Architecture Analysis & Design Language (AADL) V2 Hybrid Annex DOCUMENT

This Architecture Analysis & Design Language (AADL) standard document was prepared by the SAE AS-2C Architecture Description Language Subcommittee, Embedded Computing Systems Committee, Aerospace Avionics Systems Division.

Table of Contents

[A.1 Scope 3](#_Toc410615476)

[A.2 Hybrid Annex (HA) Sublanguage Structure 3](#_Toc410615477)

[A.3 Assert 4](#_Toc410615478)

[A.4 Invariant 4](#_Toc410615479)

[A.5 Variables 5](#_Toc410615480)

[A.6 Constants 7](#_Toc410615481)

[A.7 Channels 8](#_Toc410615482)

[A.8 Behavior 9](#_Toc410615483)

[A.8.1 Skip Activity 10](#_Toc410615484)

[A.8.2 Assignment Activity 10](#_Toc410615485)

[A.8.3 Boolean Assignment Activity 10](#_Toc410615486)

[A.8.4 Wait Activity 11](#_Toc410615487)

[A.8.5 Communication 12](#_Toc410615488)

[A.8.6 Sequential Composition 15](#_Toc410615489)

[A.8.7 Concurrent Composition 16](#_Toc410615490)

[A.8.8 Choice 16](#_Toc410615491)

[A.8.9 Continuous Evolution 18](#_Toc410615492)

[A.8.9.1 Differential Expression 19](#_Toc410615493)

[A.8.9.2 Differentials 19](#_Toc410615494)

[A.8.9.3 Derivative Expression 20](#_Toc410615495)

[A.8.9.4 Derivative Time 21](#_Toc410615496)

[A.8.10 Interrupts 23](#_Toc410615497)

[A.8.10.1 Timed Interrupt 23](#_Toc410615498)

[A.8.10.2 Communication Interrupt 24](#_Toc410615499)

[A.8.11 Repetition 26](#_Toc410615500)

[A.8.12 Boolean and Numeric Expressions 27](#_Toc410615501)

[A.9 Informative References 29](#_Toc410615502)

Hybrid Annex

Normative

* 1. Scope

1. This Hybrid Annex document defines an annex sublanguage of AADL to allow continuous behavior specifications to be attached to AADL components. The purpose of the Hybrid Annex sublanguage of AADL is modeling continuous behavior of environments external to the system being designed, sensors and actuators, and the interaction of cyber-physical control systems with their environment.
2. This document is organized as follows:
   * Section A.2 defines the structure of Hybrid Annex subclauses.
   * Section A.3 defines the syntax and semantics of assertion declarations.
   * Section A.4 defines the syntax and semantics of invariant declarations.
   * Section A.5 defines the syntax and semantics of variable declarations.
   * Section A.6 defines the syntax and semantics of constantdeclarations.
   * Section A.7 defines the syntax and semantics of communication channels within Hybrid Annex subclauses.
   * Section A.8 defines the syntax and semantics of behaviordeclarations.
   * Section A.9 provides informative references
   1. Hybrid Annex (HA) Sublanguage Structure
3. HA subclauses can be attached to AADL device component implementations to model continuous behavior of sensors and actuators or abstract component implementations to model continuous behavior of environments. HA subclauses attached to component types are assumed to apply to all their implementations.
4. Like AADL, HA is case insensitive.
5. The word “Activity” is used to described the change in the behavior of the AADL component for which the HA subclause is defined. It serves the same as the word “Process” in process algebras.

Syntax

hybrid\_annex ::=

[ **assert** { Assertion }+ ]

[ **invariant** Assertion ]

[ **variables** { variable\_declaration }+ ]

[ **constants** constant\_declaration]

[ **channels** { channel\_declaration }+ ]

**behavior** { behavior\_declaration }+

* 1. Assert

1. To facilitate analysis of AADL architectures together with their environments, HA subclauses may use one or more Assertion as behavior interface specification language (BISL) in an **assert** section. Syntax and semantics of Assertion are defined in the AADL v2 BLESS Annex Document (AS5506-- submitted for standardization).

Naming Rule

1. Assertion labels (if any) are globally-visible and must be unique.
   1. Invariant
2. An **invariant** section has a single Assertion that must always be true about behavior of the HA subclause containing it.

Example

**abstract** **implementation** InvariantExp.impl

**annex** hybrid **{\*\***

**assert**

<< NORMAL: : ( c@now < (u+Iso\_Prperties::Tolerance) ) and (c@now > (l-Iso\_Properties::Tolerance) ) >>

<< EA\_IS\_1: : forall x:BLESS\_Types::Time in 0.0 ,, now are ( c@now – c@x) <= Iso\_Properties::Heat\_Rate\*(now-x)>>

<< EA\_IS\_2: : forall x:BLESS\_Types::Time in 0.0 ,, now are (c@now – c@x) >= Iso\_Properties::Cool\_Rate\*(now-x)>>

**invariant**

<< NORMAL() and EA\_IS\_1() and EA\_IS\_2() >>

**behavior**

Null::= skip

**\*\*};**

**end** InvariantExp.impl;

* 1. Variables

1. Local variables in the scope of current annex subclause are declared in **variables** section along with data type. Variables can be used to represent values of physical properties in the entity being modeled.
2. Depending on AADL component for which the HA subclause is defined, a variable may either be discrete or continuous.
3. A variable's type is specified by data\_component\_classifier\_reference to an AADL data component.

Syntax

variable\_declaration ::=

*variable*\_identifier { **,** *variable*\_identifier }+ **:**

data\_component\_classifier\_reference

Legality Rules

1. The data component referenced must either be part of the package containing the component with the HA subclause or must be declared in scope of a package imported using with clause.

Semantics

1. Values of local variables are persistent throughout the lifetime of their containing component.

Example

-- Following example shows the use of variables section to -- declare different types of variables

**package** Variable\_Example **public**

**with Base\_Types, BLESS\_Types;**

**abstract** VarExpSys

**end** VarExpSys;

**abstract** **implementation** VarExpSys.impl

**annex** hybrid **{\*\***

**variables**

t\_clk, c\_clk : BLESS\_Types::Time

water\_level, speed : Base\_Types::Float

counter : Base\_Type::Integer

**behavior**

Null::= skip

**\*\*};**

**end** VarExpSys.impl;

**end** Variable\_Example;

* 1. Constants

1. Constants in the scope of a HA subclause are declared in a **constants** section.

Syntax

constant\_declaration ::=

behavior\_constant { **,** behavior\_constant }\*

behavior\_constant ::=

*behavior\_constant\_*identifier

**=** ( integer\_literal | real\_literal ) [*unit\_*identifier]

Legality Rules

1. Constants can only be initialized at declaration and cannot be assigned any value afterwards.
2. A constant must be initialized with either integer\_literal or real\_literal value.

Example

-- Following example shows constants declaration

**package** Constant\_Example **public**

**abstract** ConstExpSys

**end** ConstExpSys;

**abstract** **implementation** ConstExpSys.impl

**annex** hybrid **{\*\***

**constants**

pi = 3.14159, g = 9.8 mpss, u = 1

r = 0.0254 cm

**behavior**

Null::= skip

**\*\*};**

**end** ConstExpSys.impl;

**end** Constant\_Example;

* 1. Channels

1. The **channels** section is used to declare communication channels used among behavior Activities defined within the same HA subclause.
2. Channels declarations contain AADL data component classifier references to specify type of the data sent or received along a particular channel.

Syntax

channel\_declaration ::=

*channel*\_identifier {**,** *channel*\_identifier }\* :

data\_component\_classifier\_reference

Legality Rules

1. HA channels only support unidirectional communication, as a result an Activity cannot use same channel for both input and output communication events.

Example

-- Following example shows channel declarations

**package** Channel\_Example **public**

**with** Base\_Types, BLESS\_Types**;**

**abstract** ChExpSys

**features**

idp : in data port Base\_Types::Float;

odp : out data port Base\_Types::Float;

**end** ChExpSys;

**abstract** **implementation** ChExpSys.impl

**annex** hybrid **{\*\***

**channels**

Plant2Con : BLESS\_Types::Real

Con2Plant : Base\_Types::Integer

ch1, ch2 : Base\_Types::Integer

**behavior**

Null ::= skip

**\*\*};**

**end** ChExpSys.impl;

**end** Channel\_Example;

* 1. Behavior

1. The **behavior** section in HA subclause is used to specify continuous behavior of the component in terms of concurrently-executing Activities.

Syntax

behavior\_declaration ::=

*behavior\_*identifier **::=**

activity\_declaration { **&** activity\_declaration }\*

activity\_declaration ::=

**skip** |assignment | boolean\_assignment | **wait** time\_value

| communication | sequential\_composition

| concurrent\_composition | choice

| continuous\_evolution | repetition

Semantics

1. Ampersand sign (**&**) is used to separate activity declarations within a behavior declaration.
   * 1. Skip Activity
2. The skip Activity terminates immediately having no effect on variable values.
   * 1. Assignment Activity
3. The assignment Activity assigns the value of expression to the local variable declared in **variables** section.

Syntax

assignment ::=

*variable\_*identifier **:=** numeric\_expression

* + 1. Boolean Assignment Activity

1. The boolean\_assignment Activity assigns the boolean value to a local boolean type variable declared in **variables** section.

Syntax

boolean\_assignment ::=

*boolean\_variable\_*identifier **:=** boolean\_expression

* + 1. Wait Activity

1. The waitkeeps idle for time\_value time.
2. During this idle period the respective Activity does not perform any action and the variables are unchanged.

Syntax

time\_value ::=

*time\_variable*\_identifier **|** real\_literal time\_unit

Semantics

1. A *time\_variable\_*identifier can be a local variable identifier declared in **variables** section.
2. A *time\_variable\_*identifier can be a constant identifier declared in **constants** section.
3. A time\_unit defines a unit of measurement of time.
4. A time\_unit can be any time unit declared in Time\_Units enumerated property set within the project specific property set in AS5506B: **ps**, **ns**, **us**, **ms**, **sec**, **min**, **hr**.

Example

-- Following example shows use of skip, assignment and wait -- Activities declarations

**package** Mix\_Example **public**

**with** Base\_Types, BLESS\_Types**;**

**abstract** MixExpSys

**features**

idp : in data port Base\_Types::Float;

odp : out data port Base\_Types::Float;

**end** MixExpSys;

**abstract** **implementation** MixExpSys.impl

**annex** hybrid **{\*\***

**variables**

x, y : Base\_Types::Integer

**constants**

w = 200, z = 100

period = 100 ms

**behavior**

Act1 ::= wait period & x := w & Act3

Act2 ::= wait 5 sec & y := z

Act3 ::= skip

**\*\*};**

**end** MixExpSys.impl;

**end** Mix\_Example;

* + 1. Communication

1. Communication within HA subclauses uses channels. Communication with other AADL components uses ports. Both port and channel communication occurs at discrete events.
2. Communication events are of two types: input event and output event.
3. A port input event p1?(x) specifies that an input value is received from port p1 and stored in a local variable x.
4. A port output event p2!(y) specifies that an output value of variable y is sent out port p2.
5. A channel input event ch1?z specifies that an input value is received from channel ch1 and stored in a local variable z.
6. A channel output event ch2!w specifies that an output value of variable w is sent out channel ch2.

Syntax

communication ::= port\_communication | channel\_communication

port\_communication ::=

*port\_*identifier ( **?** | **!** ) **(** [ *variable\_*identifier ] **)**

channel\_communication ::=

*channel\_*identifier ( **?** | **!** )[*variable\_*identifier ]

Naming Rules

1. Port identifiers must be names of ports of the component containing the HA subclause in which they are used.
2. Channel identifiers must be names of channels declared in the **channels** section of the HA subclause in which they are used.

Semantics

1. Channel communication synchronizes involved Activities and can only occur when both sender and receiver are ready.
2. Channel communication may cause either sender or receiver to wait.
3. Port communication has semantics of AADL core language.
4. Port communication may send (!) only to out data, out event or out event data ports.
5. Port communication may receive (?) only from in data, in event or in event data ports.
6. Port communication to out data ports updates the value of the port.
7. Port communication from in data ports receives the most-recent value of the port.
8. Port communication to out event data ports issues an event (with data) on the port.
9. Port communication from in event data ports receives the most-recent fresh value of the port, or blocks until fresh data becomes available.
10. Port communication without parameter is event port transmission or receipt.

Examples

-- Following code snippet shows communication modeling based on –- input and output events on ports and channels

**package** Communication\_Example **public**

**with** Base\_Types, BLESS\_Types**;**

**abstract** ComExpSys

**features**

idp : in data port Base\_Types::Float;

odp : out data port Base\_Types::Float;

**end** ComExpSys;

**abstract** **implementation** ComExpSys.impl

**annex** hybrid **{\*\***

**variables**

x, y, z : Base\_Types:: Float

**constants**

w = 200

period = 100 ms

**channels**

ch1, ch2 : Base\_Types::Float

**behavior**

Act1 ::= wait period & idp?(x) & ch1!x

Act2 ::= ch1?y & ch2!w & odp!(y)

Act3 ::= ch2?z

**\*\*};**

**end** ComExpSys.impl;

**end** Communication\_Example;

* + 1. Sequential Composition

1. Sequential composition defines consecutively-executing behaviors.

Syntax

SequentialComposition ::=

**{** *behavior\_*identifier { **;** *behavior\_*identifier }+ **}**

Legality Rule

1. Behavior identifiers used in sequential composition must refer to behavior declarations.

Semantics

1. A sequentially composed Activity {P ; Q} behaves as P first and after its successful termination, behaves as Q.
   * 1. Concurrent Composition
2. A parallel composed Activity S1||S2 behaves as if S1 and S2 run independently except that all interactions occur through communication events.
3. Communication events between concurrently-composed behaviors, must occur along the common communication channels declared in the **channels** section connecting Activities S1 and S2.

Syntax

ConcurrentComposition ::=

**{** *behavior\_*identifier { **||** *behavior\_*identifier }+ **}**

Legality Rules

1. Behaviors defined using concurrent composition may not themselves be used in either sequential or concurrent compositions.
2. Communication channels must be shared pair-wise with complementary directions – *in* communication with *out* communication.
3. Variables used in shared communication channels must have the same type.
   * 1. Choice
4. Internal execution choice between Activities P and Q, denoted as P **[]** Q is resolved by the Activity itself.

Syntax

choice ::=

alternative { **[]** alternative }\*

alternative ::=

**(** boolean\_expression **) ->** *behavior\_*identifier

Semantics

1. Choice executes an Activity with an alternative having true boolean\_expression.
2. When more than one alternative has true boolean\_expression the choice is resolved non-deterministically.
3. When no alternative has true boolean\_expression the choice is equivalent to skip.
4. The alternative Activity (B) -> P behaves as P only if the boolean expression B is true and terminates otherwise.

Examples

-- Following code snippet shows behavior modeling using choice -- Activity construct

**package** Choice\_Example **public**

**with** Base\_Types, BLESS\_Types**;**

**abstract** ChoiceExpSys

**features**

idp : in data port Base\_Types::Float;

odp : out data port Base\_Types::Float;

idp2: in data port Base\_Types::Integer;

**end** ChoiceExpSys;

**abstract** **implementation** ChoiceExpSys.impl

**annex** hybrid **{\*\***

**variables**

w, x, y, z : Base\_Types:: Float

**channels**

ch1 : Base\_Types::Float

**behavior**

Act1 ::= idp?(x) & (x mod 2 = 0)-> EAct [](x mod 2<>0)-> OAct

EAct ::= ch1!x & odp!(x)

OAct ::= {x:=x+1 ; ch1!x ; odp!(x)}

Act2 ::= idp2?(w) & (w >=5 **and** w mod 2 = 0 )-> skip

**\*\*};**

**end** ChoiceExpSys.impl;

**end** Choice\_Example;

* + 1. Continuous Evolution

1. Specification of a continuous evolution consists of a differential equation controlled by boolean expressions.

Syntax

continuous\_evolution ::=

' differential\_expression **=** differential\_expression '

[ **<** boolean\_expression **>** ]

[ interrupt ]

Semantics

1. The continuous\_evolution statement forces values of variables declared in variables section to obey its differential equation as long as the boolean\_expression is true, until an interrupt occurs.
2. The boolean\_expression specifies the boundary condition of the variables.
3. The continuous evolution terminates as soon as boolean\_expression turns to false.
4. The interruption of continuous evolution due to boundary condition is known as boundary interrupt. Continuous evolution may be preempted due to a timed or communication interrupt as presented in A.8.10.
   * + 1. Differential Expression
5. A differential expression is composed of differential terms combined using standard multiplication (\*), addition (+), and subtraction symbols (-).

Syntax

differential\_expression ::=

differential

|differential { **\*** differential }+

|differential { **+** differential }+

| differential **-** differential

Semantics

1. Multiple differentials are combined with multiplication (\*) and additions (+) signs to form a differential expression.
2. Only one occurrence of the subtraction sign (-) is allowed within a particular differential expression. Hence, only two differentials can be combined using subtraction sign (-) to form a differential expression.
   * + 1. Differentials
3. A differential term may contain numeric\_literal or *behavior\_variable*\_ identifier with power operator (^).
4. A differential term may also contain a derivative\_expression or derivative\_time for continuous evolution of dependent variable with respect to independent variable or with respect to time respectively.
5. A differential term may be a parenthesized differentinal\_expression.

Syntax

differential ::=

[**-**] numeric\_literal

| *variable\_*identifier [ **^** numeric\_literal ]

| derivative\_expression

| derivative\_time

| **(** differentinal\_expression **)**

* + - 1. Derivative Expression

1. A derivate expression is specified using keyword **DE** followed by the order of the differential equation, a dependant variable and an independent variable.

Syntax

derivative\_expression ::=

**DE** *order*\_integer\_literal*dependent\_variable\_*identifier

*independent\_variable\_*identifier

Semantics

1. The *order*\_integer\_literal determines the order of the derivative expression, and must be positive.
2. The dependant variable is written before the independent variable. For example rate of change of variable y with respect to x denoted as is specified as **DE** 1 y x and is specified as **DE** 2 y x. Here, 1 and 2 are the order of the differential equation.



* + - 1. Derivative Time

1. Time derivation shows the rate of change of a variable with respect to the rate of change of time.
2. Time derivation is specified using keyword **DT** followed by the order of time derivation and a dependent variable.
3. Taking time as independent variable, time derivation only contains a dependant variable.

Syntax

derivative\_time ::=

**DT** *order*\_integer\_literalvariable\_identifier

Semantics

1. The *order*\_integer\_literal determines the order of the derivative time.
2. The dependant variable is written after the order in derivative time. For example rate of change of variable y with respect to time t denoted as is specified as **DT** 1 y.



Examples

-- Following code snippet illustrates specification of -- continuous evolution to using ODEs and PDEs

**abstract** **implementation** ConEvolSys.impl

**annex** hybrid **{\*\***

**variables**

s, v, x, y, z, u : Base\_Types::Float

t : BLESS\_Types::Time

**constants**

c = 0.0123, alpha = 19 mmsps

**behavior**

-- Following Activity Train specifies the behavior of a running -- train, where s is displacement, v is velocity, a is -- acceleration, and t is time.

Train ::= ' DT 1 s = v ' & ' DT 1 v = a ' & ' DT 1 t = 1 '

-- Following Activity Wave\_Equation specifies the wave behavior -- in a one dimension space using PDE. Where c is a constant and -- must be declared in constants clause with appropriate value.

Wave\_Equation ::= ' DT 2 y = (c^2) \* DE 2 y x '

-- Following Activity Heat\_Equation specifies the temperature -- change in a 3D space. Where alpha is thermal diffusivity of –- the material or substance in use, u is the temperature, and -- x, y, z are the dimensions. For air at 300 k temperature -- alpha is 19mm2/sec as declared in constants clause.

Heat\_Equation ::= ' DT 1 u - (alpha\* ( (DE 2 u x) + (DE 2 u y) + (DE 2 u z) )) = 0 '

**\*\*};**

**end** ConEvolSys.impl;

* + 1. Interrupts

1. Continuous evolution can be interrupted due to either timed and communication interrupts.

Syntax

interrupt ::= timed\_interrupt | communication\_interrupt

Semantics

1. Continuous evolution termination after specific time and communication event is realized through timed and communication interrupts respectively.
2. If the continuous evolution does not terminate due the boundary condition, timed and communication interrupts can be used for termination.
   * + 1. Timed Interrupt
3. Timed interrupt preempts continuous evolution after a specific time value and control follows the next specified Activity.

Syntax

timed\_interrupt ::=

**[>** time\_value **]>**  *activity\_*identifier

Semantics

1. An Activity with continuous evolution, boundary interrupt and time interrupt, continues its evolution if it terminates due to boundary interrupt before TimeValue time units. Otherwise, after time\_value time, the Activity behaves like the next specified Activity.

Example

-- Following code snippet illustrates specification of -- Timed Interrupt. In this example, the continuous evolution of -- the velocity of a running train is interrupted and is sent to -- out data port (speed) after every 100 ms.

**abstract** **implementation** TimedInrptSys.impl

**annex** hybrid **{\*\***

**variables**

s, v, a : Base\_Types::Float

t : BLESS\_Types::Time

**constants**

period = 100 ms

**behavior**

Train ::= ' DT 1 s = v ' & ' DT 1 v = a ' & ' DT 1 t = 1 ' [> period ]> SendSpeed

SendSpeed ::= speed!(v)

**\*\*}**;

**end** TimedInrptSys.impl;

* + - 1. Communication Interrupt

1. Communication interrupt preempts continuous evolution when the communication along any specified channel takes place.

Syntax

communication\_interrupt ::=

**[[>** *port\_or\_channel*\_identifier **~>** *activity\_*identifier { **,** *port\_or\_channel*\_identifier **~>** *activity*\_identifier }\* **]]>**

Semantics

1. An Activity with continuous evolution, boundary interrupt and communication interrupt continues its evolution except a communication event occurs on an identified channel or port, changing its behavior to the identified activity.
2. The communication event can be either input or output event.
3. As soon as the communication event takes place, the Activity behaves as the next Activity specified after **~>**.

Examples

-- Following code snippet illustrates specification of -- communication Interrupt.

**package** Communication\_Interrupt\_Exp **public**

**with** Base\_Types, BLESS\_Types**;**

**abstract** ComIntrptSys

**features**

brk : in event data port Base\_Types::Float;

**end** ComIntrptSys;

**abstract** **implementation** ComIntrptSys.impl

**annex** hybrid **{\*\***

**variables**

s, v, a, x: Base\_Types::Float

t : BLESS\_Types::Time

**channels**

ch1, ch2 : Base\_Types::Float

**behavior**

-- In the Following Activity the continuous evolution of the -- velocity of a running train is interrupted and sent across -- channel ch1 as soon as the receiver is ready.

Train ::= ' DT 1 s = v ' & ' DT 1 v = a ' & ' DT 1 t = 1 ' [[> ch1!v ~> NextAct ]]>

NextAct ::= ch1?x & skip

-- In the Following Activity, if the continuous evolution of the -- velocity of a running train is interrupted by the -- communication event across channel ch2, it behaves as -- Activity Continue. If the continuous evolution of the -- velocity is interrupted by the communication event across -- event data port brk the behavior is as Activity Brake.

STrain ::= ' DT 1 s = v ' & ' DT 1 v = a ' & ' DT 1 t = 1 ' [[> ch2!v ~> Continue , brk?x ~> Brake ]]>

Continue ::= skip

Brake ::= v := 0 & ch2!v

**\*\*}**;

**end** ComIntrptSys.impl;

**end** Communication\_Interrupt\_Exp;

* + 1. Repetition

1. The repetition executes an Activity for a finite number of times.

Syntax

repetition ::=

**REPEAT** [ **[** (integer\_literal | *integer\_variable\_*identifier) **]** ]

**(** *behavior\_*identifier **)**

Semantics

1. The statement REPEAT(P) causes Activity P to be repeated a finite, unspecified number of times.
2. The statement REPEAT [5](P) causes Activity P to be repeated five times.
3. The statement REPEAT [n](P) causes Activity P to be repeated the value of integer variable n times.

Legality Rule

1. A variable identifier used in a REPEAT statement must refer to an integer value.
   * 1. Boolean and Numeric Expressions
2. Boolean expressions are composed of boolean terms combined using standard and, or and xor boolean operators.
3. A boolean\_term may start with a not operator to specify the negation of the following boolean term.
4. A boolean\_term may consist of standard boolean values true or false and a relation.
5. A relation is defined using numeric expressions combined using standard relational operators i.e. =, <>, >, <, <=, >=.
6. A numeric\_expression is composed of numeric terms combined using standard arithmetic operators i.e. –, +, \*, /, mod.
7. A numeric\_term may start with –to specify the negative value of the following numeric\_term.
8. A term may consist of numeric\_literal or *variable\_*identifier.

Syntax

boolean\_expression ::=

boolean\_term

| boolean\_term { **and** boolean\_term }\*

| boolean\_term { **or** boolean\_term }\*

| boolean\_term { **xor** boolean\_term}\*

| relation

boolean\_term ::=

[ **not** ] ( **true** | **false**

| **(** boolean\_expression **)**

| relation )

relation ::=

**[** numeric\_expression relation\_symbol numeric\_expression **]**

numeric\_expression ::=

numeric\_term | numeric\_term **-** numeric\_term

| numeric\_term **/** numeric\_term

| numeric\_term **mod** numeric\_term

| numeric\_term { **+** numeric\_term }+

| numeric\_term{ **\*** numeric\_term }+

| numeric\_term **^** numeric\_literal

numeric\_term::=

[ **-** ]( numeric\_literal | *variable\_*identifier|

(numeric\_expression) )

numeric\_literal ::= integer\_literal | real\_literal

relation\_symbol ::= **=** | **<>** | **>** | **<** | **<=** | **>=**

Legality Rule

1. All the variables used in numeric\_expression and boolean\_expression must be declared in the **variables** section.