SHIFT
Reference Manual

Revised by: Tunç Şimşek
PRCS project SHIC version 5.59
Wed, 17 Jun 1998 14:50:41 -0700 Tue Jun 2 17:21:54 PDT 1998

Akash R. Deshpande
Aleks Göllü
Luigi Semenzato

1 Overview

SHIFT is a description language for dynamic networks of hybrid systems. Such systems consist of components which can be created, interconnected and destroyed as the system evolves in time. Components exhibit hybrid behavior, consisting of continuous-time phases separated by discrete-event transitions. They may evolve independently or jointly with each other. They interact through input-output connections and synchronization of events. The interaction network may evolve in time.

We believe this model offers the proper level of abstraction for describing complex applications such as highway and air traffic control systems, robotic shopfloors, and other systems whose operation cannot be captured by conventional models, which typically support only static interconnections.

SHIFT is extensible. We plan to use SHIFT as a mechanism for defining the Automated Highway System Tool Interchange Format (AHSTIF). An automated highway system is a hybrid system with specific characteristics and restrictions. SHIFT supports the definition and enforcement of AHSTIF-specific semantics.

This is a reference manual for SHIFT. It presents its material in a logical order, but it aims to be concise and has no pretense to be a SHIFT tutorial. Section 2 describes the model, and section 3 the language.

The name SHIFT is a permutation of HSTIF (Hybrid System Tool Interchange Format).

2 The Model

2.1 Components and Worlds

A *component* is a part of a system, or a *world*. The behavior of a component (its time evolution) depends on its *state*, and its environment (the visible part of other components).

A *world* is an evolving set of components. During the evolution of the world new components may be created, and the manner in which they interact may change. At all times the behavior of the

world is self-contained, that is, it depends only on the state of its components.

2.2 Component Model

A component is defined by its *state*, *inputs*, *outputs*, and *exported events*; its *continuous time evolution*; and its *discrete event evolution*.

At time t, the component's state $s(t)$ is the tuple $(x(t), q(t), l(t))$. The vector $x(t) \in \mathbf{R}^n$ is the continuous state. The discrete state $q(t)$ is a state in a finite state machine. The *link state* $l(t)$ is a vector of references to other components.

Links describe both physical (e.g., wires and mechanical actuators) and logical (e.g., sensors and communication channels) interconnections between components. Links are dynamic: the components they reference may change in time.

A component's inputs, outputs, and exported events define the component's interface to the rest of the world. Outputs are variables whose values are accessible (for reading) by other components. Exported events are state machine transitions which can be synchronized to those of other components. Inputs are variables whose values (during both continuous and discrete phases) are defined externally to the component.

The continuous time evolution of the continuous state x is defined by differential equations and algebraic expressions. The differential equations are in the form

$$\dot{x}_i = f_i^{q(t)}(x(t), u(t), w(t))$$

where $u(t)$ is the input of the component, and $w(t)$ the output of linked components. The algebraic definitions are in the form

$$x_j = g_j^{q(t)}(x(t), u(t), w(t)),$$

with the restriction that algebraic equations are not allowed—that is, there may not be loops in the dependency graph of algebraically-defined states. In a given discrete state q, a state variable must be defined either algebraically or differentially. However, the mode of definition for a variable may change with the discrete state.

The discrete event evolution of a component is defined by a finite state machine. Edges in the state machine are labeled with *guards*, *events* and *actions*. A guard is a predicates on the state of the component, its inputs, and the outputs of other components. A transition on an edge may be taken only when its guard is true. Events are points of synchronizations between components. State machines in linked components synchronize according to *event correspondence rules*, discussed in section 3.15. A transition may trigger the execution of actions, which change the state and output of the component, and create new components.

Associated to each discrete state is an *invariant*: a predicate on the states of a component and the output of linked components, which constrains their region of feasibility. The behavior of the system is undefined outside this region.

2.3 World Model

The world is a directed graph of components, where the labeled edges are *links*. The graph is encoded by the links $l(t)$ of components. Let $a,\ b$ be components with $b \in l_a(t)$, then there is an edge from a to b.

The $l(t)$ evolves in time. A component may change $l(t)$ through *link* or *unlink* actions associated with a transition.

3 The Language

SHIFT is a textual notation for the abstractions in section 2.

3.1 Notation

Non-terminals are in *italics*. Keywords and other literal tokens are in `typewriter`. Braces indicate repetition: $\{ X \}^*$ means zero or more repetitions of X, $\{ X \}^+$ means one or more repetitions. Brackets indicate optional parts, that is $[X]$ stands for zero or one instances of X. The vertical bar ('|') denotes alternation.

3.2 Lexical conventions

A SHIFT specification is a sequence of printable ASCII characters, including space, tab, and newline. The characters are separated into *tokens* according to the rules given below (the rules are similar, but not identical, to those of the C programming language).

- An *identifier* is a sequence of characters from the set { 'a'...'z', 'A'...'Z', '0'...'9', '_' }. The sequence must start with an alphabetic character or '_'. Certain identifiers, called *keywords*, are reserved by SHIFT, meaning they are not available as user-defined names. Table 1 lists all the keywords.

`all`	`count`	`export`	`logical`	`setup`
`array`	`create`	`find`	`maxel`	`state`
`ascending`	`default`	`flow`	`minel`	`symbol`
`by`	`define`	`function`	`number`	`then`
`choose`	`descending`	`global`	`one`	`transition`
`closed`	`discrete`	`if`	`open`	`type`
`components`	`do`	`in`	`out`	`when`
`connect`	`else`	`input`	`output`	
`continuous`	`exists`	`invariant`	`set`	

Table 1: List of SHIFT keywords.

- An *operator* is a sequence of characters from the set { '`+`', '`*`', '`-`', '`/`', '`<`', '`>`', '`=`', '`&`', '`|`', '`'`', '`` ` `` ' }, possibly enclosed in a balanced pair of parentheses, '`(`' and '`)`', '`{`' and '`}`', or '`[`' and '`]`'. For example: `(+)`.

  Note that an operator in SHIFT is different from an operator in C. In SHIFT the character sequence '`x+-3`' contains the tokens '`x`', '`+-`', and '`3`'.

- A *logical-connective* is a predefined operator from the set { '`not`', '`and`', '`or`', '`xor`' }.

- A *numeric-constant* consists of a *mantissa* followed by an optional *exponent*. The mantissa consists of one or more digits and optionally a single '`.`' (period) in any position. The exponent is in the form '`e`|`E`[`+`|`-`]{*digit*}$^+$'.

- A *symbolic-constant* is a sequence of characters starting with `$` followed by an identifier (examples: `$ONE`, `$_TWO`).
- A sequence of characters starting with '`/*`' and ending at the first occurrence of the pair '`*/`' (included) is a comment.
- A sequence of characters starting with '`//`' and ending at the first occurrence of the end-of-line character is a comment.
- Tabs, spaces, newlines, and comments terminate tokens but otherwise have no meaning.
- The characters '`"`', '`!`', '`\`', '`~`', and '`?`' are reserved and may be used internally by SHIFT.
- The characters '`@`', '`#`', '`%`' are guaranteed never to be used by SHIFT (thus they may be given special meanings by a preprocessor).
- The remaining characters are used as specified by the grammar rules in the rest of this document.

3.3 Scopes

Several SHIFT constructs establish a relationship between an identifier (for instance, '`x`') and an entity (for instance, a variable, or a type). The section of text for which the relationship is valid is called the *scope* of the identifier. SHIFT has nested scopes; a scope always starts and ends within another scope. When referring to two such scopes, the former is called the *inner scope*, the latter the *outer scope*. The outermost scope is also called the *global scope*.

This manual defines scoping rules along with each construct which defines a scope. Some rules are valid for all scopes.

- It is illegal to define an identifier twice in the same scope.
- It is permitted to redefine an identifier in an inner scope.
- Scopes are transparent: all meanings of identifiers from the outer scope are also valid in an inner scope, except those of identifiers which are redefined.

Most constructs associating identifiers to entities are called *declarations*. Unlike other languages, SHIFT has no 'declaration-before-use' rule. A declaration is valid for its entire scope, no matter where it appears.

3.4 Preprocessing

SHIFT uses the same preprocessing as the ANSI C language, for the purpose of macro substitution (`#define`), file inclusion (`#include`), and all other available facilities.

3.5 Overall structure

A SHIFT specification is a sequence of *definitions*.

specification ⇒ { *definition* }$^+$
definition ⇒ *component-type-definition*
| *external-type-definition*
| *global-variable-decl*
| *external-function-decl*
| *global-setup-clause*

3.6 Component types

A component type definition describes a set of components with common behavior.

component-type-definition ⇒ `type` *type-name* [`:` *parent*] `{` { *type-clause* `;` }$^+$ `}` [`;`]
type-name ⇒ *identifier*
parent ⇒ *identifier*
type-clause ⇒ `state` *state-declarations*
| `input` *input-declarations*
| `output` *output-declarations*
| `export` *export-declaration-list*
| `setup` *setup-clause*
| `flow` *flow-list*
| `discrete` *discrete-state-list*
| `transition` *transition-list*
state-declarations ⇒ *declaration-list*
input-declarations ⇒ *declaration-list*
output-declarations ⇒ *declaration-list*

The definition '`type` X `:` Y `{ ... }`' defines a component type named X in the global scope. The body of definition (all text between `{` and `}`) is the local scope for X.

Y, if present, must be the name of another component type called the *parent type* of X. The function P maps a type to its parent: in this case, $Y = P(X)$. Type A is a *supertype* of B when $A = P(B)$ or A is a supertype of $P(B)$. In such a case, B is a *subtype* of A.

A type represents a set of components. The statements 'component x has type X' and '$x \in X$' are equivalent.

The parent/child relationship implies certain constraints and inheritance rules between the state, input, output, and export lists of the involved types. These are described in sections 3.20.1 and 3.21.1.

In a type definition there can be multiple clauses of each kind. The items in all clauses of like kinds are concatenated in the order in which they appear, as if there were a single clause containing all of them.

3.7 External types

external-type-definition ⇒ `type` *type-name* [`;`]

The definition '`type` X' defines an external type named X in the global scope. See Section 3.11 for a description of an external type.

3.8 Export lists

export-declaration-list ⇒ *export-declaration* { ; *export-declaration* }*
export-declaration ⇒ *event-type local-event* { , *local-event* }*
local-event ⇒ *identifier*
event-type ⇒ `open`
| `closed`

Export declarations declare local events in the scope of a component type. *Local-event* identifies the event and *event-type* is its type. The *event-type* may be one of { `open`, `closed` }. If no event type is specified then the default is chosed as `closed`.

3.9 Declaration lists

declaration-list ⇒ *declaration* { ; *declaration* }*
declaration ⇒ *type variable-clause* { , *variable-clause* }*
variable-clause ⇒ *variable-name* [*init*]
type ⇒ *simple-type*
| `set` (*type*)
| `array` (*type*)
| *external-type*
variable-name ⇒ *identifier*
init ⇒ `:=` *expression*
simple-type ⇒ `number`
| `continuous number`
| `symbol`
| *type-name*
external-type ⇒ *type-name*

Input, output, and state declarations declare local variables in the scope of a component type. *Variable-name* identifies a variable, and *type* is its type. The initializing expression is evaluated at component creation time, and must be of a compatible type. If a variable is defined algebraically, its initialization is ignored. Variables are initialized to zero or `nil` unless otherwise specified.

3.10 Global Variable Declarations

global-variable-decl ⇒ `global` *declaration-list*

Global variable declaration declare global variable in the global scope of a SHIFT program. The global variable declaration follow the rules of declarations given in Section 3.9.

3.11 Types

Numbers. Variables of `number` type hold real numbers, such as 1 or 2.72. Scalar variables whose type is `continuous number` may be used in the definition of the continuous evolution of a component, as shown in section 3.12.

Symbols. Variables of type symbol hold symbolic constants (section 3.2), such as `$GO` or `$STOP`. Every different symbolic constant represents a distinct value. The only operations available for symbols are assignment and comparison.

Component types. A variable of type X, where X is a component type, is a reference to an element in T_X, that is a component whose type is X or any of its descendants.

Sets. A variable of type `set(`E`)` contains a set of elements of type E. The set might be empty. The expression $\{\}$ evaluates to the empty set.

Arrays. A variable of type `array(`E`)` contains a one-dimensional array of elements of type E. The same variable may hold arrays of different lengths at different times. The elements of the array are numbered consecutively starting from 0. An array might be empty. The expression `[]` evaluates to the empty array.

External types. Variables of the external type X can store values that are returned by the foreign functions of SHIFT. This is a pragmatic approach since a foreign function may return a values that is not of the *simple-types* allowed in SHIFT. Foreign functions are described in Section 3.24).

3.11.1 Links

When the type in a declaration is a component type, or a set or array whose element is a component type, the declared variable is a reference to another component (or a set of components). Such variable, called link variable (or simply link), identifies edges in the link graph.

Let type X define a link variable c with type Y. Then each $x \in X$ has an edge to some $y \in Y$, or to the special component `nil`. The edge is identified by the pair (x, c). In the local scope of X, c is a valid expression, and it refers to y. This is a *single-valued link.*

If c has type `set(`Y`)` or `array(`Y`, ...)`, each $x \in X$ has a set of edges to distinct components in Y. In the case of a set the edges cannot be individually named. This is a *multi-valued link.*

For simplicity, this manual often uses c to refer to the element, or set of elements, currently linked by (x, c).

3.12 Flows

flow-list	⇒	*flow* { , *flow* }*
flow	⇒	*flow-name* `{` *equation-list* `}`
flow-name	⇒	*identifier*
equation-list	⇒	*flow-or-equation* { , *flow-or-equation*}*
flow-or-equation	⇒	*differential-equation*
	|	*algebraic-definition*
	|	*flow-name*
differential-equation	⇒	*differential-equation-lhs* `' =` *expression*
algebraic-definition	⇒	*algebraic-definition-lhs* `=` *expression*
differential-equation-lhs	⇒	*continuous-number-state*
	|	*continuous-number-output*
algebraic-definition-lhs	⇒	*continuous-number-state*
	|	*continuous-number-output*
	|	*variable-name*
continuous-number-state	⇒	*variable-name*
continuous-number-output	⇒	*variable-name*

A *flow* is a named set of differential equations and algebraic definitions, used to define the continuous evolution of one or more variables in the component. *Flow-name* refers to the set of equations in *equation-list*. Its scope is the body of the enclosing component type definition.

Flows define the behavior of state or output variables of type `number`. The left-hand side of an algebraic definition, or a differential equation, is, respectively, one such variable or its derivative. (For convenience, these rules are more relaxed than those of the standard input-output-state model. The only difference between states and outputs is that outputs are visible on the outside, states are not.)

The right-hand sides of both kinds of equations are expressions of all variables of this component (states, inputs, and outputs) and the outputs of linked components.

There may be no circular dependencies in algebraic definitions.

When a *flow-name* appears in place of an equation, it stands for all equations in the corresponding *equation-list*. If a flow defines variable x more than once, all definitions of x except the last one are ignored.

Flows are used in the definition of discrete states, as shown in the next section. Two flow names have special meanings.

- The flow named `default` is the default flow for all states that do not explicitly specify one.
- The flow named `stop` sets the derivative of all variables to zero.

Note that there is a subtle difference between *differential-equation-lhs* and *algebraic-definition-lhs*. That is, differential equations require a continuous number on the left hand side of the equality where as algebraic definitions may equate any valid expression to a variable of compatible type to that expression.

3.13 Discrete states

discrete-state-list ⇒ *discrete-state-clause* { , *discrete-state-clause* }*
discrete-state-clause ⇒ *state-name* [{ *equation-list* }] [`invariant` *expression*]
state-name ⇒ *identifier*

A *state-name* is the name of the state. There must be at least one state in each machine. The first state in the list is the initial state for a newly-created component.

Flow specifies the continuous behavior at the corresponding discrete state, as given in section 3.12. The optional `invariant` expression is expected to be always true.

3.14 Transitions

transition-list ⇒ *transition* { , *transition* }*
transition ⇒ *from-set* `->` *to-state event-list transition-clauses*
from-set ⇒ *set-of-states*
set-of-states ⇒ *expression*
to-state ⇒ *state-name*
event-list ⇒ `{` [*event* { , *event*}*] `}`
event ⇒ *local-event*
| *external-event*
local-event ⇒ *identifier*
external-event ⇒ *link-var* : *exported-event* [(*set-sync-rule*)]
exported-event ⇒ *identifier*
set-sync-rule ⇒ `one` [: *temporary-link*]
| `all`
link-var ⇒ *identifier*
temporary-link ⇒ *identifier*
transition-clauses ⇒ [*when-clause*] [*action-clause*]
when-clause ⇒ `when` *expression*

A *transition* defines one or more edges in the finite state machine of a component type: one edge from each state in the *from* set to the *to-state* state. *Set-of-states* is a constant expression which evaluates to a set of states, or a single state. In this context, the identifier `all` is the set of all states for this machine. See section 3.22 for a complete list of set functions and constructors.

Event is the event associated with this transition. It must appear in the *export-list* of the component. *Exported-event* is a local event in a linked component. The link variable in an exported event may not be algebraically defined. When a transition contains exported events, the state machine of this component potentially synchronizes with the state machines in other components. The synchronization rules are given in section 3.15.

The optional *guard-clause* contains a logical algebraic expression called the *guard*. A transition may be taken only if the guard is true.

If the guard contains an expression with the quantifiers `exists`, `minel`, or `maxel`, the quantified variable defines a temporary link whose scope is the action list for the transition (see section 3.22).

The *action-clause* specifies actions which are taken concurrently with the transition, as described in section 3.17.

3.15 Synchronization rules

A component synchronizes its state machine with other state machines by labeling its own edges with *local-event*s and *external-event*s. Local events are exported; they can be used as external events by other components, and they can appear in *connection actions* (section 3.17.2). Each label of an edge E establishes conditions under which a transition may be taken along E.

When all conditions are satisfied, and the guard, if present, evaluates to true, and the component is in a state that has E as an outgoing edge, then the transition along E is taken simultaneously with other transitions as required by the conditions.

The conditions associated with each label are as follows. Let x and y be components, and Z a set of components. Let `c` be a single-valued link, and `C` a set-valued link. Let e_y be a local event for y, and e_z a local event for all components in Z.

- If `c` evaluates to `nil`, an edge labeled c:e_y may not be taken.
- If `c` evaluates to y, an edge E labeled `c`:e_y must be taken simultaneously with an edge E' labeled e_y in y.
- If e_y is of type `open` then an edge E' labeled e_y in y requires that there exists a component x with an edge E labeled y:e_y and E' must be taken simultaneously with E. However, if e_y is of type `closed` then:
  - if there is no other component x with an edge E labeled c:e_y, where c evaluates to y, then E' may be taken alone.
  - if there is at least one other component x with an edge E labeled c:e_y, where c evaluates to y, then E' must be taken simultanesouly with E.
- If `C` evaluates to the empty set, the edge labeled `C`:e_z may not be taken if *set-sync-rule* is `one`. Otherwise it may be taken.
- If `C` evaluates to Z then an edge labeled e_z in any $z \in Z$ may only be taken simultaneously with an edge labeled `C`:e_z. The following also applies.
  - If the synchronization rule is `one`, then an edge labeled `C`:e_z may only be taken simultaneously with an edge labeled e_z in a single component $z \in Z$. If a temporary link is specified, it is assigned the component z. The scope of the temporary link is the action list for the transition.
  - Otherwise, if the rule is `all`, an edge labeled `C`:e_z must be taken simultaneously with an edge labeled e_z in every $z \in Z$.

3.16 Evolution of a SHIFT system

A SHIFT system starts by executing all initializations of global variables, at time $t = 0$. Then the system evolves by alternating discrete and continuous phases, starting with a discrete phase.

In the discrete mode, all possible transitions are taken, in some serial order unless explicitly synchronized. Time does not flow in the discrete mode. The system switches to continuous mode when no more transitions are possible.

The system evolves in continuous mode according to the flow associated to the discrete state of each component. As soon as it becomes possible for one or more components to execute a transition, time stops again.

3.17 Actions

action-clause	⇒	[*define-clause*] *do-clause*
setup-clause	⇒	[*define-clause*] *do-clause* [*connect-clause*]
define-clause	⇒	`define` { { *local-definition* `;` }* }
local-definition	⇒	*temp-var-declaration* `:=` *expression*
temp-var-declaration	⇒	*type identifier* `:=` *expression*
do-clause	⇒	`do` { { *action* `;` }* }
action	⇒	*reset-action*
connect-clause	⇒	`connect` { { *connect-action* `;` }* }

Actions are used to change the continuous state, create type instances, and change the way in which components are linked.

The `setup` clause in a type definition contains several initializations and operations which are intended to make explicit the input-output connections of the components of a given type instance. This is explained in Section 3.18.

A *local-definition* declares and initializes a variable whose scope is the following local definitions, and the actions in the *do-clause* (and in the *connect-clause*.)

3.17.1 Resets

reset-action ⇒ *selector* `:=` *expression*

Selector refers to a state or output variable of this component, or an input variable of a linked component (it is defined in section 3.22). *Expression* is an expression of the old values (that is, before they are reset) of all variables (input, state, and output) of this component, and output variables of linked components. Resets have no effect on variables which are algebraically defined in the final state of the associated transition.

Note on '`:=`' vs. '`=`'. The notation '*l* `:=` *r*' represents a one-time assignment which occurs during the discrete phase. The notation '*l* `=` *r*' in certain contexts establishes equality of the left and right-hand sides as time flows. In other contexts, '*x* `=` *y*' is a logical expression which evaluates to `true` when *x* is equal to *y*.

3.17.2 Connection actions

connection-action	⇒	*external-event* `<->` *external-event* { `<->` *external-event* }*
	|	*connection*
connection	⇒	*input* `(` *link-var* `)` `<-` *expression*
input	⇒	*identifier*
link-var	⇒	*identifier*

connection actions are only allowed in the `setup` phase of a component's life. They establish static event synchronization and I/O connections for components without their direct involvement.

A *connection* makes the value of the left-hand side be that of the right-hand side at all times. If the right-hand side of a connection is not a continuously-varying number, the left-hand side may not contain continuously-varying numbers. In the *input definition* $u(a) = E$, u must be an input of

component a. The connection is static: if a later changes, the input definition refers to the value of a at setup time. **[What happens when a flow conflicts with a connection?]**

The event synchronization

$$a{:}e \texttt{ <-> } b{:}f$$

specifies that event e in a and event f in b may only occur simultaneously. Event synchronizations are also static, and the synchronized components forever remain those reached through a and b at setup time.

Some useful forms of connections are I/O connections ($u(a) = y(b)$, where y is an output of linked component b), I/I connections ($u(a) = v$, where v is an input of this component), O/I and O/O connections.

There may not be circular dependencies in discrete definitions.

3.17.3 Linking and unlinking

Links are established and removed by resets (section 3.17.1) of link variables.

In a link statement of the form X `:=` Y, X refers to a single or multi-valued link in this component. The action modifies the edge, or edges, named by X.

When X is a single-valued link, the action removes the existing link and adds a new one, from this component to the component obtained by evaluating Y (possibly to the `nil` component).

When X is a multi-valued link, the action can add or remove edges from the set, or leave it unchanged.

3.17.4 Execution of actions

The order in which actions are specified is inconsequential. Actions are executed in phases as follows.

1. All components specified by *create-expression*s are created.
2. The right-hand sides and the destinations of resets are evaluated, and so are the component initializers.
3. The previously computed values for resets and link actions and component initial values are assigned to their destinations.
4. connection actions are executed.

Definitions in the *define-clause* are evaluated in the order in which they are given. The execution sequence for the *create-expression* is: initial values, passed in arguments, define/do actions.

3.18 Setup clause

If a `setup` clause is present, its actions are executed before the finite-state machine enters the initial state. If the initial state is S_0, this is equivalent to creating an additional state S_{-1}, making it the initial state, and placing an unguarded edge from S_{-1} to S_0 with the `setup` actions. Section 3.17 lists all possible actions. The *connect-clause* of section 3.17.2) may only appear in the `setup` clause.

3.19 Global Setup Clause

global-setup-clause $\Rightarrow$ *setup-clause*

A *global-setup-clause* is analogous to the *setup-clause* of a component. Its actions are executed at the beginning of the execution of a SHIFT program right after the *global-variable-declarations* are evaluated.

3.20 Input, output, and export declarations

Variables declared in an output list may be used outside a component. They may not be reset or defined outside the component.

The input list declares local variables which can only be defined and reset from outside the component.

The export declares local events of a type. They are all visible to other components.

3.20.1 Inheritance/Subtyping of export variables

If X is the parent of Y, the export, input, and output lists of Y must be supersets of, respectively, the export, input, and output lists of X. The same name in two input or output lists must refer to the same kind of object.

Inheritance is an implemenation technique whereby the user is spared the effort of duplicating code. Inheritance and subtyping in SHIFT are implemented together and the inheritance features of SHIFT supports the subtyping restrictions specified above. If X is the the parent of Y, the export, input and output lists of X are handled respectively in the export, input and output lists of Y according to:

1. A variable a declared in X and redeclared in Y is processed according to the following rules:
   - If $a(X)$ is a variable declared in either one of the export, input or output lists of X then it must be a subtype of $a(Y)$.
   - If $a(X)$ is a variable delared in either the input or output list of X then it must be of the same kind as $a(Y)$.
2. A variable a declared in X and not declared in Y is inherited according to the following rules:
   - A new variable is instantiated in the scope of Y and given the same name, a. The scoping rules for the newly created variable obey the rules given above in section 3.20 and in section 3.3.
   - The initialization expressions of $a(X)$, if any exists, are also valid for the inherited variable.
3. Wide subtyping where variable a declared in Y and not declared in X is allowed, there are no restrictions.

3.21 State declarations

Variables declared in a state list may not be used and are not visible outside a component. They cannot be reset or defined outside a component with the exception of the creation expression, section 3.22.1.

3.21.1 Inheritance/Subtyping of state variables

State variables do not honor the inheritance mechanism. If X is an ancestor of Y the state lists of X are independent of the state lists of Y; the state lists of X are not inherited to the state lists of Y and any redeclaration of state variables in Y (overloading) is allowed.

3.22 Expressions

expression ⇒ *selector*
| *numeric-constant*
| *symbolic-constant*
| `true`
| `false`
| `nil`
| *create-expression*
| *components-of-type-expression*
| *expression binary-operator expression*
| *prefix-operator expression*
| *expression postfix-operator*
| *expression* `(` [*expression-list*] `)`
| *expression* `[` *expression*]
| `(` *expression* `)`
| *type*
| `{` [*expression-list*] `}`
| *set-former-expression*
| `[` [*expression-list*]]
| *array-former-expression*
| `all`
| `self`
| *state-name*
| `if` *expression* `then` *expression* `else` *expression*
| `exists` *identifier* `in` *expression* `:` *expression*
| `minel` *identifier* `in` *expression* `:` *expression*
| `maxel` *identifier* `in` *expression* `:` *expression*

selector ⇒ *continuous-selector*
| *link-selector*

continuous-selector ⇒ *number-var*
| *number-var* `(` *link-selector* `)`

link-selector ⇒ *link-var*
| *link-var* `(` *link-selector* `)`

expression-list ⇒ *expression* [`,` *expression-list*]

prefix-operator ⇒ *operator*
| *logical-connective*

postfix-operator ⇒ *operator*
| *logical-connective*

The indexing expression a`[`i`]` accesses the $(i+1)$-th element of array a.

The expression `{` e_1, . . ., e_n `}` evaluates to a set containing the elements $e_1, \ldots, e_n$, which must all be of the same type. Similarly, the expression `[` e_0, . . ., e_{n-1} `]` evaluates to an array of length n.

The expression `all` evaluates to the set of all states in the state machine of this type.

The expression `self` is a self-link, that is a reference to the component containing the expression.

The expression `nil` is a special component whose behavior is absolutely boring: it has no inputs, outputs, or exported events. It may be assigned to links of any type.

The logical expression `if` x `then` y `else` z evaluates to y if x is true, else it evaluates to z.

Grammar ambiguities are resolved by operator precedence. **[need precedence table here]**

3.22.1 Creation

create-expression ⇒ `create (` *type-name* { `,` *initializer* }* `)`
initializer ⇒ *variable* `:=` *expression*
variable ⇒ *identifier*

The `create (`C`, ...)` command creates a new component of type C. The left-hand side of an *initializer* is a variable (not just an input) in the new component. The initializer sets this variable to the value of the right-hand side, overriding the initial value of the variable in the declaration of the new component.

The scopes of the left and right-hand sides of an initializer differ. Thus the action `create(C,` v `:=` v`)` sets variable v in the new component to the value of variable v in the creating component.

The setup actions of the new component are executed after completion of the transition that created it.

3.22.2 Dynamic Type Evaluation

components-of-type-expression ⇒ `components (` *type-name* `)`

The `components (`C`)` command evaluates to a set containing all components compatible with type C (i.e. those components that are of type C or in the inheritance chain of type C) that exist at the time of its evaluation. Note that this feature can be used to dynamically check the type of a component: the logical expression

c `in components( T)`

evaluates to `true` if and only if c if compatible with type T.

3.22.3 Complex Set and Array Constructors

SHIFT provides more complex ways to built arrays and sets than the simple enumeration constructs listed in the *expression* rule. The set and array formers are modeled after the SETL language [1] constructs and, as such, provide a constrained, yet powerful form of "loop" within SHIFT.

The general form of a set former is the following:

`{` $F(x_1, x_2, \ldots, x_n)$ `:` x_1 `in` e_1`,` x_2 `in` e_2`,` $\ldots x_n$ `in` e_n`, [` | $P(x_1, x_2, \ldots, x_n)$ `] }`

The general array former expression is exactly the same with the outermost `{}` replaced by `[]`.

The set (resp. array) former expression evaluates to a set containing all the elements resulting from the evaluation of $F(x_1, x_2, \ldots, x_n)$ with the "free" variables x_1, x_2, $\ldots x_n$ bound in sequence in a nested way to the elements of the expressions *expression*$_i$. The result of the evaluation of F will appear in the final result if and only if the optional condition P evaluates to true (the default in the case P were missing).

Obviously the *expression*'s e_i are assumed to be either sets or arrays, or a new kind of expression called a *range*.

Here are a few examples of the use of these set and array former expression. Their full grammar will be given afterward.

Set/Array Former Example 1: Given an array of numbers, `[1, 2, 5, 12]` held in a global variable `A` of type `array(number)`, build a set containing their doubles and put it into a variable `B` of the same type.

```
B := {x * 2 :  x in A};
```

The result is to assign to `B` the value `{2, 4, 10, 24}`. Note that if the optional condition P is specified then we can obtain a different behavior.

```
B := {x * 2 :  x in A | x < 10};
```

The result stored in `B` will now be `{2, 4, 10}`.

Set/Array Former Example 2: Multiple indices. We can also write something like:

```
C := {x + y :  x in A, y in B | y > 3};
```

To understand this example, consider the Cartesian product of the *abstract* sets $A \equiv \{1,2,5,12\}$ and $B \equiv \{2,4,10\}$.

$$A \times B \equiv \{\{1,2\},\{1,4\},\ldots,\{12,2\},\{12,4\},\{12,10\}\}.$$

The condition `y > 3` will prevent all the couples with '2' as second element from appearing in the result `C`[1]

Loop Like Constructs Feedback from users of early versions of SHIFT, pointed out that in many cases it might be useful to scan an array in an orderly way. Also it would be useful to construct sets and arrays without relying on the existence of previously constructed intermediate results. Finally, there are efficiency considerations to be taken into account.

To address these problems SHIFT introduces the notion of *range*: a special form expression which may appear only at the right of the `in` keyword. As an example, let's build the set of all the numbers from 0 to 1000.

```
N_1_1000 := {x :  x in [0 ..  1000]}
```

Therefore the *range* expression constrains a variable to vary between the bounds. Moreover, the default is to step between the bounds in increments equal to 1 and in ascending order. This may be unsatisfactory for some applications, so the concept of *range* is extended in such a way to make it more "usable" from a programming language point of view.
The general form of the *range* descriptor is the following:

`[` e_1 `..` e_2 `[ :` *range-options* `] ]`

where e_1 and e_2 must be numeric expressions and *range-options* is either one of the following forms separated by commas (`,`).

[1]The implementation actually takes care of the generation of these intermediate Cartesian products, which should immediately make the reader wary of the potential for the construction of very complex algorithms which are also very costly in both time and space.

- `by` *step-expression* i.e. the numeric value of *step-expression* will be used as the "increment".
- `ascending` the range will be swept in "ascending" order.
- `descending` the range will be swept in "descending" order[2].

Set/Array Former Example 3: generating an array of numbers spaced 2.6 units form each other in the range [2.0, 36.7].

```
[i :  i in [2.0 ..  36.7 :  by 2.6]]
```

the result is

```
[2.0 4.6 7.2 9.8 12.4 15.0 17.6 20.2 22.8 25.4 28.0 30.6 33.2 35.8]
```

Range Example 1: `exists` operator. Of course it is also possible to use the range operator in conjunction with the `exists` operator and friends. In the following we will find the element of the array of instances (`IA`) which has a certain output (`out_value`) above a given threshold.

```
exists i in [0 ..  size(IA)] : out_value(IA[i]) > threshold;
```

if such an item exists in the array, the `exists` form will evaluate to *true* and `i` will be bound to the index of the first element of `IA` satisfying the condition.

Grammar The grammar of set and array formers is the following.

set-former-expression ⇒ `{` *expression* `:` *in-exprs* [`|` *expression*] `}`

array-former-expression ⇒ `[` *expression* `:` *in-exprs* [`|` *expression*] `]`

in-exprs ⇒ *in-expr* [`,` *in-exprs*]

in-expr ⇒ *identifier* `in` *aggregate-expression*

aggregate-expression ⇒ *expression*
| *range*

range ⇒ `[` *expression* `..` *expression* [`:` *step-options*] `]`

step-options ⇒ *step-option* [`,` *step-options*]

step-option ⇒ `ascending`
| `descending`
| `by` *expression*
| `discrete`
| `continuous`

Rationale These constructs were chosen because they do not interfere with the basic SHIFT 'RHS-first' rule of evaluation of actions. More traditional "loop" constructs were considered, but they all would have required much more implementation work without the semantics guarantees that the chosen constructs offer instead. Also, a "higher" order approach could have been chosen by introducing a `map` function á là LISP. However, this would have meant to introduce a notation to denote anonymous function which would have complicated the language in an unwanted way.

Lastly, there is also the argument that the adopted constructs have a very high level *mathematical* "look-n-feel", which should make many of SHIFT targeted users quite comfortable.

3.22.4 Other Set Related Operations

In order to use sets and arrays in a more convenient way, SHIFT provides some more operators to essentially perform the following operations.

- *finding* an element in a set or array.
- *counting* how many elements in a set or array satisfy a certain property.
- *choosing* an arbitrary element from a set[3]

These three operations are built syntactically in a similar way and are very expressive. They rely on the set constructor syntax in order to achieve their aim. Moreover, care has been taken to provide the user with a way to return a *default* value should the main operation fail.

Here are e few examples given the set

```
set(number) S := {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

Example: finding an element in a set.

```
number five := find {x :  x in S | x = 5};
```

Example: finding an element in a set whose square is greater than 75.

```
number squared := find {x :  x in S | pow(x, 2) > 75};
```

Example: finding an element in a set whose square is greater than 100, and if none is found return the default value 350.

```
number squared := find {x :  x in S | pow(x, 2) > 100} default { 350 } ;
```

The result will be 350.

Example: counting how many elements in a set have a square greater than 75.

```
number n_squared := count {x :  x in S | pow(x, 2) > 75};
```

The result will be 2.

Example: choosing an arbitrary element from a set.

```
number any_number := choose { x :  x in S };
```

[3]This is really a very high level operation. But it turned out to be very easy to add.

Example: choosing an arbitrary element from a set and squaring it.

```
number any_number := choose { pow(x, 2) :  x in S };
```

Example: choosing an arbitrary element from a set of numbers greater than 3.4 and squaring it.

```
number any_number := choose { pow(x, 2) :  x in S | x > 3.4 };
```

Grammar the grammar for these three *special forms* is the following.

special-form ⇒ `find` *set-former-expression default-option*
| `count` *set-former-expression*
| `choose` *set-former-expression default-option*

default-option ⇒ *empty*
| `default {` *expression* `}`

Implementation and Efficiency Notes. The three special forms are provided in order to facilitate the writing of SHIFTprograms. However, their resemblance to the standard set formers is only apparent in the case of `find` and `count`. These two forms do not allocate any intermediate data structure, and are therefore very efficient. On the other hand, `choose` does allocate an intermediate set of those element satisfying the filtering condition. As such it is more memory intensive than the other two special forms.

3.23 Predefined functions and operators

The predefined functions and operators are listed and explained in tables 2—4. Table 2 gives the standard functions on sets and logical values. Table 3 gives the arithmetic and mathematical operators. Table 4 lists miscellaneous operators.

3.24 Foreign functions

external-function-decl ⇒ `function` *function-name* `(` [*arg-list*] `)` `->` *return-type*
arg-list ⇒ *function-declaration-list*
return-type ⇒ *type*

function-declaration-list ⇒ *empty*
| *function-declaration-clause*
| *function-declaration-list* ; *function-declaration-clause*

function-declaration-clause ⇒ *type* [(`in` | `out`)] *identifier*

SHIFT does not have functions, but a SHIFT program can refer to external functions, whose implementation must be provided for the purpose of simulation. External functions are written in C. The parameter passing scheme is modeled after Ada with `in` and `out` parameters which imply a "copy in", "copy out" semantics.

The implementation of the simulator imposes further restrictions on the kind of SHIFT types that can be passed in and out a function.

Expression	Meaning
`exists` x `in` S : E	A logical expression which evaluates to `true` if and only if the set S contains at least one element which, when bound to x, causes the expression E to be true. If an `exists` expression appears in a guard, the scope of x includes the actions for that transition, where it is bound to one of the components which satisfy the guard.
`maxel` x `in` S : E `minel` x `in` S : E	The element of S which, when bound to x, respectively maximizes or minimizes expression E.
S_1 `+` S_2, S_1 `*` S_2, S_1 `-` S_2	Respectively the union, intersection, and difference of sets S_1 and S_2.
x `in` S	A logical expression which is `true` if x is a member of S.
`reduce`(S, f) `reduce`(S, f, e_0)	If $S = \{e_1, \ldots, e_n\}$, `reduce`(S, f) returns $f(e_n, f(\ldots, f(e_2, e_1)\ldots))$, and `reduce`$(S, f, e_0)$ returns $f(e_n, f(\ldots, f(e_1, e_0)\ldots))$. f is a binary function or a binary operator enclosed in double quotes (e.g., `"*"`).
`size`(S)	The number of elements in set S.
`and`, `or`, `xor`, `not`	The standard logical connectives.

Table 2: Set and logical operators.

function arguments arguments can be of type `number`, `array` or *external-type*.

return value the return value is also limited to `number`, `array` or *external-type*.

The C function which actually implements the SHIFT function must have the same name and use the following conversions:

SHIFT	C
`number`	`double`
`array(number)`	`double*`
`array(array(number))`	`double**`

There is no general conversion machinery for higher rank arrays. Moreover, a formal `array` argument declared `out` is modified *only in its contents*. Ranks and dimensions are not affected. Therefore, the array must have been allocated on the SHIFT side before it can be effectively used in a foreign function call.

3.24.1 Implementation

The implementation is straightforward. Each `function` declaration is translate into a *wrapper* C function which takes care of copying in and out the actual arguments.

Expression	Meaning
`+`, `-`, `*`, `/`, `**` `<`, `>`, `<=`, `>=`, `=`, `/=`	The standard arithmetic and relational operators (`/=` is "not equal.")
lhs `+:=` *rhs*, *lhs* `-:=` *rhs*	Increment and decrement assignment operators. Currently implemented only for set insertion and removal: i.e. *lhs* must be a set valued location and *rhs* must be either • a value of the base type of the set, in which case the object is added or removed from the set. • a set of values of the base type of the set, in which case all its elements are added from the *lhs* set valued side.

`exp(`x`)`, `ln(`x`)`, `log10(`x`)`
`sin(`x`)`, `cos(`x`)`, `tan(`x`)`
`sqrt(`x`)`
`atan(`x`)`, `atan2(`x, y`)`
	The standard elementary mathematical functions.

`abs(`x`)`	Absolute value.
`floor(`x`)`, `trunc(`x`)`, `round(`x`)`,	Coercion to integers. Floor, trunc, and round produce results rounded toward $-\infty$, toward 0, and toward nearest.
`max(`$x_1, \ldots, x_n$`)` `min(`$x_1, \ldots, x_n$`)`	Maximum and minimum.
`signum(`x`)`	Returns -1, 0, or 1, depending on whether x is negative, zero, or positive.
`random()`	`random` requires no arguments and returns a random number uniformly distributed in [0, 1].

Table 3: Arithmetic Operators and Elementary Functions

3.24.2 Rationale

The rationale of the foreign function interface is to provide a way to use C functions in SHIFT simulations with as little overhead as possible on the C programmer. The functionality is far from complete and comprehensive, but it provides a minimum of functionality that extends the usefulness of SHIFT by allowing the reuse of standard C library functions.

3.25 Global variables

global-variable-decl $\Rightarrow$ `global` *declaration-list*

Global variables have global scope, can be used in expressions, and can be set by any component.

Expression	Meaning
`narrow(`X`, `y`)`	The compile-time type of `narrow(`X`, `y`)` is X. Let Y be the compile-time type of y. If Y is X or a supertype of X, and the run-time value of y has type X or a subtype of X, then `narrow(`X`, `y`)` returns y, otherwise it is an error.

Table 4: Miscellaneous Operators

References

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

Acknowledgements

Most of the ideas in this manual are the result of discussions involving the authors and other PATH members. In particular we have greatly benefited from discussions with Datta Godbole, Raja Sengupta and Pravin Varaiya.