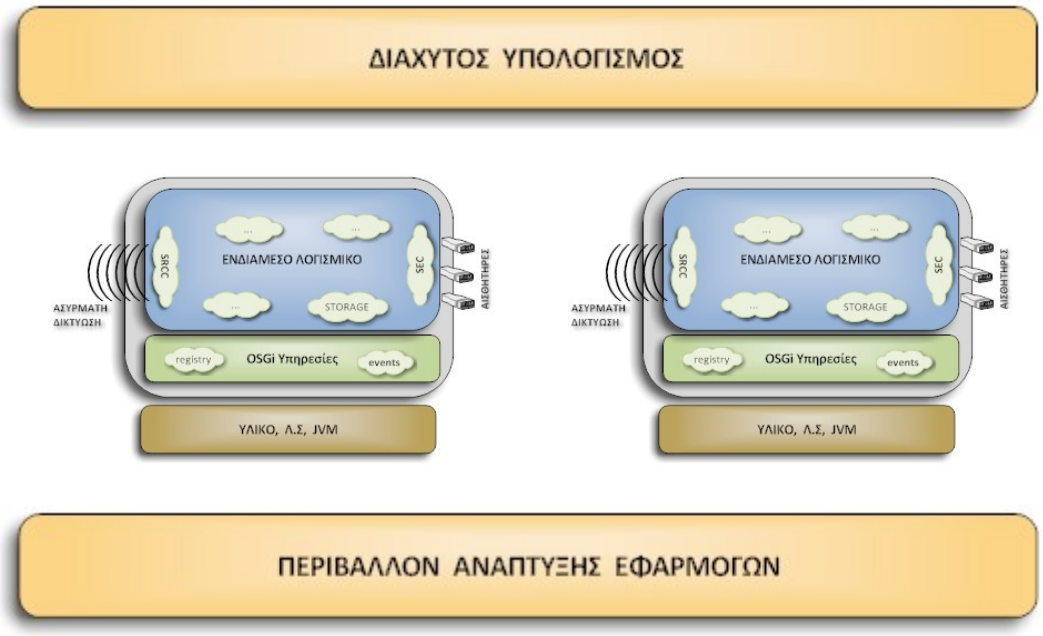

Documentation of the Application Description Language

TECHNICAL REPORT

version 1.5

Vangelis Nomikos, and Kostas Kolomvatsos

NKUA TEAM

September 2009

Contents

1. Introduction

This document is the updated version of the first release of the Documentation of the IPAC Application Description Language [2] and it is an attempt to describe the syntax and some implementation details of the Application Description Language (ADL). The ADL consists of a set of 42 EBNF-like rules describing structures and elements that the developer will be able to use in order to define a new IPAC Application.

Using the openArchitectureWare framework [1], and more specifically the Xtext framework, we take an auto generated parser and an Ecore meta-model for the definition of the ADL. These components are necessary for the correct definition of an IPAC application and the editors syntactic checking mechanism.

Moreover, some extensions are used to enrich our language and the derived meta-model with useful functionalities. For example, we have defined a set of Java classes that provide us methods used for checking purposes or in the content assist component. Just to note one, the method getServices() is used in order to have access in the service model and accordingly to retrieve all the available service names and methods.

In the following sections, we try to present the syntax of all the available elements of the ADL and give some practical examples.

2. Fundamentals

2.1 Identifiers

Rules for using identifiers are almost the same as in any common programming language. A valid identifier is a sequence of one or more characters including underscores or digits. However, identifiers should not start with a digit or to be a reserved word as well as an operator. Table 1 presents the reserved words of the ADL. These words have special meaning and they can not be used for other purposes, for example, as variable names.

In an IPAC application, identifiers can be used in declarations as well as in expressions. Also, they can be used for the definition of the application name, the names of entry points or they can represent service names and methods. In the definition of ADL, identifiers are matched with the predefined “ID” type of the XText framework. Hence, the appropriate syntax for identifiers is:

identifier : (‘a-zA-Z_’ (‘a-zA-Z_’0-9*);

where the symbol * denotes a list of zero or more appropriate characters. Some valid examples could be: variable1, vector4load, num_of_messages, _service2, etc.

Table 1. The reserved words of the ADL.

application	invoke
as	listen
blocked	method
body	nonBlocked
boolean	onFault
break	property
case	set
const	string
default	switch
double	true
else	value
entry	var
event	vector
false	wait
if	while
int	

2.2 Data Types

Data types are used to define the type and the range of values that are acceptable for variables. ADL provides support for four primitive data types:

Table 2. ADL primitive data types.

Primitive Data Types	Usage
int	for integer values
double	for real values
String	for a sequence of characters
boolean	corresponds to **true** or **false** values

The rule used in the ADL for the definition of data types is:

Enum *TypeName :*
*"**int**" | "**double**" | "**String**" | "**boolean**";*

At the current status, ADL does not support simple char values, short or long integers and float values. However, the supported primitive data types can be easily extended by defining their names in the above described ADL rule.

For the manipulation of a collection of data, the ADL supports the *"**vector**"* data type. Vectors are used to store multiple data values of primitive data types and all these values should be of the same type. This complex data type is used especially in ***invoke*** statements (see Section 3.5). When methods of middleware services are called by the application, vectors can be used as parameters. Moreover, if the method returns a collection of values these can be stored in a vector variable of the same type. Assignment statements (see Section 3.1) are also allowed between vectors (i.e. store the values of a vector to another vector).

ATTENTION: There are no means provided by the ADL for the manipulation of specific values stored in a vector. This is going to happen through specific utility services defined for such reasons.

2.3 Operators

An operator takes two operands and combines them in order to result a new value. The ADL operators are divided into five categories: logical, equality, relational, additive and multiplicative operators. A logical operator produces a boolean value (true or false) based on the logical relationship of its arguments. Equality and relational operators evaluate the relationship between the values of the operands and generate a boolean result. Equivalence and nonequivalence work with all primitives data types and are used for the creation of logical expressions. A relational operator is used to compare the values of the operands given in the specific expression producing a boolean result. Expressions are fully described in Section 2.4. Additive and multiplicative operators are used to build mathematical expressions which result a specific value. Table 3 presents the operators that belong to each category.

Table 3. Categories of Operators.

Category	Operator	Meaning
Logical Operators	and	Logical conjunction
	or	Logical disjunction
Equality Operators	==	Equal with
	!=	Not equal with
Relational Operators	>	Greater than
	>=	Greater or equal than
	<	Less than
	<=	Less or equal than
Additive Operators	+	Used for addition
	-	Used for subtraction
Multiplicative Operators	*	Used for multiplication
	/	Used for division
	%	Produces the remainder from integer division (modulo)

ATTENTION: The operators “>”, “>=”, “<”, “<=” cannot be applied for boolean values.

It should be noted that all the above described operators are binary. The ADL allows the definition of negative values using the minus operator (“-“) in front of a constant value. Examples concerning the usage of operators are given in the next section of this report. An important point is that the ADL does not support the logical negation operator (“*not*”). The developer should be aware of this issue in order to create expressions not containing logical negation.

In the Xtext model, rules defining operators use enumerations. These rules are:

***Enum** EqualityOperator: "==" | "!=";*

***Enum** RelationalOperator: "<=" | ">=" | "<" | ">";*

***Enum** AdditiveOperator: "+" | "-";*

***Enum** MultiplicativeOperator: "*" | "/" | "%";*

2.4 Expressions

Expressions are used for the combination of elements representing specific values. They result a specific result which in the most of the cases is assigned to variables or is used in other more complex expressions. In general, a typical form of an expression is:

<operand> <operator> <operand>

As mentioned before, we consider only binary operators. The expressions supported by the ADL are as complex as the developer wants, however, without the usage of parentheses. Their correctness is based on the order of operations. Operator precedence defines how an expression is evaluated when several operators are present. ADL has specific rules that determine the order of the evaluation. For example, for mathematical operators, multiplication and division are evaluated before addition and subtraction.

In order to define hierarchy in the operations in an expression we define the following types of expressions:

- A generic "*Expression*" could be a "*ConditionalExpression*" followed by zero or more pairs of the logical operator "*or*" and another "*ConditionalExpression*".
  Syntax : Expression:
  ConditionalExpression ("**or**" ConditionalExpression)*

- A "*ConditionalExpression*" could be an "*EqualityExpression*" followed by zero or more pairs of the logical operator "*and*" and another "*EqualityExpression*".
  Syntax : ConditionalExpression:
  EqualityExpression ("**and**" EqualityExpression)*

- An "*EqualityExpression*" could be a "*RelationalExpression*" followed by at most one pair of an "*EqualityOperator*" (see Table 3) and another "*RelationalExpresion*". Notice that an EqualityExpression consists of at most two relational expressions.
  Syntax: EqualityExpression:
  RelationalExpression (EqualityOperator RelationalExpression) ?

- A "*RelationalExpression*" could be an "*AdditiveExpression*" followed by zero or more pairs of a "*RelationalOperator*" (see Table 3) and another "*AdditiveExpression*".
  ***Syntax*:** RelationalExpression:

AdditiveExpression (RelationalOperator AdditiveExpression) *

- An "*AdditiveExpression*" could be a "*MultiplicativeExpression*" followed by zero or more pairs of an "*AdditiveOperator*" (see Table 3) and another "*MultiplicativeExpression*".
  Syntax: AdditiveExpression:
  MultiplicativeExpression (AdditiveOperator MultiplicativeExpression) *

- Finally, a "*MultiplicativeExpression*" could be a "*PrimitiveElement*" followed by zero or more pairs of a "*MultiplicativeOperator*" (see Table 3) and another "*PrimitiveElement*".
  Syntax: MultiplicativeExpression :
  PrimitiveElement (MultiplicativeOperator PrimitiveElement) *

Some expression examples follow:
Example 1: *The value of the variable "temperature" is less or equal than 35.0.*
temperature **<=** 35.0

Example 2: *The value of the variable "temperature" is greater than -20.0.*
temperature **>** -20.0

Example 3: *The value of the variable "humidity" not equal to 23.4.*
humidity **!=** 23.4

Example 4: *The result of the multiplication of the value of the variable "temperature" and the variable "error" is greater than the value of the constant "THRESHOLD".*
temperature ***** error **>=** THRESHOLD

Example 5: *The value of "temperature" is less than or equal of 35.0 and the value of "humidity" greater than 65.234.*
temperature **<=** 35.0 **and** humidity **>=** 65.234

Example 6: *Evaluate the expression 5 and check if the variable "isCritical" is equal to true. Then evaluate the disjunction of these expressions.*
temperature **<=** 35.0 **and** humidity **>=** 65.234 **or** isCritical **== true**

ATTENTION: The precedence of the operators in an expression is the precedence utilized by the Java programming language.

Example 7: *Multiply the "oldTemperature" with 0.2 and the "newTemperature" with 0.8 and then add the results.*
oldTemperature ***** 0.2 **+** newTemperature ***** 0.8

Example 8*: Evaluate the expression in 7 and check if the result is less than or equal to the value of the "threshold" multiplied by 1.20.*
oldTemperature ***** 0.2 **+** newTemperature ***** 0.8 **<=** threshold ***** 1.20

Example 9: *Evaluate the following mathematical expression. Multiplications go first and then additions.*
tWeight * temperature + hWeight * humidity + wWeight * windSpeed

ATTENTION: However, the following cannot be represented using the ADL:
*0.8 * (temperature * tWeight + humidity * hWeight)*
The equivalent in ADL could be
*0.8 * temperature * tWeight + 0.8 * humidity + 0.8 * hWeight*

2.5 Declarations

In the declaration part of each application (see Section 4) the developer can define variables, constants and vectors. Constants cannot change value during the execution of the application and their definition has the following form:

Syntax:
const <constantName> **as** <type> **value** <constantValue>

Hence, the declaration of a constant requires a valid name, a data type (as they defined in the Section 2.2) and a value. The type of the "*constantValue*" should match with the "*type*" defined in the declaration. For this, a specific checking mechanism is defined. It should be noted that every time the developer defines a constant the checking mechanism ensures that the constant is not previously defined. In such cases the line containing the declaration is underscored and an error message appears in the screen when the developer places the mouse on the declaration line. Some examples of constant declarations follow:

Example 10: *Define a boolean constant with name "isDeployed" and value equal to true.*
const isDeployed **as boolean value true**

Example 11: *Define a string constant with name "serviceName" and value equal to "something".*
const serviceName **as String value** "something"

Example 12: *Define an integer constant with name "numOfSensors" and value equal to 64.*
const numOfSensors **as int value** 64

Example 13: *Define a double constant with name "temperatureThreshold" and value equal to 41.223.*
const temperatureThreshold **as double value** 41.223

The second case of declarations involves the variable declarations. The general form of a variable declaration is:

Syntax:

var <variableName> **as** <type> [**value** <constantValue>]

Hence, we can define the name and the type of a variable and optionally an initial value could be assign. In contrast to the most programming languages, multiple variables cannot be defined in the same line. The developer should devote one declaration line for each variable. Some examples of variable declarations are presented below:

Example 14: *Declare a double variable with name "threshold".*
var threshold **as double**

Example 15: *Declare a string variable with name "message".*
var message **as String**

Example 16: *Declare an integer variable with name "numOfIterations" and initialize it to the value 10.*
var numOfIterations **as int value** 10

Example 17: *Declare a double variable with name "fireProbability" and initialize it to the value 1.0.*
var fireProbability **as double value** 1.0

Example 18: *Declare a boolean variable with name "canBeDeployed" and initialize it to the value true.*
var canBeDeployed **as boolean value true**

COMMENT: When an *integer* or *double* variable is declared without initial value, the value zero (0) is assigned by default.

Concerning the checking mechanism used for this part of declarations involve the name checking and the type of the value if the optional part is used.

Finally, the only available complex type, that the developer can use, is the vector declaration. A vector is used for storing a collection of primitives data types of the same type. The general form of the vector declaration is:

Syntax:

vector <vectorName> **as** <type>

Some examples of vector declarations are presented below:

Example 19: *Declare a vector variable of double elements with name "temperatures".*
vector temperatures **as double**

Example 20: *Declare a vector variable of string elements with name "sensorsNames".*

vector sensorsNames **as String**

Concerning multiple vector declarations it stands the same as for variables. Every vector is also declared in a separate line.

3. Control Statements

Before defining an IPAC Application, the developer must have a thorough understanding of the problem and a carefully planned approach to solving it. When writing the application, he also must understand the elements and structures are available and employ proven program-construction techniques. In this section, we present the basic control structures that the ADL supports. The most common statements are assignment statements, selection statements, iteration statements, and invoke statements. A more detailed analysis follows in the following Sections.

3.1 Assignment Statement

Assignment is performed with the operator = . It means "Take the value of the right-hand side (often called the rvalue) and copy it into the left-hand side (often called the lvalue)." An rvalue is any constant, variable, expression or the result of an invoke statement. An lvalue must be a distinct, named variable or vector. The general syntax of an assignment statement is:

Syntax:
set <variable> **=** < Constant | Variable | Expression | Invoke Statement>

where variable is the name of a predefined simple or vector variable.

ATTENTION: The type of the left and the right side of the assignment statement must be the same!

Some examples of assignment statements are presented below:

Example 21: *Assign to the variable "query" the value "This is a dummy query".*
set query **=** "This is a dummy query"

Example 22: *Assign to the variable "temperature" the value 35.6*
set temperature **=** 35.6

Example 23: *Assign to the variable "temperature" the value of the constant "THRESHOLD".*
set temperature **=** THRESHOLD

Example 24: *Assign to the variable "temperature" the result of the expression "temperature * 1.2 + 3.0".*

```
set temperature = temperature * 1.2 + 3.0
```

3.2 Selection Statements

An IPAC application uses selection statements in order to choose among alternative courses of action. ADL supports two types of selection statements. The if…else statement (single-selection) and the switch statement (multiple-selection). Note that the single selection statement can also be used for multiple-selection purposes through nested if … else statements.

3.2.1 Single-Selection Statement (if…else)

The if…else statement is the most common way to control program flow. This kind of statement allows the developer to specify an action to perform when the condition is true and a different action when the condition is false. The general form of such a statement is:

Syntax:

```
if (expression) {
        statements
}
[else {
        statements
}]
```

The expression must result a boolean value. The statements list should contain zero or more statements. Also, note that the else part of the statement is optional. If an else part is missing an indicated action is performed only when the condition is true; otherwise, the action is skipped.

Some examples of single selection statements are presented below:

Example 25: *Check if the value of the variable "temperature" is greater or equal than the value of "threshold" and if this is true assign to the variable "alerted" the value true (assume that the variable alerted is of boolean type).*

```
if (temperature >= threshold) {
        set alerted = true
}
```

ATTENTION: The braces are mandatory even if only one statement follows!

Example 26: *Check if the variable "isCritical" is equal to true. Then assign to the variable "probability" the value 1.0 otherwise assign the value 0.2.*

```
if (isCritical == true) {
        set probability = 1.0
}
```

```
else {
        set probability = 0.2
}
```

3.2.2 Multiple-Selection Statement (switch)

The switch statement allows the developer to specify an action to perform under multiple conditions. The syntax of a switch statement is:

Syntax:

```
switch (expression) {
        caseStatements
        [defaultStatement]
    }
```

The expression should be evaluated to integer. The default statement is optional. The syntax of a case statement is:

Syntax:

```
case intValue:
        statements
```

ATTENTION: At the current status, in a switch statement only integer values are allowed!

Some examples of the switch statement are presented below:

Example 27: *Check the value of the variable "criticality" and set the appropriate value to the variable named probability.*

```
switch (criticality) {
        case 0:
                set probability = 0.1
                break
        case 1:
                set probability = 0.5
                break
        case 2:
                set probability = 1
                break
}
```

Notice that each case ends with a break statement (section 3.4), which causes execution to jump to the end of the switch body. This is the conventional way to build a switch statement, but the break is optional. If it is missing, the code for the following case statements executes until a break is encountered.

Also, notice that the developer can succeed the desired functionality of the above example with an alternative approach by using nested if…else statements. This approach is described below for the Example 27:

```
if (criticality == 2) {
        set probability = 1.0
}
else {
        if (criticality == 1) {
                set probability = 0.5
        }
        else {
                set probability = 0.1
        }
}
```

3.3 Iteration Statements

An iteration repeats the execution of a number of statements until the controlling expression is evaluated to false. The expression is evaluated once at the beginning of the loop and every time an iteration is finished. The ADL supports only one type of iteration statement, the while-statement.

3.3.1 While Statement

The while statement allows the developer to specify that an application should repeat some actions while the condition remains true. The basic syntax of such a statement is:

Syntax:

```
        while (expression) {
                statements
         }
```

Some examples of while statements are presented below:

Example 28: *Assume that the variable "temperature" has been initialized to the value 10. Increase the temperature per 2.5 in each iteration until it exceeds the threshold of 40.*

```
while (temperature <= 40) {
        set temperature = temperature + 2.5
}
```

Note that, a break statement could be used in a while statement in order to stop execution.

ATTENTION: We do not support the statement "for" the "while" statement can be

used in cases where the number of iterations is predefined.

3.4 Break Statement

The break statement could be used only in an iteration or a switch statement. A break statement is used to alter the flow of control. When executed, it causes immediate exit from that statement. Execution continues from the first statement after the control statement. For a break statement, the developer could simply use the reserved word "break".

3.5 Invoke Statement

The IPAC middleware consists of a set of services each of them providing a set of public methods. At the definition of a newly created IPAC application, the developer is able to use these services and methods in order to give to the application the desired functionality. The developer gains access to these elements by using invocation statements. The basic syntax of an invocation statement is:

Syntax:

```
invoke ServiceName {
        method MethodName (parameterList)
        [ onFault { statements } ]
}
```

The onFault structure is optional and it is executed when an error occurs in the invoke statement.

Some examples of invoke statements are presented below:

Example 29: *Let's invoke the method "getNumOfSensors" of the "StorageService".*

```
invoke StorageService {
        method getNumOfSensors ()
}
```

Example 30: *Retrieve the top-5 temperatures from the "StorageService".*

```
invoke StorageService {
        method getTopKTemperatures (5)
}
```

Example 31: *Invoke the method "getNumOfSensors ()" of the "StorageService" and if an error occurs invoke the "soundAlert()" method of the "UIService".*

```
invoke StorageService {
        method getNumOfSensors ()
        onFault {
                invoke UIService {
                        method soundAlert()
                }
```

```
        }
}
```

We can also assign to a variable the result of an invocation statement.

Example 32: *Assign to the variable "temperature" the return value of the "getTemperature" method of the "StorageService".*

```
set temperature = invoke StorageService {
                        method getTemperature()
                    }
```

Example 33: *Assign to the vector variable "resultSet" the return value of the invocation of the "query" method of the "ReasonerService".*

```
set resultSet = invoke ReasonerService {
                        method query ("Here goes a query")
                    }
```

ATTENTION: The type of the variable and the return type of the value of the specified method must be the same!

ATTENTION: A list with all the available services and their corresponding methods which they will be available to the developer.

3.6 Listen Statement (DEPRECATED!!! – DO NOT USE IT!!!)

A listen statement allows the developer to specify what events an application is able to handle and accordingly what actions are going to be performed when the event is fired. The syntax of a listen statement is:

Syntax:

```
listen <eventName> <eventType>
```

The "eventName" specifies one of the available events which have been defined in the application profile by using the application profile editor. The "eventType" specifies the type of the event; "blocked" or "nonblocked". If a listen statement uses the "blocked" type, the execution of the IPAC application will be suspended until the event is fired, implementing a synchronous event handling. Otherwise, when the type of the event is "nonBlocked", the execution of the IPAC application will continue until the event is fired (asynchronous event handling).

ATTENTION: For each listen statement the appropriate EventHandler block must be defined before the body part. An event entry point contains the actions to be performed when an event is fired.

Some examples of listen statements are presented below:

Example 34: *The application should be able to handle "fireEvents". It should suspend its execution until this event will be fired.*
listen fireEvent **blocked**

Example 35: *The application should be able to handle "fireEvents". It should continue its execution and when this event is fired the event handler will be executed.*
listen fireEvent **nonBlocked**

Comment: *However, event handling is going to be finalized in the next realease of this report.*

3.7 Wait Statement

A wait statement allows the developer to specify a time interval for which the application suspends its execution. The syntax of such a statement is:

Syntax:
wait <numOfSeconds>

Example 36: *The application must suspend its execution for 5 seconds.*
wait 5

4. Building an Application

In this Section, we explain the basic architecture of a newly created IPAC application. The basic template of an application is listed in Listing 1.

```
application name {
        Declarations' Definition Part

        EventHanders' Definition Part

        EntryPoints' Definition Part

        body {

                // Here goes the application logic.
        }
}
```

Listing 1: The basic architecture of an IPAC application.

Each IPAC application consists of four distinct parts: the declaration, the event handling, the entrypoints' definition and the body part. After the definition of the application name, the developer has the opportunity to define the constants and variables that the application is going to use. After the declarations, the EventHandlers'

part follows. In this part, the developer specifies a set of statements that will be executed when a specific event is fired. Moreover, a set of properties can be defined in order to store values taken by events. For each application specific event the corresponding event handler must be defined. When an event is fired, the appropriate event handler is executed. As far as the entry points is concerned, the definition of them is not fully supported until now but will be used in a future version of the language. It should be noted that all the aforementioned parts are optional and should be used only when they are necessary.

Comment: The definition of entry points is, for now, similar to the definition of methods to common programming languages.

The basic syntax of an event handler is described below:

Syntax:

```
event name {
        eventAssignmentStatements
        statements
}
```

As depicted above, a list of eventAssignmentStatements are used in the definition of an event handler. Some events (especially UI events) carry specific information that should be manipulated by the application. Such a statement is used for this purpose. The basic syntax of an EventAssignmentStatement follows:

Syntax:

```
property variableReference value propertyID
```

Note that the field of "propertyID" takes values of a predefined list of properties which is related to the corresponding event. These values are available through the content-assist mechanism of the ACE editors.

An example of the definition of an event handler follows:

Example 37: *When a fire event is fired, invoke the "soundAlert" method of the "UIService".*

```
event fireEvent {
        invoke UIService {
                method SounAlert()
        }
}
```

After the definition of entry points, the developer can proceed to the creation of the core application logic. All application statements are defined in the "body" part and they implement the desired functionality. Details of all the available statements are described in Section 3. As we can see in the Listing 1, braces are used to define the

beginning and the end of a body part. The developer could use tabs in order to justify the application code and this way to be easier the optical control of the workflow. Finally, we should note that comments in the application workflow can be placed using the symbols "//" for one line comment and the symbols "/*" and "*/" for multiple line comments.

4.1 Example Application

This is a simple example of an IPAC application. The logic of this application simply shows how the developer can use some of the ADL elements described in this report.

```
application DummyApp_5643 {
        var threshold as int value 0
        vector resultSet as double

        event FireEvent {
                invoke uiService {
                        method displayAlert(0, "Fire detected")
                }
        }

        body {

                while ( threshold < 5) {
                        set resultSet = invoke reasonerService {
                                method query ( "A query 1")
                        }

                        wait 5

                        set threshold = threshold + 1
                }
        }
}
```

APPENDIX A – ADL Definition

This appendix contains the definition of the ADL in terms of EBNF-like rules. At the current status, the ADL is defined by a set of such rules.

```
Application:
      kApplication="application" appName=ID "{"
            (declarations+=Declaration)*
            (eventHandlers+=EventHandler)*
          (entryPoints+=EntryPoint)*
            body=Body
      "}";

// Specifies the code to be executed for an entry or an event.
EventHandler:
      "event" name=ID "{"
            (eventAssignments+=EventAssignmentStatement)*
            (eventStatements+=Statement)*
      "}";

EventAssignmentStatement:
      "property" assRef=[VariableDeclaration] "value" handler-
Ref=[EventHandler] "." propId=ID;

EntryPoint:
      "entry" name=ID "{"
            (entryStatements+=Statement)*
      "}";

Body:
      "body" "{"
            (statements+=Statement)*
      "}";

// ********************** //
// ***** DECLARATIONS ***** //
// ********************** //
// Abstract rule. Useful for linking references!
Declaration:
      ConstantDeclaration | VariableDeclaration | VectorDeclaration;

ConstantDeclaration:
      "const" name=ID "as" type=TypeName "value"
cValue=ConstantValue;

VariableDeclaration :
      "var" name=ID "as" type=TypeName ("value"
cValue=ConstantValue)?;

VectorDeclaration :
      "vector" name=ID "as" type=TypeName;

Enum TypeName :
      int="int" | double="double" | string="String" | boo-
lean="boolean";

// ***** CONSTANT VALUES ***** //
ConstantValue:
```

```
       IntegerLiteral | StringLiteral | DoubleLiteral | BooleanLit-
eral;

IntegerLiteral:
      intValue=INT;

StringLiteral:
      strValue=STRING;

BooleanLiteral:
      booleanValue=BooleanEnum;

Enum BooleanEnum:
      true="true" | false="false";

DoubleLiteral:
      doubleValue=DOUBLE;

// A native rule is a lexer rule!
Native DOUBLE:
      "('0'..'9')*'.'('0'..'9')+";

// Native SL_COMMENT:

// ********************* //
// ***** STATEMENTS ***** //
// ********************* //
Statement:
      AssignmentStatement | IfStatement | SwitchStatement | In-
vokeStatement | ListenStatement | WaitStatement | IterationStatement
| BreakStatement;

IfStatement :
           "if" "(" ifExpression=Expression ")" "{"
                 (ifStatements+=Statement)*
            "}"
           ("else" "{"
                 (elseStatements+=Statement)*
            "}")?;

SwitchStatement :
           "switch" "(" switchExpression=Expression ")" "{"
                 (caseStatements+=CaseStatement)*
                 (defaultStatement=DefaultStatement)?
                 "}";

// ATTENTION: Only INT values in CaseStatements!
CaseStatement:
      "case" cValue=INT ":" (caseStatements+=Statement)*;

DefaultStatement:
      "default:"  (defaultStatements+=Statement)*;

ListenStatement :
      "listen" eventName=[EventHandler] eventType=EventType;

Enum EventType :
      blocked="blocked" | nonBlocked="nonBlocked";

// ATTENTION: Java style! Space seperator at the parameter list!
InvokeStatement:
```

```
      "invoke" componentName=ID "{"
                  "method" methodName=ID
                        primitiveEL=PrimitiveElementList
                        (faultStatement=FaultStatement)?
      "}";

PrimitiveElementList :
      "(" (primitiveElements+=PrimitiveElement ("," primitiveEle-
ments+=PrimitiveElement)*)?  ")"
;

// Like try-catch block. The code to be executed.
FaultStatement :
      "onFault" "{" (fStatements+=Statement)* "}";

// ATTENTION: A Declaration reference may be a constant declaration.
Check is needed!!!
// Also a PrimitiveElement is allowed as right operand. Checks needed
(possible?) or sth more simplified.
AssignmentStatement:
       "set" assRef=[Declaration] "=" (invokeState-
ment=InvokeStatement | expression=Expression);
// primitiveElement=PrimitiveElement deleted as right operand

// e.g set v1 to 47 ???

IterationStatement:
      "while" "(" whileExpression=Expression ")" "{"
            (whileStatements+=Statement)*
      "}";

BreakStatement:
      "break";

WaitStatement:
      "wait" secs=INT;

// ***************************** //
// ***** PRIMITIVE ELEMENTS ***** //
// ***************************** //
// ATTENTION: Usage of the abstract rule! Vector reference is al-
lowed!
PrimitiveElement:
       peRef=[Declaration] | cValue=ConstantValue;
// (cValue=ConstantValue | varValue=VariableReference |
constValue=ConstantReference | vectorValue=VectorReference);

//VectorReference:
//    indexedVecRefName=IndexedVectorReference | vecRef-
Name2=[VectorDeclaration];

//IndexedVectorReference:
//    vecRefName=[VectorDeclaration] "(" index=INT ")";


// ********************** //
// ***** EXPRESSIONS ***** //
// ********************** //
Expression:
      condAndExpr=ConditionalAndExpression ("or" optCondAn-
dExpr+=ConditionalAndExpression)*;
```

```
ConditionalAndExpression:
      equalExpr=EqualityExpression ("and" optE-
qualExpr+=EqualityExpression)*;

EqualityExpression:
      relExpr=RelationalExpression (equalOps+=EqualityOperator
optRelExprs+=RelationalExpression)?;

RelationalExpression:
      addExpr=AdditiveExpression (relOps+=RelationalOperator optAd-
dExprs+=AdditiveExpression)?;

AdditiveExpression:
      multiExpr=MultiplicativeExpression (addOps+=AdditiveOperator
optMultiExprs+=MultiplicativeExpression)*;

// ATTENTION: PrmitiveElement or sth else more limited?
MultiplicativeExpression:
      primElem=PrimitiveElement (multOps+=MultiplicativeOperator
optPrimElems+=PrimitiveElement)*;

Enum EqualityOperator:
      equalOp="==" | unEqualOp="!=";

Enum RelationalOperator:
      leOp="<=" | geOp=">=" | lOp="<" | gOp=">";

Enum AdditiveOperator:
      addOp="+" | minusOp="-";

Enum MultiplicativeOperator:
      multOp="*" | divOp="/" | modOp="%";
```

APPENDIX B – Getting started with ACC

In order to start using the IPAC application creation component you have to follow the following steps:

1. Visit the url http://oaw.itemis.com/openarchitectureware/language=en/2837/downloads and at the section "Eclipse 3.4.2 Ganymede + openArchitectureWare 4.3.1" download the environment for the platform of your choice.
2. Unzip the file with the distribution to a destination folder. This folder will be the installation folder of your oAW Eclipse environment.
3. Download from the SVN of the IPAC project (http://ipad.di.uoa.gr/svnrepo/ipac) the five deployable plug-ins and all the necessary external files. All the above are located under the SVN folder ACC/ NKUA_ACC_v1/PLUG-INS.
4. Go to the installation folder of your oAW Eclipse environment, and put the five deployable plug-ins into the plugins folder.
5. Copy the folder IPAC_ROOT at the hard-disk C: of your computer.
6. Start the oAW Eclipse environment (if it is already open, just restart it).

ATTENTION: At the current status of the development, the oAW Eclipse environment and all the extended features (plug-ins, external files) have been tested only on Microsoft Windows environments (XP and Vista). Support for other OS will be a next step.

In order to define a new IPAC project, the developer should follow the following steps:

7. Choose from the main menu of the environment the options *"File → New → Project → Xtext DSL Wizards → ipacadl Project"* and then click the "*Next*" button.
8. Specify the project name and press the finish button.
9. At the left part of the eclipse workbench, all the necessary files are created. Double click, the file "*model.adl*" under the folder src.
10. The file "*model.adl*" opens with the textual editor and the developer is ready to start writing the application.
11. At any time, the developer can use the auto-completion functionality of the editor by pressing the combination of keys Ctrl+Spacebar.

A more detailed guide describing the functionality of ACC will be available at the next release of this report.

References

[1] http://www.openarchitectureware.org/

[2] V. Nomikos and K. Kolomvatsos, "Documentation of the IPAC Application Description Language", Technical Report, version 1.0, Department of Informatics and Telecommunications, University of Athens, May 2009.