5th Slovakian-Hungarian Joint Symposium on Applied Machine Intelligence and Informatics
January 25-26, 2007 ░ Poprad, Slovakia

A New Formalism Technique for OCL

Gergely Mezei, Tihamér Levendovszky, Hassan Charaf

Department of Automation and Applied Informatics, Budapest University of Technology and Economics
Goldmann György tér 3, H-1111 Budapest, Hungary
{gmezei, tihamer, hassan}@aut.bme.hu

Abstract: Modeling, especially domain-specific modeling has growing importance in many fields of software enginering, such as modeling control flows of data processing for example in man-machine systems. Customizable language dictionary and customizable notations of the model elements offered by domain-specific technologies makes software systems easier to create and maintain. Visual model definitions have a tendency to be incomplete, or imprecise, thus, the definitions are often extended by textual constraints attached to the model items. Textual constraints can also eliminate the incompleteness stemming from the limitations of the visual definitions. Object Constraint Language (OCL) is one of the most popular constraint languages in the field of Domain Specific Modeling Languages. OCL is a flexible, yet formal language with mathematical background. OCL has been formalized using set theory. Our research focuses on creating an OCL optimization solution, but the existing formalism is hard to use in the field of dynamic, constraint manipulating algorithms. The paper presents OCLASM, a new formalism for OCL based on the Abstract State Machines technique.

Keywords: Domain-specific modeling, OCL, constraints, Abstract State Machines

1 Introduction and Motivation

Domain Specific Modeling Languages (DSMLs) allow creating visual models using a high level of abstraction, the customization of model rules and notation. Visual models created by DSMLs makes easier to understand and handle the problems that made them very popular in almost every field of software engineering including, but not limited to general software modeling, feature modeling, resource editing or control flow modeling.

Although visual language definitions have many advantages, they have the tendency to create imprecise, incomplete, and sometimes even inconsistent definitions. For example, assume a domain describing the cooperation between computer networks and humans in a man-machine system. A computer can have input and output connections in the network, but these connections use the same cable with maximum *n* channels. Thus, the number of the maximum available output connections equals the total number of channels minus the current number of input channels. It is hard, or even impossible to express this relation in a visual way. The solution to the problem is to extend the visual definitions by textual constraints. There exist several textual constraint languages, the Object Constraint Language (OCL) is possible the most popular among them. OCL was originally developed to create precise UML diagrams [1] only, but the flexibility of the language made possible to reuse OCL in language engineering, such as in metamodeling [2]. Nowadays, OCL is one of the most wide-spread approaches in the field of metamodeling and model transformations. The textual constraint definitions of OCL are unambiguous and still easy to use.

Nowadays, domain-specific modeling tools usually have support for OCL either by interpreters, or by compilers. Interpreters are easier to implement, but they are not as flexible and efficient as compilers. Thus, the key of efficient constraint handling is to use optimizing OCL compilers. None of the existing tools support optimization. Our research focuses on creating a complete, system-independent optimizing constraint compiler. We have created three optimizing algorithms (presented in [3] and in [4]) that can accelerate the validation process by relocating, decomposing the constraint expressions and by caching the model queries. The algorithms can accelerate the validation by 10-12% according to our measurements. Besides the pseudo code of the algorithms, preliminary proofs of correctness were also presented in the mentioned papers. However when implementing the algorithms, we have found that the proofs are not precise enough, formal proofs are required.

OCL has a mathematical definition based on set theory with a notion of object model and system states. Although this formalism ensures that OCL constraints are unambiguous, it does not cover dynamic behavior of the constraints. Set theory is a highly flexible formalism technique that could be extended to support dynamic behavior. Another way to formalize the optimization algorithms are the Abstract State Machines (ASMs), formerly known as *evolving algebras*. ASMs were introduced by Gurevich [5]. ASM is working on abstract data structures, which comes with a simple mathematical foundation. The notation of ASM is based on the mathematically precise notion of a virtual machine execution, states and state transitions. This notation is familiar both from programming practice and mathematical standards. ASM provides a concise way to define system semantics and dynamic behavior. ASMs are very popular in the domain of formal specification. The ASM formalism has several advantages in contrast with the original formalism in this field. Firstly, the notation of ASM is easier to use for proving the correctness of algorithms given by pseudo code. Secondly, modularization and stepwise refinement is easier to accomplish in ASM. This also means that the formalism specification of the dynamic behavior can be hierarchically decomposed. Set theory is a flexible technique, but it uses a low-level description of the problem space, thus, the description the dynamic behavior would produce a considerably huge rule set. In case of ASM this problem does not occur, because ASM allows to choose the level of abstraction used in the formalism. Therefore, the formalism in ASM can be more concise.

This paper presents OCLASM, the ASM formalism of the OCL language. Our aim is to use this formalism to prove the correctness of the optimization algorithms. The paper also contains the formalism of one of the optimization algorithms, the *RelocateConstraint* algorithm to show how the formalism can be used in practice. The paper is organized as follows: Section 2 elaborates the most important projects in the field of OCL formalism and Abstract State Machines. Section 3 presents the basics of Abstract State Machines. Section 4 introduces the new formalism technique including the construction of the formalism and several basic examples. Finally, Section 5 summarizes the presented work.

2 Related Work

Abstract State Machines were used in many projects as a mathematical formalism. This section introduces only a few of these projects, more precisely, the projects in connection with OCL or modeling, and projects from where our method has borrowed some basic ideas. Some of the mathematical model formalisms not based on ASM are also presented.

The book [6] presents a precise approach, which facilitates the analysis and validation of UML models and OCL constraints. It defines a formal syntax and semantics of OCL types, operations, expressions, invariants, and pre-postconditions, and it discusses some of the main problems with the original OCL specification. Although the book does not use ASM for formalism, it gives a precise overview about the topic.

The OCL formalism available in set theory is examined in [7]. The paper collect the elements appearing in OCL standard, but not in the formalism. It presents an extension of the original formalism to solve these problems.

In [8] an ASM definition for dynamic OCL semantics is presented. This formalism focuses on the states of the modeling environment and handles the invariants as atomic units implemented in outer functions. This means that the formalism handles the effects and the result of the validation, but it does not give ASM definition for the OCL statements, such as `forall`, thus, it is not capable of describing algorithms operating with statements.

ASM definition for Java and Java Virtual Machine (JVM) is elaborated in [9]. This ASM formalism offers an implementation-independent description of the language and the execution environment. Using this abstract description, several properties of Java and JVM have been proved. The book contains several straightforward solutions. Our approach has borrowed the basic idea of formalization, namely handling the code as an annotated syntax tree from here.

3 Backgrounds

3.1 Basic ASMs

In [5], ASMs are introduced as follows. ASMs are finite sets of *transition rules* of the form

```
if (condition) then Updates
```

which transform abstract states. Where *Condition* (referred to as guard) under which a rule is applied is an arbitrary predicate logic formula without free variables. The formula of *Condition* evaluates to *true* or *false*. *Updates* is a infinite set of assignments in the form of $f(t_1..t_n)$:= t whose execution is understood as changing (or defining, if it has been not defined before) the value of the occurring function *f* at the given arguments.

The notion of *ASM states* is the classical notion of mathematical structures where data come as abstract objects, i.e., as elements of sets (domains, universes, one for each category of data) which are equipped with basic operations (partial functions) and predicates (attributes or relations). The notion of *ASM run* is the classical notion of computation in transition systems. An ASM computation step in a given state consists of executing simultaneously all updates of all transition rules whose guard is *true* in the state if these updates are *consistent*. A set of updates is called *consistent* if it contains no pair of updates with the same location.

Simultaneous execution provides of an ASM rule *R* for each *x* satisfying a given condition φ:

```
forall x with φ R,
```

where φ is a Boolean-valued expression and *R* is a rule. We freely use abbreviations, such as *where, let, if then else*, *case* and similar standard notations which are easily reducible to the above basic definitions.

A priori no restriction is imposed neither on the abstraction level or on the complexity or on the means of the function definitions used to compute the arguments and the new value denoted by t_i, *t* in function updates. The major distinction made in this connection for a given ASM *M* is that between *static* functions which never change during any run of *M* and *dynamic* ones which typically do change as a consequence of updates by *M* or by the environment. The dynamic functions are further divided into four subclasses. *Controlled* functions are dynamic functions which can directly be updated by and only by the rules of *M*. *Monitored* functions are dynamic functions which can directly be updated by and only by the environment. *Interaction* or *shared* functions are dynamic functions which can directly updated by rules of *M* and by the environment. *Derived* functions are dynamic functions which cannot be directly updated either by *M* or by the environment, but are nevertheless dynamic because they are defined in terms of static and dynamic functions.

3.2 The Mathematical Definition of ASM

In an ASM state, data is available as abstract elements of domains which are equipped with basic operations represented by functions. Relations are treated as Boolean-valued functions and view domains as characteristic functions, defined on the superuniverse which represents the union of all domains. Thus, the states of ASMs are algebraic structures, also called algebras.

Definition 1 *A vocabulary (also called signature) Σ is a finite collection of function names. Each function name has an* arity, *which is a non-negative integer representing the number of arguments the function takes. Function names can be static or dynamic. Nullary function names are often called constants; but the interpretation of dynamic nullary functions can change from one state to the next. Every ASM vocabulary is assumed to contain the static constants* undef, True *and* False.

Definition 2 *A state $\mathfrak{A}$ of the vocabulary Σ is a non-empty set X, together with the interpretation of the function names of Σ, where X means the superuniverse of $\mathfrak{A}$. If* f *is an n-ary function name of Σ, then its interpretation $f^{\mathfrak{A}}$ is a function from X^n into X; if c is a constant of Σ, then its interpretation $c^{\mathfrak{A}}$ is an element of X. The superuniverse X of the state $\mathfrak{A}$ is denoted by $|\mathfrak{A}|$.*

The *elements* of the state are the elements of the superuniverse of the state and according to the definition the parameters of the functions are also elements of the superuniverse.

Definition 3 *An abstract state machine M consists of a vocabulary Σ, an initial state $\mathfrak{A}$ for Σ, a rule definition for each rule name, and a distinguished rule name called the main rule name of the machine.*

4 The Formalization Method

4.1 The Basics

In OCLASM, ASM is used to describe the *execution* of constraints. Thus, the approach *observes* the execution. The rules of ASM are static, namely they do not depend on the constraint, which makes easier to prove the equivalence of two algorithms or constraints.

States of OCLASM describe the execution of constraint expressions, where *state* represents the state of the constraint execution at a certain point of time. For example a *state* describes which expression is under evaluation. The rules of OCLASM are used to navigate between the *states* when running the validation. This approach is similar to the method published in [9] in several aspects. The constraint expression is handled as the sequence of programming statements and expressions. The execution of the constraint is a step-by-step execution of these programming units. At each position, the corresponding expression or statement is evaluated or executed, and then the control flow proceeds to the next programming unit. This method is similar to traversing the annotated abstract syntax tree of the constraint.

The *state*s of OCLASM contains the programming units (statements or expressions, referred to as *Phrase*s) of the constraint, and the current position of the execution. It is important that the *state*s do not contain any particular information about the underlying model (the model to validate), since (i) OCL cannot change the underlaying model by the definition of OCL [1], and (ii) the validation must be platform independent. Thus, to access the model items, external functions are used as described later.

The superuniverse of OCLASM consists of (i) the universe of *Phrase*s, and (ii) the universe of possible positions in the source code. The first universe describes the type of *Phrase*s, namely the basic syntactic constructs available in OCL. This universe contains for example the definition of OclExpression

```
Exp := Property | Variable | Literal | Let |  OclMsg |
if (Exp) Exp else Exp
```

or the definition of boolean operations `Exp Bop Exp`. The second universe, called *Pos* represents the valid positions of the execution in the given constraint.

4.2 The Functions of OCLASM

Recall that constraint handling must be platform-independent. The model items are identified by a unique ID (a literal expression), but the model representation is platform independent. This requirement can be easily accomplished, if there is an *ModelInterface* to handle every model-based operation [2]. This way, the database-related implementation can be hidden. In case of OCLASM this means that accessing the model items is handled in external (*monitored*) functions:

- `meta(id)`: Obtains the meta ID of the model item identified by `id`.
- `navMultiplicity(id, dest)` : Obtains the multiplicity of the edge from the model item identified by `id` and the destination, whose navigation name is `dest`.
- `navigateTo(id, dest)` : Obtains the elements of the edge from the model item identified by `id` and the destination, whose navigation name is `dest`. The result is a *Set* of model IDs.
- `getAttribute(id, name)` : Obtains the value of the attribute `name` of the model item identified by `id`.

The execution of a constraint is handled by dynamic functions based on the superuniverse of OCLASM. The dynamic function *RestBody* contains (i) part of the current constraint still to be executed and (ii) the results and values already computed and still needed. At the beginning of the execution *Restbody* contains the representation of the constraint itself. The values in *RestBody* are *Phrase*s. Another dynamic function, called *pos* represents the current position of the execution, pointing to the current expression or statement to be executed. The values of *pos* are elements of the universe *Pos*. The expression *RestBody/pos* denotes the currently to be computed subterm of *RestBody* at *pos*. It will eventually be replaced by the computed value of the subexpression.

To navigate between subexpressions during the execution of the constraint we define the functions *Child* and *Top*. Each expression containing subexpressions (children) can use the function *Child* to reach these subexpressions. The function has two parameters: the position of the expression and the ordinal number of the subexpression. The function returns with the position of the subexpression. The function *Top* represents the inverse operation: if the execution of a subexpression is terminated, then the *Top* function is used to retrieve the position of the parent expression.

The dynamic function *Type* is used to handle the type of the different expressions uniformly. It has one input parameter, the position of the expression. The return value of the function can be one of the basic types defined in OCL, such as real, integer, or collections. Collections are handled as special types, because here the name of the type does not really define the type. For example `OrderedSet` does not hold any information about the contained model items, thus, `OrderedSet('a','b')->first() + 3` cannot be identified as an invalid expression. To solve this problem, the type of the collection contains an additional information as follows: `Collection(ItemType)`. Note, that the expression is recursive (`ItemType` is also a type), but it does not cause any difficulties, because the collections are handled flattened [1].

The value of the expressions are handled similarly to the type function: The function *Value* retrieves the position of the expression and returns its value. Using the notation of common progamming languages, such as C, the difference between `Restbody/pos` and `Value(Restbody/pos)` is the following: `Restbody/pos` is similar to a pointer. In contrast, `Value(Restbody/pos)` is the value in the pointed memory block.

OCL allows the user to define local variables. In OCLASM, these variables are handled by the function *Local*. The function has one input parameter: the name of the variable and it returns the position of a variable declaration expression. Variable declarations contain the name, type and value of the variable. If there is no local variable defined with the given name, then the function returns *undefined*. Name of the local variables are handled by the unary function *Name*, which has one input parameter, the position of the variable expression. *Name* returns undefined, if the position is not a valid variable definition.

OCLASM handles all four types of collections (`Set, OrderedSet, Bag` and `Sequence`) by arrays indexed by integer numbers. Indexing is denoted by brackets. The items in the arrays are the items in the collections, for example `myset[3]` means the third item in the collection `myset`. The arrays can be traversed by the `forall` expression of ASM, obtaining every element. The dynamic function *Length* helps to query the number of items contained by the collection.

Tuple types are also handled as arrays, but here the indices are the names of the tuple items. For example the expression `Tuple(x: Integer = 5, y: String = 'Ok')` results an array with the indices ‘x’ and ‘y’. Tuple items can have different types, thus, using the definition of the previous example `tuple["x"]` results an *Integer* value, while `tuple["y"]` results a *string.*

4.3 The Definition of OCLASM

Using the previously defined functions, the syntax of OCLASM can be defined:

Definition 4 *The vocabulary Σ_{OCLASM} of our OCLASM formalism is assumed to contain the following characteristic functions (arities are denoted by dashes):*

- *undef, true, false, pos/0*
- *RestBody/1, Child/1, Top/1, Type/1, Value/1, Name/1, Local/1, Length/1*
- *meta/1, navMultiplicity/2, navigateTo/2, getAttribute/2.*

The first two rows of functions definition describe the *dynamic* functions, while the last row contains the *monitored* functions.

Definition 5 *The superuniverse* $|\mathfrak{A}|$ *of a state* $\mathfrak{A}$ *of* Σ_{OCLASM} *is the union of two universes: (i) the possible programming Phrases of OCL, and (ii) the possible positions of the Phrases in the abstract tree representing the constraint.*

4.4 Transition Rules

The input of the formalism is the OCL constraint. It is important that the constraint must not use abbreviations. For example to query the attributes the expressions must be fully expanded. Thus, `age` is not valid, but `self.age` is valid when the formalism is applied.

Transition rules describe how the states of OCLASM change over time by evaluating expressions and executing statements of the input program. Initially, *RestBody* is the given method body, *pos* is its start position and *Local* is empty.

All *Phrase*s are evaluated from innermost to outermost. This basic behavior is implemented in the rule `eval`. Each language construct available in OCL has well-defined rules describing the right order of execution of the subexpressions. For example in the case of conditional expressions, the condition is evaluated first. The rule `eval` obtains the expression at the current position and updates the function *pos* according to the right order of execution. *Pos* is updated from unevaluated expressions to the appropriate subexpressions until an atomic expression, for example a primitive literal is reached. For binary expressions, the left-to-right evaluation strategy is used. When the computation of a given expression is finished, then the *Value* and *Type* functions is updated for the position of the expression. If the whole constraint is processed then the topmost expression holds the value of the constraint (supposing that the constraint can has a return value, like invariants). For the sake of simplicity, `eval` has a return value that is the *Value* of the evaluated expression. Language constructs appear not only in the rule `eval`, but each language constructs has its own rule. While `eval` describes the execution order, these rules describe the semantics of the expression. Although the OCLASM formalism presented in this paper is capable of describing all OCL operations, we present only a few rules showing the method in practice. Other operations can be formalized similarly. To simplify the examples, we use `r/p` instead of `RestBody/pos`.

Firstly, a very simple rule, the *VariableDeclaration* is shown. The function `eval` obtains the position of the children expressions (variable name, type and init value) and executes them before this rule is executed.

```
rule VariableDeclaration(VarName, VarType, VarInit)
{
  Name(r/p)  = VarName
  Type(r/p)  = eval(VarType)
  if (VarInit!= undef)
```

```
         Value(r/p) = eval(VarInit)
     else
         Value(r/p) = undef
  endif
}
```

Secondly, the rule for `iterate` operations is presented. It is essential to formalize this operation, because every other collection operation can be accomplished by using `iterate` [1]. For example, the collection operation `count()` can be simulated by an iterate expression

`iterate(i : Integer, r Integer = 0 | r+1).`

The node `iterate` has exactly four children in the abstract syntax tree: (i) the collection, where he operation is applied; (ii) the declaration of the iterate variable, (iii) the declaration of the result variable, and (iv) the iteration body.

```
rule iterate(CollDef, IteratorDecl, ResultDecl, Iteration)
{
  eval(CollDef);
  eval(IteratorDecl);
  eval(ResultDecl);

  Value(Local(Name(ResultDecl)))= Value(ResultDecl)
  Type(Local(Name(ResultDecl))) = Type(ResultDecl)

  forall collectionElement in Value(CollDef)
   Value(Local(Name(IteratorDecl)))= collectionElement
   Type(Local(Name(IteratorDecl))) = Type(IteratorDecl)
   eval(Iteration);
  endfor

  Value(Local(Name(IteratorDecl)))= undef;
  Value(r/p) = Value(Local(Name(ResultDecl)));
  Type(r/p)  = Type(Local(Name(ResultDecl)));
  Value(Local(Name(ResultDecl)))= undef;
}
```

The third example shows the rule constructed for navigation expressions. Here the model-based, external functions are also used. The rule evaluates the origin, namely, it obtains the model item, which is the starting point of the navigation. As next, the rule checks the multiplicity of the rule, if it allows exactly one connection, then the result is a *ModelItem*, in any other case the result is a collection of *ModelItems*. Since the function `navigateTo` always returns a list (with the IDs of the destination nodes), thus, in the first case the first element of the result array is used (`navigateTo(Value(Origin), DestName)[0]`). In this case the type of the result is the ID of the meta node of the destinations node. If the multiplicity is not 1, then a new collection is created and returned.

```
rule Navigate(Origin, DestName)
{
  eval(Origin);
  if (navMultiplicity(Value(Origin),Name(DestName))==1)
     Value(r/p)= navigateTo(Value(Origin),DestName)[0]
     Type(r/p) = ModelItem
  else
     Type(r/p) = Set(ModelItem)
     Value(r/p) = Set()
     forall ModelId in navigateTo(Value(Origin),DestName)
       Value(r/p)[Length(r/p)+1)] = ModelId
     endfor
  endif
}
```

Conclusions

Since visual model definitions cannot describe models precisely enough, thus, visual model language definitions must be extended with textual constraints. OCL is one of the most popular textual constraint language; it is essential to provide precise, unambiguous definitions in several modeling techniques such as UML, or metamodeling. One of the key features of OCL is the mathematical formalism based on set theory with a notion of an object model and system states. This formalism can prove the completeness of the models using OCL, but it does not contain the definition of constraint execution, or the dynamic behavior of the constraint expressions. Due this limitation of the OCL formalism, the correctness of the algorithms cannot be proven in general. This restriction meant that we could not prove the correctness of our optimization algorithms in a mathematical form.

This paper has presented OCLASM, a new formalism for OCL. The paper has presented the main reasons, why a new formalism was made instead of extending the existing one. The new formalism is based on the popular ASM technology, and it can be used to study the dynamic behavior. ASM allows to define the behavior in a compact, yet rigorous way. The basic idea of the formalism is to create rules for all language expressions, such as `iterate`, and use these rules to simulate the validation. OCLASM handles the constraints as a sequence of statements and expressions and it navigates through these programming units using a function pointing to the current expression. Model-based operations use monitored (external) functions showing that constraint validation must be independent from the current model representation.

The mechanism of the formalism method has been shown including how to handle language construct, such as tuple types, or collections. The formal definition of

OCLASM has also been presented and the paper also includes several rules for the most important language constructs.

Using the new formalism of OCL, it is possible to create and validate algorithms based on OCL. OCLASM can guarantee the mathematical background for advanced constraint handling, for example for optimizing algorithms. This extension for OCL can improve handling of textual constraints and, thus, the efficiency of visual model engineering. Future work mainly focuses on describing the correctness of our OCL optimization algorithms and proving their correctness using this description.

Acknowledgement

The paper is established by the support of the National Office for Research and Technology (Hungary).

References

[1] Jos Warmer, Anneke Kleppe: Object Constraint Language, The: Getting Your Models Ready for MDA, Second Edition, Addison Wesley, 2003 [2] UML 2.0 Specification, http://www.omg.org/uml/

[2] Gergely Mezei, László Lengyel, Tihamér Levendovszky, Hassan Charaf: Extending an OCL Compiler for Metamodeling and Model Transformation Systems: Unifying the Twofold Functionality, INES 2006

[3] G. Mezei, T. Levendovszky, H. Charaf: An Optimizing OCL Compiler for Metamodeling and Model Transformation Environments, Working Conference of Software Engineering, 2006

[4] G. Mezei, T. Levendovszky, H. Charaf: Restrictions for OCL Constraint Optimization Algorithms, OCL for (Meta-) Models in Multiple Application Domains (OCLApps) Workshop, 2006

[5] Egon Börger, Robert Stärk, Abstract State Machines: A Method for High-Level System Design and Analysis, Springer-Verlag, 2003 [6] Bison, Official Homepage, http://www.gnu.org/software/bison/bison.html

[6] M. Richters: A Precise Approach to Validating UML Models and OCL Constraints. PhD thesis, Universit at Bremen, Bremen, Germany, 2001

[7] Stephan Flake: Towards the Completion of the Formal Semantics of OCL 2.0. ACSC 2004: 73-82

[8] Stephan Flake, Wolfgang Müller: An ASM Definition of the Dynamic OCL 2.0 Semantics. UML 2004: 226-240

[9] Robert F. Stärk, Joachim Schmid, Egon Börger: Java and the Java Virtual Machine: Definition, Verification, Validation, Springer Verlag, 2001