The SHIFT Programming Language and Run-time System for Dynamic Networks of Hybrid Automata

Akash Deshpande, Aleks Göllü and Luigi Semenzato*
{akash,gollu,luigi}@eecs.berkeley.edu
Department of Electrical Engineering and Computer Sciences
University of California at Berkeley, Berkeley, CA 94720

Abstract

SHIFT is a programming language for describing dynamic networks of hybrid automata. Such systems consist of components which can be created, interconnected and destroyed as the system evolves. Components exhibit hybrid behavior, consisting of continuous-time phases separated by discrete-event transitions. Components may evolve independently, or they may interact through their inputs, outputs and exported events. The interaction network itself may evolve.

We believe that the SHIFT model offers the proper level of abstraction for describing complex applications such as automated highway systems, air traffic control systems, robotic shopfloors, coordinated submarines and other systems whose operation cannot be captured easily by conventional models.

We have implemented a compiler for translating a SHIFT program to a C program, and we have implemented the SHIFT run-time system for obtaining an executable program. The executable program, when run, simulates the design specified in the SHIFT source program.

1 Introduction

SHIFT is a programming language for describing dynamic networks of hybrid automata. Such systems consist of components which can be created, interconnected and destroyed as the system evolves. Components exhibit hybrid behavior, consisting of continuous-time phases separated by discrete-event transitions. Components may evolve independently, or they may interact through their inputs, outputs and exported events. The interaction network itself may evolve.

The SHIFT model was motivated by our need for tools that support dynamically reconfigurable hybrid systems. Our primary application was the specification and analysis of different designs for automatic control of vehicles and highway systems. [21] We needed to capture the behavior of vehicles, sensors, actuators, communication devices and controllers in a structured and modular specification. [3, 4] Models for these components were to be provided by different groups of experts and then integrated in different ways for comparative analysis. From our previous experience in modeling [13, 7], analysis [18, 2, 10] and implementation [5, 8], we adopted the hybrid systems approach for modeling the system components. Since spatial relationships between vehicles change as they move, our application displayed a dynamically changing network of interactions between system components.

We investigated several system description frameworks [17, 11, 12, 6, 14, 16, 20, 22] and some of their real-time extensions, but found none that suited our needs sufficiently well. Most support only

*Currently at VXtreme, Inc., 701 Welch Road, Bldg C., Palo Alto, CA 94304. E-mail: luigi@vxtreme.com.

static configurations of system components and mainly provide discrete event behavior description with limited real-time extensions. However, we needed to describe models with switched differential equations (such as a vehicle with automatic gear shift) and coordinated behaviors (such as communicating controllers). Standard simulation tools such as Matlab or MatrixX, while suitable for numerical integration of fixed sets of differential equations, are difficult to use in applications with rapidly changing sets of differential equations (due to component creation and deletion), complex stopping conditions (such as existential queries on the state of the world), and complex program logic (such synchronous compositions of state machines).

The hybrid systems approach [1] satisfied our needs for component modeling but not for modeling dynamically reconfigurable interactions between components. Most general-purpose programming languages support dynamic reconfigurability. But the abstraction facilities in general-purpose programming languages such as C or C++ would not allow us to write simple, concise descriptions of our designs. SHIFT was designed to remedy this situation by providing both high-level system abstractions and the flexibility of programming languages.

The goal of SHIFT is to be easy to learn and use. We strived to keep the language design simple and small. SHIFT has only one number type (corresponding to the C `double` type), no functions (although it can use external functions written in C) and it has no memory management primitives, relying on garbage collection in its implementation. In spite of its simplicity, SHIFT programs are surprisingly powerful yet compact. This is because of the high-level system abstractions provided by SHIFT, including differential equations, state transitions and synchronous compositions, all within the framework of dynamic networks of hybrid automata.

While our immediate application concerned vehicles and highways, we believe that the SHIFT model offers the proper level of abstraction for describing complex applications in general, including air traffic control systems, robotic shopfloors, coordinated submarines and other systems whose operation cannot be captured easily by conventional models.

In section 2, we describe a simplified version of the SHIFT model. We discuss the models of a type, a component and the world and give the formal semantics of the model. In section 3, we describe the main features of the SHIFT language—states, inputs, outputs, differential equations and algebraic definitions, discrete states and state transitions—and provide two examples illustrating these features. In section 4, we describe the run-time environment for simulating SHIFT programs; in particular, we describe the algorithm for determining synchronized transitions.

2 The SHIFT Model

The model presented in this section is austere and simplified: the language provides facilities for structured programming. In the SHIFT model, the *world* W consists of a set of hybrid *components* $h_1, \ldots, h_w$:

$$W = \{h_1, \ldots, h_w\}.$$

Each component h is in a particular *configuration* C_h, and collectively these determine the configuration of the world:

$$C_W = (C_{h_1}, \ldots, C_{h_w}).$$

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.

Each component has both continuous-time dynamics and discrete-event dynamics which depend on the configuration of the world. Components obey continuous-time dynamics within each phase and discrete-event dynamics in phase transitions.

2.1 Type Description

The components are organized into *types* according to their prototypical behaviors. The type description mechanism in the SHIFT language permits the declaration of inputs, outputs and auxiliary state variables and provides several programming conveniences. In this section, we describe a simplified SHIFT model for the description of a type. In this simplified model, a type is the tuple

$$\begin{aligned} H \;=\; & (q \in Q,\ Q \text{ finite,—the discrete state variable,} \\ & x = (x_1, \ldots, x_n),\ \text{with each } x_i \in \mathbf{R}, \text{—the continuous state variables,} \\ & C = (C_0, \ldots, C_m),\ \text{with each } C_i \subset W, \text{—the configuration state variables} \\ & L = \{l_1, \ldots, l_p\}, \text{—the event labels} \\ & F = \{F_q \mid q \in Q\} \text{—the flows, and} \\ & T \text{—the transition prototypes.)} \end{aligned}$$

To be accurate, the discrete states of H should be referred to as Q_H, and similarly for the other elements in the type description. We drop the subscript where it is obvious from context.

We assume that the cardinality of C_0 is fixed, while for $i = 1, \ldots, m$, C_i may have a different number of elements in different phases. C_0 can be thought of as the set of single-valued references to other components, and each C_i, $i = 1, \ldots, m$ can be thought of as a set-valued reference to other components in the world. SHIFT provides constructs for structuring and accessing the configuration state C; we assume that these constructs have been "flattened" to yield C in the form shown here. SHIFT imposes type restrictions on the definition of these variables; we ignore these in the presentation of the model.

Each type is a prototypical hybrid automaton A_H with Q as its discrete states. In each discrete state $q \in Q$, F_q defines the flow of each continuous variable x_i either as a differential equation or as an algebraic definition. These definitions, respectively, take the form

$$\begin{aligned} \dot{x}_i &= F_{i,q}(x, x_{C_0}) \text{ or} \\ x_i &= F_{i,q}(x, x_{C_0}). \end{aligned}$$

Here, x_{C_0} denotes the vector obtained by listing the continuous variables of all elements of C_0. We require that there be no cyclic dependencies between algebraically defined variables.

T is a finite set of tuples of the form

$$\delta = (q, q', g, E, a)$$

where $q, q' \in Q$ are respectively the *from* and *to* states of δ, g is a guard predicate, E is a set of event labels and a is an action that alters the state of the world. The guard predicate takes one of two forms:

1. $g(x, x_{C_0})$ or
2. $\exists c \in C_i\ g(x, x_c, x_{C_0}),\ 1 \le i \le m.$

In the second form, multiple quantifiers are permitted (with obvious modifications of the form), and the quantifiers may be negated. The quantified variables may be used in the action a.[1] Informally, for δ to be taken, the guard must evaluate to true.

Each event label in E takes one of four forms:

[1] It may not always be meaningful to use a quantified variable in an action. For example, in a guard of the form $\nexists c \in C_1\ g(\ldots)$, c carries no meaningful information that can be used in the action.

1. l, with $l \in L$;

2. $c : l$, with $c \in C_0$ and l an event label in the type of c;

3. $\exists c \in C_i \; c : l$, with $1 \leq i \leq m$ and l an event label in the type of c (the quantified variable may be used in the action a); and

4. $\forall c \in C_i \; c : l$, with $1 \leq i \leq m$ and l an event label in the type of c.

The event labels place synchronization constraints on the world. Informally, a transition with the label $c : l$ can be taken if and only if c takes a transition with the label l. The constraint extends naturally to labels with quantified variables.

In Shift, an "internal" transition, that is a transition which does not synchronize with others, is specified by leaving E empty. As a mathematical convenience, we assume that such internal transitions are labeled ϵ.

The action a is a map from x, x_{C_0} and any variables in the guard or event labels to (x, C). It has the direct effect of resetting the continuous state and the configuration of the component. The action may also create new components and initialize them. In Shift, actions may also reset the inputs of components in C_0.

2.2 Component Description

A component h of type H assigns, at each time, values to its state variables (q, x, C). Associated to h is a hybrid automaton A_h derived from the prototypical automaton A_H using the values of C. The discrete states of A_h are the same as the discrete states of A_H—i.e., Q.

The transitions of A_h, denoted T_h, are obtained by transforming the transition prototypes of A_H into an equivalent set of simpler transitions. The transformation rules are given here for the purpose of explaining the model—they are not actually used in the implementation of the Shift run-time system.

- A transition prototype with a guard in the form $\exists c \in C \; g(x_h, x_c, x_{C_0})$ yields $|C|$ transitions, one for each component $c \in C$, with guards $g(x_h, x_c, x_{C_0})$.

- An event label l in the transition prototype is replaced by the event label $h : l$ in the transition.

- An event label $c : l$ in the transition prototype is retained without change in the transition.

- A transition prototype with an event label in the form $\exists c \in C \; c : l$ yields $|C|$ transitions, one for each $c \in C$, with event label $c : l$.

- A transition prototype with an event label in the form $\forall c \in C \; c : l$ yields one transition with $|C|$ event labels, one for each $c \in C$, labeled $c : l$.

A transition prototype with multiple existential quantifiers in the guards or event labels yields one transition for each element in the cross product of the domains of those variables.

Finally, we assume that T_h contains a trivial self-looping transition $(q, True, \emptyset, I, q)$ for each $q \in Q$, where I is the identity function. Such a transition is not actually specified in Shift; we introduce it as a mathematical convenience.

Let $events(A_h)$ be the set of all events appearing on the transitions of A_h:

$$events(A_h) = \cup_{t \in T_h} E_t.$$

2.3 World Description

Now we describe the hybrid automaton A_W associated to the world. The set of discrete states of A_W is the cross product $Q_W = Q_{h_1} \times \cdots \times Q_{h_w}$. Similarly, the set of continuous states of A_W is the cross product of the sets of continuous states of the components in W. The dynamics of the continuous state variables are governed by the flows defined in the component automata. The transitions T_W of A_W are tuples $\Delta = (\delta_1, \ldots, \delta_w)$ with each $\delta_i \in T_{h_i}$. Let

$$events(\Delta) = \cup_{\delta_i} E_{\delta_i}.$$

T_W satisfies the following condition: $\Delta \in T_W$ if and only if

1. $events(\Delta) \neq \emptyset$—i.e., not all transitions are trivial,
2. $events(\delta_i) = events(\Delta) \cap events(A_{h_i})$ for each i—i.e., events taken by each component are exactly the events taken by the world that are of interest to that component, and
3. $\nexists \Delta' \in T_W$ such that $events(\Delta')$ is a strict subset of $events(\Delta)$—i.e., Δ is minimal.

The guard associated to $\Delta \in T_W$ is the conjunction of the guards of the component transitions in Δ:

$$g_\Delta = \wedge_i g_{\delta_i}.$$

The action associated to Δ is the parallel execution of the actions of the component transitions in Δ—i.e., each map a_{δ_i} is evaluated using the state of the world just before the transition.

Our model performs synchronous composition of multiple automata. Our choice of this definition of world transitions rested on a tradeoff between ease of use and efficiency of implementation. An alternative would be to describe pair-wise cause-effect relationships between the transitions of different components. [9, 15] Programs written using the synchronous composition approach can be exponentially smaller than those written using cause-effect relationships. However, the algorithm for determining world transitions in the synchronous composition approach is exponentially more complex than the corresponding algorithm for the cause-effect synchronization approach. Thus, we preferred ease of use over efficiency of implementation. Cause-effect synchronizations can also be described in SHIFT using symbol-valued input-output connections.

2.4 World Semantics

Let

$$\tau = [\tau'_0, \tau_1], [\tau'_1, \tau_2], [\tau'_2, \tau_3], \ldots, \ \ \tau'_0 = 0 \text{ and } \forall i \ (\tau_i = \tau'_i \text{ and } \tau_{i+1} \geq \tau'_i)$$

be a finite or infinite sequence of intervals of $\mathbf{R}_+$ or an initial segment of it. Associated to each interval $[\tau'_i, \tau_{i+1}]$, is a world automaton $A_W(i)$. The semantics of the world are given over traces (τ, s) where

$$s = (q_0, x_0, \Delta_0), \ldots, (q_i, x_i, \Delta_i), \ldots$$

where $q_i \in Q_W(i)$, $x_i : [\tau'_i, \tau_{i+1}] \to \mathbf{R}^{n_W(i)}$ and $\Delta_i \in T_W(i)$.

A trace (τ, s) is a *run* of the world iff the following conditions hold.

1. Initialization. $A_W(0)$ is constituted from the appropriately initialized set of initial components in the world.[2]

[2]SHIFT provides a mechanism for initiating the world by creating and initializing components at the start of the execution of a SHIFT program.

2. Continuous evolution. For each i, $\forall t \in (\tau_i', \tau_{i+1})$,

$$\begin{aligned} \dot{x}_{j,h_k}(t) &= F_{(j,h_k),(q_i,h_k)}(x_{h_k}(t), x_{C_0(h_k)}(t)) \text{ if } x_{j,h_k} \text{ is differentially defined, and} \\ x_{j,h_k}(t) &= F_{(j,h_k),(q_i,h_k)}(x_{h_k}(t), x_{C_0(h_k)}(t)) \text{ if } x_{j,h_k} \text{ is algebraically defined.} \end{aligned}$$

3. Discrete evolution. At each boundary point $\tau_i = \tau_i'$

   (a) $q_i = q_{\delta_i}$ and $x(\tau_i)$ satisfies the guard g_{δ_i}—i.e., δ_i is enabled at τ_i, and

   (b) $q_{i+1} = q'_{\delta_i}$ and for each component k, $x_{h_k}(\tau_i') = a_{\delta_{i,k}}(x(\tau_i))$—i.e., the state of each component is reset according to that component's transition. The side-effects of each action are realized by creating new components as indicated by the actions and initializing them using $x(\tau_i)$.

   (c) $A_W(i+1)$ is constituted from the set of components in the world after δ_i is taken.

3 The SHIFT Language

This section describes the main features of the SHIFT language and relates them to the model in section 2. Sections 3.1 and 3.2 give examples illustrating the features of the SHIFT language. The SHIFT reference manual [19] describes the SHIFT syntax.

SHIFT provides two native data types—`number` and `symbol`, and it provides mechanisms for defining additional types. Variables of type `number` have piecewise constant or piecewise continuous real-valued time traces. The latter variables are declared to be of type `continuous number`. Variables of type `symbol` have piecewise constant symbol-valued time traces.

SHIFT provides three mechanisms for defining additional types: `set`, `array` and `type`. A variable of type `set(T)`, where `T` is a native or user-defined type, contains a set of elements of type `T`. A variable of type `array(T)` contains a one-dimensional array of elements of type `T`.

A component prototype is defined by the SHIFT `type` declaration. This is similar to the C `struct` declaration. The equivalent of C structure members are the `inputs`, `outputs`, and `states` of a SHIFT type. These are the continuous state variables x and the configuration variables C in the model. Thus one can write:

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

Dividing the continuous state from the model into inputs, outputs, and state in the language allows for more structured programming. SHIFT defines the circumstances in which variables of each kind may be used. For instance, state variables are not visible outside the component, and the input variables of a component may only be defined by another component.

A component type declaration also specifies the component's continuous and discrete behavior. This is done through additional syntactic constructs called *clauses*, each starting with a meaningful keyword.

The `discrete` clause defines the possible values for q (the component's discrete state variable) and associates a set of differential equations and algebraic definitions to each discrete state. For instance:

```
type Car {
  ...
  flow default {
    position' = velocity;
    velocity' = acceleration;
  }

  discrete
    accelerating { acceleration = 3; },
    cruising     { velocity = 30; },
    braking      { acceleration = -5; };
  ...
}
```

Groups of common equations can be given a name through the `flow` clause. The special flow name `default` defines the default behavior of a set of variables in all states. In the example, 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 states 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. Suppose that we wish the car to brake when a roadside controller signals an emergency. This can be specified with the transition

```
    cruising -> braking { controller:emergency }
```

The definition of the `Controller` type must include an exported event, emergency, and at least one transition that triggers it.

```
type Controller {
  ...
  export emergency;

  transition
```

```
    normal -> panic_mode { emergency }
      when <some critical condition>;
  ...
}
```

Some other clauses provided by SHIFT are `do`, `setup`, `global` and `function`. The `do` clause associates actions to transitions. The `setup` clause defines the component's initializations, input-output connections and externally defined event synchronizations. The `global` clause defines global variables and the `function` clause declares external functions.

SHIFT types may be organized in an inheritance hierarchy. A subtype is required to conform to its supertype's interface by declaring a superset of inputs, outputs and exported events.

The following examples illustrate the main features of SHIFT.

3.1 The Particle Example

In this example, a particle source creates particles at the rate of one per second and places them at the beginning of a 1000m track. The speed of the particles is initialized from the speed posted by the track monitor. The particles travel down the track and exit when they reach the end, notifying the track monitor of their exit. The track monitor updates its posted speed based on the exit speeds of particles. At externally provided time intervals, the monitor commands all particles between 500m and 1000m to raise their speeds by 10%. Figure 1 illustrates this example.

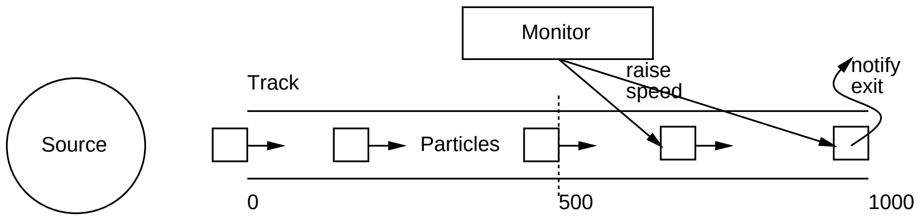

Figure 1: Particles-Source-Monitor

3.1.1 The Particle Type

The prototypical behavior of particles is shown in Figure 2. Each particle has a continuously varying

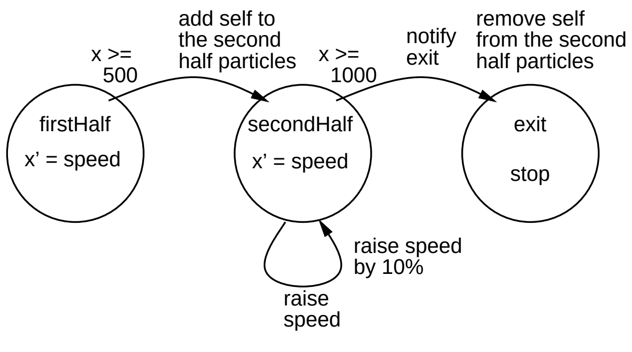

Figure 2: Particle

position along the track, given by the real-valued state variable x. Each particle has a real-valued piecewise constant output variable `speed`. The default flow of the particle is given by the equation

$$\dot{x} = speed.$$

The particle has three discrete states, `firstHalf`, `secondHalf` and `exit`, indicating that the particle is in `firstHalf` when $x < 500$, in `secondHalf` when $500 \leq x < 1000$ and in `exit` when $x \geq 1000$. This behavior is not automatic but must be programmed. The particle's flow in the `exit` state is `stop`, a Shift keyword, which assigns the rate 0 to all continuously changing variables. A "stopped" component that is unreachable and unable to participate in any world transition is, in effect, deleted from the world.

The particle exports two events, `raiseSpeed` and `notifyExit`, which may be used for synchronization by other components in the system.

The particle transitions from `firstHalf` to `secondHalf` on an internal event when $x >= 500$. As a part of this transition, it adds itself to the globally defined set `secondHalfParticles`. The particle loops in `secondHalf` on the event `raiseSpeed`, raising its speed by 10%. The particle transitions from `secondHalf` to `exit` on the event `notifyExit` when $x >= 1000$, removing itself from the set `secondHalfParticles`.

The Shift description of the particle type is given below.

```
type Particle
{
  state continuous number x;
  output number speed;
  flow default { x' = speed };
  discrete
    firstHalf,
    secondHalf,
    exit stop;
  export raiseSpeed, notifyExit;
  transition
    firstHalf -> secondHalf {}
      when x >= 500
      do {
        secondHalfParticles := secondHalfParticles + {self};
      };
    secondHalf -> secondHalf {raiseSpeed}
      do {
        speed := 1.1*speed;
      };
    secondHalf -> exit {notifyExit}
      when x >= 1000
      do {
        secondHalfParticles := secondHalfParticles - {self};
      };
}
```

3.1.2 The Source Type

The prototypical behavior of a source is shown in Figure 3. It has a continuously varying real-valued

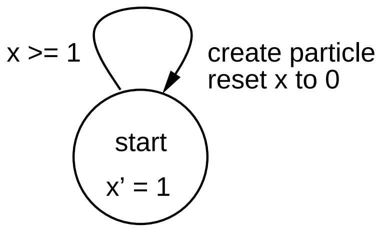

Figure 3: Source

state variable x that measures the elapsed time from the last transition. It has a state variable `monitor` of type `Monitor`, which is a link to the track monitor.

The source has one discrete state `start` in which the flow specification is

$$\dot{x} = 1.$$

The source loops in `start` on an internal event when $x >= 1$, creating a new particle whose speed is initialized from the monitor's output, and resetting the timer x to 0. The SHIFT description of the source type is given below.

```
type Source
{
  state continuous number x;
        Monitor monitor;
  discrete start { x' = 1 };
  transition
    start -> start {}
      when x >= 1
      do {
        create(Particle, speed := speed(monitor));
        x := 0;
      };
}
```

3.1.3 The Monitor Type

The prototypical behavior of a monitor is shown in Figure 4. It has a continuously varying real-valued state variable x that counts down the time to the next `raiseSpeed` broadcast. It has a real-valued output `speed`.

The monitor has one discrete state, `start`, in which the flow is

$$\dot{x} = -1.$$

It loops in `start` synchronously with exactly one particle in the set `secondHalfParticles` on the event `notifyExit`, and it updates its `speed` output to be the average of its old speed and the speed of the exiting particle. It loops in `start` synchronously with exactly all particles in the set `secondHalfParticles` on the event `raiseSpeed` when $x \leq 0$, and it resets x to the next broadcast time using an externally defined function `nextBroadcastTime()`. The SHIFT description of the monitor type is given below.

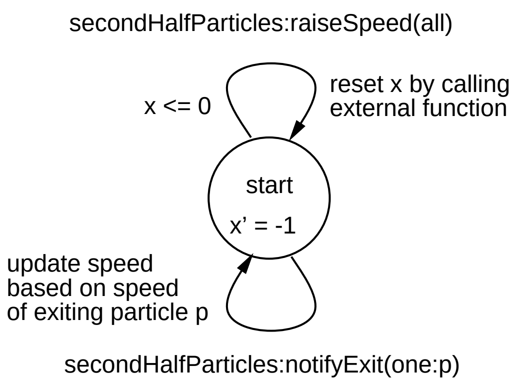
secondHalfParticles:raiseSpeed(all)
x <= 0
reset x by calling external function
start
x' = -1
update speed based on speed of exiting particle p
secondHalfParticles:notifyExit(one:p)

Figure 4: Monitor

```
type Monitor
{
  output number speed;
  state continuous number x;
  discrete start { x' = -1 }
  transition
    start -> start {secondHalfParticles:notifyExit(one:p)}
      do {
        speed := 0.5*(speed + speed(p));
      };
    start -> start {secondHalfParticles:raiseSpeed(all)}
      when x <= 0
      do {
        x := nextBroadcastTime();
      };
}
```

3.1.4 External Function Declaration

The externally defined function `nextBroadcastTime()` is declared within the SHIFT program as follows.

```
function nextBroadcastTime() -> number;
```

3.1.5 Global Variables

The program has three global variables: `monitor`, initialized to a component of type `Monitor`, `source`, initialized to a component of type `Source` and `secondHalfParticles`, a set of particles, initialized to the empty set. The SHIFT description of these global variables is given below.

```
global Monitor monitor
        := create(Monitor, speed := 100, x := nextBroadcastTime());
global Source source := create(Source, monitor := monitor);
global set(Particle) secondHalfParticles := {};
```

A Shift program begins execution by initializing global variables. The components created as part of this initialization begin exercising their dynamical behavior as soon as all global variables are initialized. In this example, the source would create the first particle after one time unit, and the monitor would execute `raiseSpeed` when the initially specified next broadcast time elapses.

3.2 Vehicle-Roadway Environment Processor Example

In this section, we display fragments of a Shift program for the Vehicle-Roadway Environment Processor (VREP). VREP computes the position of a vehicle by integrating the vehicle's speed. The vehicle's position is maintained in two coordinate frames: a global Euclidean reference frame and a road reference frame.

Let $\dot{p}^T$ be the vehicle's linear speed in its body coordinate frame and let R be a skew-symmetric matrix derived from its rotational speeds (its exact form is shown below). Let p_g^T be the vehicle's position in the global frame and let p_r^T be its position in the road frame. Let M_{vg} be the 3×3 unitary matrix which gives the rotational alignment of the vehicle body coordinate frame to the global coordinate frame and let M_{rg} be the corresponding matrix for the road frame at the vehicle's location. Let C, O, G and B be 3×3 matrices derived from the road's curvature, orientation, grade and banking (their exact forms are shown below). Then, the equations of motion are given as follows.

$$\begin{aligned} \dot{p}_g &= \dot{p} M_{vg} \\ \dot{p}_r &= \dot{p} M_{vg} M_{rg}^T C \\ \dot{M}_{vg} &= R M_{vg} \\ M_{rg} &= OGB \end{aligned}$$

3.2.1 VREP Input-Output Specifications

The VREP takes as input the vehicle's linear and rotational speeds in the body coordinate frame. The Shift description of these inputs and outputs are shown below.

```
type VREP // Vehicle Roadway Environment Processor
{
  input
    continuous number xDot, yDot, zDot; // longitudinal, lateral and vertical speeds
    continuous number wx, wy, wz; // roll, pitch and yaw speeds
  output
    continuous number gxp, gyp, gzp; // global x, y and z positions

    Section section; Segment segment; Lane lane;
      // the vehicle's section, segment and lane on the highway

    continuous number rxp, ryp, rzp, lyp; // road x, y and z positions; lane y position

    continuous number vgam11, ..., vgam33; // vehicle-global alignment matrix
    continuous number rgam11, ..., rgam33; // road-global alignment matrix
  ...
}
```

3.2.2 VREP Flow Equations

The Shift description of the VREP flow equations is shown below. Shift permits only scalar equations. However, dynamical equations are often written most conveniently in matrix form. The `mat2scalar` filter associated to the Shift tool-set translates matrix equations into scalar form by performing the matrix operations. The filter performs addition, subtraction, multiplication and transposition.

```
type VREP
{
  ...
  flow cruising {

/* $mat2scalar
                                      |vgam11 vgam12 vgam13|
    |gxp gyp gzp|' = |xDot yDot zDot|*|vgam21 vgam22 vgam23|
                                      |vgam31 vgam32 vgam33|;

    |rxp ryp rzp|' = |xDot yDot zDot| *

      |vgam11 vgam12 vgam13| |rgam11 rgam12 rgam13|~ |1/(1+ryp*curvature) 0 0|
      |vgam21 vgam22 vgam23|*|rgam21 rgam22 rgam23| *|        0           1 0|
      |vgam31 vgam32 vgam33| |rgam31 rgam32 rgam33|  |        0           0 1|;

    |vgam11 vgam12 vgam13|'   | 0  wz -wy| |vgam11 vgam12 vgam13|
    |vgam21 vgam22 vgam23| = |-wz  0  wx|*|vgam21 vgam22 vgam23|
    |vgam31 vgam32 vgam33|    | wy -wx 0 | |vgam31 vgam32 vgam33|;

    |rgam11 rgam12 rgam13|    | ccos scos 0| | cgs 0 sgs| |1   0   0 |
    |rgam21 rgam22 rgam23| = |-scos ccos 0|*|  0  1  0 |*|0  cbs sbs|
    |rgam31 rgam32 rgam33|    |  0    0   1| |-sgs 0 cgs| |0 -sbs cbs|;

$end-mat2scalar */

    ccos = cos(orientation(segment) - rxp*curvature(segment));
    scos = sin(orientation(segment) - rxp*curvature(segment));
    cgs = cos(grade(segment));
    sgs = sin(grade(segment));
    cbs = cos(banking(segment));
    sbs = sin(banking(segment));
  }
  ...
}
```

3.2.3 Sample VREP Transition

The VREP provides several transitions to maintain the position of the vehicle on the road structure—for example, recognizing whether the vehicle has crossed lane, section and segment boundaries, collided with road barriers and blocks or run off the road.

We display a representative transition for updating the vehicle's lane. The VREP transitions from `cruise` to `cruise` on the event `updateLaneRight` when the vehicle's offset from the lane center, `lyp`, is more than half the lane width, there is no barrier on the right edge of the vehicle's lane, there is a lane to the right and there is no barrier on the left edge of that lane. (The symbol-valued `side` variable of a barrier takes valued `$Right` and `$Left`.)

The action associated to this transition consists of a `define` block and a `do` block. In the Shift semantics, the statements in the `do` block of a transition (indeed, of all synchronized transitions) are executed in parallel—i.e., the right hand sides and destinations of all statements are evaluated before any assignments are made to those destinations. The statements in the `define` block are executed sequentially before the `do` block is executed. Variables defined in the `define` block are visible from the `do` block of that transition. The `define` block proves useful to eliminate common subexpressions in the statements of the `do` block, and to create and initialize new components in a sequential order. In the following example, the temporary variable `lr` is a common subexpression in the two statements in the `do` block.

```
type VREP
{
  ...
  transition
    cruise -> cruise {updateLaneRight}
    when
      lyp > 0.5*width(lane) and
      not (exists br in barriers(lane) :
           (side(br) = $Right and startXOffset(br) <= rxp and
              endXOffset(br) >= rxp)) and
      if laneRight(lane) = nil then false else
        not (exists bl in barriers(laneRight(lane)) :
             (side(bl) = $Left and startXOffset(bl) <= rxp and
               endXOffset(bl) >= rxp))
    define {
      lr := laneRight(lane);
    }
    do {
      lyp := -0.5*width(lr) + lyp - 0.5*width(lane);
      lane := lr;
    };
  ...
}
```

4 The Shift Run-time Environment

A Shift program is executed by alternating between the "continuous" and "discrete" steps. In the continuous step, all differentially defined variables are integrated simultaneously using the fourth-order Runge-Kutta numerical integration algorithm. Since all continuous variables are integrated simultaneously, Shift yields more accurate results than simulators that perform component-wise integration using piecewise constant approximations of interacting components.

During the continuous step, the state of the world is examined periodically for enabled world transitions using the synchronization algorithm described in section 4.1. When an enabled transition is found, the discrete step is executed by "taking" the transition. The run-time system takes transitions as soon as they are enabled. The SHIFT language and semantics permit but do not require the as-soon-as semantics in the implementation, and programs written with such an assumption may not work correctly under other implementations of SHIFT.

4.1 The Synchronization Algorithm

In this section, we describe the algorithm that searches for an enabled world transition in the model of section 2. The SHIFT run-time system handles set-valued event references without actually expanding them (see section 2.2).

Since a component transition with a false guard cannot be a part of the world transition, the algorithm considers transitions with true guards only.

4.1.1 Definitions

Let $(c, h : l)$ denote a reference to the event $(h : l)$ in some transition of component c. If $c = h$, $(h : l)$ is a local event of c; otherwise it is an external event of c.

Assume that the set of components are ordered and the transitions within a component are also ordered.

Define $externalSynchronization(h : l) = \{c \mid h : l \text{ is an external event of c}\}$.

The algorithm uses two stacks, the *search stack* and the *backtracking stack*. Elements on the search stack are event references of the form $(c, h : l)$. The search stack keeps track of event references for which the algorithm has not yet selected transitions. Elements on the backtracking stack are of the form (c, d, e, s, b), where c is a component, d is a transition in c, e is an event, s is the size of the search stack when the element is pushed on the backtracking stack, and b is a boolean. A transition or component is said to be *active* if it is on the backtracking stack. When pushing an active component on the backtracking stack, b is set to `old`, and otherwise it is set to `new`. The backtracking stack keeps track of the selected transitions.

The algorithm uses a global variable, `failed`, which, when true, indicates that a world transition cannot be found. The algorithm uses the following operations.

$pushSearch(z)$, $pushBacktrack(z)$—pushes z on the search stack or the backtracking stack.

$pushSearchExpand(c, d, a)$— c is a component, d is a transition in c, a is an event. $pushSearchExpand$ executes the following operations:

```
foreach $h : l$ in external events of $d$ {
    if $h : l \neq a$ {
        $pushSearch(h, h : l)$;
}
foreach $c : l$ in local events of $d$ {
    foreach $h$ in $externalSynchronization(c : l)$ {
        $pushSearch(h, c : l)$;
    }
}
```

$z = popSearch()$, $z = popBacktrack()$—pops the search stack or the backtrack stack in z.

$searchSize()$—returns the number of elements on the search stack.

```
$(c, d) = nextWorldTransition()$;
$pushSearchExpand(c, d, 0)$;
$pushBacktrack((c, d, 0, 0, new))$;
$failed$ = false;
$mark()$;
```

Table 1: The Initialization Operations

```
while not ($searchSize() = 0$ or $failed$) {
    $searchStep()$;
    $mark()$;
}
```

Table 2: The Main Loop of the Algorithm

clearSearch(i)—pops elements off the search stack until there are only i elements left.

active(h)—returns the active transition of component h if any, 0 otherwise.

nextWorldTransition(c, d)—returns the component, transition pair after (c, d) in the world; returns 0 if there is no such transition.

nextComponentTransition(h, d, a)—returns the next transition d' in component h after d that satisfies $a \in E_{d'}$; returns 0 if there is no such transition.

mark()—prints the state of the search and backtracking stacks.

4.1.2 The Algorithm

The algorithm consists of initializations, given in Table 1, followed by the main loop, given in Table 2. At the end of the algorithm loop, if the search has not failed, the backtracking stack contains the nontrivial transitions of the world transition.

The operations *searchStep*() and *backtrack*() used by the algorithm are described in Tables 3 and 4, respectively.

4.2 Example Run of the Synchronization Algorithm

To illustrate the algorithm consider a world with four components a, b, c and, d. Let a have one transition $t1$ with two external events $b : x$ and $d : z$. Let b have two transitions $t1$ and $t2$. Let $t1$ have one local event $b : x$ and one external event $d : w$. Let $t2$ have one local event $b : x$ and one external event $d : z$. Let c have one transition $t1$ with one local event $c : y$. Let d have two transitions $t1$ and $t2$. Let $t1$ have one local event $d : w$. Let $t2$ have one local event $d : z$. Let all components be in the source state of these transitions.

The states of the search and backtracking stacks at each *mark*() point of the algorithm are given in Table 5. The first column is used as an index, the second column shows the backtracking stack and the third column shows the search stack. We discuss some representative parts of the algorithm.

The algorithm picks the transition $(a1, t1)$ as the first transition in the world. The status of the search and backtracking stacks at the end of the initialization has index 1 in Table 5.

```
1   $(c, h : l) = popSearch();$
2
3   $dactive = active(c);$
4   if not $dactive$ { $dnext = nextComponentTransition(c, 0, h : l);$ }
5   if $dactive$ and $h : l \in E_{dactive}$ {
6       $pushBacktrack((c, dactive, h : l, searchSize(), old));$
7   }
8   else if $dactive$ and $(h : l \notin E_{dactive}$ or not $dnext)${
9       $pushSearch(c, h : l);$
10      $backtrack();$
11  }
12  else {
13      $pushBacktrack((c, dnext, h : l, searchSize(), new));$
14      $pushSearchExpand(c, dnext, h : l);$
15  }
```

Table 3: The *searchStep*() Operation

The first call to *searchStep*() pops the element $(b, b : x)$ off the search stack. Since component b is not *active* and has a transition with event $b : x$ lines 13 and 14 of *searchStep*() are executed. The status of the search and backtracking stacks after these operations has index 2.

After several *searchStep*() operations we obtain the search and backtracking stacks with index 5. At this point the *searchStep*() pops the element $(d, d : z)$ off the search stack. Since component d is *active* with transition $t1$ and since $d : z \notin E_{t1}$, lines 9 and 10 of *searchStep*() are executed. Line 10 invokes the *backtrack*() operation, which pops the element $(b, t1, d : w, 1, F)$ off the backtracking stack. Since this is not a *new* element, lines 4, 5, and 6 of *backtrack*() are executed. The status of the search and backtracking stacks at line 5 of backtrack() has index 6.

At the end of the search (index 14) the three transitions on the backtracking stack are $(a, t1)$, $(b, t2)$, and $(d, t2)$. The world transition is constructed by adding the trivial transition in component c.

4.3 Characteristics of the Algorithm

The proof of the correctness of the search algorithm is beyond the scope of this paper. It consists of two parts. The first part proves that the world transition constructed by the algorithm indeed satisfies the conditions in section 2.3. The second part proves that the algorithm is exhaustive and if it fails there are indeed no world transitions.

In the worst case, the space requirement of the algorithm is $O(w)$ and its time requirement is $O(exp(w))$, where w is the number of components in the world.

5 Conclusion

We described the SHIFT programming language and run-time environment for dynamic networks of hybrid automata. We presented the SHIFT models of a type, a component and the world and gave the formal semantics of the model. We described the main features of SHIFT, such as states, inputs, outputs, differential equations and algebraic definitions, discrete states and state transitions, and illustrated them using examples. We described the run-time environment for simulating SHIFT programs and gave the algorithm for determining synchronized transitions.

```
$(c, d, h : l, searchIndex, newElem) = popBacktrack();$

if not $newElem$ {
    $pushSearch(c, h : l);$
    $mark();$
    $backtrack();$
}
else {
    $clearSearch(searchIndex);$
    if $h : l \neq 0$ {
        $dnext = nextComponentTransition(c, d, h : l);$
        if not $dnext$ {
            $pushSearch((c, h : l));$
            $mark();$
            $backtrack();$
        }
        else {
            $pushBacktrack((c, dnext, h : l, searchSize(), new));$
            $pushSearchExpand(c, dnext, h : l);$
        }
    }
    else {
        $(cnext, dnext) = nextWorldTransition(c, d);$
        if ( $(cnext,\ dnext) = 0$ ){
            $failed =$ true;
        }
        else {
            $pushBacktrack((cnext, dnext, 0, 0, T));$
            $pushSearchExpand(c, d, 0);$
        }
    }
}
```

Table 4: The *backtrack* Operation

1		
	(a,t1,0,0,T)	(b,b:x)
		(d,d:z)
2		
	(b,t1,b:x,1,T)	(a,b:x)
	(a,t1,0,0,T)	(d,d:w)
		(d,d:z)
3		
	(a,t1,b:x,2,F)	(d,d:w)
	(b,t1,b:x,1,T)	(d,d:z)
	(a,t1,0,0,T)	
4		
	(d,t1,d:w,1,T)	(b,d:w)
	(a,t1,b:x,2,F)	(d,d:z)
	(b,t1,b:x,1,T)	
	(a,t1,0,0,T)	
5		
	(b,t1,d:w,1,F)	(d,d:z)
	(d,t1,d:w,1,T)	
	(a,t1,b:x,2,F)	
	(b,t1,b:x,1,T)	
	(a,t1,0,0,T)	
6		
	(d,t1,d:w,1,T)	(b,d:w)
	(a,t1,b:x,2,F)	(d,d:z)
	(b,t1,b:x,1,T)	
	(a,t1,0,0,T)	
7		
	(a,t1,b:x,2,F)	(d,d:w)
	(b,t1,b:x,1,T)	(d,d:z)
	(a,t1,0,0,T)	
8		
	(b,t1,b:x,1,T)	(a,b:x)
	(a,t1,0,0,T)	(d,d:w)
		(d,d:z)
9		
	(b,t2,b:x,1,T)	(a,b:x)
	(a,t1,0,0,T)	(d,d:z)
		(d,d:z)
10		
	(a,t1,b:x,2,F)	(d,d:z)
	(b,t2,b:x,1,T)	(d,d:z)
	(a,t1,0,0,T)	
11		
	(d,t2,d:z,1,T)	(b,d:z)
	(a,t1,b:x,2,F)	(a,d:z)
	(b,t2,b:x,1,T)	(d,d:z)
	(a,t1,0,0,T)	
12		
	(b,t2,d:z,2,F)	(a,d:z)
	(d,t2,d:z,1,T)	(d,d:z)
	(a,t1,b:x,2,F)	
	(b,t2,b:x,1,T)	
	(a,t1,0,0,T)	
13		
	(a,t1,d:z,1,F)	(d,d:z)
	(b,t2,d:z,2,F)	
	(d,t2,d:z,1,T)	
	(a,t1,b:x,2,F)	
	(b,t2,b:x,1,T)	
	(a,t1,0,0,T)	
14		
	(d,t2,d:z,0,F)	{}
	(a,t1,d:z,1,F)	
	(b,t2,d:z,2,F)	
	(d,t2,d:z,1,T)	
	(a,t1,b:x,2,F)	
	(b,t2,b:x,1,T)	
	(a,t1,0,0,T)	

Table 5: The State of the Search and Backtracking Stacks

We have implemented a compiler for translating a SHIFT program to a C program, and we have implemented the SHIFT run-time system for obtaining an executable program. The executable program, when run, simulates the design specified in the SHIFT source program.

SHIFT is being used as a system description, integration and simulation environment in the Automated Highway Systems project of the National AHS Consortium. It is also being used at Porto University for describing and simulating coordinated submarine maneuvers for ocean weather profiling.

Acknowledgements

The authors thank Marco Antoniotti, Datta Godbole, Michael Kourjanski, Raja Sengupta, Joseph Sifakis, Joao Sousa, D. Swaroop, Pravin Varaiya, Daniel Wiesman and Sergio Yovine for stimulating discussions and valuable comments, insights and contributions. This project was funded by the California PATH program of the University of California at Berkeley and by the National Automated Highway Systems Consortium.

References

[1] R. Alur, C. Courcoubetis, T. Henzinger, and P. Ho. Hybrid Automata: An Algorithmic Approach to the Specification and Verification of Hybrid Systems. In *Hybrid Systems*, LNCS 736, Springer-Verlag, 1993, pp. 209-229.

[2] A. Deshpande and P. Varaiya. Viable Control of Hybrid Systems. In *Hybrid Systems II*, LNCS 999, Springer-Verlag. 1995.

[3] A. Deshpande, D. Godbole, A. Göllü, L. Semenzato, R. Sengupta, D. Swaroop and P. Varaiya. *Automated Highway System Tool Interface Format.* California PATH Technical Report (draft). 24 January 1996.

[4] A. Deshpande, D. Godbole, A. Göllü, P. Varaiya. "Design and Evaluation Tools for Automated Highway Systems." In *DIMACS* 1995 and in *Hybrid Systems III*, LNCS, Springer-Verlag, 1996.

[5] F. Eskafi, Delnaz Khorramabadi, and P. Varaiya, An Automatic Highway System Simulator. In *Transpn. Res.-C* Vol. 3, No. 1, pp. 1-17, 1995.

[6] *Estelle – A Formal Description Technique Based on Extended State Transition Model.* ISO9074, 1988

[7] D. Godbole, J. Lygeros, E. Singh, A. Deshpande and E. Lindsey. Design and Verification of Communication Protocols for Degraded Modes of Operation of AHS. In *Conference on Decision and Control.* 1995.

[8] A. Göllü. Object Management Systems. *PhD Thesis*, UC Berkeley 1995.

[9] A. Göllü, P. Varaiya. "Dynamic Networks of Hybrid Automata", *Fifth Annual Conference on AI, Simulation, and Planning in High Autonomy Systems*, pp. 244-251, Gainesville, Florida. 1994.

[10] J. Haddon, D. Godbole, A. Deshpande and J. Lygeros. Verification of Hybrid Systems: Monotonicity in the AHS Control System. In *DIMACS*. 1995.

[11] C.A.R. Hoare. *Communicating Sequential Processes*, Prentice/Hall International, 1985

[12] G.P. Hong and T.G. Kim. The DEVS Formalism: A Framework for Logical Analysis and Performance. In *Fifth Annual Conference on AI, Simulation, and Planning in High Autonomy Systems*, pp. 170-278, Gainesville, Florida. 1994.

[13] A. Hsu, F. Eskafi, S. Sachs, P. Varaiya. Protocol Design for an Automated Highway System. In *Discrete Event Dynamic Systems: Theory and Applications 2*, (1993): 183–206.

[14] Kemal Inan and Pravin Varaiya. Finitely Recursive Process Models for Discrete Event Systems. In *IEEE Trans. Auto. Control*, vol. AC-33, no. 7, pp. 626-639, July 1988.

[15] R. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach.* Princeton University Press, 1994.

[16] *LOTOS – A Formal Description Technique Based on the Temporal Ordering of Observational Behavior.* ISO8807, 1XS989

[17] R.Milner. *A Calculus of Communicating Systems*, Springer-Verlag, 1980

[18] A. Puri and P. Varaiya, Driving safely in smart cars. In *American Control Conference*, pp. 3597–3599, 1995.

[19] L. Semenzato, A. Deshpande and A. Göllü. *Shift Reference Manual.* California PATH Technical Report (draft). 28 June 1996.

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

[21] P. Varaiya. Smart cars on smart roads: problems of control. In *IEEE Trans. Automatic Control*, vol. 38, No. 2, 1993.

[22] Bernard Zeigler. *Multifaceted modeling and discrete event simulation.* Academic Press, London, Orlando, 1984.