BLESS Syntax Card v0.9 (December 2, 2016)

```
action ::=
  basic_action
  | behavior_action_block
  | alternative | for_loop
  | forall_action
  | while_loop
  | do_until_loop
  | locking_action

actual_assertion_parameter ::=
  formal_identifier : actual_assertion_expression

actual_assertion_parameter_list ::=
  actual_assertion_parameter
    { , actual_assertion_parameter }*

actual_parameter ::= target | expression

alternative ::=
  if guarded_action { [] guarded_action }+ fi
  |
  if ( boolean_expression_or_relation ) behavior_actions
  { elsif ( boolean_expression_or_relation )
    behavior_actions }*
  [ else behavior_actions ]
  end if

array_range_list ::= natural_range { , natural_range }*

array_size ::= [ natural_value_constant  ]

array_type ::= array [ array_range_list ] of type

asserted_action ::=
  [ precondition_assertion ]
  action
  [ postcondition_assertion ]

assertion ::=
  << ( assertion_predicate
  | assertion_function
  | assertion_enumeration
  | assertion_enumeration_invocation ) >>

assertion_annex_library ::=
  annex Assertion {** { assertion }+ **} ;

assertion_enumeration ::=
  asserion_enumeration_label_identifier :
    parameter_identifier +=>
  enumeration_pair { , enumeration_pair }*

assertion_enumeration_invocation ::=
  +=> assertion_enumeration_label_identifier
  ( actual_assertion_parameter )

assertion_expression ::=
  assertion_subexpression
    [ { + assertion_subexpression }+
    | { * assertion_subexpression }+
    | - assertion_subexpression
    | / assertion_subexpression
    | ** assertion_subexpression
    | mod assertion_subexpression
    | rem assertion_subexpression ]
  | sum logic_variables [ logic_variable_domain ]
      of assertion_expression
  | product logic_variables [ logic_variable_domain ]
      of assertion_expression
  | numberof logic_variables [ logic_variable_domain ]
      that subpredicate

assertion_function ::=
  [ label_identifier : [ formal_assertion_parameter_list ] ]
   := ( assertion_expression | conditional_assertion_function )

assertion_function_invocation ::=
  assertion_function_identifier
    ( [ assertion_expression |
    actual_assertion_parameter
    { , actual_assertion_parameter }* ] )

assertion_predicate ::=
  [ label_identifier : [ formal_assertion_parameter_list ] : ]
  predicate

assertion_range ::=
  assertion_subexpression range_symbol assertion_subexpression

assertion_subexpression ::=
  [ - | abs ]  timed_expression
  | assertion_type_conversion

assertion_type_conversion ::=
  ( natural | integer | rational | real | complex | time )
  parenthesized_assertion_expression

assertion_value ::=
  now | tops | timeout
  | value_constant
  | variable_name
  | assertion_function_invocation
  | port_value

assignment ::=
  variable_name [ ' ] := ( expression | record_term | any )

(for subprograms)

basic_action ::=
  skip | assignment | simultaneous_assignment | when_throw
  | subprogram_invocation
```

```
(for threads)
basic_action ::=
  skip
  | assignment
  | simultaneous_assignment
  | communication_action
  | timed_action
  | when_throw
  | combinable_operation
  | issue_exception
  | computation_action

behavior_action_block ::=
  [ quantified_variables ] { [ behavior_actions ] }
  [ timeout behavior_time ]  [ catch_clause ]

behavior_actions ::=
  asserted_action
  | sequential_composition
  | concurrent_composition

behavior_annex ::=
  [ assert { assertion }+ ]
  [ invariant assertion ]
  [ variables ]
  states { behavior_state }+
  [ transitions ]

behavior_state ::=
  behavior_state_identifier
     : [initial] [complete] [final] state [ assertion ] ;

behavior_time ::=  integer_expression unit_identifier

behavior_transition ::=
    [ behavior_transition_label : ]
    source_state_identifier { , source_state_identifier }*
    -[ [ transition_condition ] ]-> destination_state_identifier
   [ { [ behavior_actions ] } ] [ assertion ] ;

behavior_transition_label ::=
    transition_identifier [ [ priority_natural_literal ] ]

behavior_variable ::=
  local_variable_declarator { , local_variable_declarator }*
  : [ modifier ] type [ := value_constant ] [ assertion ] ;

case_choice ::=
  ( boolean_expression_or_relation  ) -> expression

case_expression ::=
  ( case_choice { , case_choice }+ )

catch_clause ::=
  catch ( ( exception_label : basic_action ) )+

combinable_operation ::=
 fetchadd
 ( target_variable_name ,
   arithmetic_expression [, result_identifier] )
 | ( fetchor | fetchand | fetchxor )
 ( target_variable_name , boolean_expression
   [, result_identifier] )
 | swap
 ( target_variable_name , reference_variable_name
   , result_identifier )

communication_action ::=
  subprogram_invocation
  | output_port_name ! [ ( expression ) ]
  | input_port_name ? ( target )
  | frozen_input_port_name >>

completion_relative_timeout_catch ::= timeout behavior_time

component_element_reference ::=
  subcomponent_identifier | bound_prototype_identifier
  | feature_identifier | self

computation_action ::=
  computation ( behavior_time [ .. behavior_time ] )
  [ in binding ( processor_unique_component_classifier_reference
  { , processor_unique_component_classifier_reference }+ )]

concurrent_composition ::=
  asserted_action { & asserted_action }+

conditional_assertion_expression ::=
  ( predicate ?? assertion_expression
    : assertion_expression )

conditional_assertion_function ::=
    ( condition_value_pair { , condition_value_pair }* )

conditional_expression ::=
  ( boolean_expression_or_relation ??
    expression : expression )
  |
  ( if boolean_expression_or_relation then
    expression else expression )

condition_value_pair ::=
    parenthesized_predicate ->  assertion_expression

constant_number_range ::=
  [ [-] numeric_constant .. [-] numeric_constant  ]

data_component_name ::=
  { package_identifier :: }* data_component_identifier
   [ . implementation_identifier ]

declarator ::= identifier { array_size }*

dispatch_condition ::=
  on dispatch [ dispatch_expression ] [ frozen frozen_ports ]
```

```
dispatch_conjunction ::=
   dispatch_trigger  { and dispatch_trigger }*

dispatch_expression ::=
  dispatch_conjunction  { or dispatch_conjunction }*
  | stop
  | dispatch_relative_timeout_catch
  | completion_relative_timeout_catch
  | provides_subprogram_access_identifier

dispatch_relative_timeout_catch ::= timeout

dispatch_trigger ::= in_event_port_name | in_event_data_port_name
  | port_event_timeout_catch

do_until_loop ::=
  do
  [ invariant assertion ]
  [ bound integer_expression ]
  behavior_actions
  until( boolean_expression_or_relation )

enumeration_pair ::= enumeration_literal_identifier -> predicate

enumeration_type ::=
  enumeration
  ( defining_enumeration_literal_identifier
  { , defining_enumeration_literal_identifier }* )

event ::= < port_variable_or_state_identifier >

event_expression ::=
  [not] event
  | event_subexpression (and event_subexpression)+
  | event_subexpression (or event_subexpression)+
  | event - event

event_subexpression ::=
  [ always | never ] ( event_expression ) | event

event_trigger ::=
  in_event_subcomponent_port_reference
  | in_event_data_subcomponent_port_reference
  | ( trigger_logical_expression )

exception_label ::= ( exception_identifier )+ | all

execute_condition ::=
  boolean_expression_or_relation | timeout | otherwise

existential_quantification ::=
  exists logic_variables logic_variable_domain
  that predicate

expression ::=
  subexpression
  [ { + numeric_subexpression }+
  | { * numeric_subexpression }+
  |   - numeric_subexpression
  |   / numeric_subexpression
  |   mod natural_subexpression
  |   rem integer_subexpression
  |   ** numeric_subexpression
  | { and boolean_subexpression }+
  | { or boolean_subexpression }+
  | { xor boolean_subexpression }+
  |   and then boolean_subexpression
  |   or else boolean_subexpression  ]

expression_or_relation ::=
  subexpression [ relation_symbol subexpression ]

for_loop ::=
  for integer_identifier
    in integer_expression .. integer_expression
  [ invariant assertion ]
  { asserted_action }

forall_action ::=
  forall variable_identifier { , variable_identifier }*
    in integer_expression .. integer_expression
    behavior_action_block

formal_assertion_parameter ::=
  parameter_identifier [ ~ type_name ]

formal_assertion_parameter_list ::=
  formal_assertion_parameter
  { , formal_assertion_parameter }*

formal_expression_pair ::=
  formal_identifier => actual_expression

frozen_ports ::= in_port_name { , in_port_name }*

function_call ::=
  { package_identifier :: }*
  function_identifier ( [ function_parameters ] )

function_parameters ::=
  formal_expression_pair { , formal_expression_pair }*

guarded_action ::=
  ( boolean_expression_or_relation )~> behavior_actions

index_expression_or_range ::=
    integer_expression [ ..  integer_expression ]
```

```
integer_expression ::=
  [ - ]
  ( integer_assertion_value
  | ( integer_expression - integer_expression )
  | ( integer_expression / integer_expression )
  | ( integer_expression { + integer_expression }+ )
  | ( integer_expression { * integer_expression }+ ) )
internal_condition ::=
  on internal internal_port_name { or internal_port_name }*
issue_exception ::=
  exception
  ( [ exception_state_identifier , ] message_string_literal )
locking_action ::=
  *!< | *!>
  | required_data_access_name !<
  | required_data_access_name !>
logic_variable_domain ::=
  in ( assertion_expression range_symbol assertion_expression
    | predicate )
logic_variables ::=
  logic_variable_identifier { , logic_variable_identifier }*
  : type
logical_operator ::=
  and | or | xor | and then | or else
mode_condition ::=  on trigger_logical_expression
modifier ::= nonvolatile | constant |  shared |  spread | final

name ::=
  root_identifier { [ index_expression_or_range ] }*
    { . field_identifier { [ index_expression_or_range ] }* }*
natural_number ::=
 natural_integer_literal
 | natural_constant_identifier
 | natural_property
natural_range ::= natural_number [ .. natural_number ]
number_type ::=
  ( natural | integer | rational | real | complex | time )
  [ constant_number_range ]
  [ units aadl_unit_literal_identifier ]
numeric_constant ::= numeric_literal | numeric_property
parameter ::= [ formal_parameter_identifier : ] actual_parameter
parameter_list ::= parameter { , parameter }*
parenthesized_assertion_expression ::=
  ( assertion_expression )
  | conditional_assertion_expression
  | record_term
parenthesized_predicate ::= ( predicate )
port_name ::=
  { subcomponent_identifier . }* port_identifier
    [ [ natural_literal ] ]
port_event_timeout_catch ::=
  timeout ( port_identifier { [ or ] port_identifier }* )
    behavior_time
port_value ::=
  in_port_name ( ? | 'count | 'fresh | 'updated )
predicate ::=
  universal_quantification
  | existential_quantification
  | subpredicate
    [ { and subpredicate }+
    | { or subpredicate }+
    | { xor subpredicate }+
    | implies subpredicate
    | iff subpredicate
    | -> subpredicate ]
predicate_invocation ::=
  assertion_identifier
   ( [ assertion_expression | actual_assertion_parameter_list  ] )

predicate_relation ::=
  assertion_subexpression relation_symbol assertion_subexpression
  | assertion_subexpression in assertion_range
  | shared_integer_name += assertion_subexpression
property ::=
  property_constant | property_reference
property_constant ::=
  property_set_identifier :: property_constant_identifier
property_field ::=
  [ integer_value ]
  | . field_identifier
  | . upper_bound
  | . lower_bound
property_name ::= property_identifier { property_field }*
property_reference ::=
  ( # [ property_set_identifier :: ]
  | component_element_reference #
  | unique_component_classifier_reference #
  | self #  )
  property_name
quantified_variables ::= declare { behavior_variable }+
range_symbol ::= .. | ,. | ., | ,,
record_field ::= defining_field_identifier : type ;
```

```
record_term ::= ( { record_value }+ )

record_type ::= record ( { record_field }+ )

record_value ::= field_identifier => value ;

relation_symbol ::= = | < | > | <= | >= | != | <>

sequential_composition ::=
  asserted_action { ; asserted_action }+

simultaneous_assignment ::=
  ( variable_name [ ' ] { , variable_name [ ' ] }+
  :=
  ( expression | record_term | any )
    { , ( expression | record_term | any ) }+ )

subcomponent_port_reference ::=
  subcomponent_identifier { . subcomponent_identifier }*
  . port_identifier

subexpression ::=
  [ - | not | abs ]
  ( value | ( expression_or_relation )
  | conditional_expression | case_expression )

subpredicate ::=
  [ not ]
  ( true | false | stop
  | predicate_relation
  | timed_predicate
  | event_expression
  | def logic_variable_identifier )

subprogram_annex_subclause ::=
  annex Action {** subprogram_behavior **} ;

subprogram_behavior ::=
  [ assert { assertion }+ ]
  [ pre assertion ]
  [ post assertion ]
  [ invariant assertion ]
  behavior_action_block

subprogram_invocation ::=
  subprogram_name ( [parameter_list] )

subprogram_name ::=
  subprogram_prototype_name
  | required_subprogram_access_name
  | subprogram_subcomponent_name
  | subprogram_unique_component_classifier_reference
  | required_data_access_name . provided_subprogram_access_name
  | local_variable_name . provided_subprogram_access_name

target ::= local_variable_name | output_port_name

time_expression ::=
  time_subexpression
  | time_subexpression - time_subexpression
  | time_subexpression / time_subexpression
  | time_subexpression { + time_subexpression }+
  | time_subexpression { * time_subexpression }+

time_subexpression ::= [ - ]
  ( time_assertion_value
  | ( time_expression )
  | assertion_function_invocation )

timed_expression ::=
    ( assertion_value
      | parenthesized_assertion_expression
      | predicate_invocation )
    [ ' | ^ integer_expression | @ time_expression ]

timed_predicate ::=
  ( name | parenthesized_predicate | predicate_invocation )
  [ ' | @ time_expression | ^ integer_expression ]

transition_condition ::=
  dispatch_condition
  | execute_condition
  | mode_condition
  | internal_condition

transitions ::= transitions { behavior_transition }+

trigger_logical_expression ::=
  event_trigger { logical_operator event_trigger }*

type ::=
  type_name
  | number_type
  | enumeration_type
  | array_type
  | record_type
  | variant_type
  | boolean
  | string

type_name ::=
  { package_identifier :: }* data_component_identifier
    [ . implementation_identifier ]
  | natural | integer | rational | real
  | complex | time | string

unique_component_classifier_reference ::=
  { package_identifier :: }* component_type_identifier
  [ . component_implementation_identifier ]

universal_quantification ::=
  all logic_variables logic_variable_domain
  are predicate
```

(for subprograms)

```
value ::=
  variable_name | value_constant | function_call
  | incoming_subprogram_parameter_identifier | null
(for threads)
value ::=
  now | tops | timeout | null |
  | value_constant | in mode ( { mode_identifier }+ )
  | variable_name | function_call | port_value
value_constant ::=
  true | false | numeric_literal | string_literal
  | property_constant | property_reference
variables ::= variables { behavior_variable }+
variant_type ::=
  variant [ discriminant_identifier ]
  ( { record_field }+ )
when_throw ::=
  when ( boolean_expression ) throw exception_identifier
while_loop ::=
  while ( boolean_expression_or_relation )
  [ invariant assertion ]
  [ bound integer_expression ]
  behavior_action_block
```

Alphabetized Lexicon

```
base ::= digit [ digit ]
based_integer_literal ::=
  base # based_numeral # [ positive_exponent ]
based_numeral ::= extended_digit [_] extended_digit
character ::= graphic_character | format_effector
    | other_control_character
comment ::= --{non_end_of_line_character}*
complex_literal ::=
  [  [-] real_literal :  [-] imaginary_part_real_literal  ]
decimal_integer_literal ::= numeral
decimal_real_literal ::= numeral . numeral [ exponent ]
digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
exponent ::= (E|e) [+] numeral | (E|e) - numeral
extended_digit ::=
  digit | A | B | C | D | E | F | a | b | c | d | e | f
format_effector
  The control functions of ISO 6429 called character tabulation (HT),
  line tabulation (VT),  carriage return (CR), line feed (LF), and
  form feed (FF).

graphic_character ::= identifier_letter | digit | space_character
     | special_character
identifier ::= identifier_letter {[_] letter_or_digit}*
identifier_letter
   upper_case_identifier_letter | lower_case_identifier_letter
integer_literal ::= decimal_integer_literal | based_integer_literal

letter_or_digit ::= identifier_letter | digit
lower_case_identifier_letter
  Any character of Row 00 of ISO 10646 BMP whose name begins
  Latin Small Letter.
numeral ::= digit {[_] digit}*
numeric_literal ::=
  integer_literal | real_literal | rational_literal | complex_literal

other_control_character
  Any control character, other than a format_effector, that is allowed
  in a comment; the set of other_control_functions allowed in comments
  is implementation defined.

rational_literal ::=
  [ [-] dividend_integer_literal |  [-] divisor_integer_literal  ]

real_literal ::= decimal_real_literal
space_character
  The character of ISO 10646 BMP named Space.
special_character
  Any character of the ISO 10646 BMP that is not reserved for a control
  function, and is not the space_character, an identifier_letter,
  or a digit.

string_element ::= "" | non_string_bracket_graphic_character
string_literal ::= "{string_element}*"
upper_case_identifier_letter
   Any character of Row 00 of ISO 10646 BMP whose name begins
  Latin Capital Letter.
```