University of California, Berkeley

SHIFT Tutorial

A first course for SHIFT programmers

Tunç Şimşek, January 27th, 1999

Before we begin

This course will walk you through a series of examples so as to provide a detailed overview of the SHIFT language syntax and semantics. At the end of the course you should be able to write, compile and execute your own simulations. Further reading can be obtained from the SHIFT home page accessible via http://www.PATH.Berkeley.Edu/SHIFT. The *"SHIFT: Reference Manual"*, A. Deshpande, A. Göllü, L. Semenzato, PATH Report is the primary source of information for the language specifics[1]. The document *"SHIFT Programming Language and Run-Time System for Dynamic Networks of Hybrid Automata"*, A. Deshpande, A. Göllü, L. Semenzato, PATH Report should be referred to for a better understanding of the SHIFT mathematical model.

This document is organized in a narrative setting. The idea is to provide on hands experience with emphasis on semantics rather than syntax.

Abstract

SHIFT is a programming language for describing dynamic networks of hybrid automata. Hybrid automata are viewed as components that execute their behavior in continuous phases separated by discrete event transitions. Components can be created, destroyed or interconnected to produce a network of components. As such components may evolve individually or they may interact via a set of inputs, outputs and event synchronizations. The network itself may reconfigure as a result of such interactions.

Pragmatics

Specifically, the SHIFT language has simulation semantics. The procedure for generating an executable simulation will usually depend on the platform. Currently, there is a SHIFT compiler, command line debugger and visual debugger configured to operate on the Sun Solaris and SGI operating systems. We write a SHIFT program, compile it with the SHIFT compiler to generate C code, and finally link this C code with the SHIFT run-time libraries to obtain an executable simulation.

Before we continue with this tutorial we should make sure that we are comfortable with this compilation procedure. The first step is to configure our working environment. This consists of setting some environment variables[2]. Refer to the figure below and copy the code provided into your .cshrc file and then source the new .cshrc file with:

```
$source ~/.cshrc
```

[1] The latest version of the *"Shift: Reference Manual"* is also found in the distribution under the documentation directory.

[2] The environment specific initializations are given for users of the *C shell* or *Turbo C shell* under the Sun Solaris, SGI operating systems. The provided initializations should be added to your *.cshrc* file in your home directories.

Figure 1: Initialization part in the .cshrc file

```
set path = ($path /usr/local/shift/bin)
if (-f /usr/local/shift/shiftDefaults) then
        source /usr/local/shift/shiftDefaults
endif

setenv TCL_LIBRARY /usr/local/shift/lib/tcl
setenv TK_LIBRARY /usr/local/shift/lib/tk
setenv CPP /usr/ccs/lib/cpp
```

Now we should be ready to compile SHIFT programs. To try this out compile the SHIFT program provided in the figure below with the shell script *cshift*[3]:

```
$cshift test.hs

cshift: compiling to C...
cshift: compiling C intermediate code...
cshift: done compiling the SHIFT file test.hs.
        the resulting executable simulation  was left
        in file test.sim.
```

You may now execute the simulation in 2 ways; either by executing the resulting *test.sim* file which contains a built in command line debugger or by executing *TkShift* and then connecting to the simulation *test.sim*. The latter is self explanatory in the sense that the debugging options are presented graphically, hence it will not be discussed here; however, I suggest that you try it for yourself. To get the started with the command line debugger work through the following example:

```
$test.sim

Time is 0.

SHIFT debug > c 100
[1]...................................................
[51]..................................................
Time is 99.

SHIFT debug > lt
List of all the types: TestComponentType

SHIFT debug > l TestComponentType
List of components of type TestComponentType:
    TestComponentType 0

SHIFT debug > p TestComponentType 0
{TestComponentType 0
    {x 60.957}
    {a 0.100}
    {discrete loop~0}
}
```

[3] All the source files given in this tutorial can freely be downloaded from http://www.EECS.Berkeley.Edu/~simsek

It should be clear that the debugger commands **c**, **lt**, **l**, **p** stand for **C**ontinue, **L**ist **T**ypes, **L**ist components of type and **P**rint components of type respectively. This tutorial will provide the minimal debugger commands so that the examples may be inspected. You should refer to the SHIFT home page for an explanation of all the command line debugger options or simply type **help** at the debugger prompt.

Now that we are ready to compile and execute SHIFT programs we proceed to investigate the SHIFT programming language.

Figure 2: test.hs

```
type TestComponentType {

  state continuous number x := 100;
  state number a := 0.1;

  flow
    default {

      x' = -a*x;

    };
}

global TestComponentType testComponent :=
                                create(TestComponentType);
```

The SHIFT programming language

Take a moment to study the program *test.hs* listed above. The first aspect of this program that we notice is that the syntax is suggestive. Namely, there is an abstract class of hybrid automata specified within the `type TestComponentType { … }` clause and this class, henceforth referred to simply as **type**, is given the name *TestComponentType*. Within the type clause we specify the behavior, or evolution laws, of this type. In doing so we make use of **state** variables. In this case, we define the state variable *x* and say that it is of type **continuous number**. Then we define a parameter[4] *a* and say that it is of type **number**. Finally we establish the behavior of components of this type. This is done with the **flow** clause. We indicate that components of this type will evolve such that their *x* variables obey the given differential equation (flow). The identifier **default** means that the specified flow will be obeyed in any discrete mode of the hybrid machine unless specified otherwise. Note that we have not specified any discrete modes in this example. Finally, we make use of the **create** operator to instantiate a component of type *TestComponentType* in the global scope and we reference it with *testComponent*.

The above example is not intended to provide the details of SHIFT, rather it was intended for you to get some exposure to the syntax. The main result of studying the test.hs will be drawn somewhat loosely. That is, the constructs that outline a SHIFT program are of 3 kinds. These are, type definitions, global component declarations and external function specifications. A SHIFT program is thought of a closed world that consists only of components that are instances of type definitions. Global component declarations are for declaring such instances in the global scope. The external functions are for auxiliary routines such as file i/o, arithmetic routines. Formally, a SHIFT program is:

shift–program	$\rightarrow$	*top–level–construct*$^{+}$
top–level–construct	$\rightarrow$	*type–definition* | *global–component–declaration* | *external–function–definition*
type–definition	$\rightarrow$	**type** *type–identifier [***:***optional–supertype–identifier]* { … }
global–component–declaration	$\rightarrow$	**global** *variable–declaration*

Types and Components

As seen from the example above, a SHIFT program is made up of types and components. It is useful to draw an analogy to the object oriented paradigm of classes and objects. Namely that within a class description one declares a data part and a collection of methods that operate on these data. Then instances of these classes are created and

[4] One aspect of SHIFT variables is that we do not syntactically distinguish between state variables and parameters.

are called objects. Similarly in SHIFT we describe a type by associating an identifier to it, declaring state, input, output variables and events, then we formulate evolution laws in state space. A SHIFT type construct can be summarized with the following template:

```
type TypeIdentifier {

  input ...        // What is fed to it
  state ...        // Memory
  output ...       // What it feeds to others

  export ...       // A list of event names
                   // for synchronization

  flow ...         // Flow specifications for the
                   // continuous evolution

  discrete ...     // A list of discrete modes
                   // and the flow obeyed
                   // in each discrete mode

  transition ...   // The transition structure
                   // amongst discrete modes

  setup ...        // What happens when a component
                   // of this type is
                   // initially instantiated
}
```

Variables are classified as being either **input**, **state**, **output** or **global**. The state, input and output modifiers refer to the accessability of a variable declared within the scope of a type. Namely, input variables are read only, output variables are public read-write and state variables are private read-write. The global modifier refers to the accessability of a variable declared in the global scope; they are public read-write. The term ‘state’ may cause a confusion in that mathematically all of input, state and output variables are treated as state variables although only the private ones are formally referred to as state. The syntax for declaring a variable is:

<table>
<tr><td>variable–declaration–clause</td><td>$\rightarrow$</td><td>scope variable–declaration^{+}</td></tr>
<tr><td>scope</td><td>$\rightarrow$</td><td>state | input | output | global</td></tr>
<tr><td>variable–declaration</td><td>$\rightarrow$</td><td>type identifier optional–initialization ;</td></tr>
<tr><td>type</td><td>$\rightarrow$</td><td>number / continuous number / symbol
| set(type) | array(type) | identifier</td></tr>
<tr><td>optional–initialization</td><td>$\rightarrow$</td><td>: = expression | ϕ</td></tr>
</table>

A variable may be of type **number**, **continuous number, symbol**, a user defined type or nested **set**(s) and **array**(s) of these types. The initialization of a variable is optional and it can be any valid SHIFT expression. Some examples of variable definitions are:

```
input  Vehicle v := some_vehicle_defined_somewhere;
state  continuous number gamma := 2*sin(45);
       array(set(number)) as := [ { 1 , 2 } , { 0, 3.14 } ];
       symbol color := $red;
output set(number) s := { 3, 5, 12.34, 14.2335 };
       array(Vehicle) vs :=[create(Vehicle), create(Vehicle)];
```

The **export** clause is for defining events that are to be used for synchronization with other components. Note that we can export events to be of **open** or **closed** type. If we do not specify anything then the default will be closed. The difference between these two types is stated formally in the section "Synchronization Constraints". The formal syntax of the export clause is:

export–clause	$\rightarrow$	**export** *event–list* **;**
event–list	$\rightarrow$	*event–type identifier ,event–list* */ event–type identifer*
event–type	$\rightarrow$	**open** / **closed** / ϕ

The **discrete** clause is for defining the discrete modes of the finite state machine associated with this type. Namely, we label each discrete mode with an identifier and we associate an optional flow to it. The optional flow should be valid flow construct as described in the next section. An illustration of the discrete clause follow as:

```
discrete
 idle { x' = 0; },
 busy1 { x' = f_busy(x);  y = g_busy(x); },
 busy2 { busy_2_flow_expressions; },
 busy3;
```

The initial state of the finite state machine will be the first state specified in the discrete clause, in this case the *idle* state. As seen above, the flow for *idle* and *busy1* are specified directly. The flow for *busy2* is an identifier and is assumed to have been defined in the flow clause described in the next section. The flow associated with *busy3* will be the default flow if it is specified in the flow clause; otherwise all variables will have rate zero.

There is a special state called the **exit** state. It cannot be defined since it is a built into every component, however we will illustrate its use in the example *zeno.hs* later in this document. If the exit state is entered the component is destroyed and in fact this is the only destructor for a component. Formally, the discrete clause syntax is given by:

discrete–clause	$\rightarrow$	**discrete** *mode–list*;
mode–list	$\rightarrow$	*mode* , *mode–list*
		| *mode*
mode	$\rightarrow$	*mode–identifier optional–flow*
mode–identifier	$\rightarrow$	*identifier*
optional–flow	$\rightarrow$	{ *evolution–law*$^+$ } | { *flow–identifier* } | ϕ

Once we've specified a type, we need some way for creating components of this type. This is done through the **create** operator. We've already seen this operator before. However, the create operator has more capabilities than what we've seen so far. Namely, we can initialize the variables of a type upon creation. To do so we pass additional arguments to the create operator. Consider again the test program. We can modify the creation of the *testComponent* as illustrated in the figure below. In this case the default initialization for the variables *x* and *a* will be ignored and *testComponent* will start its evolution with $x = 200$ and $a = 0.05$.

Figure 3: create.hs

```
type TestComponentType {

  state continuous number x := 100;
  state number a := 0.1;

  flow
    default {

      x' = -a*x;

    };
}

global TestComponentType testComponent :=
                                create(TestComponentType,
                                       x := 200,
                                       a := 0.05);
```

As mentioned before, it is useful to draw an analogy between SHIFT types and components to the object oriented paradigms of classes and objects. So far we have discussed the data part of SHIFT types. In the next section we will work on the methods, or evolution laws, of SHIFT types. Before we continue, however, there is another SHIFT clause that may be viewed within the data part of a SHIFT type. This is the **setup** clause. It specifies the initial actions to be taken by the component once it is created. The setup clause can have 3 parts: **define**, **do** and **connect**. In the define clause we can create auxiliary, temporary, variables that exist only in the scope of the following do and connect clauses. We will see shortly why these variables are mathematically auxiliary as compared to the input, state and output variables which are mathematically state variables. In the do clause we modify the input, state and output variables of the component that is just created. Finally, in the **connect** clause we establish an algebraic equality between two variables; specifically, we connect an input, state or output variable of one component to the input variable of another component. The result of the connection is that the input of the second component follows whatever value the input, state or output of the first component takes. The setup clause is illustrated in the figure below.

Figure 4: setup_bad.hs

```
type TestComponentType {

  state continuous number x := 100;
        continuous number t;
  state number a := 0.1;
        number w;
  flow
    default {

      t' = 1;
      x' = -a*x + cos(w*t);

    };

  setup
     define {

       number pi := 3.14;
       number freq := 50 ;
     }
     do {

       w := 2 * pi * freq;
       t := 0;
     }
     connect {

       x(mon) <- x;
     };
}

type Monitor {

  input continuous number x;

}

global TestComponentType testComponent :=
                                create(TestComponentType,
                                       x := 200, a := 0.05);
global Monitor mon := create(Monitor);
```

Several modifications were done to the program *create.hs* to obtain *setup_bad.hs*. The purpose of this example is 2-fold. First, we illustrate the different ways to initialize a component upon creation and emphasize the order of initializations. Second, the semantics of input, state and output variables are differentiated from the semantics of auxiliary variables.

The first concept that is crucial to understand is that a list of assignments on input, state, output and even global variables has no ordering, specifically, they are executed in parallel. In the example listed above, *testComponent* and *mon* will be created at the same time and hence will not know of each other during their initialization phases. One requires caution here, because *testComponent*, when created, refers to *x(mon)*[5] in its do clause. But *x(mon)* is not an accessible reference and this will result in an error. This problem is fixed with *setup_good.hs*.

Figure 5: setup_bad.hs

```
type TestComponentType {

  state continuous number x := 100;
        continuous number t;
  state number a := 0.1;
        number w;
  input Monitor mon_reference;

  flow
    default {

      t' = 1;
      x' = -a*x + cos(w*t);

    };

  setup
     define {

       number pi := 3.14;
       number freq := 50 ;
       number angular_freq := 2*pi*freq;
     }
     do {

       a := 0.02;
       w := angular_freq;
       t := 0;
     }
     connect {

       x(mon_reference) <- x;
     };
}

type Monitor {
```

[5] *x(mon)* is read as “the variable *x* of *mon*”.

```
}

global TestComponentType testComponent;
global Monitor mon;

setup
  define {
    Monitor m := create(Monitor);
    TestComponentType t := create(TestComponentType,
                                  x := 200, a := 0.05,
                                  mon_reference := m);
  }
  do {
    testComponent := t;
    mon := m;
  };
```

Now, we are guaranteed that *m* will be created before *t*. For clarity I summarize here the precedence of initializations:

1. Perform the optional initializations specified in the definition of the variables.

2. Perform the optional initialization specified in the create operator.

3. Evaluate the optional setup clause.

In the above example, the global components are initialized as follows. First the variables *mon* and *testComponent* are declared but not created. Then the global setup clause is evaluated. The define clause in the global setup has sequential execution contrary to the way variable declarations were executed in parallel. Hence, we define the temporary variables *m* and *t* and create components of type Monitor and TestComponentType and in this order. Upon creation of *t*, the variables *x* and *a* of *t* take the values *200, 0.02*, and the variable *mon_reference* of *t* refers to the component *m* created above. Now, since *mon_reference* is not a nil reference we can safely access *x(mon_reference)* so that it will follow the value of the variable *x(testComponent).* Finally, in the do part of the global setup clause, the references *mon* and *testComponent* are assigned to the components that *m* and *t* refer to.

As seen from the example above, we broke up the expression *2*pi*freq*. Instead of writing *2*3.14*50* I preferred to define the value of *pi* first, then *freq* and finally *w := angular_freq*. This illustrates that the statements in the define clause are executed <u>sequentially</u>, and also that these temporary variables are available in the do clause.

Consider now the evolution semantics of state, input, output and global variables. The These semantics may be of 2 kinds, continuous or discrete. In the discrete case we have the actions[6] in the do clause. Their execution is strictly parallel. All the right-hand-side expressions are evaluated first, then they are assigned to the left-hand-sides in parallel. In the continuous case, we have the flow[7] equalities in the flow clause. Their evolution has

[6] Henceforth, statements in the do clause will be referred to as *actions*.
[7] Henceforth, statements in the flow clause will be referred to as *flow*.

flow semantics, i.e. algebraic equality or differential equality. Flow equality is distinguished from actions by using "=" in one case and ":=" in the other.

Evolution

We have seen that equality of input, state, output and global variables can be either of discrete semantics, i.e. actions, or of continuous semantics, i.e. flow. This section clarifies the framework that lets these two semantics work together. To begin with, we maintain that a SHIFT type is an abstract model of a hybrid state machine. We think of the SHIFT world as one that evolves in state space where the state variables are roughly the input, state, output, global variables and discrete modes of a finite state machine. Next state assignment is due to flows and transitions.

An individual component evolves in phases. Initially a component starts out in a discrete mode of its finite state machine and time passes while the state and output variables evolve according to the flow associated with that mode. At some time the outgoing transitions are enabled, time stops, the component executes one of these transitions and enters a new discrete mode of its finite state machine. Then time passes again and the state and output variables evolve according to the flow, possibly a different set of laws, associated with this mode.

In the case of several components each component evolves as above with the exception that they may be interacting. At first, all the components are in the initial discrete modes of their finite state machines. Time passes and the components evolve in parallel according to the flows associated with these discrete modes. In this continuous phase the components may be interacting due to input and output variables. At some time, an outgoing transition of one or more of the components may become enabled. Time stops and if these components satisfy a set of synchronization constraints then they synchronize and execute their transitions in parallel. If they do not have any synchronization constraints then they execute their transitions sequentially. Then the components evolve in the new discrete modes of their finite state machines.

Continuous Phase

The flow is a collection of differential equations in state space format:

$$\dot{x} = f(x,u)$$
$$y = g(x,u).$$

Here I have adopted the usual notation for u for input, x for state and y for output variables. The functions f and g can be arbitrarily complex. The output variable assignment $y = g(x,u)$ is also referred to as algebraic flow. This is a better term because we can have an algebraic equality of quantities other then numbers, so long as it is defined within some algebra.

We note that inputs cannot appear on the left hand side assignments. However, one convention is that we output variables may appear directly in the left-hand-side of differential equality, and similarly state variables can appear in the left-hand-side of algebraic equality.

The formal syntax for flow is listed below. Please refer to the discrete clause listed previously and note that evolution laws can also be specified directly within a discrete mode, or by means of reference to a flow identifier. If nothing is specified for a discrete mode then the default flow (if specified) will be associated with that discrete mode.

flow–clause	$\rightarrow$	**flow** *flow–declaration–list ;*
flow–declaration–list	$\rightarrow$	*flow–declaration , flow–declaration–list* *| flow–declaration*
flow–declaration	$\rightarrow$	*flow–identifier* **{** *evolution–law*$^{+}$ **}**
evolution–law	$\rightarrow$	*identifier* **'** = *expression* *| identifier = expression*
flow–identifier	$\rightarrow$	*identifier* | **default**

Another form of algebraic equality was discussed previously in the connect part of a setup clause. We will conclude the discussion for continuous evolution by providing an example of a DC motor. The example is motivated by using connect to interconnect causal dynamic systems. The overall model of the motor is illustrated in the figure below which is followed by a listing of the file *motor.hs*.

Figure 6: Causal model of a DC Motor

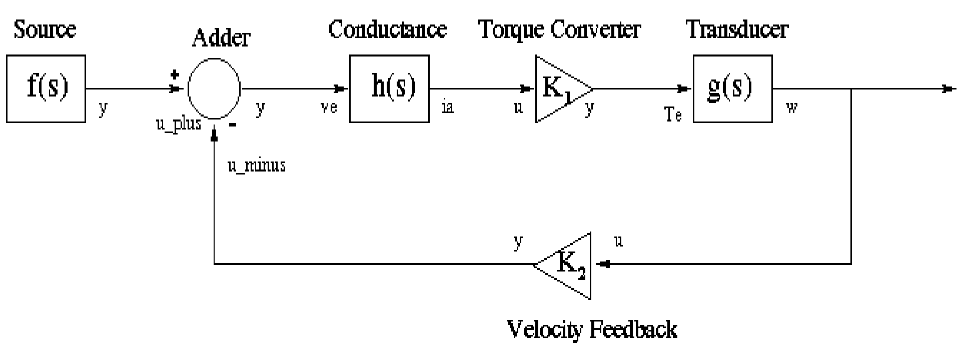


Figure 7: motor.hs

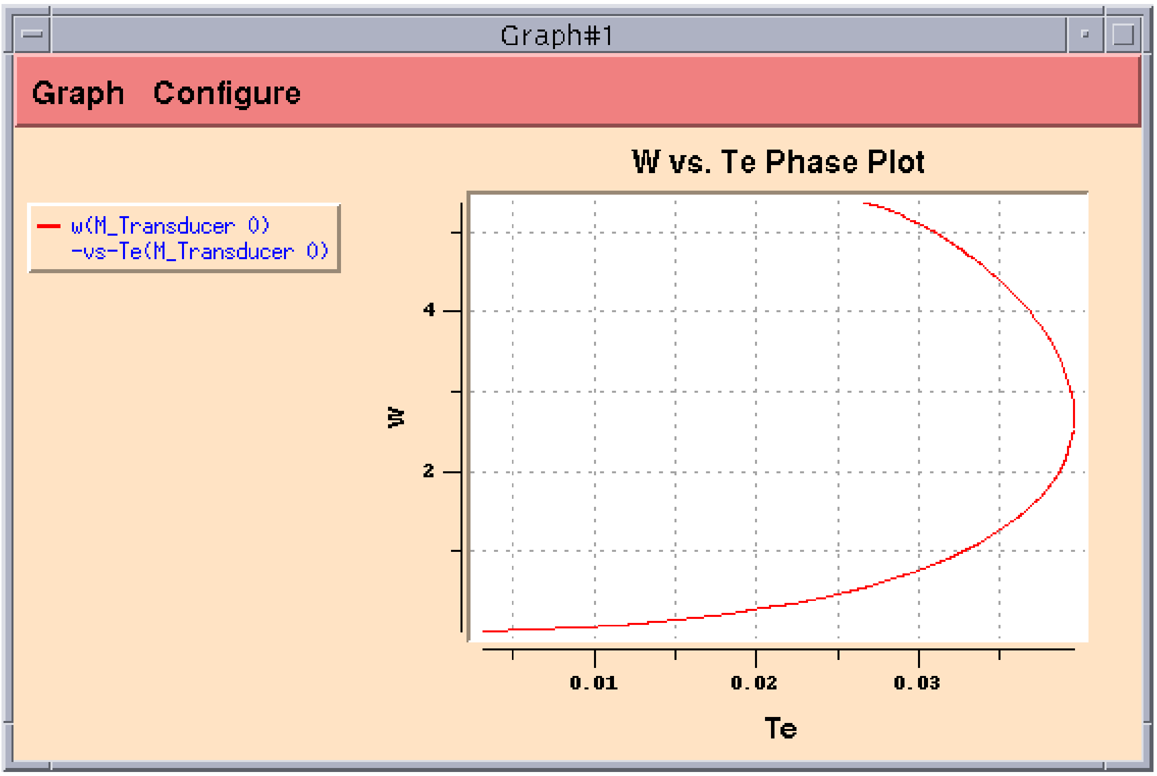
```
// This is the model of a generic multiplier.
// It multiplies the input u with a constant K and produces the
// output y.
type  Gain {

  input continuous number u ;
  output continuous number y ;
  state number K ;

  // The only discrete state which utilizes the default flow.
  discrete
    normal ;
```

```
  flow
    default {

      y = K * u ;
    };
}

// This generic adder takes the difference of
// the u_plus input and the u_minus input and produces the y output.
type  Adder {

  input continuous number u_plus ;
        continuous number u_minus ;
  output continuous number y ;

  // The only discrete state which utilizes the  flow "add".
  discrete
    normal { add; } ;

  flow
    add {
      y = u_plus - u_minus ;
    };
}

// This model is basically the conductance which takes as
// input the error signal "ve" and produces the output "ia".
type  Conductance {

  input continuous number ve ;    // The error signal
  output continuous number ia ;   // The armature current
  state number Ra := 1.75 ;       // The armature resistance
        number La := 2.833-3 ;    // The armature inductance

  // The discrete mode with the flow specified inline.
  discrete
    normal { ia' = ( ve - ia * Ra ) / La ; };
}

// This model is basically a transducer which takes as
// input the torque "Te" and produces as output the rotational speed "w".
type  Transducer {

  input continuous number Te ;  // The generated torque
  output continuous number w ; // Rotational speed of the DC motor
  state number Beq := 5e-3 ;     //  Equivalent Rotational damping constant.
        number Jeq := 30e-6 ;  // Equivalent inertia of motor and load.

  discrete
    normal { w' = ( -Beq * w + Te ) / Jeq ; };
}

// Outputs a constant number "y".
type  Source {
  output continuous number y ;
  setup  do { y := 1 ; };
}
```

```
// The Dynamic model of a DC Motor
type  DC_Motor {

  state
     Conductance conductance ;
     Transducer  transducer  ;
     Adder       adder  ;
     Gain        torque_Converter ;
     Gain        velocity_Feedback ;
     Source      source ;

  setup
    do {
       adder             := create(Adder);
       conductance       := create(Conductance,
                              Ra := 1.75,
                              La := 2.83e-3);
       transducer        := create(Transducer,
                                     Jeq := 30e-6,
                              Beq := 5e-3);
       torque_Converter  := create(Gain,
                              K := 0.0924);
       velocity_Feedback := create(Gain,
                              K := 0.093);
       source            := create(Source);
    }
    connect {
       u_plus(adder)       <- y(source);
       u_minus(adder)      <- y(velocity_Feedback);
       ve(conductance)     <- y(adder);
       u(torque_Converter) <- ia(conductance);
       Te(transducer)      <- y(torque_Converter);
       u(velocity_Feedback) <- w(transducer);
    };
}

//  Create the global DC_Motor.
global  DC_Motor motor := create(DC_Motor);
```

Here is a phase plot of the DC Motor characteristics. This plot was obtained with TkShift.

Figure 8: Phase plot of DC Motor characteristics



Discrete Phase

We have already see the data part on which the discrete machinery of SHIFT operates. These were the discrete states of a finite state machine to which we associated flows.

A transition connects two modes of the finite state machine. What is of interest to us is when this transition may be taken and what is the result of execution this transition. As a mathematical object a transition may be represented with the following notations:

$$\begin{aligned} \delta \quad &\equiv \quad s_0 \in Q \xrightarrow{\text{when}(\cdot)\in\{\text{true,false}\} \qquad e\subset\text{Exported Events} \qquad \text{do}(\cdot)} s_1 \in Q \\ &\equiv \quad (s_0, \text{when}(\cdot), e, \text{do}(\cdot), s_1). \end{aligned}$$

In what follows I will use $events(\delta) \equiv e$ to denote the exported events of a transition and $guards(\delta) \equiv when(\cdot)$ to denote the guard of a transition. Namely, a transition moves the finite state machine from the discrete mode s_0 to the discrete mode s_1. The transition is attempted when the logical formula $when(\cdot)$ evaluates to *true*. The synchronization events are a collection of the exported events that may force this transition to be executed simultaneously with other transitions in other components. The actions , $do(\cdot)$, are a collection of assignments that are executed in parallel when this transition is taken.

Transitions in SHIFT have ASAP (as soon as possible) semantics. This means that a transition will be attempted as soon as its guard evaluates to true. I use the term attempted here to imply that it may not be possible to take the transition due to synchronization constraints.

Synchronization Constraints

Let us assume that there are N instantiated components denoted by the $C = \{c_1, \ldots, c_N\}$. Let each c_i be in a discrete mode s_i. Pick an outgoing transition δ_i from each c_i, and let $\Delta=\{\delta_1, \ldots,\delta_N\}$. Note that some of the δ_i's may be *hidden* transitions. We assume that each discrete mode in every finite state machine has a self loop via a hidden transition. A hidden transition is denoted with $\delta^{\phi}=(s_i, true, \phi, identity(\cdot), s_i)$.

Then, the world transition Δ satisfies synchronization constraints if and only if:

1. There is at least one δ_i in Δ with $\delta_i \neq \delta^{\phi}$. That is, not all transitions are hidden,
2. For each i, $guards(\delta_i) = true$. The world transition is enabled,
3. Let $transitions(c_i)$ denote the set of all transitions specified in the finite state machine associated with component c_i. Let $events(\Delta) = \bigcup_{i=1,\cdots,N} events(\delta_i)$. Then, for each i:
$$events(\delta_i) = events(\Delta) \bigcap \left(\bigcup_{\delta \in transitions(c_i)} events(\delta) \right)$$
Namely, the events taken by transition δ_i are exactly the events taken by the world transition Δ that is of interest to component c_i,
4. Recall that an event may be of two types : open or closed. Say that if an event l is open then $l \in L_{open}$. Then, for each i:
If there is an event $l \in events(\delta_i)$ with $l \in L_{open}$ then there is another transition $\delta_j \in \Delta$ with $l \in events(\delta_j)$,
5. Assume that for some $i \in \{1,\ldots,N\}$ we replace $\delta_i \in \Delta$ with δ^{ϕ} so that we obtain a new world transition Δ'. Then, Δ' cannot satisfy conditions (1), (2), (3) and (4).

Conditions (1) and (2) are trivial. Essentially, (3) is the core of the synchronization logic. Condition (4) is only needed if we have declared some exported events of type **open**. Condition (5) states that the desired synchronization set (of transitions) should be minimal. It is worth mentioning the intuitive difference between open and closed events. Basically, we declare an event as open only when we want to make sure that a transition is never executed alone, that is, there is another transition that "closes" that event.

The formal syntax for describing transitions is very similar to the suggestive notation for a transition given above. Within a **transition** clause we list a series of transitions, each of which is an aggregate of **when** (the guard), **define**, and **do** clauses. The define an do clauses were discussed previously within the context of the setup clause. Their syntax and semantics are exactly the same within the transition clause and hence will not be discussed here. The most tricky part about transitions is in specifying the events of a certain transition. An event is usually specified as `c:l` where `c` is a reference to some component and `l` is the name of the event that is exported by that component. We use the abbreviation `e` instead of `self:e`[8]. However, we may be interested in synchronizing with several components and for this purpose there is syntax that supports set-to-one and set-to-all synchronization. Namely, if `S = { c1, c2, c3 }` is a set of components then the event `S:l(one)` means that this component will execute a synchronized transition on the event l with one and only one of the components `c1, c2` or `c3`. Similarly `S:l(all)` means that this component will execute a synchronized transition on the event `l` with all of the components `c1, c2` and `c3`. These concepts are illustrated in the example *zeno.hs* listed below. The syntax for the transition clause is easier understood than said, thus I will provide a suggestive outline. You should take note from the grammar that it is possible to associate a *quantifier-tag* to the synchronized component in a set-to-one synchronization. It is possible to quantify quantities in SHIFT from a given set. This, however, is discussed later.

The example, *zeno.hs*, listed below is a physical model intended to demonstrate Zeno's paradox. Specifically, we cannot get zeno's paradox because time progresses in integration steps which are assumed to be infinitesimal. However, due to implementation, there is the possibility of zero crossings; i.e. a guard on $x>a$ may not become enable as soon as $x>a$ since it is not evaluated analytically.

transition–clause	$\rightarrow$	**transition** *transition–list* **;**
transition–list	$\rightarrow$	*transition–body* , *transition–list* | *transition–body*
transition–body	$\rightarrow$	*initial–state–identifier* **-** > *final–state–identifier* **{** *synchronization–events* **}** *optional–when–clause* *optional–define–clause* *optional–do–clause*
synchronization–events	$\rightarrow$	*synchronization–event* , *synchronization–events* | *synchronizaiton–event*
synchronizaiton–event	$\rightarrow$	*component–identifier* **:** *exported–event–label* | *component–set–identifier* **:** *exported–event–label* **(one)** | *component–set–identifier* **:** *exported–event–label* **(one :** *quantifier–tag* **)** | *component–set–identifier* **:** *exported–event–label* **(all)**
optional–when–clause	$\rightarrow$	**when** *logical–connective*
optional–define–clause	$\rightarrow$	**define** *{ temporary–variable–definitions–for–use–in–do–clause }*
optional–do–clause	$\rightarrow$	**do** *{ action–list }*

[8] In SHIFT, the keyword `self` is a convenient reference by which a component may refer to itself. The keyword is basically the same as the keyword `this` in C++.

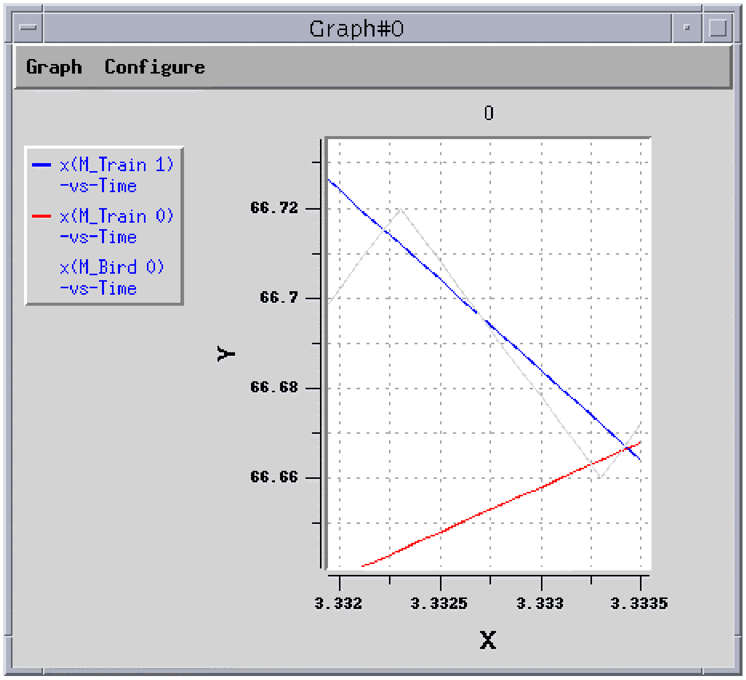
Figure 9: zeno.hs

```
// The physical model of a train in one dimension.
type Train {

  output continuous number x := 0 ; // x coordinate of the train
         number v := 0 ;            // velocity of the train
         set(Train) Trains := {}; // other trains on the same tracks

  export
    collision ; // Event that occurs when trains collide

  flow
    default {

    // The continuous evolution of the system requires
    // that the train moves at a constant velocity.
      x' = v ;
    };

  discrete
    start,  // The initial state of the train
    motion; // The state in which the dynamics
            // of the train is simulated

  transition
    start -> motion {},  // Start moving

    // The simulation ends when an  Train detects a
    // collision with another  Train.  The guard on
    // this transition evaluates to TRUE if and only if
    // there exists another  Train, t,  that is moving
    // in the opposite direction to this  Train and the
    // x coordinate of t has overlapped with the x coordinate
    // of self.
    motion -> exit { collision }
      when (
             exists t in Trains :
                 (x(t) <= x and v > 0)
                  or (x(t) >= x and v < 0)
           ),

    // The simulation may also end when an  Train other
    // than this one has detected a collision.  This transition
    // occurs if and only if another train has detected a collision.
    motion -> exit { Trains:collision(one) };
}

// The model of a bird flying back and forth amongst the  Train's.
type Bird {

  output continuous number x := 0;  // x coordinate of the bird
         number v := 0 ;            // The velocity of the bird
         set(Train) Trains := {}; // The set of trains in the simulation
         number noOfTrips := 0 ;    // The number of trips that the  Bird
```

```
  flow
    default {

      // The bird flies at constant velocity
      x' = v;
    };

  discrete
    start,  // Initial discrete state of the bird
    motion; // The discrete state in which the dynamics of the
            // bird is simulated

  transition
    start -> motion { }, // Start moving

    // The bird changes direction when it crosses path with one
    // of the trains
    motion -> motion{ }
      when (exists t in Trains :
                 (x(t) < x and v(t) < 0 and v > 0)
                  or (x(t) > x and v(t) > 0 and v < 0))
      do {

        noOfTrips := noOfTrips + 1 ; // Increment no. of trips taken
        v := -v;                     // Change direction of travel
      },

    // The simulation ends when the  Train's collide with each other.
    motion -> exit { Trains:collision(one) };
}

// A scenario where there are two  Train heading towards each
// other and an  Bird is travelling back and forth between them.
type Physical_Environment {

  state
    Train Positive_Train;
    Train Negative_Train;
    Bird  bird ;
    set(Train) Trains ;

  setup define {

      // Dynamically create an  Train travelling in the
      // positive x direction and assign its velocity and initial
      // x position.
      Train d_p_Train := create(Train,
                          x := 0,
                          v := 20);

      // Dynamically create an  Train travelling in the
      // negative x direction and assign its velocity and initial
      // x position.
      Train d_n_Train := create(Train,
                          x := 200,
                          v := -40);
```

```
      // position and velocity.
      Bird d_Bird := create(Bird,
                       x := 0,
                       v := 60,
                       noOfTrips := 0);
    } do {

      // Assign the dynamically created travelling in the
      // positive direction to Positive_Train.
      Positive_Train := d_p_Train ;


      // Assign the dynamically created travelling in the
      // negative direction to Negative_Train.
      Negative_Train := d_n_Train ;

      // Assign the dynamically created  Bird to the Bird.
      bird := d_Bird ;

      // Inform the Positive train about the Negative train.
      Trains(Positive_Train) := { d_n_Train };

      // Inform the Negative train about the Positive train.
      Trains(Negative_Train) := { d_p_Train };

      // Inform the bird about the negative and positive trains.
      Trains(bird) := { d_n_Train, d_p_Train };

      // The scenario consists of only the negative and positive
      // trains.
      Trains := { d_n_Train, d_p_Train };
    };

   discrete
     normal ; // The only discrete state of the scenario.

   transition

     // The scenario exits when one of the  Trains
     // detects a collision.
     normal -> exit { Trains:collision(one) };
}


global Physical_Environment scenario
                       :=create(Physical_Environment);
```

Try running this simulation with TkShift with an integration time step of 0.0001. Then, try to get the time-plot of the x coordinates of the Positive_Train, Negative_Train and the Bird. The figure below illustrates the zero crossings.



Operators, constants, built-in functions

There are 3 constants in SHIFT that should be noted. These are **true, false** and **nil**. True and false are logical constants, whereas any object may be nil. A symbolic constant is indicated by prefixing a symbolic name with a `$`; for example, `$red`, `$blue`, `$msg`. Numeric constants can be specified in floating point or exponential format; some examples are `3.14, 1, 8.34e-10`. A constant set is a comma separated list delimited with braces, and a constant array is a comma separated list delimited with brackets. Sets and arrays may be nested. Examples are `{ $red, $blue, $green }, [{1,2}, {5,4}]`.

The basic SHIFT operators and functions are listed below. We take special note to the overloading of the = operator, namely the distinction between actions and flows. Namely, an action, `x := a`, is done via the `:=` lexicon, and the flow, `x = a`, is done via the = lexicon. However, there is one more place where = finds use; in logical connectives. For example, we may say when `x = a` in a transition to mean “attempt this transition when x is equal to a”.

Operator / Function	Meaning
`+ , - , * , / , **`	Standard arithmetic operators on numbers and sets
`< , > , <= , >= , /= , = , not ,`	Standard logical connectives (/= means not equal

`and, or, xor`	to)
`= , <-`	Algebraic equality and algebraic connections
`:=`	Assignment (action)
`+ , - , *`	Union, intersection and difference of sets
`exists x in S : p(x)`	A logical connective that evaluates to true whenever there exists an $x \in S$ such that $p(x)$ is true. S must be a set or array. The temporary variable x need not be declared.

`maxel x in S : f(x)`
 	
`minel x in S : f(x)`
	Returns the element of the set or array S that maximizes or minimizes the expression $f(\cdot)$. The temporary variable x need not be declared. Will not work if S is nil.

`x in S`	Evaluates to true whenever x is a member of the set or array S. x is not a temporary variable here. If S is a set of numbers then x is in S iff there is a y in S such that x-y=0. If S is a set of components then x is in S iff there exists a y in S such that x and y refer to the same component.
`size(S)`	Returns the number of elements of the set or array S.
`exp(x), ln(x), log10(x), sin(x), cos(x), tan(x), sqrt(x), atan(x), atan2(x,y), abs(x), signum(x), max(x,y), min(x,y), random()`	Elementary mathematical functions, random() takes no arguments and returns a number uniformly distributed in [0,1].
`floor(x), trunc(x), round(x)`	Coercion of integers

In addition to these operators there are two other operators that take as arguments type names. These are the **create** and **components** operators: *create(type-name, …)* creates an instance of type-name, *components(type-name)* evaluates to a set consisting of the current components of type (or sub-type) type-name.

The conditional assignments:

`x := if p(a,b) then y else z ;` (action)

`x = if p(a,b) then y else z ;` (flow equality)

are useful. Note that this construct can be nested:

```
x := if p(a,b) then y
        else if q(a,b) then z
```

```
                 else w;

x := if p(a,b) then

                if q(a,b) then x

                 else y

          else z;
```

Sets and Arrays

Sets and arrays are the most powerful data structures in SHIFT. It is possible to quantify elements from a set or array, and they also serve as iteration constructs. In their most general forms, the set and array constructors are:

Set constructor:

```
S := { f(x1,x2,xn) : x1 in S1, x2 in S2, xn in Sn | p(x1,x2,xn) } ;
```

Array constructor:

```
S := [ f(x1,x2,xn) : x1 in S1, x2 in S2, xn in Sn | p(x1,x2,xn) ];
```

These expression are read as follows: Assign to the set all $(x_1, x_2, \ldots , x_n) \in S_1 \times S_2 \times \ldots \times S_n$ such that $p(x_1 ,x_2, \ldots , x_n)$ is *true*. Then, apply $f(\cdot)$ to each element of S. Here, the S_i's are assumed to be either sets, arrays or ranges. Once important fact is that these constructors can be nested, and as such we can construct nested sets and arrays. Note that the predicate p is optional.

The range expression is a convenient way for forming indices. It can be of the following forms:

Expression. ei's, and inc must be numeric constants or numbers. Take note that the two dots are SHIFT syntax, and that there must be at least a single space on both sides of the two dots.	Meaning
`[ e1 .. e2 ]`	Sweep the range *e1 … e2* in ascending order and with increments of 1.
`[ e1 .. e2 : ascending ]`	Sweep the range *e1 … e2* in ascending order and with increments of 1. s of 1.
`[ e1 .. e2 : descending ]`	Sweep the range *e1 … e2* in descending order and with increments of 1.
`[ e1 .. e2 : by inc ]`	Sweep the range *e1 … e2* with increments of *inc*.

It is also possible to combine the options **ascending**, **descending** and **by** as a comma separated list.

Examples

Assume that number n is declared somewhere in the program.

1. `[ 1 .. n ]` evaluates to an array as suggested,

2. `[ 1 .. 10 : by 2 ]` evaluates to the array [1, 3, 5, 7, 9],

3. `{2 * x : x in [1 .. 10 : by 2]}` evaluates to the set { 2, 6, 10, 14, 18 } without the assumed ordering,

Assume that `S:={x : x in [1 .. 3]}` and `A:=[1 .. 3]` are defined somewhere in the program.

1. `{[x,y] : x in S, y in A}` evaluates to the cross product of S and A. The result is arrays nested in sets, i.e. { [1,1], [1,2], [1,3], [2,1], [2,2],[2,3],[3,1],[3,2],[3,3] } without the ordering assumed here,

2. `{[x,y]: x in S,y in A | x <= 2,y >= 2}` evaluates to the set { [1,2], [1,3], [2,2], [2,3] }.

Once sets and arrays are created we need syntax for accessing their elements. Array elements are accessed directly. For example, assume the array `A:=[[ 0,1],[2,3],nil]` is defined, then we may access its elements as follows: `A[0][0]` will be `0`, `A[1][1]` will be `3` and `A[2]` will be `nil`. Note that arrays are zero indexed.

Accessing the elements of a set requires the use of primitive functions. There are 3 basic functions: **find**, **choose** and **count**. Find and choose returns a single element, and count counts the number of elements. These are best understood through examples. (Assume that S and A are as above):

1. `find { x : x in S | x = 2 } default {0}` returns 2,

2. `find { x : x in S | x = -1 } default {10}` returns 10,

3. `choose { x : x in S } default { -1}` returns an arbitrary element of S,

4. `choose { x : x in S | x > 1 }default{ 0 }` returns arbitrarily either 2 or 3,

5. `count { x : x in S }` returns 3,

6. `count { [x,y] : x in S , y in A }` returns 9.

Finally, we discuss quantification from sets. Quantification is normally done through the find and choose operators as illustrated above. However there is another form of quantification that is useful. In this form we quantify components from a set of synchronizing components. The syntax was illustrated in the formal syntax of the transition clause. I conclude this document with the example listed below, *quantify.hs*.

Figure 10: quantify.hs

```
type Circuit {

  state continuous number time;

  discrete
    on { time'= -1; },    // Specify 'on' first to ensure that
    off,                  // all circuits are initially 'on'.
    hysteresis ;

  export
    busted,
    shutdown;

  transition

    // Go into hysteresis when lifetime expires
    on -> hysteresis  { busted }
    when time <= 0,

    // Shutdown when you receive command from the monitor
    hysteresis -> off { shutdown };

  setup
    do {

      time := 100 * random(); // Circuits have a lifetime
                              // uniformly distributed
                              // in [0,100] seconds.
    };
}

type Monitor {

  // Create 100 circuits for simulation
  state set(Circuit) circuits
         := { create(Circuit) : i in [1 .. 100] };

  // A variable that will store a reference to a busted circuit
        Circuit bustedCircuit := nil;

  discrete
    monitor_circuits,
    shutdown_circuit;

  transition

    // Monitor circuits and find one that is busted
    monitor_circuits -> shutdown_circuit { circuits:busted(one:c) }
    when size(circuits) /= 0
     do {

       bustedCircuit := c;           // c is quantified from the
                                     // set of all circuits
       circuits := circuits - { c }; // remove c from the set
                                     // of all circuits
     },
```

```
    // Shut down the busted circuit and then keep monitoring
    // the remaining circuits
    shutdown_circuit -> monitor_circuits { bustedCircuit:shutdown };

}

global Monitor mon := create(Monitor);
```

In this example, a monitor creates 100 circuits, each with a lifetime randomly distributed in [0,100] seconds. The circuits are initially *on*, and they go into *hysteresis* when their lifetime expires. It is the monitors responsibility to find an offending circuit and to turn it off. The monitor synchronizes with one circuit from all the circuits on the event *busted*, it then tags and quantifies that circuit with the temporary reference *c*. This temporary reference need not be declared.