organized as follows. In Section 3 we discuss the framework and the general methodology for large scale system development. In Section 3 we summarize requirements for simulation frameworks and describe the SHIFT language that is used as the specification and simulation language of the overall framework. In Section 4 we describe our approach to the interface between the SHIFT specification language and KRONOS, a software tool for verification of hybrid automata. In Section 5 we describe the real-time code generation from SHIFT specification. Section 6 has the conclusions.

# 3 General Framework for Large Scale System Design

The overall methodology we use for large scale system design, prototype, and deployment is depicted in Figure 1.

Large scale system development goes through several stages. The first stage typically consists of feasibility studies that lead to models and designs on paper. The second stage broadens the scope with prototype experiments.

## 3.1 Stage One

Any large scale system design task requires the specification of an overall architecture within which controllers can be designed to coordinate the system. The architecture design decomposes the control design problem into the control of sub-systems.

The highway automation architecture of PATH is discussed in [5, 6]. An architecture for Air Traffic Control is proposed in [7], and an architecture for Autonomous Underwater Vehicles is proposed in [8].

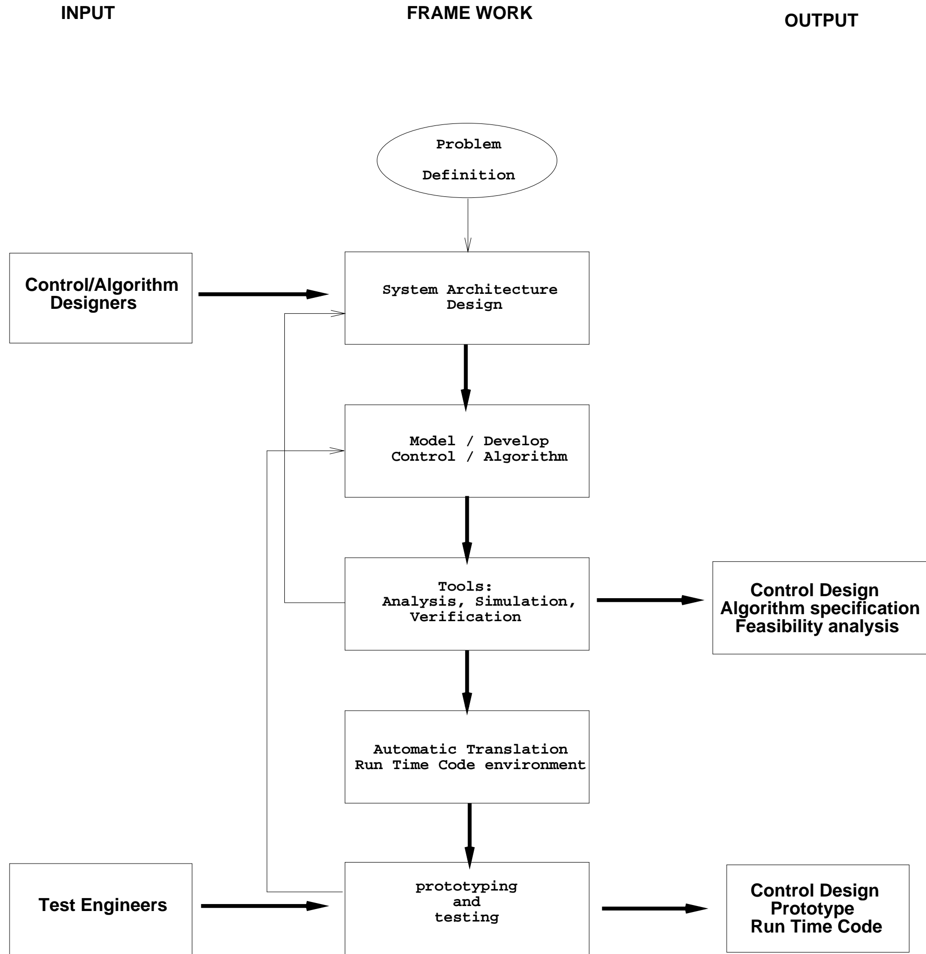


Figure 1: A General Framework for System Development

Analysis, simulation, and verification tools are needed within which these tasks can be carried out. Verification tools provide guarantee conditions for specific designs. In general, verification is possible only with very high level abstractions or very isolated subsystems. Analysis tools are useful for high level models for which closed form solutions can be obtained. Simulation is needed to see the detailed performance of a design.

These tools are used in successive iterations until the first stage starts delivering satisfactory architecture and control designs.

### 3.2 Stage Two

As the confidence in the system architecture and control designs grows the second stage starts which involves physical prototype experiments.

At this stage it is essential to leverage the control specifications of the earlier stage into actual code that can be used in physical experiments. This requires the implementation of a real-time control environment in the actual devices that can execute control instructions. Such a run-time environment must provide the necessary abstractions to interface with sensory and communication inputs and outputs to actuators.

Once a real-time control environment is implemented, it becomes possible to automati-

cally translate specifications from the design stage into the target physical prototype system. Such automatic translation increases efficiency of experiments and prevents loss of information between the two stages.

Physical experiments may require further refinement of the control designs or the overall architecture.

# 4 Simulation Framework

Frameworks shield their users from software implementation details and allow them to concentrate on their particular specification or evaluation task. In [9] the requirements for simulation frameworks were discussed in detail.

These frameworks must address the needs of several categories of users who use it in successive stages. System engineers develop automation architectures, control and communication engineers design, implement, and test individual control algorithms; system analysts test and evaluate the overall system; and system planners select the automation strategy for deployment based on evaluation results.

In addition to traditional software engineering requirements, these frameworks must allow the designers to use a specification language that fits their domain, in this case differential equations and finite state machines; they must provide a structured specification, simulation, and evaluation environment with formal semantics; they must represent dynamic interaction dependencies among components in the system; and last but not least they must provide models of entities that are particular to the application domain.

Our experience with generations of simulation frameworks that we have implemented [1, 2] lead us to the conclusion that simulation frameworks are best implemented in a programming language with explicit simulation syntax and semantics.

Simulation frameworks developed in a language like C or C++ impose semantic notions such as inputs, outputs, events, differential equations etc. onto the C++ syntax and expect the user to follow framework rules for using the class libraries. This approach does not provide the user with any syntactic support for large scale system development. Block-diagram based simulation tools are easy to use, but do not provide the necessary expressive power to represent dynamic interaction patterns among the simulated objects.

To address these shortcomings we have developed the SHIFT language.

## 4.1 SHIFT Programming Language

SHIFT is a special-purpose object-oriented programming language designed to simulate large dynamical systems. It bridges the gap between system and control theory, formal methods, and programming languages for a focused yet large class of applications [3, 4].

SHIFT users define types (classes) with continuous and discrete behavior. A simulation starts with an initial set of components that are instantiations of these types. The world-evolution is derived from the behavior of these components.

The world evolves in a sequence of phases. During each phase, time flows while the configuration of the world remains fixed. In the transition between phases, time stops and the set of components in the world and their configurations are allowed to change.

The data model of a type consists of numerical variables, link variables, a set of discrete states, and a set of event labels. The variables are grouped into input, internal (state), and output variables. A type has read only access to its input variables and read/write access to

its internal and output variables. Types can access other types through their link variables. Such access is limited to write only access for inputs and to read only access for outputs.

In SHIFT, write access to a variable constitutes the ability to define the evolution of that variable. Read access constitutes the ability to use that variable in specifying the evolution of variables.

The data model supports inheritance. A subtype inherits the input and output variables and event labels of its parents. It may add new variables and event labels.

The behavior model is hybrid, i.e., the model has both continuous and discrete behaviors. Each discrete state has a set of differential equations and algebraic definitions that govern the continuous evolution of numeric state and output variables. These equations are based on all numeric variables of this type and outputs of other types accessible through link variables. The algebraic definitions cannot have cyclic dependencies.

The discrete behavior is given by a set of transitions among the discrete states. A transition is given by a from state, to state, a set of events, a guard, and reset actions. Events consist of event labels of this type (local events), and event labels on types accessible through link variables (external events). External events create a connection (synchronization) between transitions in different components and require concurrent execution of such transitions. Transitions are executed when guard conditions on variables and synchronization requirements on events hold. When a transition is executed numerical and link variable values may be changed and new components can be created as part of the reset actions.

A type can establish input output connections among variables of types accessible through its link variables. Alternatively, a type can provide algebraic or constant definitions for other types' inputs.

At this time the behavior model does not support inheritance.

Components evolve in time according to their continuous behavior rules until a discrete transition becomes possible. At that point the discrete transition is executed in zero time. Several transitions can be executed before time passage resumes.

Under the current implementation SHIFT programs are translated into C code and linked with SHIFT run-time libraries to create an executable. SHIFT programs can link in C functions. The run-time executable supports programmatic, command-line, and graphical interfaces for user interaction. For further information, we refer the interested parties to [11].

## 5 Verification Framework

Undoubtly, safety is the most important factor in developing software and hardware for critical systems such as automated highways and vehicles. The controllers used to execute commands and maneuvers (such as lane changes, merging, and exiting) must be completely reliable, since errors could lead to lethal accidents. Though other factors, such as performance and efficiency, can be measured quite accurately by analysing some typical behaviors of the system, ensuring safety requires all the possible trajectories to be studied.

In general, the task of showing that the system meets the safety criteria cannot be achieved by simulation, and we have to use formal verification methods. These methods provide means to verify that a mathematical model of the system has some formally stated property.

In our framework, the language SHIFT is the formalism used for describing system models. SHIFT has a formally defined semantics, that is, to every SHIFT program corresponds a mathematical object that characterizes all the possible behaviors of the program. More precisely, such object is a network of *hybrid automata* [12].

The verification problem is too hard to be solved in a fully automatic way, decision and semi-decision algorithms have been devised for useful subclasses of hybrid automata. The development of algorithms have been followed by the implementation of verification tools. KRONOS [13] is one such tool. [1]

### 5.1 The verification tool

KRONOS is a software tool developed at VERIMAG in Grenoble, France. One major design objective of KRONOS is to provide a verification engine that can be integrated into modeling frameworks of the real-time systems in a wide range of application areas. Real-time communication protocols, timed asynchronous circuits, and hybrid systems are domains where KRONOS is currently being used.

The mathematical foundations of KRONOS come from the theory of timed automata. This formalism is a special case of hybrid automata where the behavior of a continuous variable $x$ is defined by the simple differential equation $x' = 1$. Such variables, called *clocks*, are used to measure the time elapsed between events. Timing constraints such as propagation delays, execution times and response times, are expressed as predicates on the values of the clocks.

The state-space defined by a timed automaton is infinite because clocks are real-valued variables. However, the state-space can be symbolically characterized by a finite disjunction of linear inequalities on the clocks. This exact symbolic characterization of the state-space is the most important property of the formalism. It makes the verification problem to be tractable not only in theory, but also in practice.

### 5.2 Verification methods

The verification methods provided by KRONOS can be classified in two general categories according to the strategy applied to explore the state-space: backward analysis and forward analysis.

Backward-analysis methods are based on algorithms that perform a backward search of the reachability graph of the model to compute the set of predecessors of a given set of states. Given the set of *unsafe* states (e.g., those characterizing a rear-end collision), these algorithms compute the initial conditions that potentially lead to such an undesired situation. A state not in the computed set is guaranteed to never reach an unsafe condition. Thus, these methods can be used to synthesize initial conditions that are proved to maintain safety.

Forward-analysis methods rely on algorithms that construct the set of successors by performing a forward exploration of the reachability graph. These algorithms analyze all the possible evolutions of the system starting at a given set of initial conditions. The safety requirement is violated if an unsafe state is encountered during the search; otherwise, it is fulfilled. When safety is not verified, these algorithms generate examples of violating evolutions which can be used to modify the original model.

### 5.3 Verification of SHIFT programs

In order to verify a SHIFT program with KRONOS, we translate it into the KRONOS input format. For this purpose, the latter has been extended to simplify the translation procedure between the two formalisms.

[1] KRONOS is freely distributed through the web for academic non-profit use. Please refer to the KRONOS home page `http://www.imag.fr/VERIMAG/PEOPLE/Sergio.Yovine/kronos/kronos.html.` to download the package or to know more about KRONOS.

The syntactic translation from one format to the other is a completely automatic procedure. However, not every SHIFT program can be automatically verified since the verification problem for general hybrid automata has been proven to be undecidable. Some restrictions need to be placed and some abstractions need to be envisaged.

### 5.3.1 Restrictions

For the time being, we restrict the verification to those SHIFT programs not involving the dynamic creation of objects and the use of SHIFT built-in primitives to deal with sets of objects. Ideally, the SHIFT program should consist of an initial object that creates, once and forall, all the objects of the system.

The main reason for preventing objects to be created while the system evolves is that the number of objects in the system may then grow without bound. KRONOS only supports verification of systems made up of a fixed number of components. Verification of systems consisting of an unbounded number of process is a very active area of research today, so we expect these restrictions could be removed in the future.

### 5.3.2 Abstractions

SHIFT programs may contain continuous variables with very complex dynamics. In general, an analytical (symbolic) solution of a differential equation cannot be computed by an algorithm. The analysis can be simplified by finding a tube of trajectories such that the solution of the differential equation lies inside the tube. For instance, it is possible to approximate a non-linear trajectory in the plane by the area defined by two piecewise linear trajectories lying below and above it. Such an approximation is safe, i.e., if no state in the tube is found to be unsafe, then the original trajectory satisfies the safety criteria. However, if a state in the tube is unsafe, it doesn't mean that the original trajectory is unsafe, it might be the case that the approximation is just too rough and needs to be reviewed.

One way of applying this approach is to partition the time line in a finite number of intervals $I_0, \ldots, I_n$ and replace the differential equation $x' = f(x,t)$ by a set of differential inclusions $x' \in g_k(x,t)$ for $k = 0, \ldots, n$, such that for all $k$, the differential inclusion contains all the solutions of the differential equation in the interval $I_k$. Under some (minor) assumptions, hybrid automata with differential inclusions of the form $x' \in [a,b]$, where $a$ and $b$ are rational constants, can be automatically analyzed with KRONOS [14].

## 5.4 Implementation

In order to be able to syntactically recognize those SHIFT programs that are indeed timed systems, we have extended the syntax of SHIFT. To avoid confusion, we call this "new" language SHIFT/KRONOS. The added features are indeed macro definitions. That is, they do not introduce new concepts into the language as SHIFT/KRONOS can be fully and syntactically translated into SHIFT.

The syntax SHIFT/KRONOS is the one of SHIFT, without any flow declarations and with the following additional statements:

- `kronosclocks x_1, ..., x_n` which corresponds to the SHIFT variable declaration `state continuous number x_1, ..., x_n`, plus the SHIFT flow declaration `x' = 1` associated with every discrete mode.

- `kronoswhen <cond>` where `<cond>` is a condition over the set of variables declared as `kronosclocks`. This statement corresponds to the SHIFT guard `when <cond>`. SHIFT guards are also allowed, which amounts of taking the conjunction of the conditions stated in the `when` and the `kronoswhen`.

- `kronosinvar <cond>` where `<cond>` is a condition over the set of variables declared as `kronosclocks`. This statement corresponds to the SHIFT invariant `invariant <cond>`. SHIFT invariants are also allowed, which amounts of taking the conjunction of the conditions stated in the `invariant` and the `kronosinvar`.

- `kronosreset { x_1 := v_1; ...; x_n := v_n; }`, where v_i is either 0 or another variable x_j. declared as a `kronosclock`. This statement corresponds to the SHIFT statement `do { x_1 := v_1; ...; x_n := v_n; }`.

Appendix 7 shows an example of SHIFT/KRONOS program: the Fischer's mutual-exclusion protocol. Appendix 7 shows the equivalent SHIFT specification of `type Fischer` (the rest of the program remains the same).

### 5.4.1 The compiler

Kish is the compiler that takes as input a SHIFT/KRONOS program and generates as output the files needed for the verification. Kish is actually a modification of the SHIFT compiler Shic. The additional work carried out by Kish with respect to Shic is the following.

- Type-checking required by the added syntactical features.
- Generation of the input files, called `Timed Automata`, to the verification tool KRONOS.
- Generation of C code that integrates the code generated by Shic for simulation purposes with code needed for verification.

### 5.4.2 The verifier

We have developed a verification tool called Grizzly that integrates both KRONOS and the SHIFT simulator. Grizzly is indeed the tool that explores all the possible behaviors of a SHIFT/KRONOS program. KRONOS provides all data-structures and associated manipulation functions required to store, update and check the consistency of the timing constraints. The run-time library of the SHIFT simulator is the one in charge of manipulating the corresponding SHIFT variables and dealing with the synchronization of transitions.

Grizzly essentially works as follows. Given a state, which is composed of a KRONOS and a SHIFT data-structures, it calls the appropriate functions of the SHIFT run-time library to construct all the possible successors of the state (i.e., the states reachable by all the possible outgoing transitions from the state) only taking into account the constraints imposed by the pure SHIFT statements (i.e., without considering the timing information). Then Grizzly calls the appropriate functions of KRONOS to check whether the transitions found in the previous step meet the timing constraints (i.e., the condition imposed by the `kronoswhen` is satisfied), in which case the transition is taken and a new state is created, otherwise the transition is not taken. Grizzly keeps repeating this procedure until all the states have been visited.

The algorithm described above generates all the possible states in which the SHIFT/KRONOS program may stay. We can also ask Grizzly to check whether some given state is indeed reachable from the initial state.

### 5.4.3 Example

We illustrate here using the example in Appendix A. a typical verification session. The shell-script called `kshift` should be used to generate the C code, the KRONOS timed automata and to compile and link all the files together with Grizzly. The result is an executable binary file named as the input file with the extension `.grz`. This file is the one to be executed to perform the verification.

For instance, we can use `fischer.grz` to check whether the component `mcs` (an instance of the type `MonitorCS`) reaches the discrete mode `bad`. To do so, we use the following command:

```
fischer.grz -REACH mcs_bad -DFS -h
```

The option `-REACH mcs_bad` indicates that we are looking whether the component `mcs` reaches the discrete state `bad`, whereas `-DFS` and `-h` are options provided by KRONOS that correspond respectively to perform the exploration using a depth-first search and to generate a trace if the state is reachable. The output is the following:

```
grizzly: Evaluating reachability: _init AND E<> _reach
grizzly: Using depth-first search with max stack size: 1000.
grizzly: Max symbolic-states set size: 1000.
grizzly: Symbolic states visited: 9
grizzly: reachability successful
   f4   f3   f2   f1   mcs
0: idle idle idle idle good F4_X=F3_X and F4_X=F2_X and F4_X=F1_X
1: idle idle idle wait good F4_X<=10 and F4_X=F3_X and F4_X=F2_X and F1_X<=F4_X
2: idle idle wait wait good F4_X<=10 and F4_X=F3_X and F4_X=F2_X and F1_X<=F4_X
3: idle idle wait test good F4_X<=10 and F4_X=F3_X and F4_X=F2_X and F1_X+1<F4_X
4: idle idle wait cs   good F4_X<=10 and F4_X=F3_X and F1_X<=F4_X and F2_X<=F1_X
5: idle idle test cs   good F4_X<=10 and F4_X=F3_X and F2_X<=F4_X and F1_X<=F2_X
6: idle idle cs   cs   bad  F4_X<=10 and F4_X=F3_X and F2_X<=F4_X and F1_X+1<F2_X
```

### 5.4.4 Practical experiments

We have used Grizzly to verify a distributed fault-diagnosis protocol. The tool has demonstrated to be very useful, specially for finding bugs (mainly deadlocks due to synchronization problems) in early phases of the design. The final specification of the protocol consisted of 2500 lines of SHIFT code. We have formally verified two properties:

- *bounded-time fault detection* : if a fault happens, the fault is eventually detected within some time, and
- *absence of false alarms* : if a fault is diagnosed, then the fault have occurred.

The verification has been carried out for a platoon consisting of 4 cars: leader, second, middle and last, corresponding to the 4 possible modes of the protocol (i.e., extra cars in the platoon will behave as the middle one).

## 6 Real-Time Code Generation

We describe here the work carried out concerning the generation of executable code from SHIFT programs. We discuss here the main underlying ideas and the prototype implementation.

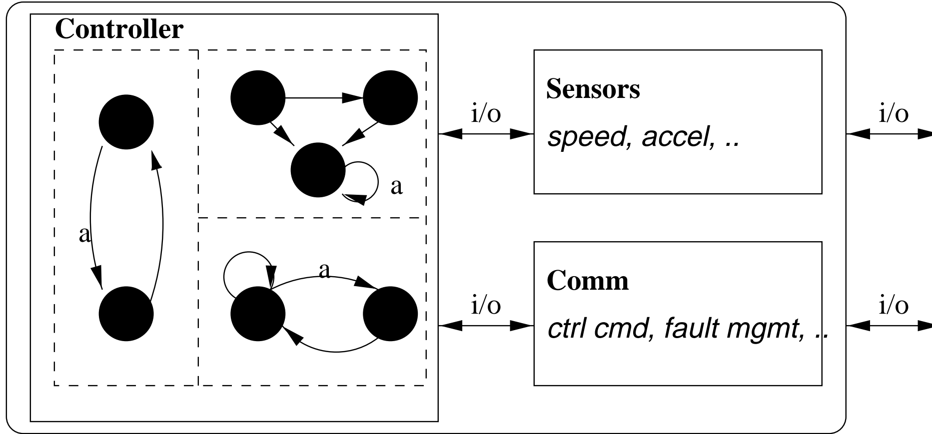


Figure 2: Structure of a SHIFT application.

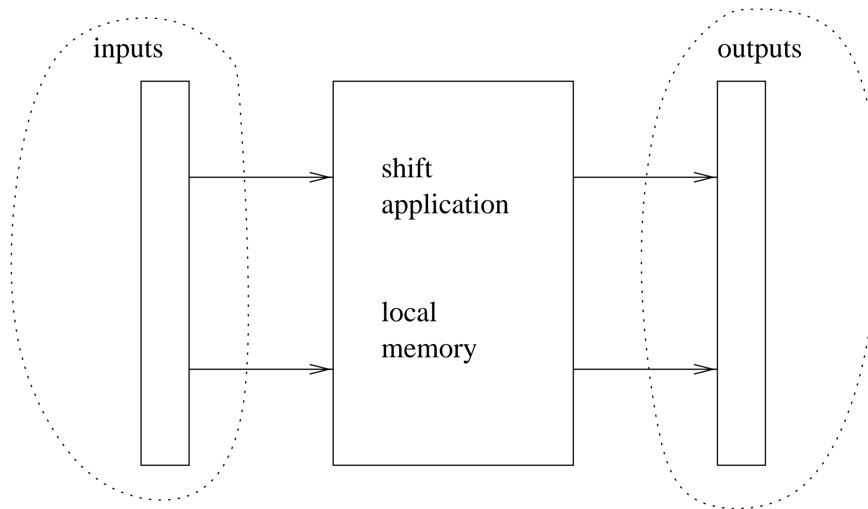


Figure 3: Architecture of the executor.

We refer to a Shift application as a collection of a fixed number of Shift components. The external interface of the application is defined to be the set of inputs and outputs of the components. The components are allowed to synchronize with each other through events, but they are only allowed to interact with the external world using the external interface. Such an application is therefore open in the sense that its behavior depends on externally provided inputs. (See Fig.2).

Given a Shift application, we are interested in generating code to be executed on a real-time operating system (QNX, for instance). We think of a Shift application as a single task, i.e., the application itself, rather than as a collection of tasks, i.e., one task per component. With this in mind, generating code for the application consists in implementing a program called "Shift executor" (Fig.3) which is going to execute the Shift application more or less is the same way that the simulator does.

## 6.1 Benefits of the approach

This approach allows us to avoid directly implementing a protocol for synchronizing the SHIFT components on top of the operating system. The synchronization between components inside the application is ensured by the SHIFT executor using the same algorithm implemented in the simulator. Another advantage of this approach is that the execution of the SHIFT application on the real-time operating system will be a single thread instead of the interleaving of multiple threads which results when implementing each component as a separate task. Besides, the granularity of the execution time can be set up to be equal to the time increment used for the simulation. Thus, if the executor is scheduled in time, it will reproduce the same behavior observed during the simulation, provided that the time needed to execute the code is smaller than the simulation time-step (in other words, if the code can be simulated in real time). This property can be checked during the simulation.

## 6.2 Extension of the syntax

Typically, for the purpose of the simulation, the "external world" is also model as a collection of SHIFT components. Such components are likely to be replaced by sensors and actuators when moving from the specification to the implementation. In order to identify such components we propose to extend the syntax of SHIFT. For historic reasons, we call such components *real-time components*.

The main issue regarding real-time components is that we have to insure that the real device that implements the component provides the same functionality as defined by the SHIFT model of the component. Clearly, the actual functionality of the device may vary from one manufacturer to another and it is impossible to forsee all the possible implementations and to generate code for all of them.

In order to overcome this problem, we propose to add to the type declaration a section for implementation-dependent (or device-dependent) information. This section consists in a list of assignments. The left-hand side of the assignment is an *output* variable. The right-hand side is an *external function call*. There must be one and only one assignment for each output.

To define an implementation-dependent behavior, we extend SHIFT with the `rtimplementation` declaration within a type descriptor. The syntax is the following:

| | | |
|---|---|---|
| *rtimplementationdecl* | ::= | `rtimplementation` *rtimplementation*$^*$ |
| *rtimplementation* | ::= | *rtid* { rtassign$^*$} |
| *rtassign* | ::= | *var* `:=` *externalfunctioncall* |

The compiler checks that *var* is an output variable and that all and only the output variables get assigned. The idea here is that the right-hand side is the function (external to the SHIFT specification) that provides the value of the corresponding output variable. There can be more than one `rtimplementation` declarations.

When generating code for execution, we need to distinguish which components are going to be compiled as is and which are going to be replaced by an implementation-dependent device. If a component is left as is, then nothing needs to be done. To generating code for implementation-dependent components we need a mechanism that allows the programmer to choose the adequate implementation. To do so, we extend SHIFT with the `rtcreate` operator which is esentially the same as the `create` except that it only accepts as parameter the identifier associated with an `rtimplementation` declaration within the type. The `rtcreate` has

the following syntax:

$$rtcreateexpr \quad ::= \quad \texttt{rtcreate ( } type \texttt{ , } rtid \texttt{ )}$$

where *type* must be the name of a type and *rtid* must be the name of an `rtimplemenation` declaration within that type. The components created with `rtcreate` are called *rt-components*.

Appendix B. shows an example of the use of the `rtimplementation` and the `rtcreate` mechanisms.

### 6.3 Extension of the SHIFT compiler and simulator

We have extended the SHIFT compiler Shic to accept the new syntactic constructs. Furthermore, we have modified the SHIFT simulator in such a way that implementation-dependent components can be fully integrated into the simulation. This allows the designer to test and debug the code to be executed on-board off-line. This is because in our approach, the code to be executed and the one used for the purposes of the simulation are the same.

The modified simulator works as follows. Before each discrete-continuous step, it goes through the list of all the rt-components and executes the code correspondingto the `rtimplementation` declaration. All the output variables are updated according to the values returned by the external functions called. These output variables are indeed inputs to other components. Once this step is completed, the simulator proceeds in the usual way.

The main loop of the simulator is synchronized with real time. For the system to be able to run in real time it is required that without the timing adjustment the computations run faster than real time. After each pass through the main loop the simulator kernel has to wait for the timer signal to continue. The time in between the signals is used to process discrete events which according to the model occur in zero time. For the remaining difference between simulated and real time the simulation kernel remains in the wait state. There are major factors which affect the accuracy of the simulation with respect to the pace of real time. One - the simulation time step which normally advances virtual time ahead the real time in a jump-like manner. This difference is in the order of 20-100 milliseconds, depending on the hardware platform and the integration step setting. Another parameter is the clock granularity supported by the computer clock. That granularity is in the order of microseconds or less. The accuracy of the timing subroutine is achieved by recording time stamps at the beginning of execution and adjusting the pace of virtual time to that of real time. Provided that the hardware platform of choice has enough computing power, the accuracy of the real-time system stays within the accuracy of the computer clock plus or minus the value of integration step.

### 6.4 Real Time SHIFT for Solaris and QNX operating systems

Currently the real-time SHIFT system is available for two operating systems: Solaris and QNX.

For Solaris it is a simple extension of the basic SHIFT package. It required an addition of the Posix-4 compatibility library and otherwise relies solely on the Solaris standard library functions and system calls.

Porting real-time SHIFT for QNX is more daunting. Since many of the support libraries used for regular SHIFT are not readily available for the QNX environment, a working prototype only was developed by the time of this report. This prototype includes bare bones run-time system without debugger and garbage collection modules.

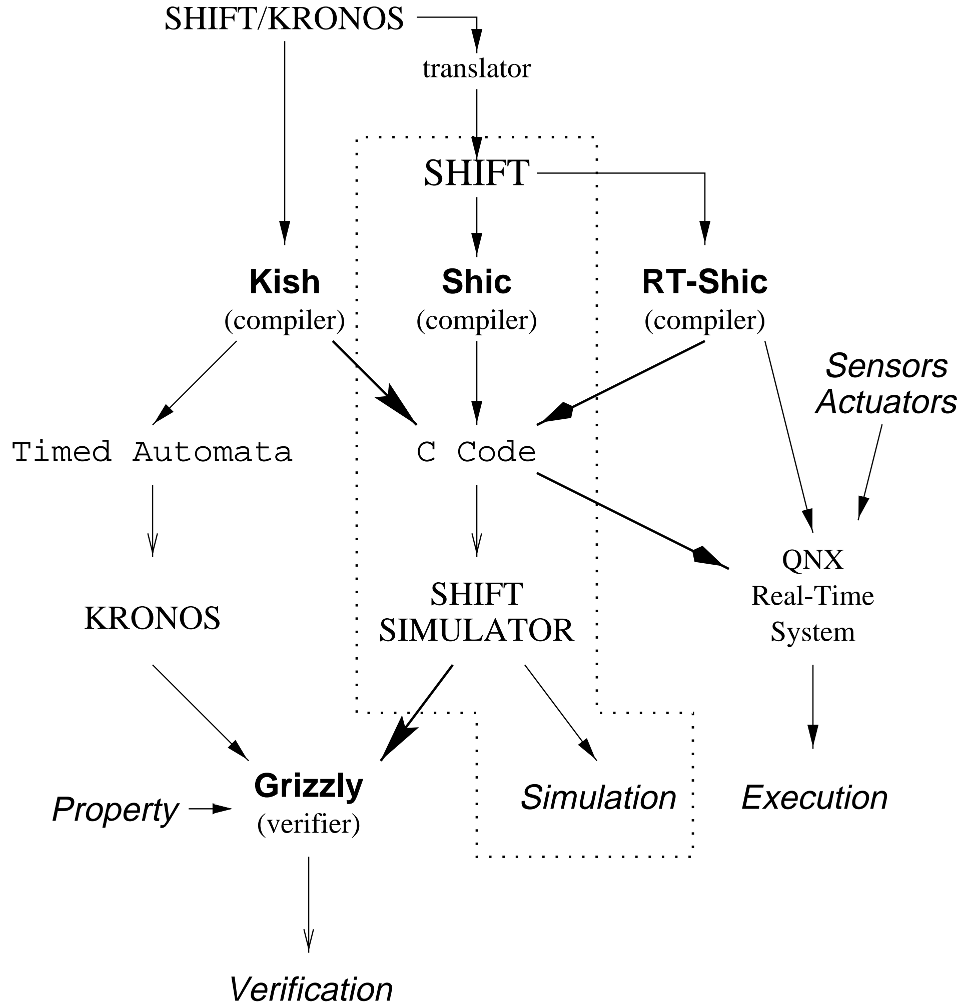


Figure 4: Simulation, verification and execution of SHIFT programs.

## 7 Conclusion and Current Status

We have presented a general framework for modeling, simulation, verification, and prototyping of controllers in a large scale system. We have illustrated the framework for the automated highway system application. Other similar systems that are using this approach include ATMS, autonomous underwater vehicles, and wireless communication systems.

Fig.4 shows the tool support and the functional diagram for verification and code generation.

## References

[1] Farokh H. Eskafi. " Modeling and Simulation of the Automated Highway System", *PhD Thesis*, UC Berkeley 1996. Also Path Report UCB-ITS-PRR-96-19.

[2] Aleks O. Göllü. "Object Management Systems", *PhD Thesis*, UC Berkeley 1995. Also Path Report UCB-ITS-PRR-95-19.

[3] A. Deshpande, A. Göllü, and L. Semenzato. "The SHIFT Programming Language and Run-time System for Dynamic Networks of Hybrid Automata". California PATH Technical Report UCB-ITS-PRR-97-7.

[4] A. Deshpande, A. Göllü, and L. Semenzato. "SHIFT Reference Manual", California PATH Technical Report UCB-ITS-PRR-97-8.

[5] P. Varaiya and S. Shladover. "Sketch of an IVHS systems architecture", *Proceedings of the Vehicle Navigation and Information Systems Conference*pp. 1117-1124, Oct. 20-23, 1991.

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

[7] Tak-Kuen Juhn Koo, Yi Ma, George J. Pappas and Claire Tomlin. "SmartATMS: A Simulator for Air Traffic Management Systems" Submitted to Winter Simulation Conference 1997.

[8] Joao Sousa and Aleks Göllü. "A Simulation Environment of the Coordinated Operation of Multiple Autonomous Underwater Vehicles" Submitted to Winter Simulation Conference 1997.

[9] F. Eskafi and A. Göllü. "Simulation Requirements and Methodologies in Automated Highway Planning", to appear in *TRANSACTIONS of the Society for Computer Simulation.*

[10] A. Deshpande. "AHS components in SHIFT", *PATH technical report*, 1996.

[11] SHIFT URL address: `http://www.path.berkeley.edu/shift`.

[12] R. Alur, C. Courcoubetis, N. Halbwachs, T. Henzinger, P. Ho, X. Nicollin, A. Olivero, J. Sifakis and S. Yovine. The Algorithmic Analysis of Hybrid Systems. *Theoretical Computer Science*, 138:3–4, 1995.

[13] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In *Hybrid Systems III, Verification and Control*, pages 208–219. Lecture Notes in Computer Science 1066, Springer-Verlag, 1996.

[14] A. Olivero, J. Sifakis, and S. Yovine. Using abstractions for the verification of linear hybrid systems. In D. Dill, editor, *Proc. 6th Computer-Aided Verification*, pages 81–94, California, June 1994. Lecture Notes in Computer Science 818, Springer-Verlag.

## Appenidx A. Fischer's protocol in SHIFT/KRONOS

```
type Fischer
{
state
        number id ;
kronosclocks
        x ;
discrete
```

```
        idle ,
        wait kronosinvar x<=1 ,
        test ,
        cs ;
export
        closed enter_cs ;
transition
        idle -> wait {}
        when N = 0
        kronosreset { x:=0 ; } ;
transition
        wait -> test {}
        when true
        do { N := id ; }
        kronoswhen x<=1
        kronosreset { x:=0 ; } ;
transition
        test -> cs { enter_cs }
        when N = id
        kronoswhen x>1
        kronosreset { x:=0 ; } ;
}

type MonitorCS
{
state
        number c := 0 ;
discrete
        good,
        bad ;
transition
        good -> good { S:enter_cs(one) }
        when c = 0
        do {
                c := 1 ;
        } ;
transition
        good -> bad { S:enter_cs(one) }
        when c = 1 ;
}

global Fischer f4 := create(Fischer, id := 4) ;
global Fischer f3 := create(Fischer, id := 3) ;
global Fischer f2 := create(Fischer, id := 2) ;
global Fischer f1 := create(Fischer, id := 1) ;
global number N := 0 ;
global set(Fischer) S := { f1, f2, f3, f4 } ;
global MonitorCS mcs := create(MonitorCS) ;
```

## Fischer's protocol in Shift

```
type Fischer
{
state
        number id ;
state
        continuous number x ;
flow
        kronosflow { x' = 1 ; } ;
discrete
        idle { kronosflow },
        wait { kronosflow } invariant x<=1 ,
        test { kronosflow },
        cs   { kronosflow };
export
        closed enter_cs ;
transition
        idle -> wait {}
        when N = 0
        do { x := 0 ; } ;
transition
        wait -> test {}
        when x<=1
        do { N := id ;
             x := 0 ;
        } ;
transition
        test -> cs { enter_cs }
        when N = id and x>1
        do { x := 0 ; } ;
}
```

## Appendix B. Implementation-dependent declarations

```
function xrandom() -> number
function yrandom() -> number
function xoutput(number a) -> number

type Car
{
     state
        Car car_in_front ;
```

```
     output
        continuous number x ;

     setup
        do
        {
                x := 1 ;
        } ;

     flow
        flow_1 {
                x' = 1 ;
        } ;

     discrete
        s1 { flow_1; } ,
        s2 { flow_1; } ;

     transition
        s1 -> s2 { }
        when x(car_in_front) = 1
        do
        {
                x := xoutput(x) ;
                x := 0 ;
        } ,

        s2 -> s1 { }
        when x >= 10
        do
        {
                x := 0 ;
        } ;

     rtimplementation
        impl_1 {
                x := xrandom() ;
        } ,
        impl_2 {
                x := yrandom() ;
        } ;

}

global Car d := rtcreate(Car, impl_1) ;
global Car c := create(Car, x:=0, car_in_front := d) ;
```