BIP Language : Concrete Syntax

VERIMAG
Centre équation - 2, avenue de Vignate
38610 Gieres, FRANCE

June 11, 2007

1 Lexical tokens

An *identifier* consist of letters, digits and '_', and begins with a letter.
The list of keyword is:**behavior**, **class**, **complete**, **component**, **connector**, **contains**, **data**, **delayable**, **do**, **eager**, **else**, **end**, **extern**, **from**, **if**, **include**, **incomplete**, **init**, **initial**, **lazy**, **on**, **port**, **priority**, **provided**, **rename**, **state**, **timed**, **to**, **trans**, **when** .

A special directive starting with **{#** and ending with **#}** is provided to allow an arbitrary piece of C code wrapped within this directive to be specified in the BIP description.

Single line comment, starting with **//** as in C++, is allowed.

Files can be included within a BIP description using the directive **include** followed by the file name between double quotes.

2 The BIP Model

A model defines the component hierarchy and in addition, optional global shared data and C declarations (data types, functions).

model-decl ::=
    { *model-item* }*

model-item ::=
    component-def |
    global-decl |
    c-decl

A model should contain at least a component definition.

3 Components

A Component, which can be of either an *Atom* or a *Compound*, is defined as:

component-def ::=
    component *component-id* [**(** *formal-args* **)**]
        [{ *atom-item* }* | { *compound-item* }*]
    end

component-ref ::=
    component-id { **.** *component-id* }*

formal-args ::=
    type-id arg-id { **,** *type-id arg-id* }*

An atomic component define its ports, data and behavior, and an optional C declaration.

atom-item ::=
    c-decl |
    data-decl |
    port-decl |
    behavior

4 Ports

port-decl ::=
    port [**complete** | **incomplete**]
        port-id { **,** *port-id* }*

port-ref ::=
    [*component-ref* **.**] *port-id*

5 Data

For data, basic C primitive types can be specified. Pointer types, aggregate types or any arbitrary types or typedefs can be specified within a **{#... #}** directive. If data are declared in a C section, and used in the BIP part, these data must be declared in the BIP part with the reserved word **extern**. Data

declared with the reserved word **timed** are timed data used for timed guard expression. If the data is of type <T>, it must exist a function with the following profile:
<T> <T>_increment(<T> timeVal, int increment) ;

data-decl ::=
[**extern**] [**timed**] **data** *type-id var-id* { **,** *var-id* }*

c-decl ::=
{# plain C/C++ decl **#}**

6 Behavior

A behavior is defined by a Petri net or an automaton. In both cases, it is a set of states/places and transitions.
In the case of an automaton the keyword **state** is followed by a control state and the list of outgoing transitions from this state. Each transition is labelled by a port identifier followed by its guard, function (i.e. statement) and a target state. The target state is either a simple state or a selected list of target states, in this case all list elements must have a condition except the last one which must be unconditional.
In the case of a Petri net, the keyword **trans** is followed by a port identifier, the keyword **from**, the list of incoming places, its guard, function, and the target places.
The target places are either a simple list of places or a selected list of couples (condition, target places) followed by unconditional target places.

behavior ::=
behavior
[*auto-behaviour* | *net-behaviour*]
end

auto-behaviour ::=
auto-init-transition { *auto-state* }*

auto-init-transition ::=
init
[**do** *action-block*]
to *auto-target*

auto-state ::=
state *state-id*

{ *auto-transition* }*

auto-transition ::=
on *port-id*
[**provided** *expression*]
[**when** **(** *expression, urgency* **)**]
[**do** *action-block*]
to *auto-target*;

auto-target ::=
{ *state-id* **if** *expression* **;** }*
state-id

net-behaviour ::=
net-init-transition { *net-transition* }*

net-init-transition ::=
init
[**do** *action-block*]
to *net-target*

net-transition ::=
trans *port-id*
from *place-id* { **,** *place-id* }*
[**provided** *expression*]
[**when** **(** *expression, urgency* **)**]
[**do** *action-block*]
to *net-target*

net-target ::=
{ *place-id* { **,** *place-id* }* **if** *expression* **;** }*
place-id { **,** *place-id* }*

urgency ::=
eager |
delayable|
lazy

A compound component defines new components from existing components (atoms or compounds) by creating their instances, defining the connectors between the instances and defining the priorities between the interactions of the connectors. The contents of a compound component is defined by:

compound-item ::=
c-decl |

data-decl |
subcomponent-decl |
connector-decl |
priority-decl

subcomponent-decl ::=
contains *component-id* [**(** *actual-args* **)**]

The instances can have parameters providing initial values to their variables through positional association.

actual-args ::=
expression { **,** *expression* }*

7 Connectors

A connector definition includes its set of ports followed by an optional **complete** expression describing its minimal complete interactions. If the **complete** expression is omitted, the connector is taken as maximally incomplete. A connector may optionally belong to a predefined type specified by the keyword **class** followed by the type name. Connectors may have a behavior specified similarly as the behavior of transitions, by a set of guarded commands associated with the feasible interactions.

connector-decl ::=
connector *connector-id* **=** *interaction*
[**complete =** *expression*]
[**class** *class-id*]
behavior
{ *connector-transition* }*
end

connector-ref ::=
[*component-ref* **.**] *connector-id*

connector-transition ::=
[**on** *interaction*]
[**provided** *expression*]
[**do** *action-block*]

interaction ::=

port-ref { **,** *port-ref* }*

class-id ::=
Singleton |
Handshake

8 Priorities

Given a system of interacting components, priorities are used to filter interactions amongst the feasible ones depending on given conditions. The syntax for priorities is the following:

priority-decl ::=
priority { *priority-rule* }*

priority-rule ::=
priority-id [**if** *expression*]
interaction-set < [*interaction-set* | *****]

interaction-set ::=
qualified-interaction |
{ *qualified-interaction* { **;** *qualified-interaction*}* **}**

qualified-interaction ::=
[*connector-ref*] *interaction*

9 Expressions

Expressions are C-like expressions. All data involved in expressions must be declared as BIP data. The BIP data whose type is user-defined may be followed by the **.** or − > C operator and field or function expression as in usual C++. In this case during code generation, the BIP tool reproduces the end of the expression without any type check.
A state name may be used as a boolean expression which is true if the component is in this state.

expression ::=
c-constant |
obj-ref |
c-op$_1$ *expression* |
expression c-op$_2$ *expression* |

(*expression*)

c-constant ::=
useful C constants

obj-ref ::=
[*component-ref* **.**] *var-id* |
func-id |
obj-ref **.** *field-id* |
obj-ref − > *field-id* |
obj-ref [*expression*] |
obj-ref **(** [*expression* { **,** *expression* }*] **)** |
***** *obj-ref*

c-op$_1$ ::=
common C unary operator

c-op$_2$ ::=
common C binary operator

10 Actions

An action is a C-like statement. The set of possible actions is a subset of C statements.

action-block ::=
{ *action* | *c-action* }*

action ::=
[*obj-ref* **=**] *expression* **;** |
if (*expression* **) { ***action-block* **}**
[**else { ***action-block* **}**]

c-action ::=
{# plain C/C++ action **#}**