AADL Model Behavior: Rapid-Prototype XOR Exactitude?

Brian R Larson

7 June 2011

SAE International AS-2C Standard Committee Meeting

TELECOM ParisTech 6-9 June 2011

brl@multitude.net
Why Model AADL Component Behavior?

What is the purpose of modeling behavior of AADL components?

- prototyping - non-functional, early representation of behavior: 
  \( \text{computation}(100 \ \text{us}) \)

- exact modeling - makes 2 claims: model behavior accurately reflects the “real” system; formally-verified properties of the models therefore hold for the “real” system too

- both - ... 

Are prototyping and exact modeling domain-specific languages mutually exclusive?
**Introduction**

Why model behavior?

**Purposes of AADL Component Behavior Modeling**

*Prototyping* is used early in the system design process to represent coarse, high-level behavior to facilitate analysis tools, but is later supplanted by source code in a traditional programming language as the actual design article.

*Exact modeling* defines mathematically-precise behavior for components to facilitate verification tools, remains the actual design article throughout the design process. Trustworthy code generation from formally-verified models obviates need traditional source code.

Consequently, exact modeling needs many features of compiled programming languages.
(1) The Behavior Annex (BA) document provides a standard sublanguage extension to allow behavior specifications to be attached to AADL components.

The aim of the Behavior Annex is to refine the implicit behavior specifications that are specified by the core of the language.
The Behavior Annex targets the following goals:

- Describe the internal behavior of component implementations as a state transition system with guards and actions. *However, the aim is not to replace software programming languages or to express complex subprogram computations.*

- Extend the default run-time execution semantics that is specified by the core of the standard, such as thread dispatch protocols.

- Provide more precise subprogram calls synchronization protocols for client-server architectures.
Without BA, and this SAE International AS-2C standard committee, BLESS would not exist.

Drafts of proposed annex documents are sent to committee members for comment and questions.

While reviewing an early draft of the BA annex document, I noticed that its action grammar was similar to that used by DANCE, a concurrent programming language that may be augmented with assertions forming a proof outline. A proof tool would transform the proof outline into a complete, formal proof, or list unsolved proof obligations.
So I tried to formalize BA

Behavioral Language for Embedded Systems with Software (BLESS) was created from BA to prove embedded software/system conformance to specification.

Initially, BLESS was to be exactly like BA, but add assertions that form a proof outline which otherwise would be ignored like comments.

Formalizing the semantics of BLESS (exact modeling) required divergence from BA (prototyping).

This presentation explains those divergences and their reasons.
BLESS Syntax Matches BA When Possible

- Mostly addition of optional bnf clauses.
- A few clauses are mandatory for BLESS, but optional for BA.
- Biggest difference: BLESS subprogram behaviors have annex subclause restricted to action execution.
- BA and BLESS thread behaviors look deliberately similar.
Behavior Actions

The *behavior actions* may be a single asserted action, an ordered sequence of actions, or an unordered set of actions.

\[
\text{behavior actions ::= }
\]

\[
\text{asserted_action } \\
| \text{behavior action sequence } \\
| \text{behavior action set }
\]

\[
\text{behavior action sequence ::= } \\
\text{asserted action } \{ ; \text{ asserted action } \} +
\]

\[
\text{behavior action set ::= } \\
\text{asserted action } \{ \& \text{ asserted action } \} +
\]

(same grammar as BA except \text{asserted action replacing} \text{behavior action})
(Just) Add Optional Assertions to Actions

By inserting assertions into BA grammar, ignored as whitespace like comments otherwise, formal proofs could be created from proof outlines of BA too.

asserted_action ::= 
  [ precondition_Assertion ]
behavior_action 
  [ postcondition_Assertion ]
BLESS Assertions Include Time

BLESS Assertions are first-order predicates augmented by simple temporal operators surrounded by << and >>.

\( p@t \) means predicate \( p \) is true at time \( t \).

\( p^k \) means predicate \( p \) is true \( k \) thread periods of duration \( c \) from now.

\( p^k \equiv p@ (\text{now} + (k*c)) \)

\( p' \) means predicate \( p \) will be true one thread period from now.

\( p' \equiv p^1 \equiv p@(\text{now}+c) \)

*(only periodic and hybrid AADL components have periods)*
Assertion ::= << assertion_body >>

assertion_body ::= 
  label_identifier : { parameter_identifier }* 
  ( : predicate | ::= predicate_expression ) 
| predicate

full Assertion grammar in More: Assertion.
BLESS/BA Grammar Differences

Remove confusing two-character tokens

! < ! > can use Get_Resource and Release_Resource instead*

p >> can use freeze(p) instead**

BLESS uses >> to terminate Assertions.

*D.5(12) p. 29  **D.6(10) p. 31
Names and Types

BA has no types. Uses unique component classifier references to data component types directly.

BA has grammar for complex names, but no semantics for what they mean.

BLESS must type-check every name and operator.
BLESS Types

- simple grammar extension to permit type expressions and type operators
- defines semantics for compound names
- small, simple set of type rules
- supplies missing semantics for names
Type Grammar

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

full type grammar in More: Type.
BLESS restricts subprogram behavior to execution of an action.

```plaintext
subprogram_behavior ::= 
  [ availability ] [ assert { Assertion }+ ]
  [ pre predicate ] [ post predicate ]
existential_lattice_quantification

existential_lattice_quantification ::= 
  [ quantified_variables ] { asserted_action }
  [ catch_clause ]
```

(more about this, next section)
Thread Behavior

Thread behavior is defined using the BLESS annex sublanguage of AADL using a state transition system.

A *state transition system* has a set of states, a set of local variables some of which may have initial values, and a set of transitions.

The transitions of a state transition system specify behavior as a change of the current state from a source state to a destination state.

Threads may have an *availability clause* used with product-line engineering to determine whether this thread behavior is applicable for any particular product.

Threads may have an *assert clause* listing labelled Assertions to be invoked by other Assertions.

Threads may have an *invariant clause* that must be true of every state.
Thread Behavior Grammar

thread_behavior ::= [ availability ] [ assert { Assertion }+ ] [ invariant Assertion ] [ variables ] states { behavior_state }+ [ transitions ]
A behavior state may be one of initial, complete, and final, otherwise execution. A behavior state may have an Assertion that holds when that state is current.

\[
\text{behavior\_state} ::= \text{behavior\_state\_identifier} \\
\quad \{ \_ \text{behavior\_state\_identifier} \}^* \\
\quad : [ \text{initial} \mid \text{complete} \mid \text{final} ] \text{ state} \\
\quad [ \text{Assertion} ] ;
\]
Behavior Transitions Must Have { and }, Even If Empty

Mandatory brackets for transitions much improves intelligibility of ANTLR-generated parser error messages.

```
behavior_transition ::= [ behavior_transition_label : ]
source_state_identifier
{ , source_state_identifier }*
-[ [ behavior_condition ] ]-->
destination_state_identifier
( { behavior_actions } | {} )
[ Assertion ] ;
```
Allow Timeout on Subset of Event Ports

Needed to define timeout dispatch triggers that reset/started on events at a subset of ports, not just since last suspension.
A *dispatch trigger* is an event which causes the dispatch condition to be evaluated.

The value of a dispatch condition is a boolean expression of dispatch triggers.

Event arrival at either event ports or event data ports causes a dispatch trigger referenced by the port's identifier.

dispatch_trigger ::= 
  timeout [ ( event_port_identifier+ ) ]
  behavior_time ] 
| in_event_port_identifier
Availability

Availability is used with *product-line engineering* (PLE) models to control which features any particular product possesses.

Used effectively, PLE permeates everything: architecture, hardware, firmware, test, manufacturing, even technical publication.

Given a set of features for a particular product the availability expression is evaluated by replacing each identifier with *true* if that feature is in the set, and *false* otherwise, then applying the boolean operators.

Many BLESS constructs have optional availability clauses. When the availability is false, the construct “disappears” becoming effectively a comment, skip, or whitespace.
Availability Grammar

availability ::= availability availability_subexpression

availability_expression ::= availability_subexpression
[ { and availability_subexpression }+ | { or availability_subexpression }+ | { xor availability_subexpression }+ | ]

availability_subexpression ::= [ not ]
( product_line_engineering_identifier
| ( availability_expression )
)
An *alternative formula* makes the proof semantics symmetric. A boolean expression *guards* each alternative; guards may be evaluated in any order. At least one of the guards must be true. If more than one guard is true, any of their alternatives may be performed. An alternative formula may have an availability clause.

```
alternative ::= if [ availability ] guarded_action { [] guarded_action }+ fi

guarded_action ::= ( boolean_expression_or_relation )-> behavior_actions
```
Extend behavior_action

Grammar for behavior_action is extended to allow if-fi.

\[
\text{behavior}\_\text{action} ::= \\
\quad \ldots \\
\quad | \ \text{alternative} \quad \text{Retain BA construct if-elseif-else in grammar, but convert to if-fi for analysis.}
\]
BLESS expressions have no operator precedence

BLESS expressions need parentheses where BA doesn’t because all operators in BLESS have the same precedence, so require parentheses to disambiguate usage.
Expression Grammar

expression ::= subexpression
[ { \texttt{+ numeric\_subexpression} }+ \\
| { \texttt{* numeric\_subexpression} }+ \\
| \texttt{- numeric\_subexpression} \\
| \texttt{/ numeric\_subexpression} \\
| \texttt{mod natural\_subexpression} \\
| \texttt{rem integer\_subexpression} \\
| \texttt{** numeric\_subexpression} \\
| { \texttt{and boolean\_subexpression} }+ \\
| { \texttt{or boolean\_subexpression} }+ \\
| { \texttt{xor boolean\_subexpression} }+ \\
| \texttt{cand boolean\_subexpression} \\
| \texttt{cor boolean\_subexpression} ]
A *subexpression* allows negation and grouping with parentheses.

```plaintext
subexpression ::= [ _ | not ]
( value | ( expression_or_relation ) )

expression_or_relation ::= expression [ relation_symbol expression ]
```
BA Subprograms Have Uncontrollable Power

BLESS restricts what subprograms can do in BA, so that their effect may be rigorously composed.

BLESS expands what subprograms can do in BA, a little.
Pure Function Subprograms

AADL does not define function subprograms.

Subprograms whose last parameter is “out”, and all previous parameters “in”, are deemed to be functions and may be invoked within expressions.

Such functions must be side-effect free; the only effect of function execution is determining the return value.
BLESS Subprograms Have Only Actions

BA allows definition of a state transition system for subprograms.
BLESS allows only an action for subprograms.
BLESS Subprograms have no Ports

BLESS subprograms have only parameter features.
BLESS subprograms do just computation.

For timing use a device, system, or thread component instead.
BLESS Subprograms must have no Side-Effects

The assume-guaranty composition of BLESS subprograms rely on the entire effect of computation be reflected in the “out” parameters the subprogram returns when it terminates.

When the subprograms precondition applied to the actual invocation parameters is true, then the postcondition applied to the actual returned parameters will be true.

Side-effects nullify the promise that the subprogram made its postcondition true, and nothing else!
Multiple-kind BA States

In BA, states may be any or all of initial, complete, and final. This causes all sorts of semantic overlapping.
In BLESS, state may have only one kind, initial, complete, final or execute.

Enforced by parser.
A *type* is a set of values.

The universe of all values, $V$, contains all simple values like integers and strings, and all compound values like arrays, records, and variants.

A type is a set of elements of $V$.

Moreover when ordered by set inclusion, $V$ forms a lattice of types.

The top of this lattice is the set of all values or $V$ itself.

The bottom of the lattice is the empty set.
Subtypes

Since types are sets, subtypes are subsets.

Moreover the semantic assertion “$T_1$ is a subtype of $T_2$” corresponds to the mathematical condition $T_1 \subseteq T_2$ in V.

Subtyping in the basis for type checking.
Atomic Types

Atomic types are AADL data component classifiers.

Instead, an AADL package is provided, BLESS_Types that extend those in Base_Types package defined in the Data Model Annex document.

BLESS_Types have a BLESS_Properties::Supported_Operators list of operator symbols for types that support arithmetic.

A BLESS_Properties::Supported_Relations list of relation symbols defines what relations can be applied to the type.
Type Grammar

type ::= type_name | enumeration_type
| type_operator_invocation | number_type
| array_type | record_type | variant_type
| boolean
A *type name* must refer to a data component or a data component implementation.

Data components in other packages may be referenced by a sequence of package identifiers separated by double colons.

Implementation names are formed by suffixing an identifier to the name of the data component implemented separated by a period.

```plaintext
type_name ::= { package_identifier :: }+
data_component_identifier
[ . implementation_identifier ]
```
An *enumeration* type is a sequence of identifiers.

Enumeration types are expressed as the reserved word `enumeration` followed by a sequence of identifiers enclosed in parentheses.

```plaintext
enumeration_type ::= enumeration ( { defining_enumeration_literal_identifier }+ )
```

The Data Model equivalent to `enumeration (a b c)` is

```plaintext
data EnumType
    Data_Model::Data_Representation => Enum;
    Data_Model::Enumerators => ("a", "b", "c");
end EnumType;
```
In general, where $s$ is a sequence of identifiers separated by spaces, $s'$ is that same sequence of identifiers enclosed in double quotes separated by commas, $N$ is an data component identifier, and $P$ is a package prefix so that $P::N$ is a legal type name, $\text{enumeration}(s) \equiv P::N$ such that in package $P$ there is,

```plaintext
data N
    Data_Model::Data_Representation => Enum;
    Data_Model::Enumerators => (s');
end N;
```
Each enumeration type has five pre-defined functions: first, last, next, previous, and position. Where $s$ is a sequence of identifiers, $i_1 \ i_2 \ \ldots \ i_n$,

\[
\begin{align*}
\text{first}(\text{enumeration}(s)) & \equiv i_1 \\
\text{last}(\text{enumeration}(s)) & \equiv i_n \\
\text{next}(i_j) & \equiv i_{j+1} \text{ for } j < n \\
\text{previous}(i_j) & \equiv i_{j-1} \text{ for } j > 1 \\
\text{position}(i_j) & \equiv j
\end{align*}
\]
Units Type

A *units type* represents an explicitly-listed set of measurement unit identifiers as the set of legal values.

The second and succeeding unit identifiers are declared with a multiplier representing the conversion factor that is applied to a preceding unit to determine the value in terms of the specified measurement unit.

```
units_type ::= units units_list
units_list ::= ( defining_unit_identifier { another_unit }* )
another_unit ::= defining_unit_identifier =>
unit_identifier * numeric_literal
```

---

1 AS5506A §11.1.1 Property Types
The core language equivalent to

\[
\text{units} \ (u_1, \ u_2 \Rightarrow u_1 \times i_1, \ u_3 \Rightarrow u_2 \times i_2, \ . \ . \ .) \ \text{is}
\]

```plaintext
property set myPropertySet is
    My_Units : type units (u_1, \ u_2 \Rightarrow u_1 \times i_1, \ u_3 \Rightarrow u_2 \times i_2, \ . \ . \ .);
end myPropertySet;
```

A *number type* is the name of a data component that behaves like an indivisible number, possibly restricted to a subrange, and may have units.

```
number_type ::= number_component_classifier_reference [ range ] [ units units_designator ]
units_designator ::= units_unique_property_type_identifier | units_list
```
Array Type

An *array type* is a collection indexed by natural numbers.

The natural numbers in an array type expression denote the size of the array in successive dimensions.

The sizes may be expressed as natural literals, or identifiers of natural number values.

```plaintext
array_type ::= array [ array_range_list ] of type
array_range_list ::= natural_range
                    { , natural_range }*

natural_range ::= natural.literal
               | ( natural.literal | natural.identifier )
               .. ( natural.literal | natural.identifier )
```
The Data Model for arrays uses the property `Data_Model::Slice` to define ranges for each array dimension rather than the property `Data_Model::Dimension` which only defines the array size.

The Data Model equivalent to

```
array [5, 0..15, May..October] of MyPackage::MyElementType is
```

is

```
data My_Three_Dimensional_Array
  properties
  Data_Model::Data_Representation => Array;
  Data_Model::Base_Type => (classifier
    (MyPackage::MyElementType));
  Data_Model::Slice => (0..4, --5 becomes 0..4
    0..15, --same range of natural literals
    May..October); --May and October must identify natural values
end My_Three_Dimensional_Array
```
Array Type

In general, where \( n \) is a sequence of positive integer literals, integer ranges (i.e. 1..10), \( n' \) is that same sequence separated by commas having single integer literals replaced by integer ranges starting at zero, and \( E \) and \( T \) are data component identifiers, and \( P \) and \( R \) are package prefixes so that \( P::T \) and \( R::E \) are legal type names,

\[
\text{array } [n] \text{ of } R::E \equiv P::T \text{ such that in package } P \text{ there is,}
\]

```plaintext
data T
    Data_Model::Data_Representation => Array;
    Data_Model::Base_Type => (classifier (R::E));
    Data_Model::Slice => (n);
end T;
```
A record type is a collection indexed by identifier labels.

record_type ::= record ( { record_field }+ )
record_field ::= defining_field_identifier : type ;
The Data Model equivalent to `record ( l1:T1; l2:T2; )` is:

```plaintext
data My_Record
  properties
    Data_Model::Data_Representation => Struct;
    Data_Model::Base_Type =>
      (classifier (T1), classifier (T2));
    Data_Model::Element_Names => ("l1", "l2");
end My_Record
```
Record Type

In general, where $S$ is a sequence of pairs of labels and type names, where each label is separated from its type name by a colon and followed by a semicolon,\(^2\) $B$ is a sequence of the second elements of those pairs (type names) of $S$ enclosed in parentheses prefaced by \textit{classifier} separated by commas,\(^3\) and $L$ is a sequence of the first elements of those pairs (labels) of $S$ enclosed in double-quotes and separated by commas,\(^4\) and $P$ is package prefix so that $P :: T$ is a legal type name, \texttt{record} $(S) \equiv P :: T$ such that in package $P$ there is,

```data
T
  Data_Model::Data_Representation => Struct;
  Data_Model::Base_Type => (B);
  Data_Model::Element_Names => (L);
end T;
```

\(^2\) i.e. $l_1:T_1; l_2:T_2; l_3:T_3$

\(^3\) i.e. classifier $(T_1)$, classifier $(T_2)$, classifier$(T_3)$

\(^4\) i.e. "$l_1$", "$l_2$", "$l_3$"
The Data Model Annex shows an alternate way to represent records (structs) using subcomponents of data component implementations to represent record elements.

These are not supported by BLESS.

Use the Data Model properties instead.
A *variant type* holds a value of varying type specified by the value of a discriminant.

A discriminant holds the value of one of the labels of the record fields, which then determines the type of the variant.

\[
\text{variant\_type ::=} \\
\text{variant [ discriminant\_identifier ]} \\
\text{( \{ record\_field \}+ )}
\]
The Data Model equivalent to `variant [d] (c1:T1; c2:T2; )` is

```plaintext
data My_Variant
    properties
    Data_Model::Data_Representation => Union;
    Data_Model::Base_Type => (classifier (T1), classifier (T2));
    Data_Model::Element_Names => ("c1", "c2");
    Data_Model::Discriminant => "d";
end My_Variant
```
In general, where $S$ is a sequence of pairs of labels and type names, where each label is separated from its type name by a colon and followed by a semicolon, $B$ is a sequence of the second elements of those pairs (type names) of $S$ enclosed in parentheses prefaced by `classifier` separated by commas, and $L$ is a sequence of the first elements of those pairs (labels) of $S$ enclosed in double-quotes and separated by commas, $d$ is a discriminant identifier, and $P$ is package prefix so that $P::T$ is a legal type name, $\text{variant}[d](S) \equiv P::T$ such that in package $P$ there is:

```
data T
   Data_Model::Data_Representation => Union;
   Data_Model::Base_Type => (B);
   Data_Model::Element_Names => (L);
   Data_Model::Discriminant => "d";
end T;
```
A *type declaration* associates an identifier with a type expression.

Other type expressions may use the identifier in place of its type expression.

```plaintext
type_declaration ::=  
  type_identifier : type type ;
```

Brian R Larson  ()  Rapid-Prototype XOR Exactitude?  7 June 2011  68 / 114
A *type operator* creates new types from existing types. Formal parameters in type operator’s type expression are replaced by actual type expressions when the type operator is invoked.

```plaintext
type_operator ::= type_identifier [ type_parameter_identifier+ ] : type type ;
type_operator_invocation ::= type_operator_identifier [ type { , type }* ]
```
Type Operator

Where $T$ is a type operator label, $t_1 t_2 \ldots t_n$ are formal parameters, and $E$ is a type expression that uses $t_1 t_2 \ldots t_n$, and its definition

$$T[t_1 t_2 \ldots t_n]: \text{type } E;$$

(there is type operator $T$ with formal parameters $t_1 t_2 \ldots t_n$ with type expression $E$)

is visible in the scope of its use, the meaning of type operator invocation is

$$\mathcal{M} \left[ T(a_1 a_2 \ldots a_n) \right] \equiv \mathcal{M} \left[ E \mid t_1 \mid t_2 \mid \ldots \mid t_n \right]$$

(the meaning of a type operator invocation is the meaning of the type expression of the type operator definition (with the same label) having actual parameters substituted for formal parameters)
Type Inclusion Rules

A type is included in another type $t \subseteq s$ when every value of one type is also a value of the other.

$$t \subseteq s \equiv \forall v \in t | v \in s$$
Type Inclusion Rules

In the following type rules,

- type expressions are denoted by $s$, $t$, and $u$,
- $s \rightarrow t$ is a function with domain $s$ and range $t$.
- type names by $a$ and $b$, and element labels by $L$;
- $V$ is the set of all values;
- $d$ is a discriminant label;
- $C$ is a set of inclusion constraints for types;
- $C.a \subseteq b$ is the set $C$ extended with the constraint that type $a$ is included in $b$;
- $C \models t \subseteq s$ is an assertion that from $C$ we can infer $t \subseteq s$. 
Type Inclusion Rules

**TOP ([TOP])**
\[ C \vdash t \subseteq V \text{ (every type is included in the set of all values)} \]

**VAR ([VAR])**
\[ C.a \subseteq t \models a \subseteq t \text{ (what it means to extend a type constraint)} \]

**BAS ([BAS])**
\[ C \vdash a \subseteq a \text{ (every type includes itself)} \]
**Type Inclusion Rules**

**TRANS ([TRANS])**

\[
C \vdash s \subseteq t \land C \vdash t \subseteq u \quad \Rightarrow \\
C \vdash s \subseteq u
\]

*(type inclusion is transitive)*

**FUN ([FUN])**

\[
C \vdash s \subseteq s_1 \land C \vdash t \subseteq t_1 \\
C \vdash (s \rightarrow t) \subseteq (s_1 \rightarrow t_1)
\]

*(a function type includes another when its domain includes the other’s domain and its range includes the other’s range)*
Array Type

**CAR ([CAR])**

\[
C \vdash s \subseteq t \land n \leq m \\
C \vdash \text{array}[n] \ of \ s \subseteq \text{array}[m] \ of \ t
\]

(an array type includes another when its element type includes the other’s element type, and the other has at most as many elements)

**CARM ([CARM])**

\[
C \vdash s \subseteq t \\
C \vdash \text{array}[n_1, n_2, \ldots, n_k] \ of \ s \subseteq \text{array}[n_1, n_2, \ldots, n_k] \ of \ t
\]

(a multi-dimensional array includes another when its element type includes the other’s element type, and has exactly the same dimensions)
Slice Type

**SLICE ([SLICE])**

\[
\begin{align*}
C &\models s \subseteq t \land d \leq a \land b \leq e \\
C &\models \text{array}[a..b] \text{ of } s \subseteq \text{array}[d..e] \text{ of } t
\end{align*}
\]

(an array slice includes another when its element type includes the other’s element type, and its range includes the other’s range)

**SLICEM ([SLICEM])**

\[
\begin{align*}
C &\models s \subseteq t \land \forall i \in \{1,\ldots,k\} \left| d_i \leq a_i \land b_i \leq e_i \right. \\
C &\models \text{array}[a_1..b_1,\ldots,a_k..b_k] \text{ of } s \subseteq \text{array}[d_1..e_1,\ldots,d_k..e_k] \text{ of } t
\end{align*}
\]

(a multi-dimensional slice includes another when its element type includes the other’s element type, and for each dimension its range includes the other’s range)
RECD ([RECD])

\[ C \models s_1 \subseteq t_1 \land \cdots \land C \models s_n \subseteq t_n \]

\[ C \models \text{record} (L_1 : s_1 ; \ldots \; L_n : s_n ; \ldots L_m : s_m ;) \subseteq \text{record} (L_1 : t_1 ; \ldots \; L_n : t_n ;) \]

(a record type includes another when the other has elements the same labels, and perhaps additional others, and for each label the corresponding element type includes the other’s element type for that label)
Variant Type

VART ([VART])

\[ C \models s_1 \subseteq t_1 \land \cdots \land C \models s_n \subseteq t_n \]

\[ C \models \text{variant}[d](L_1 : s_1; \ldots L_n : s_n;) \subseteq \text{variant}[d](L_1 : t_1; \ldots L_n : t_n;) \]

(a variant type includes another when the other has elements the same labels, and for each label the corresponding element type includes the other’s element type for that label)
Type Rules for Expressions

Type rules for expressions determine types of expressions, especially complex names.

Relation symbols, $= \neq$, are treated as functions of pairs of the same element type to $\text{boolean}$, $(s, s) \rightarrow \text{boolean}$, and are defined for every type $s$.

Relation symbols, $< <= >$, are treated as functions of pairs of the same element type to $\text{boolean}$, $(s, s) \rightarrow \text{boolean}$, and are pre-defined for types $\text{natural integer rational real complex}$. 
Type Rules for Expressions

Numeric operator symbols, $+, \times$, are treated as functions of sequences of the same element type to that element type, $(s, \ldots, s) \rightarrow s$, and are pre-defined for types natural integer rational real complex.

Numeric operator symbols, $-, / \mod \rem \times\times$, are treated as functions of pairs of the same element type to that element type, $(s, s) \rightarrow s$, and are pre-defined for types natural integer rational real complex.

Unary $-$ is arithmetic negation, $s \rightarrow s$, and is pre-defined for types integer rational real complex.
Logical operator symbols, `and`, `or`, and `xor`, are treated as functions of sequences of `boolean` to `boolean`, `(boolean, ..., boolean) → boolean`.

Logical operator symbols, `cand` and `cor`, are treated as functions of pairs of `boolean` to `boolean`, `(boolean, boolean) → boolean`.

Unary `not` is complement, `boolean → boolean`. 
In the following type rules,

\( A \) is a set of type assumptions for variables;

\( C \) is a set of inclusion constraints for types;

\( V \) is the set of all values;

\( e \) is an expression;

\( s, t \) are types;

\( s \rightarrow t \) is a function with domain \( s \) and range \( t \);
Type Rules for Expressions

- \( x \) is a variable;
- \( L \) is a field label;
- \( d \) is a discriminant label;

\[ A.x : t \] is the set \( A \) extended with the assumption that variable \( x \) has type \( t \);

\( C, A \vDash e : t \) means that from the set of constraints \( C \) and the set of type assumptions \( A \), we can infer that expression \( e \) has type \( t \);

\( f : s \rightarrow t \) means \( f \) is a function with domain type \( s \) and range type \( t \):

\[ \text{subprogram } f \text{ features } x:\text{in parameter } s; \]
\[ y:\text{out parameter } t; \ldots \text{end } f; \]
Type Rules for Expressions

ETOP ([ETOP])
\[ C, A \models e : V \] (the type of every expression is included in the set of all values)

EVAR ([EVAR])
\[ C, A . e : t \models e : t \] (define extending a type assumption)

ETRANS ([ETRANS])
\[
C, A \models e : t \land C \models t \subseteq u \\
\hline
C, A \models e : u
\]
(type inclusion is transitive for expressions too)
Type Rules for Function Application

**APPL ([APPL])**

\[ C, A \vdash f : s \to t \land C, A \vdash x : s \]

\[ \frac{}{C, A \vdash f(x) : t} \]  

(a function of type \( s \to t \), applied to a parameter with type \( s \), has type \( t \))
Type Rules for Array Names

ECAR (ECAR)

\[
C \models x : \text{array}[n] \text{ of } s \land 0 \leq m < n \\
C \models x[m] : s
\]
(indexing a variable of array type has the array’s element type)

ECARM (ECARM)

\[
C \models x : \text{array}[n_1, n_2, \ldots, n_k] \text{ of } s \\
0 \leq m_1 < n_1 \land \ldots \land 0 \leq m_k < n_k \\
C \models x[m_1, m_2, \ldots, m_k] : s
\]
(indexing a variable of multi-dimensional array type has the array’s element type)
Type Rules for Record and Variant Names

SEL ([SEL])

\[ C, A \models x : \text{record}(L_1 : t_1; \ldots L_n : t_n;) \]
\[ \quad \Rightarrow \quad C, A \models x.L_i : t_i \quad i \in 1..n \]  
(selecting a label of a variable having record type, has the type of the labeled element)

VSEL ([VSEL])

\[ C, A \models x : \text{variant}[d](L_1 : t_1; \ldots L_n : t_n;) \]
\[ \quad \Rightarrow \quad C, A \models x.L_i : t_i \iff x.d = L_i \quad i \in 1..n \]  
(selecting a label of a variable having variant type, has the type of the labeled element, only when the label is same as the discriminant)
In BLESS, an *Assertion*\(^5\) is a temporal logic formula enclosed within `<< >>`.

\(^5\)Capital ‘A’ when used as a proper name for the assertions in BLESS, small ‘a’ when talking about assertions generally.
A BLESS assertion annex subclause contains a predicate, or multiple Assertions.

A BLESS assertion annex library contains at least one Assertion.

Annex subclauses can only be declared in component types, component implementations, and feature group types; annex libraries must be declared in packages.

Any entity may have its \texttt{BLESS::Assertion} property associated with the label of an Assertion in a visible BLESS assertion annex library.
Assertion Grammar

assertion_annex_subclause ::=  
    { Assertion }+ | predicate

assertion_annex_library ::= { Assertion }+  

Assertion ::= << assertion_body >>

assertion_body ::=  
    label_identifier : { parameter_identifier }*  
    ( : predicate | ::= predicate_expression )  
    | predicate
Assertion Labels and Parameters

An Assertion may have a label by which other Assertions can refer to it.

An Assertion may have formal parameters. If so an Assertion’s meaning is textual substitution of actual parameter for formal parameters throughout the body of the Assertion’s predicate or predicate expression.

If an Assertion has no parameters, occurrences of its invocation may be replaced by the text of its predicate.

If an Assertion has parameters, its label and actual parameters, may be replaced by its predicate with formal parameters replaced by actual parameters.
**Predicate**

A *predicate* is a boolean valued function, when evaluated returns *true* or *false*. An Assertion claims its predicate is *true*.

The meaning of the logical operators within a predicate have customary meanings.

\[
\text{predicate ::=}
\begin{align*}
\text{universal\_quantification} & \mid \\
\text{existential\_quantification} & \mid \\
\text{predicate\_atom} & \mid \\
\{ \text{and} \ \text{predicate\_atom} \}^+ & \mid \\
\{ \text{or} \ \text{predicate\_atom} \}^+ & \mid \\
\{ \text{xor} \ \text{predicate\_atom} \}^+ & \mid \\
\text{implies} \ \text{predicate\_atom} & \mid \\
\text{iff} \ \text{predicate\_atom} & \mid \\
\Rightarrow \ \text{predicate\_atom} & \}
\end{align*}
\]
Where $i$ is an interval, and $A, B$ are predicate atoms:

$$M_i[A \text{ and } B] \equiv M_i[A] \land M_i[B]$$ (the meaning of \textit{and} is conjunction)

$$M_i[A \text{ or } B] \equiv M_i[A] \lor M_i[B]$$ (the meaning of \textit{or} is disjunction)

$$M_i[A \text{ xor } B] \equiv M_i[A] \oplus M_i[B]$$ (the meaning of \textit{xor} is exclusive-disjunction)

$$M_i[A \text{ implies } B] \equiv M_i[A] \Rightarrow M_i[B]$$ (the meaning of \textit{implies} is implication)

$$M_i[A \text{ iff } B] \equiv M_i[A] \Leftrightarrow M_i[B]$$ (the meaning of \textit{iff} is if-and-only-if)

$$M_i[A \Rightarrow B] \equiv M_i[A] \Rightarrow M_i[B]$$ (the meaning of \textit{=>} is implication)
Predicate

The meaning of `true`, `false`, and `not` within a predicate have customary meanings.

Both parenthesized predicate and name may be followed by a time expression.

```
predicate_atom ::= [ not ]
( true | false
| predicate_invocation
| predicate_relation
| parenthesized_predicate [ time_expression ]
| name [ time_expression ]
)```
Where \( i \) is an interval, and \( A \) is a predicate atom:

\[
M_i[\text{not } A] \equiv \neg M_i[A] \quad (\text{the meaning of not is negation})
\]
\[
M_i[\text{true}] \equiv \top \quad (\text{the meaning of true is true})
\]
\[
M_i[\text{false}] \equiv \bot \quad (\text{the meaning of false is false})
\]
In a time expression, the ‘ means next clock cycle, the @ means when the subexpression, in seconds, is true, and the ^ means an integer number of clock ticks from now.

Grammatically, subexpression and integer value are time-free (no ‘ @ or ^).

time_expression ::= 
   ' | @ subexpression | ^ integer_value
Time Expression

Where $P$ is a name or a parenthesized predicate, $t$ is a time, $d$ is the duration of a thread’s period, and $k$ is an integer:

$M_t[P'] ≡ M_{t+d}[P]$  (the meaning of $P'$ at time $t$, is the meaning of $P$ a period duration hence)

$M[P@t] ≡ M_t[P]$  (the meaning of $P@t$ is the meaning of $P$ at time $t$)

$M_t[P^k] ≡ M_{t+dk}[P]$  (the meaning of $P^k$ at time $t$, is the meaning of $P$, $k$ period durations hence, or earlier if $k < 0$)
Predicate invocation allows labeled Assertions to be used by other Assertions.

\[
predicate_invocation ::= 
\begin{align*}
\text{assertion_identifier} \\
( [ \text{predicate_expression} \\
\{ , \text{predicate_expression} \}^* ] )
\end{align*}
\]
Predicate Invocation

Where $B$ is an Assertion label, $f_1 f_2 \ldots f_n$ are formal parameters, and $P$ is a predicate that uses $f_1 f_2 \ldots f_n$, and

$$\langle B : f_1 f_2 \ldots f_n : P \rangle \quad (\text{there is Assertion } B \text{ with predicate } P \& \text{ formal parameters } f)$$

then the meaning of predicate invocation is

$$\mathcal{M} _i [ B (a_1 a_2 \ldots a_n) ] \equiv \mathcal{M} _i [ P | f_1 a_1 | f_2 a_2 | \ldots | f_n a_n ]$$

(the meaning of a predicate invocation is the meaning of the predicate of the Assertion with the same label having actual parameters substituted for formal parameters)
Relations have conventional meanings. The `in` operators tests membership of a range.

```plaintext
predicate_relation ::= predicate_expression ( in range | relation predicate_expression )
relation ::= = | != | < | > | <= | >=
range ::= predicate_expression range_symbol
    predicate_expression
range_symbol ::= .. | ,, | .. | ,,  
```
Predicate Relation

Where c, d, l, and u are predicate expressions,

\[ M_i[c = d] \equiv M_i[c] = M_i[d] \]
\[ M_i[c \neq d] \equiv M_i[c] \neq M_i[d] \]
\[ M_i[c < d] \equiv M_i[c] < M_i[d] \]
\[ M_i[c > d] \equiv M_i[c] > M_i[d] \]
\[ M_i[c \leq d] \equiv M_i[c] \leq M_i[d] \]
\[ M_i[c \geq d] \equiv M_i[c] \geq M_i[d] \]
\[ M_i[c \text{ in } l..u] \equiv M_i[c] \geq M_i[l] \land M_i[c] \leq M_i[u] \]
\[ M_i[c \text{ in } l',..u] \equiv M_i[c] > M_i[l] \land M_i[c] \leq M_i[u] \]
\[ M_i[c \text{ in } l,..u] \equiv M_i[c] \geq M_i[l] \land M_i[c] < M_i[u] \]
\[ M_i[c \text{ in } l,..u] \equiv M_i[c] > M_i[l] \land M_i[c] > M_i[u] \]
Parentheses disambiguate precedence.

\[
parenthesized\text{\_}\text{predicate} ::= ( \text{predicate} )
\]

Where \( P \) is a predicate,

\[
\mathcal{M}_i[(P)] \equiv \mathcal{M}_i[P] \quad (\text{the meaning of parenthesis is its contents})
\]
Universal quantification claims its predicate is true for all the members of a particular set. Logic variables must have types.

Bounding the domain of quantification to a range, or when some predicate is true, defines the set of values that variables may take.

```
universal_quantification ::= all logic_variables logic_variable_domain are predicate
logic_variables ::= identifier { _, identifier }* : type
logic_variable_domain ::= in ( expression __ expression | predicate )
```
Universal Quantification

Where $v$ is a logic variable, $T$ is a type, $R$ is a range, and $P(v)$ is a predicate that uses $v$,

$$
M_i[\text{all } v:T \text{ in } R \text{ are } P(v)] \equiv \forall v \in M_i[R] \subseteq M_i[T] \mid M_i[P(v)]
$$

(for all $v$ in $R$, a subset of $T$, $P(v)$ is true)
Existential Quantification

Existential quantification claims its predicate is true for at least one member of a particular set.

\[
\text{existential\_quantification} ::= \exists \text{ logic\_variables} \text{ logic\_variable\_domain} \text{ that predicate}
\]
Existential Quantification

Where \( v \) is a logic variable, \( T \) is a type, \( R \) is a range, and \( P(v) \) is a predicate that uses \( v \),

\[
\mathcal{M}_i \left[ \exists v : T \text{ in } R \text{ that } P(v) \right] \equiv \\
\exists v \in \mathcal{M}_i[R] \subseteq \mathcal{M}_i[T] \mid \mathcal{M}_i[P(v)]
\]

(there exists \( v \) in \( R \), a subset of \( T \), for which \( P(v) \) is true)
Predicate Expression Grammar

predicate_expression ::=  
  sum logic_variables [ logic_variable_domain ]  
        of predicate_expression  
  | product logic_variables [ logic_variable_domain ]  
         of predicate_expression  
  | numberof logic_variables [ logic_variable_domain ]  
        that predicate_atom  
  | predicate_subexpression  
        [ { + predicate_subexpression }*  
        | { * predicate_subexpression }*  
        | − predicate_subexpression  
        | / predicate_subexpression  
        | ** predicate_subexpression  
        | mod predicate_subexpression  
        | rem predicate_subexpression ]
Sum, Product, and Number-Of Quantifiers

Where \( v \) is a logic variable, \( T \) is a type, \( R \) is a range, \( P(v) \) is a predicate that uses \( v \), \( E(v) \) is a predicate expression that uses \( v \), and \( e, f \) are predicate subexpressions,

\[
\begin{align*}
M_i\left[\text{sum} \ v : T \ in \ R \ of \ E(v)\right] & \equiv \sum_{v \in R} M_i\left[E(v)\right] \\
(\text{sum the value } E(v) \text{ for each } v \text{ in the range } R) \\
M_i\left[\text{product} \ v : T \ in \ R \ of \ E(v)\right] & \equiv \prod_{v \in R} M_i\left[E(v)\right] \\
(\text{multiply the value } E(v) \text{ for each } v \text{ in the range } R) \\
M_i\left[\text{numberof} \ v : T \ in \ R \ that \ P(v)\right] & \equiv \| \{v \in M_i[R] \mid M_i[P(v)]\} \| \\
(\text{cardinality of the set of } v \text{ in } R \text{ for which } P(v) \text{ is true}) \\
M_i\left[e + f\right] & \equiv M_i\left[e\right] + M_i\left[f\right] \quad (\text{the meaning of } + \text{ is addition}) \\
M_i\left[e \times f\right] & \equiv M_i\left[e\right] \times M_i\left[f\right] \quad (\text{the meaning of } \times \text{ is multiplication})
\end{align*}
\]
More: Assertion  Predicate Expression

**Predicate Expression**

\[ M_i[e - f] \equiv M_i[e] - M_i[f] \]  (the meaning of - is subtraction)
\[ M_i[e / f] \equiv M_i[e] \div M_i[f] \]  (the meaning of / is division)
\[ M_i[e ** f] \equiv M_i[e]^{M_i[f]} \]  (the meaning of ** is exponentiation)
\[ M_i[e \mod f] \equiv M_i[e] \mod M_i[f] \]  (the meaning of mod is modulus)
\[ M_i[e \rem f] \equiv M_i[e] \rem M_i[f] \]  (the meaning of rem is remainder)
Negation has the usual meaning.

Assertions that define functions using := may be referenced by its label together with a list of actual parameters, if any.

The reference and parameters mean the textual substitution of formal parameters with actual parameters in the text of the labeled Assertion’s predicate expression.

Parenthesis define precedence.

The present instant is now.
Predicate Subexpression Grammar

predicate_subexpression ::= 
[ = ]
( property_constant
| { package_identifier ::= }*
assertion_function_identifier
_ [ parameter_list ] _
| now
| string_literal
| numeric_literal
| ( predicate_expression )
[ time_expression ]
| ( variable_name | port_name )
[ time_expression ] )
Parameter List

```
parameter_list ::=  
expression_or_range { expression_or_range }*

expression_or_range ::=  
expression [ .. expression ]
```
Predicate Subexpression

Where $S$ and $E$ are predicate expressions,

- $M_i[\text{not } S] \equiv \neg M_i[S]$ \textit{(the meaning of not is complement)}
- $M_i[-S] \equiv \neg M_i[S]$ \textit{(the meaning of - is negation)}
- $M_i[(E)] \equiv M_i[E]$ \textit{(the meaning of parenthesis is the meaning of its contents)}
Assertion Function Invocation

Where \( C \) is an Assertion-function label, \( f_1 f_2 \ldots f_n \) are formal parameters, and \( E \) is a predicate expression that uses \( f_1 f_2 \ldots f_n \), and

\[
\ll C : f_1 f_2 \ldots f_n := E \rr \text{ (there is Assertion-function } C \text{ with predicate expression } E \text{ and formal parameters } f)\]

the meaning of Assertion-function invocation is

\[
\mathcal{M}_i\left[ C(a_1 a_2 \ldots a_n) \right] \equiv \mathcal{M}_i\left[ E \mid f_1^{a_1} \mid f_2^{a_2} \ldots \mid f_n^{a_n} \right]
\]

(the meaning of a predicate function invocation is the meaning of the expression of the Assertion-function with the same label having actual parameters substituted for formal parameters)