Grammar for BOSSA

December 29, 2004

scheduler ::= [default] ([high_res] | [low_res]) scheduler id =
{sched_decl handlerdef interfacedef functiondef }
| [default] ([high_res] | [low_res]) virtual_scheduler id =
{vsched_decl handlerdef interfacedef functiondef }
sched_decl ::= (constdef)* (typedef)* [processdef] (fundecl | valdecl)* statedef [orderdef]
[admissiondef] [tracedef]
vsched_decl ::= (constdef)* (typedef)* [schedulerdef] (fundecl | valdecl)* statedef [orderdef]
[admissiondef] [tracedef]

constdef ::= const bossa_type_expr id = expr ;
typedef ::= (enumdef | rangedef)*
enumdef ::= type enum_name = enum { id (, id)* } ;
rangedef ::= type range_name = [expr .. expr] ;

processdef ::= process = { (process_var_decl ;)+ }
schedulerdef ::= scheduler = { (process_var_decl ;)+ }
process_var_decl ::= type_expr id | type_expr system id | timer id

fundecl ::= non_proc_type fn_name ([parameter_types]) ; | void fn_name ([parameter_types]) ;
valdecl ::= non_proc_type id = expr ; | non_proc_type system id ; | timer id ;
parameter_types ::= (type_expr | timer) (, (type_expr | timer))*

statedef ::= states = { (class_name id [: storage] ;)+ }
class_name ::= READY | RUNNING | BLOCKED | TERMINATED
storage ::= process | [state_visibility] scheduler | [state_visibility] [queue_type] queue
state_visibility ::= public | private
queue_type ::= select | select fifo | select lifo

orderdef ::= ordering_criteria = { (key_crit_decls , crit_decls | key_crit_decls | crit_decls) }
key_crit_decls ::= key crit_decl (, key crit_decl)*
crit_decls ::= crit_decl (, crit_decl)*
crit_decl ::= critop id | critop (expr ? expr : expr)
critop ::= lowest | highest

admissiondef ::= admit = { (valdef)* adm_crit [attach_detach] }
valdef ::= type_expr id = expr ;
adm_crit ::= admission_criteria ([param_var_decl (, param_var_decl)*]) = { expr }
param_var_decl ::= type_expr id
attach_detach ::= admission_attach proc_param = seq_stmt admission_detach proc_param = seq_stmt
proc_param ::= ((process | scheduler) id)

```
tracedef        ::= trace integer { [ trace_events ] [ trace_exprs ] [ trace_test ] }
trace_events    ::= events = { event_name (, event_name)* };
trace_exprs     ::= expressions = { id (, id)* };
trace_test      ::= test = { expr };

handlerdef      ::= handler (event id ) { (On event_name (, event_name)* seq_stmt)+ }
interfacedef    ::= interface = { (type_or_void id ( [ param_var_decl (, param_var_decl)* ] ) seq_stmt)+ }
functiondef     ::= function = { (type_or_void fn_name ( [ param_var_decl (, param_var_decl)* ] ) seq_stmt)+ }

bossa_type_expr ::= int | bool | time | cycles | port | process | scheduler | enum_name | range_name
type_expr       ::= bossa_type_expr | system struct id
type_or_void    ::= type_expr | void
non_proc_type   ::= int | bool | time | cycles | port | enum_name | range_name | system struct id

stmt            ::= if_stmt | for_stmt | return_stmt | switch_stmt | seq_stmt | assign_stmt | move_stmt
                    | defer_stmt | prim_stmt | error_stmt | break_stmt
if_stmt         ::= if ( expr ) seq_stmt [ else seq_stmt ]
for_stmt        ::= foreach ( id [ in class_state (, class_state)* ] ) seq_stmt
                    | foreachIncreasing ( id in state ) seq_stmt
                    | foreachDecreasing ( id in state ) seq_stmt
class_state     ::= state | class_name
return_stmt     ::= return [ expr ] ;
switch_stmt     ::= switch loc_expr in { (case class_state (, class_state)* : seq_stmt)* }
seq_stmt        ::= { (valdef)* (stmt)* }
assign_stmt     ::= loc_expr assign_unop | loc_expr assign_binop expr
assign_unop     ::= ++ | --
assign_binop    ::= = | += | -= | *= | /= | %= | &= | |= | <<= | >>=
move_stmt       ::= move_expr => state_ref [ .head | .tail ] ;
                    | move_expr => forwardImmediate() [ .head | .tail ] ;
defer_stmt      ::= defer();
prim_stmt       ::= fn_name ( [ expr (, expr)* ] );
error_stmt      ::= error( string );
break_stmt      ::= break;

expr            ::= integer | id | state | true | false | unop expr | * expr | expr . id | select()
                    | fn_name ( [ expr (, expr)* ] ) | empty( class_state ) | srcOnSched()
                    | schedulerOf( expr ) | expr binop expr | expr in class_state | ( expr )
unop            ::= + | - | ! | ~
binop           ::= + | - | * | / | % | && | || | & | | | == | != | < | > | <= | >= | << | >>
loc_expr        ::= (id | state_name) (. id)*
move_expr       ::= select() | state_name | id | id . source | id . target
```

Operator precedence is as follows:

```
{,} < {=, +=, -=, *=, /=, %=, &=, |=, <<=, >>=} < {||} < {&&} < {|} < {&} < {==, !=} < {<, >, <=, >=}
  < {<<, >>} < {+, -} < {*, /, %} < {!, ~, ++, --} < {.}
```

The associativity of the binary operators is as follows:

- Left associative: `{,, ||, &&, |, &, ==, !=, <, >, <=, >=, <<, >>, +, -, *, /, %, .}`
- Right associative: `{!, ~}`

These definitions are based on the rules of C, and simplified according to the needs of Bossa. In particular, there is no associativity specified for the various assignment operators, because an assignment is not an expression in Bossa.

Primitives

The following primitive time functions are defined for both the version of Bossa with high-resolution timers and for the Bossa without high-resolution timers:

- `now() : unit -> time`
  The current time.
- `start_relative_timer(timer,offset) : timer * time -> unit`
  Set a timer for offset time units in the future.
- `start_absolute_timer(timer,time) : timer * time -> unit`
  Set a timer for the time time.
- `stop_timer(timer) : timer -> unit`
  Stop a timer.
- `time_to_ticks(t) : time -> int`
  Convert a time to a number of ticks (on Bossa with high-resolution timers, this is equivalent to time_to_jiffies, but is included for portability).
- `ticks_to_time(n) : int -> time`
  Convert a number of ticks to a time.

The following primitive time functions are only defined for the version of Bossa with high-resolution timers:

- `make_time(sec,nsec) : int * int -> time`
  Convert a pair of a number of seconds and a number of nanoseconds to the corresponding time.
- `make_cycle_time(jiffies,cycles) : int * cycles -> time`
  Convert a pair of a number of jiffies and a number of cycles to the corresponding time.
- `make_cycles(n) : int -> cycles`
  Cast an integer to a number of cycles.
- `time_to_jiffies(t) : time -> int`
  Drop the subjiffies component of a time.
- `time_to_subjiffies(t) : time -> cycles`
  Drop the jiffies component of a time.

The following primitive time functions are planned, but are unfortunately not currently implemented:

- `time_to_seconds(t) : time -> int`
  Drop the nanoseconds component of a time.
- `time_to_nanoseconds(t) : time -> int`
  Drop the seconds component of a time.

Other miscellaneous primitive functions are as follows:

- `print_trace_info() : void -> void`
  Print the accumulated trace information. Only defined if tracing is defined.
- `get_user_int(t) : port -> int`
  Get an integer value from a user-level address.