

Table des matières

1	Introduction.....	3
2	Installation.....	3
2.1	Prerequisite.....	3
2.2	Structure of the installation.....	4
2.3	Linux installation.....	4
2.4	Windows/cygwin installation.....	4
3	BIP language.....	4
3.1	Introduction.....	4
3.2	Basic Language Elements.....	5
3.2.1	Identifiers.....	5
3.2.2	Reserved Words.....	5
3.2.3	Comments.....	5
3.2.4	Special Pragma.....	6
3.3	BIP types.....	6
3.3.1	port type.....	6
3.3.1.1	description.....	6
3.3.1.2	example.....	7
3.3.2	atomic type.....	7
3.3.2.1	description.....	7
3.3.2.2	syntax.....	7
3.3.3	compound type.....	7
3.3.3.1	description.....	7
3.3.3.2	syntax.....	8
3.3.4	connector type.....	8
3.3.4.1	description.....	8
3.3.4.2	syntax.....	8
3.4	Atomic component description.....	9
3.4.1	port.....	9
3.4.1.1	description.....	9
3.4.1.2	syntax.....	9
3.4.1.3	example.....	9
3.4.2	variable.....	9
3.4.2.1	Description.....	9
3.4.2.2	syntax.....	10
3.4.2.3	example.....	10
3.4.3	place.....	10
3.4.3.1	Description.....	10
3.4.3.2	syntax.....	10
3.4.3.3	example.....	10
3.4.4	initial block.....	10
3.4.4.1	Description.....	10
3.4.4.2	syntax.....	10
3.4.4.3	example.....	10
3.4.5	transition.....	11

3.4.5.1	Description.....	11
3.4.5.2	syntax.....	11
3.4.5.3	example.....	11
3.4.6	Priority in atom.....	12
3.4.6.1	Description.....	12
3.4.6.2	syntax.....	12
3.4.7	export port in atomic component	12
3.4.7.1	Description.....	12
3.4.7.2	syntax.....	12
3.4.8	atom type example.....	12
3.5	Compound component description.....	13
3.5.1	component instance.....	13
3.5.1.1	Description.....	13
3.5.1.2	syntax.....	14
3.5.1.3	example.....	14
3.5.2	connector instance.....	14
3.5.2.1	Description.....	14
3.5.2.2	syntax.....	14
3.5.2.3	Structured Connector.....	14
3.5.2.4	Example.....	15
3.5.3	priority in compound.....	15
3.5.3.1	description.....	15
3.5.3.2	syntax.....	15
3.5.4	export port in compound component	15
3.5.4.1	Description.....	15
3.5.4.2	syntax.....	16
3.5.4.3	example.....	16
3.6	Connector description.....	17
3.6.1	definition.....	17
3.6.1.1	description.....	17
3.6.1.2	syntax.....	17
3.6.2	variables.....	17
3.6.2.1	Description.....	17
3.6.2.2	syntax.....	17
3.6.3	interactions.....	18
3.6.3.1	Description.....	18
3.6.3.2	syntax.....	18
3.6.4	data transfer.....	18
3.6.4.1	Description.....	18
3.6.4.2	syntax.....	18
3.6.5	export port in connector	18
3.6.5.1	Description.....	18
3.6.5.2	syntax.....	18
3.6.6	example.....	19
3.7	Actions.....	20
3.7.1.1	Description.....	20
3.7.1.2	Syntax.....	20
3.8	Expressions.....	21

3.8.1.1	Description.....	21
3.8.1.2	Syntax.....	21
3.9	Array instance.....	21
3.9.1.1	description.....	21
	Array of Component.....	22
	Array of Connector.....	22
3.10	Timed models.....	22
3.10.1.1	Description.....	22
3.10.1.2	Urgency semantics.....	23
	Eager.....	23
	Delayable.....	23
	Lazy.....	23
3.10.1.3	Recommendation.....	23
3.10.1.4	Syntax.....	24
3.11	Package and system.....	24
3.11.1	package declaration	24
	description.....	24
3.11.1.1	syntax.....	25
3.11.2	system declaration.....	25
3.11.2.1	description.....	25
3.11.2.2	syntax.....	25
3.11.3	The use policy.....	25
3.11.3.1	description.....	25
3.11.3.2	syntax.....	25
3.11.4	the root definition.....	25
3.11.4.1	Description.....	25
3.11.4.2	Syntax.....	26
3.12	Syntax summary.....	26
4	Engine feature.....	29
4.1	Running an example.....	29
4.1.1	Analyze source and generate code.....	29
4.1.2	Link with the engine.....	30
4.1.3	Run the example.....	30
4.1.4	The interactive mode.....	31
4.1.4.1	example.....	32

1 INTRODUCTION

2 INSTALLATION

2.1 Prerequisite

The toolset is available for Linux (Debian family) PC's.

The executable need java 1.4 (or above) and gcc 4.x.

The BIP parser may works on command line with a java 1.4 or upper, or under eclipse version 3.1 or upper.

The BIP command line only is available under Windows/cygwin.

2.2 Structure of the installation

The archive obtained in the download page on extraction gives the following structure under folder BIP:

- bin contains the BIP to C++ code generator command line
- plugins contains specific BIP plugins
- importedplugins contains the libraries needed for BIP
- include contains BIP specific header files
- lib contains the BIP engine library
- util contains the Makfile for building the application
- examples contains the prodcons example
- aadl contains library used by AADL to BIP translation

2.3 Linux installation

- Fill the BIP2 load fom at the address :

<http://www-verimag.imag.fr/~async/bip2Download.php>

and follow instructions to load the archive.

- Untar the archive at the installation place.
- Define the environment variable BIP2_HOME to the expended directory.
- to use BIP in eclipse environment, copy the plugins of directories "importedplugins" (if not already installed) and "plugins" to the eclipse plugin directory. Restart your eclipse environment, and define the BIP2_HOME in the BIP preference window.

2.4 Windows/cygwin installation

BIP platform is not totally enable on windows/cygwin system.

3 BIP LANGUAGE

3.1 Introduction

The BIP language provides syntactic constructs for describing systems . It is a co-ordination language, and is an extension of the C programming language. It leverages on C style variables and data type declarations, expressions and

statements, and provides additional structural syntactic constructs for defining component behavior, specifying the co ordination between components and describing the priorities. The language enables us to express the concurrent and sequential behavior of systems, as an interconnection of components. A system can be described hierarchically, and timing can also be explicitly modeled in the same description.

The principal constructs are:

- port: to specify data associated with a port type
- atom: to specify behavior, with an interface consisting of ports. Behavior is described as a set of transition. Transitions are labeled by ports.
- connector: to specify the co-ordinations between the ports of components, and the associated guarded actions.
- priority: to restrict the possible interactions, based on conditions depending on the state of the integrated components.
- compound: to specify systems hierarchically, from other atoms or compounds, with connectors and priorities.
- package : to specify reusable BIP types.
- model : to specify the entire system, encapsulating the definition of the components, and specify the top level instance of the system.

The language allows defining types for port, atom, connector, and compound, defining priorities, and instantiating objects of the defined types. Every instantiated object has a scope. The scope in a BIP description are:

- atom
- compound
- model (global)
- package

3.2 Basic Language Elements

3.2.1 Identifiers

An identifier is composed of a sequence of one or more characters. A legal character is an upper case letter (A . . . Z), a lower-case letter (a . . . z), a digit (0 . . . 9) or the underscore (_) character. The first character of an identifier may be a letter. Identifiers are case sensitive.

3.2.2 Reserved Words

The following identifiers are reserved words in the language (also called keywords), and therefore cannot be used as basic identifiers in a BIP description.

atomic	component	compound	connector	data	define
delayable	do	down	eager	else	end
export	is	extern	from	header	if
in	initial	lazy	model	priority	use
multishot	package	place	on	port	type
provided	singleshot	timed	to		
up					

3.2.3 Comments

The language supports C style comments. Single line comments begin with two consecutive front slashes (//) and extends until the end of the line. Multi-line comments can be enclosed between a pair of /* and */. Examples are:

```
// This is a single line comment
/* This is a
multi line
comment */
```

3.2.4 Special Pragma

A description enclosed within the character pairs {# and #} is treated specially for code

generation purpose. It acts as a special directive for the BIP compiler to pass the description as it is to the code generator. This is useful for enclosing standard C header files, arbitrary C type and function definition, or to specify any arbitrary C code. For example:

```
{# #include<stdarg.h>
typedef char* String; #}
```

3.3 BIP types

3.3.1 port type

3.3.1.1 description

Ports define the interaction points between components. They are used to synchronize components, and have associated data.

Ports in BIP are typed. The language allows the definition of typed ports, i.e., ports associated with typed variables. A port type definition is characterized by the number the name and type of its associated variables. These data, and only these ones, will be accessible in the data transfer defined in connectors using the ports. The port type is used

- to declare ports in atomic component,
- to define a connector type profile
- to export a port from a component.

A default port type named “Port” without any associated data is predeclared.

syntax

```
<port type> ::= port type <port type name> ( [<data parameter>]
                                                (<data parameter>)* )
```

```
<data parameter> ::= <data type> <data name>
```

<port type name> ::= <identifier>
<data type> ::= <identifier>
<data name> ::= <identifier>

3.3.1.2 example

The intPort declare a port type associated with an integer variable accessible with the name x.

```
port type intPort(int x)
```

3.3.2 atomic type

3.3.2.1 description

Atomic components are the leaves of the components hierarchy. Their types define the interaction points between them and their environment. An atomic type defines an atom. Atoms are the leafs in the component hierarchy. An atomic type is characterized by :

- its name
- the exported ports (name and type)
- the parameters.

The atomic type defines also the component behavior as an automaton or a Petri net.

3.3.2.2 syntax

```
<atomic type> ::= atomic type <atomic type name> [<component parameters>]  
                [<component_activation>]  
                [<header code>]  
                [<body code>]  
                ( <data definition> | <port definition> | <place definition> )+  
                [<initial block>]  
                (<transition definition> | <priority definition> )+  
                ( <export definition> )*  
                end
```

<atomic type name> ::= <identifier>

<component activation> ::= **singleshot** | **multishot**

3.3.3 compound type

3.3.3.1 description

A compound type defines a new component type from existing components (atom

or compound)

by creating their instances, instantiating connectors between them and specifying the priorities.

Hence it defines a structural model from existing components. A compound offers the same

interface as an atom, hence externally, there is no difference between a compound and an atom.

Inner ports can be exported, to complete the component profile. Inner ports that are not exported are not visible elsewhere.

3.3.3.2 *syntax*

<compound type> ::= **compound type** <compound type name> [<component parameters>]

[<component_activation>]

[<header code>]

[<body code>]

definition>)+ (<component definition> | <connector definition> | <priority

<export definition>)*

end

<compound type name> ::= <identifier>

3.3.4 **connector type**

3.3.4.1 *description*

Connectors define the synchronization protocol (rendez-vous, broadcast, ...) between ports, and the data transfers associated with them. The connector type is characterized by :

- its name
- the support set of ports (name and type)
- the exported port, if any (used to build hierarchical connectors)

3.3.4.2 *syntax*

<connector type definition> ::= **connector type** <connector type name>

<port type profile> [<data type profile>]

[<header code>]

[<body code>]

<define connector type>

< export connector port definition>

end

<define connector type> ::= <define connector expression>
(<data definition> | <interaction definition>)*

3.4 Atomic component description

3.4.1 port

3.4.1.1 description

Ports define events which trigger transitions between two states. Implicitly, a port is internal, i.e., representing an internal action which does not need explicit synchronization for its triggering. However, a port may be exported by the keyword `export` to make it visible in the component interface. Exporting a port has an option to specify the exported name of the port, i.e., a name by which a port is known in the component interface. Moreover, internal ports are by default priority than exported ports.

3.4.1.2 syntax

<port definition> ::= **export port** <port type reference> <port name> [<data port>]
[= <export name>]

<data port> ::= (<data name> (,<data name>)*)

<export name> ::= <identifier>

<port type reference> ::= [<library name> .] <port type name>

3.4.1.3 example

For example, the port `p` associated with an integer variable `x` can be exported with a name `out` as:

```
export port intPort p(x) = out
```

3.4.2 variable

3.4.2.1 Description

Variables used in an atom are declared as data objects. They are typed, as in the C language. They may have initial values. A variable may be qualified as `extern` if its definition is in an opaque C code. A variable may be qualified as `timed`, when it is supposed to be used as a clock. A variable may be exported to be visible outside of the component (in particular in priority definition).

Data variables are defined by their name and type. Additional information may be added on the variable:

- time variables (which evaluate with the time progress)
- extern variable: declared elsewhere in C code


```
initial to idle do x=0;
```

3.4.5 transition

3.4.5.1 Description

The behavior is given by a set of transitions modeling atomic computation steps.

A transition is specified by a label (a port name) following the keyword **on**. The source(s) of

a transition follow the keyword **from**, while the destination(s) follows to. The guard of the

transition is specified after the **provided** keyword. The guard has two parts, an untimed guard and a timed guard. The untimed guard is a boolean expression of the atomic variables. Timed

guards are useful for modeling timed systems. A timed guard is an expression of the timed

variables of the atom. Timed guards are elaborated in section 3.11. The guard of a transition is

optional, and if not provided, it is assumed to be true. The action statements of the transition

are specified following the **do**.

3.4.5.2 syntax

```
<transition definition> ::= on port_expression from <places> to <places>  
                           [provided <expression>]  
                           [<timed condition>]  
                           [ do <action> ]
```

```
<places> ::= <place name> (, <place name>)*
```

3.4.5.3 example

The action statements are C statements. An example of a transition from the place empty to full is

```
on in  
  from empty to full  
  provided 0<x  
  do y=f(x);
```

We assume that the C expressions and statements used in the provided and do sections are

adequately restricted to respect the atomicity assumption for transitions e.g., no side effects,

guaranteed termination.

3.4.6 Priority in atom

3.4.6.1 Description

The language offers the option for specifying priority between the ports of an atom.

The priority defines scheduling constraints between ports and associated transitions: if the atom is in a state where two transitions are enabled, the higher priority port transition is preferred, while the lower port transition is disabled.

Remark 1: the priority rule is applied locally and may lead to deadlocks due to outer synchronization constraints. For example, if the higher priority port cannot be triggered because of external interactions (it is not connected, or connected with a strong synchronization not yet ready), the lower port will be disabled anyway.

Remark 2: Internal ports have by default higher priority than exported ports.

3.4.6.2 syntax

<atom priority definition> ::= priority <priority name> <port name> < <port name>
[**provided** <expression>]

3.4.7 export port in atomic component

3.4.7.1 Description

The port export may be done either in the port declaration, or with an additional export directive. In order to be triggered, an exported port must be connected outside of the component within at least one connector.

3.4.7.2 syntax

<export definition> ::= **export port** <port type reference> <export name> **is**
<port name> (, <port name>)*

<port type reference> ::= [<library name> .] <port type name>

3.4.8 atom type example

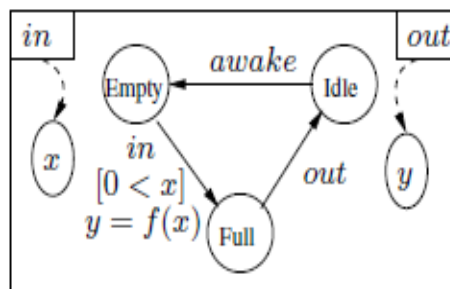


figure 3.1: An atomic type

Figure 3.1 shows an atomic reactive component with three ports in, out, awake, variables x, y, and control states Idle, Empty and Full. The automaton is initialized to control state Idle, from which it can move to Empty by the internal transition awake. At Empty, the transition labeled in is possible if $0 < x$. When an interaction through in takes place, the variable x is eventually modified and a new value for y is computed. From Full, the transition out can occur. The omission of guard and function for this transition means that the associated guard is true and the internal computation step is empty.

The BIP description of the reactive component of figure 3.1 is:

```
atomic type Reactive
  data int x, y
  export port intPort in(x) = in
  export port intPort out(y) = out
  port ePort awake
  place Idle, Full, Empty
  initial to Idle do x=0;
  on awake
    from Idle to Empty
  on in
    from Empty to Full provided  $0 < x$  do  $y=f(x)$ ;
  on out
    from Full to Idle
end
```

The variables associated with a port may be modified on executing the the transfer function of an interaction, in which the port participates. A pure event port does not have any associated variables, and provides the mechanism for event synchronization only (i.e., without any data transfer). The port type ePort used in the code-sample above is an example of a pure event port.

3.5 Compound component description

3.5.1 component instance

3.5.1.1 Description

A component to be instantiated must be first defined, either as an atomic type or a compound type.

The compound component represents an assembly of elements:

- sub-components
- connectors

The component instance represents a sub-component.

Using compound components, BIP models become hierarchical. The compound components may be used in the same way as atomic components.

As for atomic components, inner ports (ports of subcomponents, or connectors) must be exported in order to be used in outer compositions. There is no difference between exported inner port of compound components and exported port of atomic components.

3.5.1.2 syntax

```
<component definition> ::= component <component type name>
                           <component name> ( index )*
                           [ <parameters> ]
<parameters>             ::= ( <expression> ( , <expression> )* )
<index>                  ::= [ <expression> ]
```

3.5.1.3 example

To instantiate a component of type Reactive, we would write:

```
component Reactive R1
```

which creates an instance of Reactive named R1.

3.5.2 connector instance

3.5.2.1 Description

Connectors are used to define the interactions between components. Connector types define the number and type of ports interacting.

A connector instance associates the ports of instantiated components through the interactions defined by the connector type. The ports specified in the connector instantiation are mapped positionally to the port parameters in the connector type definition. Each connector port parameter must be mapped to a component port of the same type.

3.5.2.2 syntax

```
<connector definition> ::= connector <connector type name>
                           <connector name> ( index )*
                           ( <instance port reference> ( , <instance port reference> )* )
                           [ <parameters> ]
<parameters>             ::= ( <expression> ( , <expression> )* )
<instance port reference> ::= <connector name> [ .<port name> ]      |
                           <component name> .<port name>          |
                           ( <expression> ? <true instance port reference> : <false instance
port reference> )
```

3.5.2.3 Structured Connector

Structured connectors are created by the combined mechanism of exporting port from a connector and instantiating connectors, where a port of the connector is an exported port of another instantiated connector.

3.5.2.4 Example

As an example, an instance C0 of connector type Broadcast, parameterized by the ports out and in of the components R1 and R2 respectively, is written as:

```
connector Broadcast C0(R1.out, R2.in)
```

3.5.3 priority in compound

3.5.3.1 description

Priority rules in compound components have the same intuitive meaning as priority rules in atomic component. Here, the priority rules are used to select among two enabled interactions or two enabled connectors. In the second case, it prefers all interactions of one connector to all interactions of the other.

As usual, the lower priority interaction is inhibited if a higher interaction is locally possible.

When connector type is used, the priority is defined for all interactions of all connectors of the referenced type instantiated in the compound component of the priority.

Remark: as for atomic components, the choice is made locally and may lead to deadlocks.

3.5.3.2 syntax

<compound priority definition> ::= priority <priority name> <index>*

<priority element> < <priority element>

[**provided** <expression>]

<priority element> ::= <interaction> | <connector type name>

<interaction> ::= <connector reference> [: <port reference>*]

<connector reference> ::= <connector name> (<index>*)

<port reference> ::= <subcomponent reference>.<port name>

<subcomponent reference> ::= <subcomponent name> (<index>*)

3.5.4 export port in compound component

3.5.4.1 Description

The language allows a compound to export a connector as a port, similar to exporting ports from atoms. Exporting makes the port (and hence the connector) visible in the compound interface, and hence allows for its further synchronization with other components. This also provides a homogeneous interface for both atoms and compounds, facilitating their seamless usage.

In order to export a connector as a port, the connector itself must have an exported port.

The type of this port will be the type of the compound port.
 If a port is exported, it must be connected to at least one connector, otherwise it will never be triggered. If a connector port is exported, the connector becomes a sub-connector of a hierarchical connector.

3.5.4.2 syntax

```

<export definition> ::= export port <port type reference> <export name> is
                       <port reference> ( , <port reference>)*
<port type reference> ::= [<library name> . ] <port type name>
<port reference>      ::= <connector name>[.<port name>] |
                       <component name> . <port name>
  
```

3.5.4.3 example

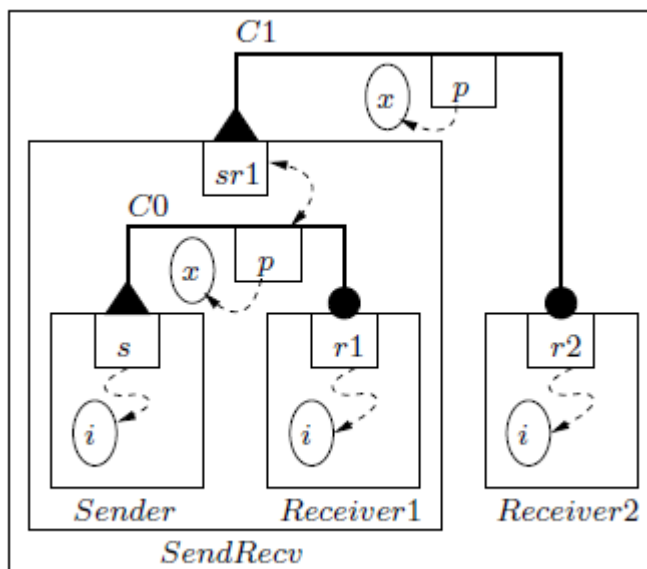


figure 3.2: Exporting connector as compound port

We provide an example (figure 3.2) of exporting a connector as a port from a compound component. The connector C0 in the compound SendRecv is exported as a port named sr1, of type intPort. The BIP description is:

```

compound type XXXX
  component Reactive Sender
  component Reactive Receiver1
  connector Broadcast C0 (Sender.s, Receiver1.r1)
  export intPort srl is C0
end

```

3.6 Connector description

3.6.1 definition

3.6.1.1 description

The port expression defines the set of legal interactions of the connector

- port1 port2 means that the connector has a legal interaction if both port1 and port2 are active
- port1' means that the connector has a legal interaction if port1 is active, the other ports being optional
- port1 + port2 means that the connector has a legal interaction if either port1 or port2 are active

For more details, see the algebra of connectors in [1].

3.6.1.2 syntax

```

<define connector expression> ::= define <port expression>
<port expression> ::= <port name> |
                   <port expression> ' |
                   <port expression> (+ <expression>)* |
                   <port expression> (<port expression>)* |
                   [ <port expression> ]

```

3.6.2 variables

3.6.2.1 Description

Data variables are used in connectors to store temporary values during the connector execution, for data transfer, or guard computation. They are transient – their values should not change the state of system, from one interaction to another.

3.6.2.2 syntax

```

<data definition> ::= data <data type name>
                   <data name> = <expression> ( , <data name> = <expression>)*

```

3.6.3 interactions

3.6.3.1 Description

The interaction definition is used to describe the specific guard and behavior for each possible interaction (list of active port). For a rendez-vous connector, there is only one possible interaction with all ports.

3.6.3.2 syntax

<interaction definition> ::= on (<port name>)+ [**provided** <expression>]
<data transfer>

3.6.4 data transfer

3.6.4.1 Description

For every interaction, the data transfer is specified by an up{ } and a down{ } method. The method up{ } is supposed to update the local variables of the connector, from the values of variables associated with the ports. This function must not have side-effects on data associated to ports. Conversely, the method down{ } is supposed to update the variables associated with the ports, from the values of the connector variables. The down function is called on successful execution of the interaction.

Remark: The up function may be called more often than the down action.

3.6.4.2 syntax

<data transfer> ::= [**up** <action>] [**down** <action>]

3.6.5 export port in connector

3.6.5.1 Description

A connector has an option to define a port and export it. This allows a connector to be used as a port in other connectors, and create structured connectors. Local variables of the connector can be associated with its exported port. This allows to manage hierarchical data transfer mechanism.

For a connector type definition, the type of its exported port forms an integral part of its signature, in addition to the list of its ports along with their types, and the C parameters.

3.6.5.2 syntax

<connector export definition> ::= **export port** <port type reference> <export name>
[<data port>]

<port type reference> ::= [<library name> .] <port type name>

3.6.6 example

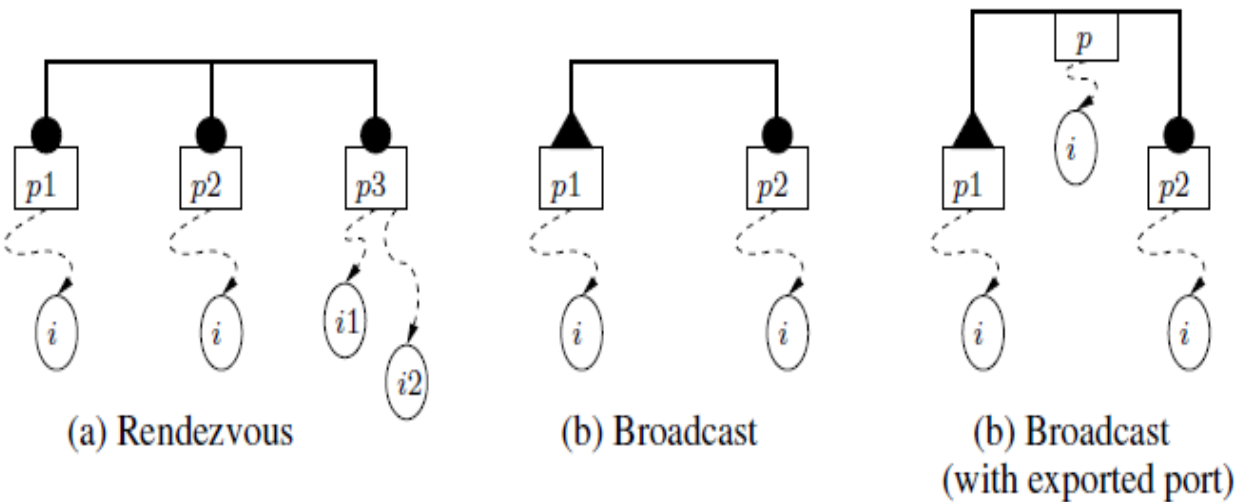


Illustration 3.3: connector types

Figure 3.3 shows graphically, examples of simple connector types. The connector type named Rendezvous (figure 3.3(a)) is parameterized by three ports, $p1$, $p2$, and $p3$. $p1$ and $p2$ are of type `intPort`, whereas $p3$ is of type `int2Port`, which associates a port with two integer variables. The connector type defines a rendezvous between the ports, specified by the expression $p1p2p3$, with associated guard and transfer functions as described below:

```
connector type Rendezvous (intPort p1, intPort p2, int2Port p3)
define [p1 p2 p3]
data int x
on p1 p2 p3 provided (p1.i + p2.i != p3.i1 + p3.i2)
up {x = MAX(p1.i, p2.i, p3.i1, p3.i2);}
down {p1.i = p2.i = p3.i1 = p3.i2 = x;}
end
```

The connector declares a local integer variable x . The interaction involving the three ports can occur only when the guard expression $(p1.i + p2.i != p3.i1 + p3.i2)$ is true. The data transfer consists of the combined effect of the up and down methods. In the up method, the MAX of the variables associated with the ports is calculated and stored in the connector variable x . In the down method, the value of x is assigned to each port variable. As a result of the data transfer, the variables associated with the ports are set to the maximum of their values.

The second example (figure 3.3(b)) defines a connector type named Broadcast, between two `intPort`s, $p1$ and $p2$. $p1$ is the trigger of the broadcast, defined by the expression $p1p2$. The BIP description is:

```

connector type Broadcast (intPort p1, intPort p2)
define [p10 p2]
data int x
on p1
up {x = p1.i;}
down {}
on p1 p2
up {x = p1.i;}
down {p2.i = x;}
end

```

The data transfer methods describe transfer of value from the variable associated with the trigger p1 to the variable associated with the synchron p2. Also in this example, as no guards are provided with the interactions, they are, by default, taken as true. Notice that contrary to other formalisms, BIP does not allow explicit distinction between input and output ports. For simple data flow relation, variables can be interpreted as inputs or outputs. For instance, in the connector type Broadcast, p1 can be thought as an output port and p2 as an input port.

The third example (figure 3.3(c)) illustrate the export of port in connector. Consider the connector type definition of Broadcast, which consists of two intPorts and exports an intPort.

```

connector type Broadcast (intPort p1, intPort p2)
define [p10 p2]
data int x
on p1
up {x = p1.i;}
down {}
on p1 p2
up {x = p1.i;}
down {p2.i = x;}
export intPort p(x)
end

```

The export statement creates a port p of type intPort, and associates the connector variable x with it. This leads to two possibilities: 1) a connector of type Broadcast can be used as an intPort in another connector, leading to a structured connector, 2) it can be exported as a port of type intPort in a compound type.

3.7 Actions

3.7.1.1 Description

The actions are used to describe the data transformations associated with the transitions, and the data transfer associated with interactions on connectors. The actions are a subset of C++ instructions. The variables used in the actions must be declared as BIP data or as external data. External functions or methods can be called in the BIP action. In this case the BIP parser does not check the function profiles. The check is postponed to the C++ compiler at the C++ compile step.

3.7.1.2 Syntax

```

<action> ::= <expression> [= <expression>]; |

```

```

if (<expression>) action [else <action>] |
{# <any C++ code> #} |
{ {<action>} }

```

3.8 Expressions

3.8.1.1 Description

Expressions may appear within guards, component or connector parameters, or within actions.

3.8.1.2 Syntax

```

<expression> ::= [<unary operator>] <postfix expression> [ <binary operator> < expression>]

```

```

<unary operator> ::= + | - | not | * | &

```

```

<binary operator> ::=

```

```

    * | / | % |
    + | - |
    < | <= | > | >= |
    == | != |
    or |
    and

```

```

<postfix expression> ::= <literal expression> |

```

```

    <primary expression> {
        [<expression>] |
        .<identifier> |
        -> <expression> |
        (<expression> {, <expression>})
    }

```

```

<literal expression> ::= <integer> | <float> | <string> | <index reference>

```

```

<index reference> ::= $<0-9>

```

Index expressions can only be used to refer multiple element instances (component or connector) – as usual to represent the nth element of the multiple instance.

3.9 Array instance

3.9.1.1 description

The language allows to create static arrays of component and connector. This is

similar to defining arrays, as in the C language, by specifying a static dimension. Individual elements of the array can be accessed by indexing.

Array of Component

The syntax for creating an array of component instance is:

```
component component-type-name component-instance-name [ dimension ]  
[ ( component-parameter-list ) ]
```

Below we define a compound type named System that instantiates an array of Reactive components.

```
compound type System (int length)  
component Reactive R[length]  
. . .  
export port intPort input is R[0].in  
export port intPort output is R[length-1].out  
end
```

It also exports port in and out of the 0th and length–1th instance respectively as ports of the compound, named input and output.

Array of Connector

```
connector connector-type-name connector-instance-name [ dimension ]  
( component-port-list ) [ ( connector-parameter-list ) ]
```

We also introduce a macro notation, denoted as \$ which refers to all the instances of the array obtained by substituting \$ by indices of the array dimension. This is useful to specify the component port instances for the connectors. An example of an array of Broadcast connectors is:

```
compound type System (int length)  
component Reactive R[length]  
connector Broadcast Cnx[length-1] (R[$].out, R[$+1].in)  
. . .  
end
```

Here Cnx is an array of Broadcast connector, where the ith connector associates the ports R[i].out and R[i+1].in.

3.10 Timed models

3.10.1.1 Description

Timed systems are modeled as automata extended with real-valued variables (clocks) measuring the time elapsed since their initialization. Time can progress at a control state only if there are no enabled transitions. As soon as a transition is enabled, time cannot progress any more and the execution of the enabled transition(s) are enforced. Control states have time progress conditions specifying how time can advance, and to specify urgency of transitions. The BIP language allows transitions to be associated with deadlines. They are conditions on the clock variables that determine when a transition can be triggered by stopping time progress.

The transition urgency defines priority between the time interaction and other

regular interactions. When a transition is eager, or delayable with clock equal to the upper bound, it will prevent the time interaction to take place.

This case can lead to deadlock. Default urgency is lazy.

3.10.1.2 Urgency semantics

The relative position of the deadline with respect to the guard determines the urgency of the action. Urgency can be of the following types:

Eager

When the deadline is equal to the guard, time cannot evolve and an enabled transition must be taken.

Delayable

When the deadline is considered as the falling edge of the guard, so time can progress, but limited to the upper bound of the guard, when an interaction must be taken.

Lazy

The third case represents the situation when the deadline is false, so there is no constraint on the time progress.

3.10.1.3 Recommendation

Generally, in order to avoid deadlock, it is preferable that an interaction depends on at most one eager transition. The following example may fall to a deadlock situation

```

model example
atom type sender
  export port Port com = communicate
  port Port work
  timed data clock
  place working, communicating
  initial to working
  on work from working to communicating delayable clock in (2,5)
  on comm from communicating to working eager do clock=0 ;
end
atom type receiver
  export port Port com = communicate
  port Port work
  timed data clock
  place working, communicating
  initial to working
  on work from working to communicating delayable clock in (4,5)
  on comm from communicating to working eager do clock=0 ;
end

connector type rendezvous(Port p1, Port p2)
  define p1 p2
end

compound type example
  component sender s
  component receiver r
  connector rendezvous(s.communicate, r.communicate)
end

component example root
end

```

3.10.1.4 Syntax

```

<timed condition> ::= <urgency> [<clock> in (<low bound>, <high bound>)]
<urgency>          ::= eager | delayable | lazy
<clock>            ::= <identifier>
<bound>            ::= _ | <expression>

```

3.11 Package and system

3.11.1 package declaration

description

the package is used to declare BIP types to be shared to several systems. The package declaration may declare any BIP type.

Remark: two types with the same name and profile declared in two different package are not compatible.

3.11.1.1 syntax

```
<package declaration> ::=      package <package name>
                                {<use package> }
                                {<bip_definition> | <opaque_code> }
                                end

<bip definition> ::=           <port type definition> |
                                <connector type definition> |
                                <component type definition>

<opaque code> ::=              [header] {# <any C++ code> #}
```

3.11.2 system declaration

3.11.2.1 description

System declaration allows to define a BIP model, including the top level component defines as root definition.

3.11.2.2 syntax

```
<system declaration> ::=      model <package name>
                                {<use package> }
                                {<bip definition> | <opaque code> }
                                <root definition>
                                end
```

3.11.3 The use policy

3.11.3.1 description

The use directive allow to import the bit types declared in a package. The BIP parser looks for a model file of name <lib name>.model (generated by the BIP compilation of a <lib name>.bip source) in the includes directories specified in the command line.

3.11.3.2 syntax

```
<use package> ::= use <lib name>
```

3.11.4 the root definition

3.11.4.1 Description

The root definition defines the top level of the BIP model.

3.11.4.2 Syntax

<root definition> ::= <component definition>

3.12 Syntax summary

Here is the syntax rules in the alphabetic order.

<action> ::= <expression> [= <expression>]; |
if (<expression>) action [else <action>] |
{# <any C++ code> #} |
{ {<action>} }

<atomic type> ::= **atomic type** <atomic type name> [<component parameters>]
[<component_activation>]
[<header code>]
[<body code>]
(<data definition> | <port definition> | <place definition>)+
[<initial block>]
(<transition definition> | <priority definition>)+
(<export definition>)*
end

<atomic type name> ::= <identifier>

<atom priority definition> ::= priority <priority name> <port name> < <port name>
[**provided** <expression>]

<binary operator> ::=
* | / | % |
+ | - |
< | <= | > | >= |
== | != |
or |
and

<bip definition> ::= <port type definition> |
<connector type definition> |
<component type definition>

<bound> ::= _ | <expression>

<clock> ::= <identifier>

<component activation> ::= **singleshot** | **multishot**

<component definition> ::= component <component type name>

```

        <component name> ( index )*
        [ <parameters> ]
<compound priority definition> ::= priority <priority name> <index>*
        <interaction> < <interaction>
        [provided <expression>]
<compound type> ::= compound type <compound type name> [<component
parameters>]
        [<component_activation>]
        [<header code>]
        [<body code>]
        ( <component definition> | <connector definition> | <priority
definition> )+
        ( <export definition> )*
        end
<compound type name> ::= <identifier>
<connector definition> ::= connector <connector type name>
        <connector name> ( index )*
        ( <instance port reference> ( , <instance port reference> )* )
        [ <parameters> ]
<connector export definition> ::= export port <port type reference> <export name>
[<data port>]
<connector reference> ::= <connector name> (<index>*)
<connector type definition> ::= connector type <connector type name>
        <port type profile> [<data type profile>]
        [<header code>]
        [<body code>]
        <define connector type>
        < export connector port definition>
        end
<data definition> ::= [export] [extern] [timed] data <data type name>
        <data name> [= <expression>] ( , <data name> [= <expression>] )*
<data name> ::= <identifier>
<data parameter> ::= <data type> <data name>
<data port> ::= ( <data name> ( , <data name> )* )
<data transfer> ::= [up <action> ] [down <action> ]
<data type> ::= <identifier>
<define connector expression> ::= define <port expression>

```

```

<define connector type> ::= <define connector expression>
                          (<data definition> | <interaction definition> )*
<export definition> ::= export port <port type reference> <export name> is
                          <port name> ( , <port name>)*
<export name> ::= <identifier>
<expression> ::= [<unary operator>] <postfix expression> [ <binary
operator> < expression>]
<index> ::= [<expression>]
<index reference> ::= $<0-9>
<initial block> ::= to <places>
                  [ do <action> ]
<instance port reference> ::= <connector name> [.<port name>] |
                             <component name>.<port name> |
                             (<expression>?<true instance port reference>:<false instance
port reference>)
<interaction> ::= <connector reference> [ : <port reference>]*
<interaction definition> ::= on (<port name>)+ [ provided <expression> ]
                             <data transfer>
<literal expression> ::= <integer> | <float> | <string> | <index reference>
<opaque code> ::= [header] {# <any C++ code> #}
<parameters> ::= (<expression> ( , <expression>)* )
<place definition> ::= place (<place name>)*
<places> ::= <place name> ( , <place name>)*
<port definition> ::= export port <port type reference> <port name> [<data port>]
                  [= <export name>]
<port expression> ::= <port name> |
                    <port expression> ' |
                    <port expression> (+ <expression>)* |
                    <port expression> (<port expression>)* |
                    [ <port expression> ]<port reference>
                    ::= <connector name>[.<port name>] |
                    <component name> . <port name>
<package declaration> ::= package <package name>
                          {<use package> }
                          {<bip_definition> | <opaque_code> }
                          end
<port type> ::= port type <port type name> ( [<data parameter>]

```

(,<data parameter>)*)

<port type name> ::= <identifier>

<port type reference> ::= [<library name> .] <port type name>

<postfix expression> ::= <literal expression> |

<primary expression> {
 [<expression>] |
 .<identifier> |
 -> <expression> |
 (<expression> {, <expression>})
}

<root definition> ::= <component definition>

<subcomponent reference> ::= <subcomponent name> (<index>*)

<system declaration> ::= **model** <package name>
 {<use package> }
 {<bip definition> | <opaque code> }
 <root definition>
 end

<timed condition> ::= <urgency> [<clock> **in** (<low bound>, <high bound>)]

<transition definition> ::= **on** port_expression **from** <places> **to** <places>
 [**provided** <expression>]
 [<timed condition>]
 [**do** <action>]

<unary operator> ::= + | - | **not** | * | &

<urgency> ::= **eager** | **delayable** | **lazy**

<use package> ::= **use** <lib name>

4 ENGINE FEATURE

4.1 Running an example

4.1.1 Analyze source and generate code

There is two possibilities to analyze BIP source, and generate code for the BIP engine:

- call BIP command line with options
- in the eclipse environment, activate the compile and generate command through BIP menu

The BIP command line options are:

- --genC-execute : generate code for simulation only
- --genC-explore : generate code for simulation and state exploration
- -l <directory location> : add a directory to look for imported library
- -g : generate debug function to display state variables when running interactive simulation
- -f <source file name> : specify the source file
- -m or --model : read a model instead of e BIP source file
- -r or --reverse <destination name>

4.1.2 Link with the engine

During the generation phase, the BIP parser generate a makefile to compile and link the C code generated. The makefile name is “<source file name>.mk”. The command :

```
make -f <source file name>.mk [debug=yes]
will generate the executable.
```

4.1.3 Run the example

The generated executable have the following option in the command line:

- --execute: simulate the model
- --explore: explore the model states
- --quiet: do not print interaction trace
- --trace: generate the trace of visited states (in simulation mode)
- --dfs: order of state exploration, go to deep first
- --bfs: order of state exploration, go to different branches first
- --interactive: in execution mode, stop at each interaction, and let user choose the interaction to execute
- --opt1: optimization mode
- --opt2: optimization mode
- --disable-output: do not generate output files
- --output / -o <name>: specify the root name of generated files
- --limit / -l <number>: limit the number of step (number of executed interactions)
- --help / -h: get the executable options
- --simout / -s:
- --seed / -r <number>: use a particular seed for random selection. Default is 1. several run with the same seed will produce the same behavior.

4.1.4 The interactive mode

In the interactive mode (with `-execute` `-interactive` options), at each step the user have the list of candidate interactions, and may chose a command:

- `<n>`: chose to execute the interaction number `<n>`
- `r`: chose a random interaction to execute
- `p <name>`: print a state variable. The variable name is :
 `{<component name>.<component name>.<variable name>`
 where the first component name is a component of the root component type
 and the variable name is a place name, a user data name, or `BIP_STATE`

4.1.4.1 example

for the source:

```
model prodcons
header {# typedef char* String; #}

/* definition of a port type */
port type IntPort(int x)

/* definition of a connector type */
connector type SendRec(IntPort s, IntPort r)(int p1, int p2)
  define [ s r ]
  data int val
  on s r provided true down {val = s.x; r.x = s.x;}
  export IntPort p(val)
end

/* definition of an atomic component type */
atomic type Worker(String cname)
  data int value = 0
  export port IntPort comm(value) = communicate
  port IntPort work(value)

  place working
  place sending

  initial to working do printf("init\n");
  on work
    from working to sending
    provided true do {
      printf("working %s\n", cname) ;
      value = (value+1) % 3 ;
    }
  on comm
    from sending to working
    provided true do printf("comm %s\n", cname) ;
end

/* definition of a compound component */
compound type Team
  component Worker producer("producer")
  component Worker consumer("consumer")
  connector SendRec c(producer.communicate, consumer.communicate)(1,2)
end

/* instantiation of the root component */
component Team t

end
```

Here is the execution trace:

```
$/ex.bip.x --interactive --execute
```

```
*****  
*      BIP Engine (Version 1.0)      *  
*      Verimag, France                *  
* (www-verimag.imag.fr/~async/BIP/bip.html) *  
*****
```

```
init
```

```
init
```

```
working producer
```

```
working consumer
```

```
List of legal connectors are :
```

```
[1]    BIP_Top/c/producer:comm|consumer:comm
```

```
Enter connector number (Hit 0 to exit) : 1
```

```
scheduler : BIP_Top/c/producer:comm|consumer:comm
```

```
comm producer
```

```
working producer
```

```
comm consumer
```

```
working consumer
```

```
List of legal connectors are :
```

```
[1]    BIP_Top/c/producer:comm|consumer:comm
```

```
Enter connector number (Hit 0 to exit) : p producer.BIP_STATE
```

```
producer.BIP_STATE = sending
```

```
Enter connector number (Hit 0 to exit) : p producer.sending
```

```
producer.sending = true
```

```
Enter connector number (Hit 0 to exit) : p producer.value
```

```
producer.value = 2
```

```
Enter connector number (Hit 0 to exit) :
```