The SHIFT Programming Language 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
http://www.path.berkeley.edu/shift

1 Introduction

SHIFT is a programming language for describing and simulating 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: continuous-time phases separated by instantaneous discrete-event transitions. Components may evolve independently, or they may interact through selected state variables and events. The interaction network itself may evolve.

The SHIFT model and language were motivated by our need for tools that support dynamically reconfigurable hybrid systems. Our primary application is the specification and analysis of different designs for automatic control of vehicles and highway systems [19]. We need to capture the behavior of vehicles, sensors, actuators, communication devices and controllers in a structured and modular specification [4]. Models for these components are provided by different groups of experts and then integrated in different ways for comparative analysis. From our previous experience in modeling [10, 6], analysis [17, 3] and implementation [5, 7], we adopted the hybrid systems approach for modeling the system components. Since spatial relationships between vehicles change as they move, our application is characterized by a dynamically changing network of interactions between system components. SHIFT has since also been used in coordinated autonomous submarines [18], air traffic control systems [11], and material handling systems.

In the rest of this section, we examine other work related to the SHIFT approach. In section 2, we describe the main features of the SHIFT language—states, inputs, outputs, differential equations and algebraic definitions, discrete states and state transitions. In section 3, we give 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.

1.1 Related Work

SHIFT is used to describe models with switched differential equations (such as a vehicle with automatic gear shift) and coordinated behaviors (such as communicating controllers). Standard 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 the evolution of relationships among components), complex event-triggering conditions (such as existential queries on the state of the world), and complex program logic (such as synchronous compositions of state machines).

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

The Omola/Omsim [15] language has a very similar approach to hybrid system modeling as SHIFT. However, Omola is designed to represent statically interconnected objects, and it does not provide the means of querying sets of components using existential quantifiers.

Statecharts [9] and Argos [13], based on Statecharts, are approaches for synchronous discrete event modeling. Their focus is on hierarchical specification of finite state machines. SHIFT does not provide explicit facilities for hierarchical behavior specification; instead, it provides a sub-typing mechanism which can be used to organize components hierarchically. SHIFT adds continuous time semantics and dynamic reconfigurability of the synchronization structure.

Recent extensions to the DEVS [21] formalism have introduced notions of dynamic reconfiguration [2, 16]. However, the DEVS formalism is primarily aimed at untimed discrete event simulation and the extensions provided for continuous evolution laws are limited.

Our mathematical model for the discrete event semantics is similar to Milner's π-calculus [14]. Both models achieve reconfiguration by a renaming of event labels used in synchronization.

2 The SHIFT Language

A SHIFT program describes a set of interacting objects called *components*, grouped into *component types*. The SHIFT construct called the *type declaration* specifies the prototypical behavior of all components of a given type.

SHIFT additionally supports a small set of primitive data types (`number` and `symbol`) and of constructed types (`array` and `set`).

- Variables of type `number` and `continuous number` have, respectively, piecewise constant and piecewise continuous real-valued time traces.
- Variables of type `symbol` have piecewise constant symbol-valued time traces.
- 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 (or, with some abuse of terminology, a *type*) is defined by the SHIFT `type` declaration. The structure of a type is partly given by its `inputs`, `outputs`, and `states`, illustrated in the following example.

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

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 value of an input variable of a component may only be specified by another component.

The `discrete` clause defines the possible values for the type's discrete state variable and associates a set of differential equations and algebraic definitions to each discrete state. Groups of common flow equations are given a name through the `flow` clause. The special flow name `default` defines the default behavior of a set of variables in all states. For instance:

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

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

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

Transitions between 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 road-side controller signals an emergency. This can be specified with the transition

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

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

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

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

SHIFT supports more complex synchronizations using the `one` and `all` mechanisms. The synchronization rules are stated in section 3.

The transition guards, given by the `when` clause, may contain existential quantifiers that query the state of sets of components (possibly all existing components). For example, let `Cars` be the set of all the components of the `Car` type and let the road consist of a single lane. Then, the variable `car_in_front` is updated as follows.

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

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

```
type Car {
   ...
   setup do { Cars := Cars + {self}; }
   ...
}
```

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

3 The SHIFT Model

The model presented in this section is austere and simplified. 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.

The SHIFT mathematical model is class based [20]. The components are organized into *types* that describe the prototypical behavior of a component. We say that a component is an instantiation of its type. Type descriptors are an implicit and static part of the world. A given world may contain many definitions $W_H = \{H_1, \ldots, H_k\}$. The subset of the world that consists of components of type H_i is denoted by $W{:}H_i$.[1]

In a class based model one has to distinguish a variable's *name* from its *value*. Below when we refer to a variable x in the context of a type, we refer to the variable's name and when we refer to a variable x in the context of a component, we refer to its value.

Another important concept in a class based model is that of a *component id*. We assume that each component in the world has a unique identifier.

3.1 Type Description

The components are organized into *types* according to their prototypical behaviors. In this simplified model, a type is the tuple

$$
\begin{aligned}
H \;=\; & (q \in Q,\ Q \text{ finite,—the discrete state variable,} \\
& x = (x_1, \ldots, x_n),\ \text{with each } x_i \in \mathbf{R}, \text{—the continuous state variables,} \\
& C = C_{M_0} \cup (M_1\, C_1, \ldots, M_m\, C_m), \text{—the configuration state variables} \\
& L = \{l_1, \ldots, l_p\}, \text{—the event labels}
\end{aligned}
$$

[1]The mathematical model does not represent sub-typing which is part of the SHIFT language. In the mathematical model, $W{:}H_i$ are disjoint, in the language $W{:}H_i$ are partially ordered with respect to subset inclusion.

$F = \{F_q \mid q \in Q\}$—the flows, and

T—the transition prototypes.)

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.

The continuous state variables of a type are given by x with each $x_i \in \mathbf{R}$.[2] The continuous evolution laws for these variables are given by F as described below.

C_{M_0} is a tuple $(M_{0_1}\ C_{0_1}, \ldots, M_{0_p}\ C_{0_p})$ where $M_{0_i} \in W_H$ and $C_{0_i} \in W{:}M_{0_i}$ and p is a constant. In essence, C_{0_i} is a reference to another component of type M_{0_i}.[3] We define $C_0 = \{C_{0_1}, ..., C_{0_p}\}$.

For $i = 1, \ldots, m$, $M_i \in W_H$ and $C_i \subset W{:}M_i$. In a component, C_i may have a different number of elements in different phases. $C_i,\ i = 1, \ldots, m$ can be thought of as a set-valued reference to other components in the world.

The event labels L in a type are values.

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.[4]

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

[2]Recall that in a type a continuous state variable x_i is a name and in a component it takes on a value in $\mathbf{R}$.

[3]In a type, C_{0_i} is a name; in a component it takes on a value in $W{:}M_{0_i}$. Note that M_{0_i} and p are values in a type.

[4]The implementation detects cycles by imposing a limit on the depth of algebraic dependencies.

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.[5] Informally, for δ to be taken, the guard must evaluate to true.

Each event label in E takes one of four forms:

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 the language, a transition which does not synchronize with others is specified by leaving E empty. 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 the language, actions may also reset the inputs of components in C_0.

3.2 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 SHIFT run-time system.

[5] 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.

- 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 $(q, \text{True}, \emptyset, I, q)$ 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:

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

3.3 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

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

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 [8, 12]. 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.

3.4 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.[6]

[6]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.

Acknowledgements

The authors thank Marco Antoniotti, Peter Cooke, Datta Godbole, Tolga Goktekin, Michael Kourjanski, Raja Sengupta, Joseph Sifakis, Tunc Simsek, Joao Sousa, D. Swaroop, Pravin Varaiya, Daniel Wiesman, Sergio Yovine and Marco Zandonadi 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] F. Barros. Dynamic Structure Discrete Event Specification Formalism. In *Transactions for the Society for Computer Simulation*. No 1, pp. 35-46 1996.

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

[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] 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.

[7] A. Göllü, P. Varaiya. "SmartAHS: An Object Oriented Simulation Framework for Highway Systems", To Appear in Iternational Journal of Mathematical and Computer Modeling.

[8] 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.

[9] D. Harel. Statecharts: a visual approach to complex systems. Science of Computer Programming, 8(3):231–275, 1987.

[10] 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.

[11] 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.

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

[13] F. Maraninchi. The Argos language: Graphical Representation of Automata and Description of Reactive Systems. *Proc. IEEE International Conference on Visual Languages*, Kobe, Japan, 1991

[14] R. Milner, J. Parrow, D. Walker. A Calculus of Mobile Processes (Parts I and II). *Information and Computation*, 100 Vol. 1 pp 1–77. 1992.

[15] S.E. Mattsson and M. Anderson. The Ideas Behind Omola. In *Proc. IEEE Symposium on Computer Aided Control System Design, CADCS '92*. Napa, CA, March 17-19 1992.

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

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

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

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

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

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