STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LIII**, Number 2, 2008

EXTENSION OF AN OCL-BASED EXECUTABLE UML COMPONENTS ACTION LANGUAGE

S. MOTOGNA, B. PÂRV, I. LAZĂR, I. CZIBULA, L. LAZĂR

Abstract. Executable models allow precise description of software systems at a higher level of abstraction and independently of a platform or a programming language. In this paper we explore the use of a Procedural Action Language based on OCL to specify executable UML components and we propose an extension that will include array types and corresponding operations.

1. Introduction

Model Driven Architecture (MDA) development represent a pertinent solution to design and control of large software systems, while UML establihed itself as a standard for software models. UML2 and its Action Semantics [6] provide the foundation to construct executable models. In order to make a model executable, the model must contain a complete and precise behavior description. But, creating a model that has a complete and precise behavior description is a tedious task or an impossible one because of many UML semantic variation points.

We have introduced COMDEVALCO a framework aimed to support definition, validation, and composition of software components, that allows the construction and execution of UML structured activities [3]. The framework refers only to UML structured activities because our first objective was to allow model transformation from PIM (Platform Independent Model) to procedural constructs of imperative languages. It includes a modeling language, a component repository and a set of tools. The object-oriented modeling language contains finegrained constructions, aimed to give a precise description of software components. Component repository is storing valid components,

Received by the editors: October 12, 2008.

2000 *Mathematics Subject Classification.* 68N30.

1998 *CR Categories and Descriptors.* D.2.4 [**SOFTWARE ENGINEERING**]: Software/Program Verification – *Formal methods, Model checking, Validation*; D.2.13 [**SOFTWARE ENGINEERING**]: Reusable Software – *Reuse models*; I.6.5 [**SIMULATION AND MODELING**]: Model Development – *Modeling methodologies* .

Key words and phrases. Software components, Executable models, OCL.

ready to be composed in order to build more complex components or systems. The toolset contains tools dedicated to component definition, validation, and composition, as well as the management of component repository.

Our approach uses Procedural Action Language (PAL) that is a concrete syntax for UML structured activities and defines graphical notations for some UML structured activity actions [7]. PAL simplifies the process of constructing executable models by simplifying the construction of UML structured activities.

The execution of the model is performed without any transformation, and by using this approach the time delay between the model changes and the model execution is minimized.

The repository should contain executable models, ready to be used in any further development. One aspect that can guarantee this principle is to use some extra conditions, such as preconditions, postconditions and invariants to the model definition that would describe in a formal way, the behavior of the model.

Object Constraint Language - OCL - has been extensively used for models of UML [6], representing a well suited specification language for defining constraints and requirements in form of invariants, pre- and post- conditions.

So, we add pre- and post-conditions to the model in the form of OCL expressions. In such a way, we obtain the desired descriptions in terms of OCL expressions, we then could use them in searching queries, and the layout of the repository can be standardized.

The repository will store different types of models, and in the initial phase, we have designed it for simple arithmetical and array problems. The OCL specification [5] doesn't contain array types, which are necessary in our approach. So, we have two options to tackle this problem: to express arrays using the existing constructions or to extend OCL Expressions.

The first approach has two main disadvantages: it restricts the type of the elements of the arrays and array specific operations should be re-written any time they are needed. We would prefer to work with a more generic construction, and do not worry about operations' implementations each time they are used. Array operations are defined once, and then called any time they are needed.

The rest of the paper is organized as follows: the next section presents some related works in the domain and compare them with our approach. Section 3 describes the action language defined as part of ComDeValCo framework and then section 4 presents the extension of PAL with array types and associated operations, and an example of an executable model that benefits from the use of our extension. The next section draws some conclusions and suggests some future development directions.

2. Related work

The xUML [8] process involves the creation of platform independent, executable UML models with the UML diagrams being supported by the action semantics-compliant Action Specification Language (ASL). The resulting models can be independently executed, debugged, viewed and tested. The action semantics extension to UML defines the underlying semantics of Actions, but does not define any particular surface language. The semantics of the ASL are defined but the syntax of the language varies. ComDeValCo is compliant with UML 2.0 and uses structured activities for models [3].

According to several domain experts, a precise Action Semantics Language (ASL) and a specified syntax are required. Unfortunately, actions defined in UML do not have a concrete syntax and OMG does not recommend a specific language, so there is not a standard ASL. Object Constraint Language (OCL) is a formal language used to describe expressions on UML models. The great overlap between ASL and OCL (large parts of the Action Semantics specification duplicates functionality that is already covered by the OCL) suggests that OCL can be used partly for ASL. OCL for Execution (OCL4X) [2] is defined based on OCL to implement operations that have side effects and provide the ability for model execution. By mapping from ASL to OCL, OCL is used to express some actions in ASL. This approach has identified some open problems when using OCL in specification of the executable models, and offered solutions based on extending OCL to include actions with side effects in order to model behavior. Our approach is, in many ways, similar to this one. We are also proposing some extensions of OCL, but based on identifying some other problems and suggesting more efficient approaches of executable model specification.

According to Stefan Haustein and Jorg Pleumann, since the OCL is a subset of the ASL, there are two options for building an action surface language based on OCL [1]: map all OCL constructs to actions, then add new syntax constructs for actions that are required, but not covered, or embed OCL expressions in new syntax constructs for actions.

The first option requires a complete mapping of the abstract OCL syntax to actions. This would mean to give up declarative semantics in OCL, or to have two flavours of OCL with different specifications that would need to be aligned carefully.

The second option can be implemented by referring to the existing OCL surface language, without modifying it, maintaining a clean syntactical separation between plain queries and actions that may influence the system state.

ComDeValCo is oriented on this second approach.

3. Procedural Action Language - description and features

As part of ComDeValCo framework we have defined a procedural action language (PAL), that is a concrete syntax for UML structured activities, and graphical notations for some UML structured activity actions [7].

The framework also includes an Agile MDA approach for constructing, running and testing models. Debugging and testing techniques are also included according to the new released standards. In order to be able to exchange executable models with other tools, a UML profile is also defined. The profile defines the mapping between PAL and UML constructs and is similar to the profile defined for AOP executable models.

In order to develop a program we construct a UML model that contains functional model elements and test case model elements. Functional model elements correspond to the program and its operations and are represented as UML activities. Test case model elements are also UML activities and they represent automated tests written for some selected functional model elements.

The Procedural Action Language (PAL) is introduced to simplify the construction of UML structured activities. PAL defines a concrete syntax for representing UML structured activity nodes for loops, sequences of actions and conditionals. The PAL syntax is also used for writing assignment statements and expressions in structured activity nodes. PAL also includes assertion based constructs that are expressed using OCL expressions.

The syntax of the language is given in Appendix A.

The framework accepts user-defined models described in UML-style or using PAL, validates them according to UML metamodel and construct the abstract syntax tree, which is then used to simulate execution. For each syntactical construction of PAL there exists a rule corresponding to the construction of the abstract syntax tree.

4. Extending PAL with array type

The intention is to store different types of models in the repository, but, in the initial phase, we have considered small models for simple arithmetical problems, and we face the problem of dealing with arrays. As mentioned before PAL uses OCL-based expressions, but the OCL specification language does not allow arrays.

There are two things that should be taken into consideration when designing types for models [9]:

- Languages that manipulate and explore models need to be able to reason about the types of the objects and properties that they are regarding within the models.

- There is also a need to reason about the types of artifacts handled by the transformations, programs, repositories and other model-related services, and to reason about the construction of coherent systems from the services available to us. While it is possible to define the models handled by these services in terms of the types of the objects that they accept, we argue that this is not a natural approach, since these services intuitively accept models as input, and not objects.

At the first attempt, it would have looked simpler to add a new type array that could create arrays with elements of any existing type in the system, but taking a deeper look, creating an array of integers is totally different from creating an array of components. Consequently, we have though at the approach that is also taken in different strongly typed programming language (Java, .NET), and that will guarantee an easy extension of the type system.

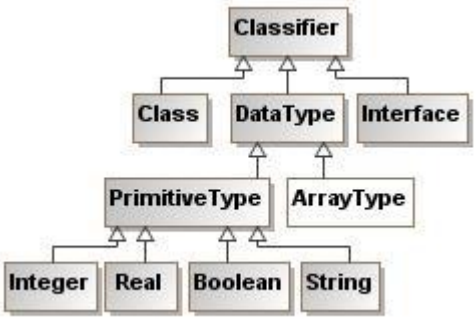

FIGURE 1. Type metaclasses

We have started from the type hierarchy from OCL [5] and refined it to integrate *ArrayType*, as depicted in Figure 1. A *Classifier* may be a *DataType*, a *Class* or an *Interface*. *ArrayType* and *PrimitiveType* are specializations of *DataType*. The most important feature of *DataType* is that a variable of this type can hold a reference to any object, whether it is an integer, a real or an array, or any other type.

We highlight only the modifications of the grammar such that our models will be able to handle arrays and records. Types can be arrays whose elements can be of any type. Records will be structures that will group together a

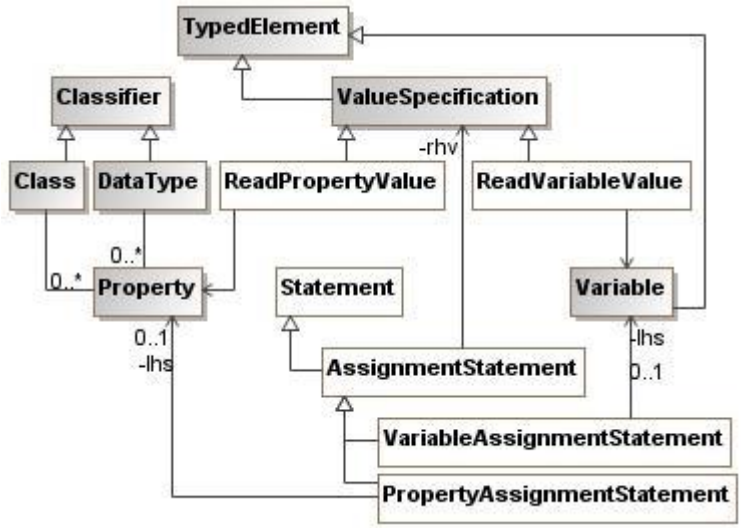

FIGURE 2. Assignment statements

number of fields, such that a field is similar to a variable declaration.

$$\begin{aligned} TYPE &: DataType|Class|Interface \\ DataType &: PrimitiveType|ArrayType \\ PrimitiveType &: Integer|Boolean|String|Real| \\ ArrayType &: TYPE[DOMAIN] \\ DOMAIN &: INT..INT|INT..* \end{aligned}$$

In the specification of the domain, first case describes an array of known size at declaration, and the second case specifies an array whose size is not known when declared.

Consequently, we will allow expressions to contain values of the newly introduced types, namely the value of an element of the array, and the value of a field from a record:

$$\begin{aligned} atom &: ID|INT|STRINGLITERAL| \\ ID'('(expr)?(',' expr) * endN =')'|e &= TRUE| \\ e &= FALSE|condition|ID'[INT']' \end{aligned}$$

The statements that involve expressions need also to be revised. Figure 2 shows part of the syntax, without specifying all the statements. The complete syntax is presented in the Appendix. The dashed components are the ones defined in UML and the white-box components are the ones introduced in ComDeValCo. Assignment statement is further specialized in two categories, depending on its left-value: for variables or for properties. According to UML 2.1 *Property* can be associated to a *Class* or to a *DataType*.

The syntactical rules corresponding to these statements are:

$$AssigmentStatement : VarAssignStatement | PropAssignStatement$$
$$VarAssignStatement : IDASSIGNexpr$$
$$PropAssignStatement : Classname.PropASSIGNexpr |$$
$$ID[ID]ASSIGNexpr$$

We adopt the same approach as the OMG Specification of OCL: we consider that there is a signature $\Sigma = (T, \Omega)$ with T being a set of type names, and Ω being a set of operations over types in T. The set T includes the basic types *int, real, bool* and *String*. These are the predefined basic types of OCL. All type domains include an undefined value that allows to operate with unknown or null values. Array types are introduced to describe vectors of any kind of elements, together with corresponding operations. All the types and operations are defined as in OCL.

DataType: is the root datatype of PAL and represents anything. Therefore, its methods apply to all primitive types, array type and record type. It is defined to allow defining generic operations that can be invoked by any object or simple value. It is similar to *AnyType* defined in OCL, but we have preffered this approach since *AnyType* is not compliant with all types in OCL, namely Collection types and implicitely its descendants. Defining *DataType* and its operation *new* we can create uniformly any new value as a reference to its starting address.

Operations on the type:

isType(element : DataType) : Boolean Checks if the argument is of the specified type,

new() :DataType Creates a new instance of the type.

Operations on instances of the type:

isTypeOf(type) : Boolean Checks if the instance is of the specified type

isDefined() : Boolean Checks if the instance is defined (not null).

Array Type inherits from *DataType*.

Operations:

size() : Integer Returns the size of the array

isEmpty() : Boolean Checks if the array has no items.

Operation *[]* takes an integer i as argument and returns the i-th element of the array.

The operations regarding the variable declarations are implemented in the *DataType*. In such a way, we may create new instances of array type, and we may check if the instance is not null.

A type is assigned to every expression and typing rules determine which expressions are well-formed. There are a number of predefined OCL types and operations available for use with any UML model, which we considered as given. For the newly introduced type constructions and its associated operations we will give typing rules. The semantics of types in T and operations in Ω is defined by a mapping that assigns each type a domain and each operation a function.

The following rule states that we may create arrays of any existing type in the system:

$$\frac{G|-A{:}T}{G|-Array(A){:}T}$$

An array M is defined with elements of a type and an integer as index:

$$\frac{G|-N{:}Int,G|-M{:}A}{G|-array(M,N){:}Array(A)}$$

If i is an index of an array then i is of type $Integer$.

$$\frac{G|-M{:}Array(A)}{G|-indexM{:}Integer}$$

The following rule specifies the way we can infer the type of an element of an array knowing the type of the array:

$$\frac{G|-N{:}Int,G|-M{:}Array(A)}{G|-M[N]{:}A}$$

The last rule states the constraints imposed on assignment to an element of an array:

$$\frac{G|-N{:}Int,G|-M{:}Array(A),G|-P{:}A}{G|-M[N]{:=}P{:}array(A)}$$

In order to illustrate our workbench support for defining and executing platform-independent components we consider a simple case study that prints a given product catalog. The class diagram presented in Figure 3 shows an extract of an executable UML model developed using COMDEVALCO Workbench [7]. The *Product* entity represents descriptive information about products and the *ProductCatalog* interface have operations that can be used to obtain product descriptions as well as the product prices. The *CatalogPrinter* component is designed to print the catalog, so it requires a reference to a *ProductCatalog*. The model contains a *SimpleProductCatalog* implementation that has an array of products and an array of *prices*.

The model defined in Figure 3 uses the stereotypes defined by the iCOMPONENT UML profile for dynamic execution environments [4]. According to the iCOMPONENT component model, these model elements can be deployed

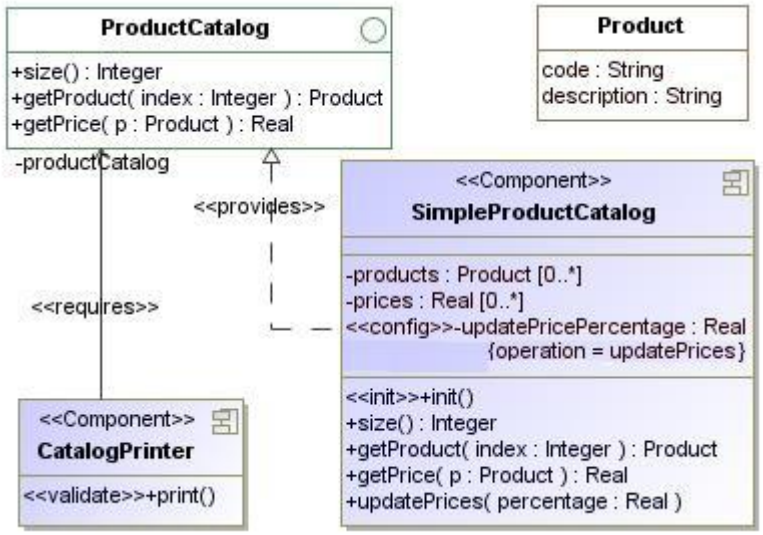

FIGURE 3. Executable iCOMPONENTS

as three modules (units of deployments): a module that contains the *Product* class and the *ProductCatalog* interface, another one that contains the *CatalogPrinter* component, and finally a module containing the *SimpleProductCatalog* component. After deployment, the dynamic execution environment applies the dependency inversion principle in order to inject the *ProductCatalog* reference required by the *CatalogPrinter* component.

Using the *validate* stereotype the *CatalogPrinter* component register the *print* method as a callback method that is executed when the component acquire the required interface. The execution starts with this method.

The *init* method of the *SimpleProductCatalog* component is executed immediately after an instance of the component is created. The *updatePricePercentage* property is a configuration property that specifies that the *updatePrices* operation will be executed when a running component instance is reconfigured.

Figure 4 and 5 show the code written using the proposed extended OCL-based action language.

5. CONCLUSION AND FUTURE WORK

We have presented an Action Language based on procedural paradigm and defined an extension with arrays, that can be succesfully used in specifying executable UML components. The approach taken in extending the PAL,

```
operation print() {
  Integer productCount = productCatalog.size();
  for(int index = 0; index < productCount; index++) {
    Product product = productCatalog.getProduct(index);
    write(product.code);
    write(product.description);
    write(productCatalog.getPrice(product));
  }
}
```

FIGURE 4. *CatalogPrinter* operation

```
operation size(): Integer {
  return products.size();
}
operation getProduct(index: Integer): Product {
  assert (0 <= index) and (index < products.size());
  return products[index];
}
operation getPrice(product: Product): Real {
  return prices[product.code];
}
operation init() {
  products = new Product[] {
    new Product(0, "A"), new Product(1, "B")
  };
  prices = new Real[] {5, 7};
}
operation updatePrices(percentage: Real) {
  for(int index = 0; index < prices.size(); index++)
    prices[index] = (1 + percentage) * prices[index];
}
```

FIGURE 5. *SimpleProductCatalog* operation

can be used in adding new features to it, and integrating them in the framework. The main application of such specifications is to completely describe executable components for storing in a repository, as suggested in [4].

As future developments we intend to add, when necessary, further extensions to the PAL and integrate them in ComDeValCo workbench, and to use information from these specifications to validate the components.

6. ACKNOWLEDGEMENTS

This work was supported by the grant ID_546, sponsored by NURC - Romanian National University Research Council (CNCSIS).

7. References

[1] S. Haustein and J. Pleumann. *OCL as Expression Language in an Action Semantics Surface Language*. OCL and Model Driven Engineering, UML 2004 Conference Workshop, 2004.

[2] K. Jiang, L. Zhang, and S. Miyake. *OCL4X: An Action Semantics Language for UML Model Execution*. Proc. of COMPSAC, pages 633-636, 2007.

[3] I. Lazar, B. Parv, S. Motogna, I. Czibula, and C. Lazar. *An Agile MDA Approach for Executable UML Structured Activities*. Studia Univ. Babes-Bolyai Informatica, 2:101-114, 2007.

[4] I. Lazar, B. Parv, S. Motogna, I.-G. Czibula, and C.-L. Lazar. *iCOMPONENT: A platform-independent component model for dynamic execution environments*. In 10th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing. submitted, 2008.

[5] Object Management Group. *Object Constraint Language Specification, version 2.0*. http://www.omg.org/cgibin/ apps/doc?formal/06-05-01.pdf, 2006.

[6] Object Management Group. *UML 2.1.1 Superstructure Specification*. http://www.omg.org/cgi-bin/doc?ptc/07-02-03/, 2007.

[7] B. Parv, I. Lazar, and S. Motogna. *COMDEVALCO Framework - the Modeling Language for Procedural Paradigm*. Int. J. of Computers, Communications and Control, 3(2):183- 195, 2008.

[8] C. Raistrick, P. Francis, J. Wright, C. Carter, and I. Wilkie. *Model Driven Architecture with Executable UML*. Cambridge University Press, 2004.

[9] J. Steel and J.-M. Jezequel. *On model typing*. International Journal of Software and System Modeling (SoSyM), 2008.

8. Appendix A - PAL grammar

```
prog: program (operation)* | (operation)+
program: PROGRAM ID pre_post_conditions statement_block
operation : OPERATION ID operation_parameter_list
    (':' TYPE)? pre_post_conditions statement_block
operation_parameter_list : aux='(' (operation_parameter)?
    (','operation_parameter)* ')'
operation_parameter : (PARAM_TYPE)? aux=ID ':' TYPE
pre_post_condition:(precondstatement)?(postcondstatement)?
statement_block : startN='{' statement* endN='}'
statement: asignstatementstandalone | callstatement | ifstatement |
    declstatement | whilestatement | forstatement | assertstatement |
     readstatement | writestatement | returnstatement
callstatement : CALL expr endN=';'
```

readstatement : READ ID endN=';'
writestatement : WRITE expr endN=';'
asignstatement : ID ASSIGN expr
asignstatementstandalone : ID ASSIGN expr endN=';'
declstatement :VARDECLR ID':'TYPE(':='expr)?endN=';'
ifstatement : IF '(' expr ')' b1=statement_block
(ELSE b2=statement_block)?
whilestatement : WHILE '(' expr ')' loop_statement_block
forstatement : FOR '(' e1=asignstatement ';' e2=expr ';'
e3=asignstatement ')' loop_statement_block
assertstatement : ASSERT ':' expr endN=';'
returnstatement : RETURN expr? endN=';'
precondstatement : PRECOND ':' oclexpr endN=';'
postcondstatement : POSTCOND ('(' ID ')')? ':' oclexpr endN=';'
loopinvariant : LOOPInv ':' expr endN=';'
loopvariant : LOOPVa ':' expr endN=';'
loop_statement_block : startN='{' (loopinvariant)?
(loopvariant)? statement* endN='}'
condition : '(' expr ')'
oclexpr : expr
expr: sumexpr
sumexpr : (conditionalExpr) (OP_PRI0 =conditionalExpr)
conditionalExpr : (multExpr) (OP_PRI1 e=multExpr)
multExpr : (atom) (OP_PRI2 e=atom)*
atom: ID | INT |STRINGLITERAL | ID '(' (expr)? (',' expr)* endN=')'|
e=TRUE | e=FALSE | condition
PARAM_TYPE: 'in'| 'out' | 'inout'
TYPE : 'Integer' |'Boolean' | 'String' | 'Real'| DataType |
TYPE[DOMAIN]
DOMAIN : INT..INT | INT.. *
OP_PRI0 :('and' | 'or' | 'not' |′ <′ |′ >′ |′ <=′ |′ >=′ |
′ ==′ |′ <>′)
OP_PRI1 : ('+' |'-')
OP_PRI2 : ('*' |'/' | 'div')
ID : ('a'..'z' |'A'..'Z') ('a'..'z'|'A'..'Z' | '0'..'9')*
INT : '0'..'9' +
STRINGLITERAL : ' " ' (options {greedy=false;} : .)* ' " '
BOOLEAN_CONST : 'true' | 'false'

Department of Computer Science, Faculty of Mathematics and Computer Science, Babeş-Bolyai University, 1, M. Kogălniceanu, Cluj-Napoca 400084, Romania

E-mail address: bparv,motogna,ilazar,czibula@cs.ubbcluj.ro