SAE Aerospace
An SAE International Group

AS5506B/X

AEROSPACE STANDARD

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

Hybrid Annex

Normative

A.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 constant declarations.
- Section A.7 defines the syntax and semantics of communication channels within Hybrid Annex subclauses.
- Section A.8 defines the syntax and semantics of behavior declarations.
- Section A.9 provides an extensive example of using Hybrid Annex
- Section A.10 provides informative references
-

A.2 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.

(4) Like AADL, HA is case insensitive.

Syntax

```
hybrid_annex ::=
[ **assert** { Assertion }+ ]
[ **invariant** Assertion ]
[ **variables** { variable_declaration }+ ]
[ **constants** constant_declarations ]
[ **channels** { channel_declaration }+ ]
 **behavior** { behavior_process_declaration }+
```

A.3 Assert

(5) 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 Assertion Annex Document (AS5506-- submitted for standardization).

Naming Rule

(6) `Assertion` labels (if any) are globally-visible and must be unique.

Examples

```
-- Following code snippet shows the use of Assertion annex as
-- BISL within HA based on the Isolette system explained in REMH
-- by FAA.

abstract implementation AssertionExpSys.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)>>

behavior

Null::= skip

**};

end AssertionExpSys.impl;
```

A.4 Invariant

(7) An invariant section has a single Assertion that must always be true about behavior of the HA subclause containing it.

Examples

```
-- Following example shows the use of Assertion annex as BISL
-- within HA based on the Isolette system explained in REMH by
-- FAA.

abstract implementation InvariantExpSys.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 AssertionExpSys.impl;
```

A.5 Variables

(8) Local variables in the scope of current annex subclause are declared in variables clause along with data type.

(9) Depending on AADL component for which the HA subclause is defined, a variable may either be discrete or continuous.

(10) A data 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

(11) The referenced external data component 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

(12) Values of local variables are persistent throughout the lifetime of their containing component.

Examples

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

package Variable_Example public

with Base_Types, BLESS;

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;
```

A.6 Constants

(13) Constants in the scope of current annex subclause are declared in a `constants` section.

Syntax

```
constant_declarations ::=

  behavior_constant { , behavior_constant }*
behavior_constant  ::=
  behavior_constant_identifier
  = ( integer_literal | real_literal ) [unit_identifier]
```

Legality Rules

(14) Constants can only be initialized at declaration and cannot be assigned any value afterwards.

(15) A constant must be initialized with either `integer_literal` or `real_literal` value.

Examples

```
-- 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;
```

A.7 Channels

(16) The `channels` section is used to declare communication channels used among behavior processes defined within the same HA subclause.

(17) Channel declarations contain AADL data component classifier references to specify type of the data sent or received along a particular channel.

Syntax

```
channel ::=

  channel_identifier {, channel_identifier }* :

  data_component_classifier_reference
```

Legality Rules

(18) HA channels only support unidirectional communication, as a result a process cannot use same channel for both input and output communication events.

Examples

```
-- Following example shows channels 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;
```

A.8 Behavior

(19) The `behavior` section in HA subclause is used to specify continuous behavior of the component in terms of concurrently-executing processes.

Syntax

```
behavior_declaration ::=

  behavior_identifier ::=

  process_declaration { & process_declaration }*


process_declaration ::=

  skip | assignment | wait time_value | communication

  | sequential_composition | concurrent_composition

  | choice | continuous_evolution | repetition
```

Semantics

(20) Ampersand sign (**&**) is used to separate process declarations within a behavior declaration.

A.8.1 Skip Process

(21) The `skip` process terminates immediately having no effect on variable values.

A.8.2 Assignment Process

(22) The `assignment` process assigns the value of expression to the local variable declared in `variables` clause.

Syntax

```
assignment ::=

  variable_identifier := numeric_expression
```

A.8.3 Wait Process

(23) The **`wait`** keeps idle for `time_value` time.

(24) During this idle period the respective process does not perform any action and the variables are unchanged.

Syntax

```
time_value ::=

  time_variable_identifier | real_literal time_unit
```

Semantics

(25) A *time_variable*_identifier can be a local variable identifier declared in `variables` clause.

(26) A *time_variable*_identifier can be a constant identifier declared in `constants` clause.

(27) The `time_unit` defines a unit of measurement of time.

(28) The `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`**.

Examples

```
-- Following example shows use of skip, assignment and wait
-- processes 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

proc1 ::= wait period & x := w & skip

proc2 ::= wait 5 sec & y := z & skip

proc3 ::= skip

**};

end MixExpSys.impl;
end Mix_Example;
```

A.8.4 Communication

(29) Communication within HA subclauses uses channels. Communication with other AADL components uses ports. Both port and channel communication occurs at discrete events.

(30) Communication events are of two types: input event and output event.

(31) A port input event `p1?(x)` specifies that an input value is received from port `p1` and stored in a local variable `x`.

(32) A port output event `p2!(y)` specifies that an output value of variable `y` is sent out port `p2`.

(33) A channel input event `ch1?z` specifies that an input value is received from channel `ch1` and stored in a local variable `z`.

(34) 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

(35) Port identifiers must be names of ports of the component containing the HA subclause in which they are used.

(36) Channel identifiers must be names of channels declared in the `channels` section of the HA subclause in which they are used.

Semantics

(37) Channel communication synchronizes involved processes and can only occur when both sender and receiver are ready.

(38) Channel communication may cause either sender or receiver to wait.

(39) Port communication has semantics of AADL core language.

(40) Port communication may send (`!`) only to `out data`, `out event` or `out event data` ports.

(41) Port communication may receive (`?`) only from `in data`, `in event` or `in event data` ports.

(42) Port communication to `out data` ports updates the value of the port.

(43) Port communication from `in data` ports receives the most-recent value of the port.

(44) Port communication to `out event data` ports issues an event (with data) on the port.

(45) Port communication from `in event data` ports receives the most-recent fresh value of the port, or blocks until fresh data becomes available.

(46) Port communication without parameter is `event port` transmission or receipt.

Examples

```
-- Following code snippet shows communication modeling based on
-- input and output event 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
proc1 ::= wait period & idp?(x) & ch1!x
proc2 ::= ch1?y & ch2!w & odp!(y)
proc3 ::= ch2?z
**};
end ComExpSys.impl;
end Communication_Example;
```

A.8.5 Sequential Composition

(47) Sequential composition defines consecutively-executing behaviors.

Syntax

```
SequentialComposition ::=
  { behavior_identifier { ; behavior_identifier }+ }
```

Semantics

(48) A sequentially composed process `{P ; Q}` behaves as `P` first and after its successful termination, behaves as `Q`.

Legality Rule

(49) Behavior identifiers used in sequential composition must refer to behavior declarations.

A.8.6 Concurrent Composition

(50) A parallel composed process `S1||S2` behaves as if `S1` and `S2` run independently except that all interactions occur through communications events.

(51) Communication events between concurrently-composed behaviors, must occur along the common communication channels declared in the `channels` section connecting processes `S1` and `S2`.

Syntax

```
ConcurrentComposition ::=

  { behavior_identifier { || behavior_identifier }+ }
```

Legality Rules

(52) Behaviors defined using concurrent composition may not themselves be used in either sequential or concurrent compositions.

(53) Communication channels must be shared pair-wise with complementary directions – in communication with out communication.

(54) Variables used in shared communication channels must have the same type.

Semantics

(55) More than two processes can take part in a particular parallel composition.

A.8.7 Choice

(56) Internal execution choice between processes `P` and `Q`, denoted as `P` **[]** `Q` is resolved by the process itself.

Syntax

```
choice ::=

  alternative { [] alternative }*


alternative ::=

  ( boolean_expression ) -> process_identifier
```

Semantics

(57) Choice executes a process with an `alternative` having true `boolean_expression`.

(58) When more than one `alternative` has true `boolean_expression` the `choice` is resolved non-deterministically.

(59) When no `alternative` has true `boolean_expression` the `choice` is equivalent to `skip`.

(60) The alternative process `(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
-- process 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
proc1 ::= idp?(x) & (x mod 2=0)-> Eproc [] (x mod 2<>0)-> Oproc
Eproc ::= ch1!x & odp!(x)
Oproc ::= {x:=x+1 ; ch1!x ; odp!(x)}
Proc2 ::=  idp2?(w) & (w >=5 and w mod 2=0 )-> skip
**};
end ChoiceExpSys.impl;
end Choice_Example;
```

A.8.8 Continuous Evolution

(61) 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

(62) The `continuous_evolution` statement forces values of variables declared in `variables` section to obey the differential equations as long as the `boolean_expression` is true.

(63) The `boolean_expression` specifies the boundary condition of the variables.

(64) The continuous evolution terminates as soon as `boolean_expression` turns to false.

(65) The interruption of continuous evolution due to boundary condition is known as boundary interrupt.

(66) Continuous evolution can also be preempted due to timed interrupt and communication interrupt as presented in A.8.9.

A.8.8.1 Differential Expression

(67) 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

(68) Multiple differentials are combined with multiplication (*) and additions (+) signs to form a differential expression.

(69) 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.

A.8.8.2 Differentials

(70) A differential term may contain `numeric_literal` or `behavior_variable_identifier` with power operator (^).

(71) 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.

Syntax

```
differential ::=
 [-] numeric_literal
  | variable_identifier [ ^ numeric_literal ]
  | derivative_expression
  | derivative_time
  | ( differentinal_expression )
```

A.8.8.3 Derivative Expression

(72) 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

(73) The `order_integer_literal` determines the order of the derivative expression.

(74) The dependant variable is written before the independent variable. For example rate of change of variable `y` with respect to `x` denoted as $\frac{dy}{dx}$ is specified as `DE 1 y x` and $\frac{d^2y}{dx^2}$ is specified as `DE 2 y x`. Here, `1` and `2` are the order of the differential equation.

A.8.8.4 Derivative Time

(75) Time derivation shows the rate of change of a variable with respect to the rate of change of time.

(76) Time derivation is specified using keyword **DT** followed by the order of time derivation and a dependent variable.

(77) Taking time as independent variable, time derivation only contains a dependant variable.

Syntax

```
derivative_time ::=

  DT order_integer_literal variable_identifier
```

Semantics

(78) The `order_integer_literal` determines the order of the derivative time.

(79) 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 $\frac{dy}{dt}$ 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 process 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 process 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 process Heat_Equation for 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 19mm^2/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;
```

A.8.9 Interrupts

(80) Continuous evolution can also be interrupted due to either timed and communication interrupts.

Syntax

```
interrupt ::= timed_interrupt | communication_interrupt
```

Semantics

(81) Continuous evolution termination after specific time and communication event is realized through timed and communication interrupts respectively.

(82) If the continuous evolution does not terminate due the boundary condition, timed and communication interrupts can be used for termination.

A.8.9.1 Timed Interrupt

(83) Timed interrupt preempts continuous evolution after a specific time value and control follows the next specified process.

Syntax

```
timed_interrupt ::=
```

[> time_value **]>** { *behavior*_identifier }$^+$

Semantics

(84) A process 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 process behaves like the next specified process.

Examples

```
-- 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;
```

A.8.9.2 Communication Interrupt

(85) Communication interrupt preempts continuous evolution as soon as the communication along any specified channel takes places.

Syntax

```
communication_interrupt ::=

[[> port_or_channel_identifier { , port_or_channel_identifier }*
]]> { process_identifier }+
```

Semantics

(86) A process with continuous evolution, boundary interrupt and communication interrupt continues its evolution except that the continuous evolution is interrupted whenever any communication event takes place along any channel.

(87) The communication event can be either input or output event.

(88) As soon as the communication event takes place, the process behaves as the next process specified after `]]>`.

Examples

```
-- Following code snippet illustrates specification of
-- communication Interrupt. In this example the continuous
-- evolution of the velocity of a running train is interrupted
-- and sent to channel ch1 as soon as the receiver is ready.

abstract implementation TimedInrptSys.impl

annex hybrid {**

variables

s, v, a, nv: Base_Types::Float

t : BLESS_Types::Time

constants
period = 100 ms

channels

ch1 : Base_Types::Float

behavior

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

GetSpeed :: ch1?nv

NextProcess ::= skip

**};

end TimedInrptSys.impl;
```

A.8.10 Repetition

(89) The repetition executes a process for a finite number of times.

Syntax

```
repetition ::=

  repeat [ [ (integer_literal | integer_variable_identifier) ] ]

  ( process_identifier )
```

Semantics

(90) The statement `repeat(P)` causes process `P` to be repeated a finite, unspecified number of times.

(91) The statement `repeat [5](P)` causes process `P` to be repeated five times.

(92) The statement `repeat [n](P)` causes process `P` to be repeated the value of integer variable `n` times.

Legality Rule

(93) A variable identifier used in a `repeat` statement must refer to an integer value.

A.8.11 Boolean and Numeric Expressions

(94) Boolean expressions are composed of Boolean terms combined using standard `and`, `or` and `xor` Boolean operators.

(95) A `boolean_term` may start with a `not` operator to specify the negation of the following Boolean term.

(96) A `boolean_term` may consist of standard Boolean values `true` or `false` and a `relation`.

(97) A `relation` is defined using numeric expressions combined using standard relational operators i.e. **=**, **<>**, **>**, **<**, **<=**, **>=**.

(98) A `numeric_expression` is composed of numeric terms combined using standard arithmetic operators i.e. **–**, **+**, *****, **/**, **mod**.

(99) A `numeric_term` may start with **–** to specify the negative value of the following `numeric_term`.

(100) 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 }+


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

(101) All the variables used in `numeric_expression` and `boolean_expression` must be declared in the `variable` clause.

A.9 Extensive Example

(1) Following example depicts the use of HA to model the continuous behavior of the Air component of the Isolette system explained in the Requirement Engineering Management Handbook (REMH) by Federal Aviation Administration (FAA). The behavior specification is based on the following two environmental assumptions;

EA-IS1: When the Heat Source is turned on and the Isolette is properly shut, the Current Temperature will increase at a rate of no more than 1^{o}F per minute**.**

EA-IS2: When the Heat Source is turned off and the Isolette is properly shut, the Current Temperature will decrease at a rate of no more than 1^{o}F per minute**.**

```
abstract Air

features

hss : in data port Iso_Variables::on_off;

hin : in data port Iso_Variables::Heat;

tout : out data port Iso_Variables::Heat;

end Air;


abstract implementation Air.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()>>

variables
h : Iso_Variables::on_off -- heat control command value
q : Iso_Variables::Heat -- heat source energy value
c : Iso_Variables::Heat -- current Air heat energy value
l : Iso_Variables::LdtTemp -- lower desired temperature value
u : Iso_Variables::UdtTemp -- upper desired temperature value

constants

r = 73.0 f -- constant room temperature
k = 0.026 wpmk -- average thermal conductivity of air



behavior

Heating ::= 'DT 1 c = -k*(c - q)' & 'DT 1 q = 1'
            [[> tout!(c) ]]> Continue
Cooling ::= 'DT 1 c = -k*(c - q)' & 'DT 1 q = -1'
            [[> tout!(c) ]]> Continue
AirTemp ::= hss?(h) & (h=on) -> Heating [] (h=off) -> Cooling

Continue ::= skip

WorkingIsolette ::= repeat(AirTemp)

**};

end Air.impl;
```

A.10 Informative References

(1) Hybrid Annex subclauses use differential equations and process algebra notations to model continuous behavior.

(2) Process algebras are used to model the behavior of reactive systems as communicating *Processes*.

(3) Calculus of Communicating Systems (CCS), Algebra of Communicating Processes (ACP), and Communicating Sequential Processes (CSP) are the important process algebras.

(4) Hybrid Annex subclause grammar and semantics are inspired by an extension of CSP known as Hybrid Communicating Sequential Processes (HCSP), but intended to support other continuous behavior modeling tools and methodologies impartially.