SAE Aerospace An SAE International Group	**AEROSPACE STANDARD**	**AS5506/TBD**
		Issued 2013-09 Draft
SAE Architecture Analysis and Design Language (AADL) Annex Volume TBD: Annex TBD: Unit Relations Annex		

RATIONALE

The purpose of this annex is to provide a way to specify relations between unit types and unit type combinations. These relations can be used to automate building of conversion functions between proportional unit types.

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.

1. Unit Relations Sublanguage

1. Scope

1. The Unit Relations Sublanguage Annex provides a standard AADL sublanguage extension to define relations between different Unit Types declared in the core AADL.
2. The core AADL provides a way to define a Unit Type that is a set of measurement unit identifiers with conversion factors between them.

```
Length_Unit : type units ( mm, cm => mm * 10, m => cm * 100, km => m * 1000 );
Time_Unit : type units ( sec, min => sec * 60, h => min * 60 );
```

3. There are relations between various Unit Types that cannot be expressed in the core AADL. For example, one measurement unit can represent a relation between a couple of others.

```
Speed_Unit : type units ( mph, mpsec => mph * 3600,
                          kmpsec => mpsec * 1000, kmph => mph * 1000 );
```

4. Also it may happen that two different AADL models developed by two different teams define different Unit Types to represent the same physical quantity. If these models are used within another AADL model there is no way to compare properties of that types.
5. The Unit Relations Annex is aimed to provide facilities to handle the both situations.
6. Another aim of Unit Relations Annex is to automate conversion between values of different Unit Types and multiplicative combinations of Unit Types (additionally to conversions between values in different measurement units of the same Unit Type defined in the Unit Type declaration).
7. Unit Relations Annex does not support relations between Units with affine dependency between them (for example, Kelvins and Fahrenheits for Temperature Units) because it is hard to define strict semantics of multiplicative operations on such Units.
8. *It would be useful to support standard systems of measurement units (for example, SI, or CGS) to not redefine units from these systems manually in model for its using.*

2. Overview of Unit Relations Sublanguage Concepts

9. Basic Unit Type is a Unit Type declared in the core AADL.
10. Derived Unit Type is formed by multiplication and division of Basic Unit Types.

```
-- 'Length_Unit * Length_Unit / Time_Unit' is an example of Derived Unit Type
```

11. A normal form of a Derived Unit Type is an expression with powers, multiplications and divisions, where each Base Unit Type is met just once.

```
-- 'Length_Unit^2 / Time_Unit' is a Derived Unit Type in a normal form
```

12. A set of measurement units of a Derived Unit Type is formed by all combinations of measurement units of the corresponding Basic Unit Types taken in a normal form.

```
-- Measurement units of 'Length_Unit^2 / Time_Unit' consist of
-- Length_Unit[mm]^2/Time_Unit[sec], Length_Unit[cm]^2/Time_Unit[sec],
-- Length_Unit[m]^2/Time_Unit[sec],  Length_Unit[km]^2/Time_Unit[sec],
-- Length_Unit[mm]^2/Time_Unit[min], Length_Unit[cm]^2/Time_Unit[min],
-- Length_Unit[m]^2/Time_Unit[min],  Length_Unit[km]^2/Time_Unit[min],
-- Length_Unit[mm]^2/Time_Unit[h],   Length_Unit[cm]^2/Time_Unit[h],
-- Length_Unit[m]^2/Time_Unit[h],    Length_Unit[km]^2/Time_Unit[h].
```

13. Basic concepts of the Unit Relations Annex sublanguage consist of a unit types equation and a unit type independence assertion.
14. Unit types equation postulates an equality of a value in one Derived Unit Type with specific measurement unit to a value in another Derived Unit Type with specific measurement unit.
15. Unit type independence assertion helps to prevent unintended consequences of unit type relation definitions. The assertion enumerates unit types that are designed to be independent of each other. If an error in unit type relation makes a subset of the types dependent, the assertions allows to detect it and to warn users.

16. A recommended way to use Unit Relations Annex is to augment each unit type declaration clause that defines a derived unit type with a Unit Relations Annex subclause that explicitly describes relation of the newly defined unit type with earlier defined ones.

3. Unit Relations Sublanguage Grammar

17. Unit Relations Annex can be declared either as an annex library or as an annex subclause attached to a property set.
18. Unit Relations Annex consists of one or more unit type equations and unit type independence assertions.

Syntax

```
unit_relations_annex ::=
  { unit_types_assertion | unit_types_equation }+

unit_types_assertion ::=
  assert independence unit_type { , unit_type }+;

unit_types_equation ::=
  unit_expr = unit_expr ;

unit_expr ::= unit_mul_expr

unit_mul_expr ::=
  unit_div_expr { * unit_div_expr }*

unit_div_expr ::=
  unit_pow_expr [ / unit_pow_expr ]

unit_pow_expr ::=
  unit_basic_expr [ ^ unitless_hi ]

unit_basic_expr ::=
    unit_value
  | unitless
  | ( unit_expr )

unit_value ::=
  unit_type [ unit ]

-- expressions with unitless value
unitless ::= unitless_sum

unitless_sum ::=
  unitless_mul { SUM_OP unitless_mul }*

unitless_mul ::=
  unitless_power { MUL_OP unitless_power }*

unitless_power ::=
  unitless_hi [ ^ unitless_hi ]

unitless_hi ::=
    basic_unitless_value
  | ( unitless )

basic_unitless_value ::=
    signed_aadlreal_or_constant
  | signed_aadlinteger_or_constant
```

```
MUL_OP ::= * | /
SUM_OP ::= + | -


-- unit types

unit_type ::= unit_unique_property_type_identifier

unit ::= unit_identifier
```

19. *unit*`_unique_property_type_identifier`, *unit*`_identifier`, `signed_aadlreal_or_constant` and `signed_aadlinteger_or_constant` are syntactic categories inherited from the core AADL as is.

Naming Rules

1. *unit*`_unique_property_type_identifier` of `unit_type` must refer to a unit type.
2. `unit` part of `unit_value` must refer to a unit identifier defined in the declaration of the unit type referred by `unit_type`.
3. unit_type must be resolved by standard scope rules from AADL (unqualified identifier means identifier from the same package as annex declared, qualification of qualified identifier must be consist with "with" clauses of the package or property set where annex declared).

Legality Rules

1. No one unit type may be included in the left part of the equation and the right part of the same equation.
2. All unit types included in the same assertion must be different.
3. Units Relation Annex may be placed to package-level only or property set-level only.

Consistency Rules

1. Equations (with respect to unit types declarations) must not allow to infer equivalence of a Unit Type to unitless value. It means the system of equations must not be enough to infer an equality *T* [*u*] = *const* where *T* is a Unit Type, *u* is a measurement unit of T, *const* is a unitless value. Inferring from unit types equations is defined as the same as inferring from system of equations on variables and real numbers and product of variables and real numbers.
2. Equations (with respect to unit types declarations) must not allow to infer equivalence of two different unitless values. It means the system of equations must not be enough to infer an equality *const1* = *const2* where *const1* and *const2* are unequal unitless values.
3. Equations must not violate unit type independence assertions, i.e. the system of equations must not be enough to infer an equality (after all possible simplification), where a set of Unit Types collected from both sides of the equality is a non-empty subset of a set of Unit Types enumerated in the independence assertion.

Semantics

20. An equation means possibility of proportional conversion a value of a Derived Unit Type from the left part of the equation to a value of a Derived Unit Type from the right part of the equation, and vice versa. For example, equation Square[kmq] = Length[km] * Length[km] means if we have a value with derived unit types Length[m] * Length[m] (as a result of intermediate computations, for example) where m is a measurement unit of unit type Length such as km => m*1000; then we may compute a value of unit type Square in kmq measurement unit by multiplication $(1/1000)^2$ to value in unit type Length[m] * Length[m], and result value will be of unit type Square with kmq unit.
21. A system of Unit Relation Annex equations defines an equivalence relation between proportional Derived Unit Types. Equivalence of two Derived Unit Types means that there is a proportional conversion function between values of that types. The form of the conversion function is F(u) = C * u, where u - a value of

one Derived Unit Type, C - unitless constant. One of possible algorithms to infer a conversion function is presented in Appendix C.

22. Checking of unit relations consistency is a global operation. I.e., while checking a unit relation we must concern to all the unit relations and unit type declarations we have in a model. For example, if an AADL model imports packages from different projects and that packages defines semantically equivalent unit types, Unit Relations Annex allows to define an equation between the types in the top model. In this case system of equations of the top model is a union of equations from all imported packages and equations from the model itself.

Examples

```
property set SI is

  Length_Unit : type units ( mm, cm => mm * 10, m => cm * 100, km => m * 1000 );

  Time_Unit   : type units ( sec, min => sec * 60, h => min * 60 );

  Speed_Unit  : type units ( mph, mpsec => mph * 3600,
                             kmpsec => mpsec * 1000, kmph => mph * 1000 );
  annex unit_relations {**
      Speed_Unit[mph] = Length_Unit[m] / Time_Unit[h];
      assert independence Length_Unit, Time_Unit;
  **}

end SI;
```

Examples

```
property set Project1_Property_Set is

  Length_Unit : type units ( mm, cm => mm * 10, m => cm * 100, km => m * 1000 );

end Project1_Property_Set;

property set Project2_Property_Set is

  Length_Unit : type units ( mm, cm => mm * 10, m => cm * 100, km => m * 1000 );

end Project2_Property_Set;

package Project3 is

  with Project1_Property_Set;
  with Project2_Property_Set;

  annex unit_relations {**
      Project1_Property_Set::Length_Unit[m] = Project2_Property_Set::Length_Unit[m];
  **}

end Project3;
```

Appendix A. Suggested algorithm of checking system of equations consistency

1. Say system of Unit Type Equations is normalized if each Unit Type is included in an equation no more than once, and each unit at unit_value is the least unit of unit_type of this unit_value, and system of equations doesn't have equations like *T* [*u*] = *const,* or like *T* [*u*] = *T* [*u*]. To become system of Unit Type Equations normalized it is need to replace each T[u'] by C * T[u] if u' is declared as C * u in declaration of T, remove equations like T[u] = T[u], simplify each equation (compute value of expressions on unitless values, move all powers of the same unit to the same part of the equation, simplify products of the same unit ($x^p * x^q \rightarrow x^{p+q}$), replace x^0 by constant 1 for any unit x).
2. Suggesting algorithm of checking consistency is iterative on sequence of equations. The first step of iteration is choosing equation to be processed as current (at the beginning it is the first equation, at the next step it is the next equation after equation from previous step). The next step of iteration is choosing any unit type from the current equation (from left or right part of equation in any degree) and expressing this unit type throught other unit types from the current equation (by multiplication and division all parts of equation, and exponentiation to become unit type chosen with degree 1). Say now the current equation is a kind of $T[u] = g(T_1[u_1], T_2[u_2], \ldots, T_n[u_n])$, where T, T_1, T_2, …, T_n are unit types, and u, u_1, u_2, …, u_n are theirs measurement units respectively, and g is a non-constant function (it is used all arguments essentially). The next step of iteration is replacing all occurrences of T[u] in other equations by $g(T_1[u_1], T_2[u_2], \ldots, T_n[u_n])$. The last step of iteration is normalization of the system processed (without changing an order of equations). The iteration process must be stopped if all equations already processed by iterations or if the system of equations can't be normalized due to inferring an equation like T[u] = const. The last case means the initial system of equations was inconsistent. The first case means the initial system of equations was consisted.

Appendix B. Suggested algorithm of checking system of equations and independence assertions consistency

1. This algorithms assumes the system of equations have been normalized by algorithm from appendix A and consistent. So each equation from the system is a kind of $T[u] = g(T_1[u_1], T_2[u_2], \ldots, T_n[u_n])$, where T, T_1, T_2, …, T_n are unit types, and u, u_1, u_2, …, u_n are theirs measurement units respectively, and g is a non-constant function (it is used all arguments essentially), and T[u] occur only in one equation ($T_1[u_1]$, $T_2[u_2]$, …, $T_n[u_n]$ may occur several times in equations but only in the right part of equations). The system of equations and assertions are consistent if the system is consistent with each assertion. So it is needed to looping through sequence of assertions and check consistency of normalized system of equations with each assertion. The algorithm suggested the following process of inferring an equality like *const* = $F(T_1[u_1], T_2[u_2], \ldots, T_n[u_n])$, where *const* is a constant, *F* is a non-constant function, $T_1[u_1]$, $T_2[u_2]$, …, $T_n[u_n]$ are unit types with their measurement units from A. The system of equations is inconsistent with assertion A iff this equation is inferred.
2. Suggesting algorithm of checking consistency with assertion A is iterative on sequence of equations (it is like the algorithm from appendix A, so modified steps are highlighted by italic). The first step of iteration is choosing equation to be processed as current (at the beginning it is the first equation, at the next step it is the next equation after equation from previous step). The next step of iteration is choosing any unit type from the current equation (from left or right part of equation in any degree) *which is not included into A* and expressing this unit type throught other unit types from the current equation (by multiplication and division all parts of equation, and exponentiation to become unit type chosen with degree 1). Say now the current equation is a kind of $T[u] = g(T_1[u_1], T_2[u_2], \ldots, T_n[u_n])$, where T, T_1, T_2, …, T_n are unit types, and u, u_1, u_2, …, u_n are theirs measurement units respectively, and g is a non-constant function (it is used all arguments essentially). The next step of iteration is replacing all occurrences of T[u] in other equations by $g(T_1[u_1], T_2[u_2], \ldots, T_n[u_n])$. The last step of iteration is normalization of the system processed (without changing an order of equations). The iteration process must be stopped if all equations already processed by iterations or if the system of equations can't be normalized due to inferring an equation like T[u] = const *or if the system of equations doesn't allow to choose the current unit type at the second step of iteration* (it is means the algorithm infers an equality like *const* = $F(T_1[u_1], T_2[u_2], \ldots, T_n[u_n])$).

3. The algorithm may be optimized by the following way. Say S is a normalized system of equation. Then each assertion A can be divided into 3 subsets: $A \cap L$, $A \cap R$, $A \setminus (L \cup R)$, where L and R are sets of unit types from the left and right parts of S respectively. So to optimize algorithm for A it is safe to remove from S all equations which unit type from the left part doesn't included into $A \cap R$.

Appendix C. Suggested algorithm of building conversion functions

1. The following algorithm checks existence of conversion function between two Derived Unit Types $T_1[u_1]$ and $T_2[u_2]$ and (in case of existence) builds this function. At the first the system of equations must be normalized by the algorithm from appendix A. So each equation of the system obtained likes $t[u] = g(t_1[u_1], t_2[u_2], \dots, t_n[u_n])$, where $t, t_1, t_2, \dots, t_n$ are unit types, and $u, u_1, u_2, \dots, u_n$ are theirs measurement units respectively, and g is a non-constant function (it is used all arguments essentially), and $t[u]$ occur only in one equation ($t_1[u_1], t_2[u_2], \dots, t_n[u_n]$ may occur several times in equations but only in the right part of equations). The next step of algorithm is replacing $T_1[u_1]$ by $T_1[u_1'] * C1$ where u_1' is the least measurement unit of T_1 and u_1 is defined as $u_1' * C_1$, and replacing $T_2[u_2]$ by $T_2[u_2'] * C_2$ where u_2' is the least measurement unit of T_2 and u_2 is defined as $u_2' * C_2$. And finally it is need to replace each basic unit type in the expression $(C_1 / C_2) * (T_1[u_1']/T_2[u_2'])$ which included in the normalized system as the left part of an equation by the right part of this equation and simplify the expression obtained. If the expression has been simplified to unitless constant (named as C), than conversion function from $T_2[u_2]$ to $T_1[u_1]$ exists and equals to $F(x) = C * x$ where x is a value of $T_2[u_2]$. If expression obtained is non-constant, $T_2[u_2]$ can't be converted to $T_1[u_1]$ by unit relations.