ADL++
Reference Manual

Alankar Khara, UNI: ask2206
COMS W4115 – winter 2014

Contents

1. Introduction

ADL++ is an Architecture Description Language based on the concepts of General Systems Theory. This language and its compiler are different from the set of current languages in ADL category in three major ways: 1) it is not domain specific, 2) it attempts to remove communication gap existed among various stakeholders, 3) it captures design rationale and history of the design decisions. In addition, this language provides analytical tools to solve the problem of architecture description of a concerned Systems based on real-time constraints.

ADL++ is complaint with **ISO/IEC/IEEE 42010** *Systems and software engineering — Architecture description*, which provides international standard to develop architecture description of system.

2. Lexical conventions

There are four kinds of tokens:

a. Identifiers
b. Keywords
c. Constants
d. Operators
e. Punctuation

2.1 Comments

The character `::` starts a comment and the characters `::` terminates it. The content between these characters is ignored by the parser.

2.2 Identifiers

An identifier is a sequence of letters, digits and underscores "_" .

For example: *Heat_Energy, Boiler_No_980*

2.3 Keywords

The following identifiers are reserved for use as keywords, and may not be used otherwise:

```
system
sub-system
environment
constraint
vulnerability
viewpoint
formula
int
char
float
bool
if
double
```

```
otherwise
while
continue
case
return
then
```

2.3 Constants

There are four kinds of constants:

a. Integer: sequence of digits
b. Strings: sequence of characters between double quotes " "
c. Float: has three parts – integer, fraction and optionally signed integer exponent
d. Character: set of strings [\n, \r, \t] for next line, return, and tab representations respectively

2.4 Operators

There are three types of operators

a. Mathematical Operators: Set of characters [+, -, *, /, ^, %]
b. Functional Operators: There are two types-
    a. :=> called reference operator (its role is defined further in the manual)
    b. = is assignment operator
c. Logical Operators: Set of characters [==, <=, >=, !=, &&, ||]

2.4.1 Operators Precedence

Following table is highest to lowest precedence table:

^	Power Operator
* / %	Multiplication and Division
+ -	Addition and subtraction
< <= > >=	Inequality relational
== !=	Equality relational
&&	Logical AND
||	Logical OR
=	Assignment
:=>	Reference

2.5 Punctuation

The symbol defined below increase the readability of the language.

Symbols	Name	Use
,	Comma	It is used to separate elements in Tuples, one of the derived data-types explained in the next section. It is also used in separating sub-systems in system declaration expression
<<	Start	It is used to tell parser that tuple definition starts
>>	End	It is used to tell parser that tuple definition ends
!	Call	It is used to expand the scope of variable defined within sub-system tuple to the outside program

3.0 The role of identifiers with Declaration Expressions Syntax

Identifiers used in the program are mainly called variables. In this section, types of identifiers are defined along with the expressions, which define them. Syntax is presented in *italics* with symbol >> representing as a line in the program.

3.1 Types

Mainly there are two types of identifier variables: called Basic Types and Derived Types.

3.1.1 Basic Types

There are six basic types:

Basic Type	**Definition Syntax Example**	**Representation**
Integers	*no_people (int)*	16-bit 2's compliment notation
Floats	*money (float)*	Range $10^{+/-38}$ or 0; precision: seven decimal digits
Doubles	*volume (double)*	Same range as float; precision: 17 decimal digits
Characters	*info (char)*	ASCII digit
Strings	*check (string)*	Set of ASCII digits between quotes
Boolean	*needed (bool)*	True, False or Null values

Syntax of variable declaration expression of basic type is *Identifier (keyword)* , where keyword is *int* for Integer, *float* for Float, *double* for Double, *char* for Characters, *string* for Strings, *bool* for Boolean.

Type conversion is not allowed in these basic types. Variables with basic types are always declared as a part of the concerned system and cannot be used outside of its scope independently. To access it in the program, one need to use ! Call Operator, as shown in the example below. It is also explained in the next section.

For example: Water is a system defined by two variables → Density and Mass.
>> Water :=> <<Density(float) , Mass (float)>>
>>....
>>if Water!Density > 1 then Flask!Clock = False Otherwise True

3.1.2 Derived Types

Derived Types makes ADL++ domain independent. Following are the derived types introduced:

3.1.2.1 Formula

In Systems theory, admissible Input and Output values are often defined as a parameterized or non-parameterized function. Moreover, there are pre and post conditions at system's run-time which can be functions as well. To capture that abstraction, derived data type Formula is introduced. This will allow independent architecture description without defining function itself.

Syntax of Formula is *Identifier (keyword)*, where keyword is Formula

For example, Energy is defined as formula in the program, which is later assigned to mathematical formula as below. This will enhance to readability of the Language as function can be used in the program prior to its definition.

>> Steam :=> <<Energy (Formula), Volume (int)>>
>> ...
*>> Steam!Energy = Temperature_Change * specific heat * mass*
:: Right hand side include other declared variables in the program ::

3.1.2.2 System

Language is based on System Oriented Approach, and System will follow the strict mathematical definition under System theory. In simple words, Objects (as defined in Object Oriented Paradigm) will be replaced by Systems.

Syntax for System declaration is: *keyword :=> Identifier*, where keyword is *System*

For example, below one can see System declaration.
>> System :=> Boiler
:: This is somewhat like Class declaration under Object Oriented Paradigm::

System is composed of sub-systems, also called components. Hierarchy, which is one of the most important properties of ADLs is built-in with this conception.

Syntax for subsystem declaration is: *Identifier :=> Identifier, Identifier ...*., where left hand side Identifier is variable defined as *system*, and right hand side identifiers will be defined as *sub-systems*.

For example, as below:
>> Boiler :=> Burner, Flask, Water, Steam, Heat_Sensor
:: Right side is the set of sub-systems in a system::

3.1.2.3 Tuple

Sub-System, as a system itself will be defined as a set of variables, called Tuple (as advanced data type, much like array or list).

Syntax for Tuple declaration is: *Identifier :=> << Identifier (keyword) , ... , Identifer (keyword)>>*,where, left-hand side Identifier is variable defined as sub-system and right hand side identifiers are basic and derived types of variables. Keywords are the data types of defined variables.

Like below:
>> Heat_Source :=> <<Energy_Given (float), Cost (formula), Burner_Factor (Constant)> >

Access to Tuple variable is based on the index, which starts with 1 or with the name of the variable. Syntax of expression to access Tuple variable is : *Identifier![index-number]* or *Identifier![Identifer]*.

Representation like below:
>> Real_Cost = Heat_Source![2]
>> Real_Cost = Heat_Source!Cost

4.0 Expressions

Identifiers along with operators form expressions. This language has one simple expression for every single operator, defined in lexical conventions. In addition, many expressions can be built with the combination of

expressions. Simple expression forms include expressions with binary arithmetic operators, logical operators, assignment operator and reference operator as defined below:

4.1 Binary arithmetic operators

Many kinds of mathematical constructs can be developed using combination of expressions below. Simple types are defined:

4.1.1 expression / expression

This operator returns the division of the two arguments, which must be integers, floats or doubles.

4.1.2 expression * expression

This operator returns the multiplication of the two arguments, which must be integers, floats or doubles.

4.1.3 expression - expression

This operator indicates subtraction of two arguments, which must be integers, floats or doubles.

4.1.4 expression + expression

This operator indicates addition of two arguments, which must be integers, floats or doubles.

4.2 Relational operators

These operators evaluate the comparisons and return the result (true, false or null).

4.2.1 expression == expression

This operator evaluates whether the two expressions are equal. The expressions can be integer, strings, or Booleans.

4.2.2 expression < expression

The < operator evaluates if the first expression is less than the second one. The arguments must be integers, floats or doubles.

4.2.3 expression <= expression

The <= operator evaluates if the first expression is less or equal than the second one. The arguments must be integers, floats or doubles.

4.2.4 expression > expression

The > operator evaluates if the first expression is greater than the second one. The arguments must be integers, floats or doubles.

4.2.5 expression >= expression

The >= operator evaluates if the first expression is greater or equal than the second one. The arguments must be integers, floats or doubles.

4.2.6 expression != expression

The != operator evaluates if the two expressions are different. The expressions can be integers, floats, doubles, string or boolean.

4.3 Logical operators

4.3.1 boolean-expression && boolean-expression

This operator evaluates if both boolean expressions are true.

4.3.2 boolean-expression || boolean-expression

This operator evaluates whether at least of the two expressions is true.

4.4 Assignment operator

4.4.1 identifier = expression

This operator declares a variable with the same name as the identifier. Then, it evaluates the expression and assigns its result and type to the variable.

4.5 Reference operator

4.5.1 identifier :=> expression

This operator declares a variable with the same name as the identifier. Then, it references the expression whenever that variable is used in the program.

5.0 Statements

There are two main kinds of statements in this language, namely: conditions and loops. Both are represented below.

5.1 Conditional statements

Conditional statements take one of the two following forms:

If boolean-expression statements otherwise statements.

If the boolean expression evaluates to true, the first set of statements is executed. Otherwise (and if the otherwise statement is used), the second set of statements is executed.

For pre and post conditions, logic plays an important part. Standard Logical operators will be used like below:
If/otherwise statement ends with period ".".

>> if Boiler!Heat_Source!Cost > MAX_Cost && Boiler!Heat_Source!Energy < MAX_ENERGY
>> Set State = TRUE
>> Otherwise
>> State = FALSE .

5.2 Loops

To facilitate analysis of systems design, loops play the central role. One can simulate run-time environment using loops. There is only while loop defined in this language, and its syntax is:

While Boolean-expression continue statement

This will be used as bellow:

>> while Burner!Energy!Cost >= MAX_COST
>> continue Burner!Burning_State.

Loops will end with '.' period.

6.0 Functions

The parallel to function in this language is the construct of sub-system tuple. There is no separate definition for functions.

7.0 Program structure

Readability is the key to any successful ADL. To enhance readability, following structure is proposed

a. System Declaration with sub-system identification
b. Sub-system definitions
c. Interconnections and interdependencies of sub-systems with assignment expressions
d. Vulnerabilities defined with control statements
e. Run-time state transitions simulated using loops
f. Develop test cases
g. Run Analysis
h. LP Solver

Using systems engineering approach, which is inherently interdisciplinary, to the architecture description problems, ADL++ will be first domain independent Architecture description language.

8.0 References

[1] Skyttner, Lars. General systems theory: ideas & applications. World Scientific, 2001.

[2] Polderman, Jan Willem, and Jan C. Willems. Introduction to mathematical systems theory: a behavorial approach. Vol. 26. Springer, 1998.

[3] Hinrichsen, Diederich, and Anthony J. Pritchard. Modelling, State Space Analysis, Stability and Robustness. Vol. 1. Springer, 2004.

[4] Lin, Yi, ed. General systems theory: A mathematical approach. Vol. 12. Springer, 1999.

[5] Braga, Christiano, and Alexandre Sztajnberg. "Towards a rewriting semantics for a software architecture description language." Electronic Notes in Theoretical Computer Science 95 (2004): 149-168.

[6] Rademaker, Alexandre, Christiano Braga, and Alexandre Sztajnberg. "A rewriting semantics for a software architecture description language." Electronic Notes in Theoretical Computer Science 130 (2005): 345-377.

[7] Garlan, David. "Software architecture and object-oriented systems."Proceedings of the IPSJ Object-Oriented Symposium. 2000.

[8] Von Bertalanffy, Ludwig. "{General System Theory}." General systems 1 (1956): 1-10.

[9] Austin M.A., Modelling Systems Structure and Behaviour, Presented at Institute for Systems Research, University of Maryland, College Park, 2012.