UNIVERSITY OF COLUMBIA

ADL++

Architecture Description Language

Alan Khara

5/14/2014

This report is submitted to fulfill final requirements for the course COMS W4115 at Columbia University.

Contents

Chapter 1: Introduction

ADLs were mainly graphical in nature, with numerous Line and Box representations that were later standardized by Object Management Group (OMG) in Unified Model Language (UML). Being quickly adopted by industry, UML was infested with ambiguous features: a given relationship between components of a system could be ambiguous and yield two different interpretations of the same drawing. The underlying semantics of ADL needs to be based on some general theory of Architecture Description. An abstraction of Systems Theory (ST) can provide a common language for all stakeholders while also allowing the capacity needed for domain-specific extensions. Systems theory considers system as a set of components, which are smaller systems itself. System is dynamic when it is changing its states. It accepts input, and provides output, which is observable state of the system.

ADL++ is an Architecture Description Language based on strictly defined semantics of Systems Theory. A sub-set of this language is implemented as part of this project.

Chapter 2: Language ADL ++

2.1 Program Execution

To run an .adl program no setup is necessary, simply use the out command with your .adl file as the only argument.

```
./adl < gcd.adl
```

This command will create output of the file on the command line as well.

2.2 Variables

Variables in ADL++ are not declared as a datatype. In the strict systems semantics, they are considered as states. To declare the variable, one needs to use the keyword “state”, like below. This is not followed by a semicolon.

```
state control
```

Here, state is a keyword and control is an identifier. Once declared, variables can be assigned values using assignment operator, “<-“in ADL ++, as followed. This is followed by a semicolon.

```
control <- 10;
```

2.3 Components

System is divided into components, and as such components are specified as functions, which takes input and produce output. This component takes one input (parameter) and sends (return) back change in the system state:

```
component boiler_test  << heat >> [

        state sensor

        sensor <- 1;

        send sensor $

]
```

A component can have no inputs (parameters) or as many number of inputs (paramenters) as required.

2.4 Control Flow

ADL++ support if -then /otherwise statements as follows:

```
if sensor != on then alert <- alarm;

otherwise alert <- no; :: otherwise clause is optional ::
```

It is important to note that both statements end with semicolon. ADL++ also supports only if-then statement without otherwise clause.

Loops are used in ADL++, however they are given the terminology of unified model language (UML). There are two types of looping constructs. First one is called constraint-satisfy clauses, which work like below:

```
constraint sensor <= 100 satisfy [

        sensor <- sensor + 1; :: This statement will run until sensor <= 100 ::

        ] …
```

Second looping construct works like for loop, and it is called repeat, which works like below:

```
repeat [

        state s ;

        s <= 100;

        s <- s + 1

        ]
```

2.5 Checking System States

In ADLs, it is important to check states of the system and components during runtime. This is facilitated by built in function view. This function prints the output to stdout like below:

```
view <<  burner << heat >> >>;
```

The above function will print the state of component burner, as represented by integer.

Chapter 3: Language Reference Manual ADL ++

3.1 Types

3.1.1 Primitive Data Types

There are only two primitive types in ADL ++: integers, Boolean. This is in strict accordance to the unified model language, which restricts the use of other primitive datatypes. A boolean in ADL ++ is defined by the true and false keywords. In addition, integer is standard 32 bit long primitive datatype. In ADLs the main use of integers is when defining the state of a component or a system. After declaring state variable, an integer value can be assigned to it.

3.1.2 Non-Primitive Types

In ADL++ components are the first class objects. They can be assigned to variables, and passed as arguments to other components. The main component is called system. Within system, multiple views of the components can be built and tested. ADL ++ is statically typed , as opposed to dynamically typed.

3.2 Lexical Conventions

3.2.1 Identifiers

An identifier is a sequence of letters, digits, or underscores. The first character must be a letter; the underscore is not considered a letter. Upper and lower case letters are different.

3.2.2 Keywords

Following is the list of keywords currently implemented

- System
- Component
- State
- Send
- Constraint
- Satisfy
- Repeat
- If
- Otherwise
- Print
- True
- False

3.2.3 Literal

Literals or constants are the values written in a standard form whose value is obvious. In contrast to variables, literals do not change in values. For example, 3, 28, “hello”

3.2.4 Punctuation

Punctuator	Use	Example
,	Component input / function parameters	component gcd << a, b >>
[]	Statement List delimiter	constraint a > b satisfy [..]
<< >>	Component input delimiter	view << >>
;	Statement end	i <- 10 ;
$	Send Statement end	Send a $

3.2.5 Comments

The characters :: introduce a multi-line comment, which terminates with the characters :: . Multi-line comments cannot be nested within multi-line comments.

```
:: This program is written by Alan Khara

and this is a comment ::

Component system <<>> [..]
```

3.2.6 Operators

Operator	Use	Associativity
*	Multiplication	Left
/	Division	Left
+	Addition	Left
<-	Assignment	Non-Associative
==	Equality	Left
!=	Not Equal	Left
<	Less	Left
>	Greater than	Left
<=	Less than or equal	Left
>=	Greater than or equal	Left

Precedence of the operators is as follows

* /

+ -

<><= >=

<- !=

^

==

3.3 Syntax

3.3.1 Program Structure

A Program in ADL ++ consists of a system as a main program, and components as functions called from the main program. 'Constraint and if' statements are used for the control logic inside the components.

3.3.1 Expressions

An expression in ADL++ is a sequence of operators and operands that produce an output (value) and may have side effect. The order of evaluation of subexpressions, and therefore the order in which side effects take place, is left to right. Various forms of valid ADL++ expressions are as below:

:: a+b evaluated first, then u+bv, then division ::

<<x + y>> / <<a + b>>;

:: First – Comp2 is evaluated and then Comp 3, finally Comp 1 ::

Comp1 << Comp2<<>>, Comp3<<>> >>;

Operands always have compatible types (states of the components); therefore type-check is not required.

Constants

As discussed in the lexical conventions, constants can be Integer or Boolean.

Identifiers

An identifier designates states or components. The type and value of an identifier is determined by this expression:

```
:: State defined and value assigned ::

state x

x <- 4;

:: component declaration ::

component bar << foo <<0, 0>> >>
```

Binary Operators

Binary operators can be used with variables and constants to create complex expressions. A binary operator is of the form:

```
expression binary-operator expression
```

Arithmetic operators

Arithmetic operators include multiplication (*), division (/), addition (+), and sub-traction (-). The operands to an arithmetic operator must be integers.

Relational operators

Relational operators include less than (<), less than or equal to (<=), greater than (>), greater than or equal to (>=), equal to (==), and not equal to (!=).The operands to a relational operator must be integers. The type of a relational operator expression is a boolean and the value is true if the relation is true. For example, the less than operator has a value of true if the left operand is less than the right operand.

Component development

Component development is an expression whose type is Component and whose value is a reference to the newly created Component. Because Components in ADL++ are the first class objects, they can be declared anywhere having an expression would be appropriate. Because of this, Components must be stored in variables to be accessed later in the system. A Component declaration is made clear with the component keyword. Specifying the result of Component is done with a send statement and void Components are not allowed. Parameter declaration is surrounded by parentheses (<< >>) and consists of a list of identifiers separated by commas.

```
:: Component Development ::

component gcd<<a, b>> [

    constraint a !=b satisfy [

        if a > b then a <- a - b;

        otherwise b <- b -a;

    ]

    send a $ ]
```

Component Call

A component call is an expression whose type and value are determined by the send type and value of the component. Calling a component executes the component and blocks program execution until the component is complete. Parameters are expressions that are separated by commas, surrounded by parenthesis and placed after the identifier representing the component. If there are no parameters, the parenthesis (<< >>) are still required for the component call.

```
:: component declaration ::

component boa << e, r >> [ … ]

:: component call::

Boa << 4, 5 >>;
```

3.3.4 Statements

ADL++ has assignment statement, if-then/otherwise logic statements and loops.

Assignment

Assignment statements consist of a modifiable value and an expression. A value is either an identifier, or an object access expression, or a collection access expression that is not a subset. When an assignment statement is executed, the expression is evaluated and the result is assigned to the value <- expression;

Control Flow Statements

- if-then-otherwise statements takes multiple Boolean expressions and then use a statement list to execute them.

- if expression-1 then statement-1-list;
- otherwise statement-2-list;

Iterations Statements

- Constraint-satisfy loop: This statement evaluates an expression before each execution of the body. The expression must be of type boolean, and the value of the expression typically changes in the body of the loop. If the expression is true, the loop body inside [] (square brackets) is executed. If the expression is false, this statement terminates. he while statement has the following syntax:
  - Constraint expression satisfy [expression]
- Repeat loop: This statement evaluates two assignments and one boolean expression, and executes the body until the expression evaluates to false. Following is the syntax, like For loop:
  - Repeat (assignment1-opt ; expression-opt ; assignment2-opt)

Jump Statements

Send: A send statement is specified with the send keyword, followed by an expression and ending with a dollar-sign. Syntax like: send expression $

3.3.5 Scope

Component Scope

Components only have access to the identifiers in their input (parameter) list and identifiers declared within their body. Example of scoping can be described by gcd.adl example as below:

```
:: Program GCD calculation with right scoping convention::

component gcd<<a, b>> [
      constraint a !=b satisfy [
            if a > b then a <- a - b;
            otherwise b <- b -a;
      ] send a $
]
:: Start of the System Program ::
component system<<>>
[
state  i
i <- 9;
view <<i>>;

view << gcd<<2,14>>^"check" >>;
view << gcd<<3,15>> >>;
view << gcd<<99,121>> >>;
]
```

Chapter 4: Project Planning

4.1 Project Timeline

Deadlines	Project Milestones
11 Feb, 2014	Proposal Submission
13 Mar, 2014	Language Reference Manual
27 Mar, 2014	Developed Lexer.ml (lexical analysis completed)
3 Apr, 2014	Ast.ml created along with the Parser.mly version 1.1
10 Apr, 2014	Ast.ml created along with the Parser.mly version 1.9
22 Apr, 2014	Ast.ml created along with the Parser.mly version 2.7
2 May , 2014	Compiler version 1.8 completed with bytecode integrated from microc
12 May, 2014	Run computation programs like gcd and binary search

4.2 Software Development Environment

This project is developed on MS Windows using Cygwin 4.1.10. Makefiles are created in every source directory. And tests are developed to check the functionality of the language.

Chapter 5: Architectural Design

Steps can be divided into following sections:

- Lexing
- Parsing and AST Creation
- Evaluation and Compilation
- Toplevel integeration

5.1 Lexing

The ADL++ lexer tokenizes the input into ADL++ readable units. This process involves discarding whitespace and comments. Illegal character combinations, such as malformed escape sequences, are caught in this phase. The scanner was written with ocamllex.

5.2 Parsing and Abstract Syntax Tree

The parser generates an abstract syntax tree (AST) from the tokens provided by the scanner. Syntax errors are found here. The scanner was written with ocamlyacc. The AST describes the statements and their associated expressions.

5.3 Evaluation, Compilation and Code Generation

The analyzer walks the abstract syntax tree produced by the parser, generates a typesafe, semantically checked abstract syntax tree. The semantic checking portion checks for other errors, such as scope errors, and the reassignment of special functions. This module walks the AST and generates assembly code corresponding to the program.

5.4 Toplevel Integeration

This phase involves Makefile creation and top-level integration. After this compiler package was capable of running .adl files.

Chapter 6: Test Plan

The test suite include two programs, one small and other significant to show the capability of ADL ++ language

How ADL++ can do a computation like calculating GCD:

6.1 GCD.adl

```
:: Program written By Alan Khara::

component gcd<<u, v>> [
      constraint u !=v satisfy [
            if u > v then u <- u – v; otherwise v <- v -u;  ]
      send a $
]
:: Start of the System Program ::

component system<<>>
[
view << gcd<<2,14>> >>;
view << gcd<<3,15>> >>;
view << gcd<<99,121>> >>;
]
```

Output on Cygwin :-

```
Administrator@LAPTOP-SPARE08
$ ./adl < gcd.adl
2
3
11
```

6.2 Boiler.adl

```
:: Program written By Alan Khara::

component pressure_sensor << pressure, transfered_heat>>[
state alert
alert <- 100;

    if transfered_heat*4 < alert then send pressure $ ;
    otherwise pressure <-0; send pressue $
]

component boiler<<burner_heat, glass_capacity, sys_value>> [
state wear_tear_factor
      wear_tear_factor <- sys_value;
```

```
        constraint burner_heat >= glass_capacity satisfy [
         burner_heat <- heat - 1;
         wear_tear_factor <- wear_tear_factor + 1;
    ]

        if burner_heat > 100 then send burner_heat/2;
        otherwise send burner_heat $
]

component burner << knob_value, heat_generate >> [
state Quality_Mark


        if knob_value > 10 then heat_generate <- knob_value*10;
        otherwise heat_generate <- knob_value * 5 ;

        Quality_Mark <- heat_generate /10;

        send Quality_Mark $
]

:: Start of the System Program ::

component system<<>>
[
view << model1<< >> >>;
view << model2<< >> >>;
]

component model1 <<>>[
state quality
          quality <- pressure_sensor << burner<< 10, 60>> , <<boiler <<80, 50, 10>>>>
        send quality $


]

component model2 <<>>[
state quality
          quality <- pressure_sensor << burner<< 20, 50>> , <<boiler <<70, 20, 10>>>>
        send quality $

]
```