```
action ::=
  basic_action
  | behavior_action_block
  | alternative
  | for_loop
  | forall_action
  | while_loop
  | do_until_loop
actual_assertion_parameter ::=
  formal_identifier : actual_assertion_expression
actual_assertion_parameter_list ::=
  actual_assertion_parameter { , actual_assertion_parameter }*
actual_parameter ::=
  [ formal_parameter_identifier : ]
  ( variable_name | value_constant )
add_subtract ::=
  multiply_divide
  [ { + multiply_divide }+
  | - multiply_divide ]
alternative ::=
  if guarded_action { [] guarded_action }+ fi
  |
  if ( boolean_expression ) behavior_actions
  { elsif ( boolean_expression )
    behavior_actions }*
  [ else behavior_actions ]
  end if
array_range_list ::= whole_range { , whole_range }*
array_type ::=
  array [ array_range_list ] of type_or_reference
asserted_action ::=
  [ precondition_assertion ]
  action
  [ postcondition_assertion ]
assertion ::=
  << ( assertion_predicate
  | assertion_function
  | assertion_enumeration ) >>
assertion_add_subtract ::=
  assertion_multiply_divide
  [  { + assertion_multiply_divide }+
     |  - assertion_multiply_divide ]
assertion_annex_library ::=
  annex Assertion {** [ ghost_variables ]
   { labeled_assertion }+ **} ;
assertion_enumeration ::=
  asserion_enumeration_label_identifier :
    parameter_identifier ~ enumeration_type_identifier +=>
  enumeration_pair { , enumeration_pair }*
assertion_exponentiation ::=
  assertion_subexpression [ ** assertion_subexpression ]
```

```
assertion_expression ::=
  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_add_subtract
assertion_function ::=
  [ label_identifier : [ variable_list ] ]
   returns type_or_reference
   := ( assertion_expression | conditional_assertion_function )
assertion_function_invocation ::=
  assertion_function_identifier
    ( [ assertion_expression |
    actual_assertion_parameter
    { , actual_assertion_parameter }* ] )
assertion_multiply_divide ::=
  assertion_exponentiation
  [ ( / | div | mod | rem ) assertion_exponentiation
    | { * assertion_exponentiation }+ ]
assertion_predicate ::=
  [ label_identifier : [ variable_list ] : ]
  predicate
assertion_range ::=
  assertion_subexpression range_symbol assertion_subexpression
assertion_subexpression ::=
  [ - | abs | truncate | round ] timed_expression
assertion_value ::=
  now
  | tops
  | timeout
  | value_constant
  | variable_name
  | port_value
assignment ::=
  assignment_target := ( expression | record_term | any )
(for subprograms)
assignment_target ::=
  variable_name
  | outgoing_subprogram_parameter_identifier
(for threads)
assignment_target ::=
  ( variable_name
  | outgoing_port_name
  | internal_port_name )
  [ ' ]
```

(for subprograms)
basic_action ::=
skip
| assignment
| simultaneous_assignment
| when_throw
| subprogram_call
| combinable_operation

(for threads)
basic_action ::=
skip
| assignment
| simultaneous_assignment
| when_throw
| combinable_operation
| communication_action
| **setmode** *mode*_identifier

behavior ::=
[**assert** { assertion }+]
[**invariant** assertion]
[**variables** { variable_declaration }+]
[**states** { behavior_state }+]
[**transitions** { behavior_transition }+]

behavior_action_block ::=
[local_variables] **{** behavior_actions **}**
[**timeout** behavior_time] [catch_clause]

behavior_actions ::=
asserted_action
| sequential_composition
| concurrent_composition

behavior_state ::=
*behavior_state*_identifier **:**
[**initial** | **complete** | **final**] **state**
[assertion] [**;**]

behavior_time ::=
numeric_constant
| *variable*_name
| port_value
| parenthesized_expression

bless_annex_subclause ::= **annex BLESS** {** behavior **} **;**

case_choice ::= **(** *boolean*_expression **->** expression **)**

case_expression ::= **case** { 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_call
| *output_port*_identifier **!** [**(** expression **)**]
| *input_port*_identifier **?** **(** *local_variable*_name **)**

completion_relative_timeout ::= **timeout** behavior_time

component_element_reference ::=
*subcomponent*_identifier
| *bound_prototype*_identifier
| *feature*_identifier
| **self**

concurrent_composition ::=
asserted_action { **&** asserted_action }+

conditional_assertion_expression ::=
(predicate **??** assertion_expression **:** assertion_expression **)**
| **(** **if** predicate **then** assertion_expression
else assertion_expression **)**

conditional_assertion_function ::=
[condition_value_pair { **,** condition_value_pair }* **]**

conditional_expression ::=
(*boolean*_expression **??** expression **:** expression **)**
| **(** **if** *boolean*_expression **then** expression **else** expression **)**

condition_value_pair ::= **(** predicate **)->** assertion_expression

conjunction ::=
relation
[{ **and** relation }+
| { **and then** relation }+]

disjunction ::=
conjunction
[{ **or** conjunction }+
| { **or else** conjunction }+
| { **xor** conjunction }+]

dispatch_condition ::= **on dispatch** [dispatch_expression]

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

dispatch_expression ::=
dispatch_conjunction { **or** dispatch_conjunction }*

dispatch_relative_timeout ::= **timeout**

```
dispatch_trigger ::=
  in_event_port_name
  | in_event_data_port_name
  | port_event_timeout_catch
  | dispatch_condition_reletive_timeout
  | completion_reletive_timeout
  | stop
do_until_loop ::=
  do
  [ invariant assertion ]
  [ bound integer_expression ]
  behavior_actions
  until ( boolean_expression )
enumeration_pair ::=
  enumeration_literal_identifier -> predicate
enumeration_type ::=
  enumeration ( { enumeration_literal_identifier }+ )
enumeration_value ::=
  enumeration_type_identifier ' enumeration_literal_identifier
exception_label ::= { exception_identifier }+ | all
execute_condition ::= boolean_expression
existential_quantification ::=
  exists logic_variables
  ( in assertion_subexpression range_symbol
      assertion_subexpression
  | which predicate )
  that predicate
exponentiation ::=
  subexpression
  [ ** subexpression ]
expression ::=
  disjunction
  [ ( iff | implies ) disjunction ]
for_loop ::=
  for integer_identifier
    in integer_expression .. integer_expression
  [ invariant assertion ]
  { behavior_actions }
forall_action ::=
  forall { variable_identifier }+
    in integer_expression .. integer_expression
    behavior_action_block
function_call ::=
  { package_identifier :: }*
  function_identifier (
    [ function_parameter {  , function_parameter }+
    | expression ] )
function_parameter ::= formal_identifier : actual_expression
ghost_variables ::= ghost variables { ghost_variable }+
ghost_variable ::= def ghost_variable
guarded_action ::= ( boolean_expression )~> asserted_action
```

```
index_expression ::=
  period_shift
  [ - period_shift
  | div period_shift
  | mod period_shift
  | { + period_shift }+
  | { * period_shift }+ ]
integer_expression_or_range ::=
  integer_expression [ ..  integer_expression ]
internal_condition ::=
  on internal internal_port_name { or internal_port_name }*
issue_exception ::=
  exception ( exception_identifier [ message_string_literal ] )
local_variables ::= declare { variable }+
logic_variable_domain ::=
  in assertion_expression range_symbol assertion_expression
  | which predicate
logic_variables ::= logic_variable { , logic_variable }*
modifier ::=
  nonvolatile | constant |  shared |  spread | final
multiply_divide ::=
  exponentiation
  [ { * exponentiation }+
  | ( / | div | mod | rem ) exponentiation ]
name ::=
  root_identifier { [ integer_expression_or_range ] }*
    { . field_identifier { [ integer_expression_or_range ] }* }*
numeric_constant ::=
  quantity
  | property_constant
  | property_reference
parameter_list ::= actual_parameter { , actual_parameter }*
parenthesized_assertion_expression ::=
  ( assertion_expression )
  | conditional_assertion_expression
  | record_term
parenthesized_predicate ::= ( predicate )
period_shift ::= [ - ] ( integer_value | ( index_expression ) )
port_name ::= port_identifier [ [ natural_literal ] ]
port_event_timeout ::=
  timeout ( port_identifier { [ or ] port_identifier }* )
    behavior_time
port_value ::=
  in_port_name [ ? | 'count | 'fresh | 'updated ]
predicate ::=
  universal_quantification
  | existential_quantification
  | predicate_disjunction
    [ ( implies | iff ) predicate_disjunction ]
```

```
predicate_conjunction ::=
  subpredicate
  [ { and subpredicate }+
  | { and then subpredicate }+ ]
predicate_disjunction ::=
  predicate_conjunction
  [ { or predicate_conjunction }+
  | { or else predicate_conjunction }+
  | { xor predicate_conjunction }+ ]
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_constant ::=
  property_set_identifier :: property_constant_identifier
property_field ::=
  [ integer_value ]
  | . field_identifier
  | . upper_bound
  | . lower_bound
property_name ::=
  [ property_set_identifier ::  ]
  property_identifier { property_field }*
property_reference ::=
  ( #
  | component_element_reference #
  | unique_component_classifier_reference #
  | self #  )
  property_name
quantity ::= numeric_literal [ unit_identifier | scalar ]
quantity_type ::=
  quantity ( unit_identifier | scalar | whole )
  [ [ number .. number ] ]
  [ step number ]
  [ representation property_constant ]
range_symbol ::= .. | ,. | ., | ,,
record_field ::= defining_field_identifier : type_or_reference
record_term ::= ( { record_value }+ )
record_type ::= record ( { record_field }+ )
record_value ::= field_identifier => value [ ; ]
relation ::=
  add_subtract
  [ relation_symbol add_subtract
  | in range ]
relation_symbol ::= = | < | > | <= | >= | != | <>
sequential_composition ::= asserted_action { ; asserted_action }+
```

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

subexpression ::=
  [ - | not | abs | truncate | round ]
  ( value
  | ( expression )
  | conditional_expression
  | case_expression )

subpredicate ::=
  predicate_relation
  | [ not ] timed_predicate

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

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

subprogram_call ::= subprogram_name ( [parameter_list] )

subprogram_name ::= required_subprogram_access_name

timed_expression ::=
  ( assertion_value
  | parenthesized_assertion_expression
  | assertion_function_invocation )
  [ ' | ^ period_shift | @ time_subexpression ]

timed_predicate ::=
  ( name
  | parenthesized_predicate
  | predicate_invocation )
  [ ' | @ time_subexpression | ^ period_shift ]

transition ::=
  transition_label :
  source_state_identifier { , source_state_identifier }*
  -[ [ transition_condition ] ]->
  destination_state_identifier
  [ { [ behavior_actions ] } ] [ ; ]

transition_condition ::=
  dispatch_condition
  | execute_condition
  | internal_condition
```

```
type ::=
  quantity_type
  | enumeration_type
  | array_type
  | record_type
  | variant_type
  | boolean
  | string
  | null
type_declaration ::= type type_identifier is type
type_or_reference ::= type | type_identifier
typedef_annex_library ::=
  annex Typedef {** { type_declaration }+ **} ;
unit_annex_library ::=
  annex Unit {** { unit_declaration }+ **} ;
unique_component_classifier_reference ::=
  { package_identifier :: }* component_type_identifier
  [ . component_implementation_identifier ]
unit_declaration ::=
  ( base | unit_formula ) [ { descriptive_identifier }+  ]
  unit_name { unit_factor }* ;
unit_factor ::= , unit_name ( * | / ) positive_numeric_literal
unit_formula :=
  { base_unit_identifier }+ [ / { base_unit_identifier }+ ]
unit_name ::= [ < { descriptive_identifier }+ > ] unit_identifier
universal_quantification ::=
  all logic_variables logic_variable_domain
  are predicate
(for subprograms)
value ::=
  variable_name | value_constant | function_call
  | enumeration_value | incoming_subprogram_parameter_identifier
(for threads)
value ::=
  now | tops | value_constant | port_value
  | variable_name | function_call | enumeration_value
value_constant ::=
  true | false | null | quantity | string_literal
  | property_constant | property_reference
variable ::= identifier ~ type_or_reference
variable_declaration ::=
  variable [ modifier ] [ := constant_expression ]
    [ assertion ] [ ; ]
variable_list ::= variable { , variable }*
variant_type ::=
  ( variant | union ) [ discriminant_identifier ]
  ( { record_field }+ )
when_throw ::=
  when ( boolean_expression ) throw exception_identifier
  [ message_string_literal ]
```

```
while_loop ::=
  while ( boolean_expression )
  [ invariant assertion ]
  [ bound integer_expression ]
  behavior_action_block
```

```
whole_range ::= whole_number [ .. whole_number ]
```

Lexicon

```
character ::= graphic_character | format_effector
    | other_control_character
comment ::= --{non_end_of_line_character}*
digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
digits ::= { digit }+
escaped_character ::= \b | \t | \n | \f | \r | \u | \` | \' | \\
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
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.
numeric_literal ::= [-] digits [ . digits [ e [-] digits ] ]
other_control_character
  Any control character, other than a format_effector,
  that is allowed in a comment
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 | escaped_character
string_literal ::= `{string_element}*'
upper_case_identifier_letter
   Any character of Row 00 of ISO 10646 BMP whose name begins
  Latin Capital Letter.
```