Formalizing the Evaluation of OCL Constraints

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: Domain-specific modeling has growing importance in many fields of software enginering, such as modeling control flows of data processing, or in man-machine systems. Customizable language dictionary and customizable notations of the model elements offered by domain-specific technologies make software systems easier to create and maintain. However, visual model definitions have a tendency to be incomplete, or imprecise; the definitions can be extended by textual constraints attached to the model items. Textual constraints can eliminate the incompleteness stemming from the limitations of the structural definition as well. The Object Constraint Language (OCL) is one of the most popular constraint languages in the field of UML and Domain Specific Modeling Languages. OCL is a flexible, yet formal language with a mathematical background. Existing formalisms of OCL does not describe dynamic behavior of constraints. Our research aims at creating an OCL optimization solution and prove its correctness formally. However, the shortcomings of the existing formalism has led us to create a new formalism. The paper presents OCLASM, a new formalism for OCL, which can describe both the semantics and the dynamical behavior of the language constructs, thus, it is capable of describing proofs of optimization algortihms. OCLASM is based on the Abstract State Machines technique.

Keywords: OCL, optimization, constraints, Abstract State Machines

1 Introduction and Motivation

Visual languages can accelerate the develepement and maintenance of software systems. Moreover, the ability to illustrate the structure of a system graphically means that even non-programmer users can understand the underlying logic. One family of visual languages are the Domain-Specific Modeling Languages (DSMLs), which allow creating visual models using a high level of abstraction, the customization of model rules and notation. The customization abilities of DSMLs makes easier to understand and handle the problems that made DSMLs very popular in almost all fields of software engineering including, but not limited

to general software modeling, feature modeling [1], resource editing [2] or control flow modeling.

Besides the advantages of visual languages, they have weaknesses as well. Visual model definitions have the tendency to be imprecise, incomplete, and sometimes even inconsistent. 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 [3] only, but the flexibility of the language made possible to reuse OCL in language engineering, such as in metamodeling [4]. 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.

There exists several commerical and non-commerical domain-specific modeling tools, which 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. The key of efficient constraint handling is to use optimizing OCL compilers. To our knowledge, currently 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 [5] and in [6]) that can accelerate the validation process by relocating, decomposing the constraint expressions and by caching the model queries. We have implemented these algorithms in our tool Visual Modeling and Transformaion Tool [7]. 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 formal proofs are required to ensure the correctnes of the compiler.

OCL has a mathematical definition based on set theory with a notion of object model and system states. Although this formalism defines the syntax and semantics of OCL constraints, it does not cover dynamic behavior of the constraints. This paper presents OCLASM, a new formalism of OCL based on Abstract State Machines [8]. Our aim is to use this formalism to describe the OCL language and to prove the correctness of our optimization algorithms. The paper 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 explains why we have decided to create a new formalism for OCL instead of extending the existing. The section elaborates the most important projects in the field of OCL formalism and Abstract State Machines as well. 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. Section 5 shows how OCLASM can be used in case of Relocateonstraint algorithm. Finally, we summarize the presented work in section *Conclusions*.

2 Related Work

2.1 Set Theory and ASMs

The first question to answer is why we have decided to create a new formalism for OCL instead of using the existing one. Existing formalism does not define the dynamic behavior of constraints, however set theory is a highly flexible formalism technique, thus the formalism could be extended to support dynamic behavior. To show the differences between such an extension and OCLASM, we have to introduce the properties of ASMs.

Abstract State Machines (ASMs) are formerly known as *evolving algebras*. ASM is working on abstract data structures, which are provided 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 from programming practice. 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 extension of 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 for our purposes with respect to OCL optimization.

2.2 OCL and ASM

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 [9] 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 [10]. The paper collects the elements appearing in OCL standard, but not in the formalism. It presents an extension of the original formalism to solve these problems.

In [11], 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 [12]. 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 ASM Basics

In [8], 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* denotes an 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 is provided 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 either 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 the environment only. *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

over 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. The new elements come from *reserve*, which is a set whose role is to provide new elements whenever needed.

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 ASM for OCL

4.1 Overview

OCLASM has been created to formalize the evaluation of constraints, OCLASM acts as an *interpreter* for OCL constraints. The same functions and the same rule set are used regardless of the constraint or the underlying model. This property of OCLASM is essential, since the OCLASM has been created as a generic formalism for OCL.

States of OCLASM represent the state of execution at a certain point of time. A *state* can be considered as an internal state of the evaluating environment, which is evolved by the rule set of OCLASM. *States* describe for example which expression is under evaluation or which local variables are available. The rules of OCLASM are used to navigate between the *states* when running the validation.

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 underlying model by the definition of OCL [3], and (ii) the validation must be platform independent. Similarly, *states* do not describe the constraints directly, but the expressions of the constraints are obtained by a monitored function.

The presented approach is similar to the method published in [12] in several aspects, where 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 evaluation proceeds to the next programming unit. The method is also similar to traversing the annotated abstract syntax tree of the constraint.

4.2 Monitored and Shared Functions

Obtaining the model items and the phrases of constraint expressions is handled by monitored functions. This solution ensures that the modeling environment and the evaluation environment are independent from the dynamic behavior of the constraint formalized by the OCLASM.

The underlying model extends the original model structure definition used in OCL: it consists of model nodes, attributes and relationships (not restricted to UML types). Attributes are either of primitive types or of complex attributes containing several sub-attributes. Model nodes can contain attributes (both primitive and complex attributes). The attribute and modeling structure described in [13] is used, which can extend the UML-based modeling structure to an n-level metamodeling hierarchy. This generic structure allows extending OCL to domain-specific languages as well [4]. The extension of the underlying modeling structure means that OCLASM can be used not only to describe the dynamics of constraint evaluation, but to formalize constraints for domain-specific models as well.

Both model nodes and attributes are identified by a unique ID (a literal expression), the universe of IDs is shared between the two different constructs. This uniformity helps reducing the number of monitored functions (for example, node-node and node-attribute navigations can be handled uniformly). To differentiate attributes and normal model nodes, there is a *IsModelNode(ID)* function defined.

To simplify handling of attributes the function *AttrValue(ID)* is used. The function returns *undef* for complex attributes, but returns a primitive value for primitive attributes. Primitive value in this context means that the type of the value is Boolean, number or string.

To express the meta level- instance level relationship the *Meta(ID)* function is used, which returns the meta item of the instance level item identified by ID. This

function is essential to capture different instantiation techniques (for example the instantiation of UML [14] and VMTS [15]).

Navigation between model items is handled by two functions. The function *To(ID, Dest)* works on the model level, the ID identifies the source model item, while *Dest* selects target items. The function returns all nodes or attributes which can be reached from the model item using a relation where the destination name is *Dest*. The result of the function is a list of possible destinations. Note that the list contains either model item IDs, or attribute IDs, but not both of them, because OCL does not allow this kind of polymorphism (attribute queries and navigations could not be distinguished).

The *Mul(ID, Dest)* ('Mul' stands for Multiplicity) function is another function to handle relations between model items. It works on the metamodel level, thus, the model definition is checked instead of the concrete models. The function *Mul* checks the minimum and the maximum multiplicity of the given relation according to the metamodel. It returns four integers as a list, the minimum/maximum multiplicity of the source/destination side.

OCLASM uses a shared function *GetPhrase(Position)* as well. *Phrase*s are basic syntactic constructs (programming statements and expressions) available in OCL. A *Phrase* has a string attribute *PhraseType* (e.g. 'NavigationCall'). *Phrase*s can contain other *Phrase*s as children. For example, an iteration *Phrase* can have an iterator variable declaration *Phrase*, an iteration condition *Phrase* and an iteration core block *Phrase*. The function *GetPhrase* returns a *Phrase* of the constraint identified by the parameter Position. Position is a value from the universe of all possible positions of *Phrase*s of the constraints (the universe is referred to as *Pos*). If the constraint is handled as a syntax tree then *Phrase*s are the nodes of this tree and the universe of *Pos* contains the pointers to the nodes. Note that the function is marked as shared, which means that it can be updated by the environment, or by the rules of OCLASM. This duality is required, because in general, constraints are defined outside the scope of OCLASM, but certain algorithms can modify the original constraints. For example an optimization algorithm can restructure the expressions of constraints in order to improve the performance.

Child(Position, I) is another shared function. It obtains the *I*th children *Phrase* of a complex *Phrase* (a *Phrase* which has children). The first parameter of the function is the (valid) position of the complex *Phrase*. If the *Phrase* does not have a child at the selected index, then the function returns *undef*. The function *Parent(Position)* implements the reverse direction: it obtains the position of a *Phrase* and returns the position of the container *Phrase*. These functions are shared functions for the same reason as *GetPhrase*. Note that *Child* and *Parent* functions are always synchronized automatically by the framework.

4.3 Dynamic Functions

Constructs of OCL, for example iterate or navigate, are mainly defined as rules in OCLASM. Dynamic functions help to store the current state of the evaluation environment when the rules are applied. For example, navigate operation selects a new model item, which is used as the origin of all further operations. Dynamic functions are just like helper variables in the environment framework.

To obtain the current position of evaluation, the position of the *Phrase* currently under execution, the function *CurrentPos()* is used. The return value of the function is a position from the universe *Pos*.

The dynamic function *Type(Pos)* is used to handle the type of the different (OCL) expressions uniformly. Its parameter is the position of the target expression. The return value of the function can be one of the basic types defined in OCL, such as real, integer, or tuple.

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

OCL allows the user defining local variables. In OCLASM, these variables are handled by the function *Local(Name)*. The function has one input parameter: the name of the variable. The function *Local* returns the position of a variable declaration expression. When defining a new local variable, then a new position is created using the *reserve*. Variable declarations contain the name, type and value of the variable. If a local variable is requested by its name and there is no local variable defined with the given name, then the function returns *undef*. The 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 the name of the local variable as a string, or 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, *Value(Position)[3]* means the third item in the collection: expression at the position *Position*. The arrays can be traversed by the *forall* expression of ASM, obtaining every element. According to the default notation of ASM [8] the length of collection is denoted as *l(Value(Position))*, where *Position* is the position of the expression.

Tuple types are also handled as arrays indexed by integer values, the definition of tuple items are stored in lists with two elements (name – value pairs), these lists

are list items of the array representing the tuple. For example, the expression `Tuple(x: Integer = 5, y: String = 'Ok')` results an array with two items: `TupleArray[1] = [‘x’, Integer = 5]`, while `TupleArray[2] = [‘y’, String = 'Ok']`. When a tuple item is queried by its name, then OCLASM tries to find an item in the associated array with the name and updates the *Type – Value* functions. In the previous example if the tuple item with name ‘y’ is requested, then OCLASM checks `TupleArray[1]`, but its name (‘x’) does not match, thus, it advances to `TupleArray[2]`. Since the name is found, OCLASM sets the *Type* of the current position to ‘String’ and the *Value* to ‘Ok’.

4.4 Vocabulary and Universes

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

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

- *Monitored functions: IsModelItem/1, AttrValue/1, Meta/1, To/2, Mul/2*
- *Shared functions: GetPhrase/1, Child/2, Parent/1*
- *Dynamic functions: CurrentPos/0, Type/1, Value/1, Name/1, Local/1*

Definition 5 *The superuniverse $|\mathfrak{A}|$ of a state $\mathfrak{A}$ of Σ_{OCLASM} is the union of six universes:*

- *The universe of Phrases (basic syntactic constructs of OCL)*
- *The universe of possible positions of Phrases in the constraints*
- *The universe Boolean (true/false/undef)*
- *The universe of finite lists of numbers*
- *The universe of finite lists of finite strings*
- *The universe of finite lists of possible identifiers (IDs) for model items and attributes*

4.5 Transition Rules

Transition rules describe how the states of OCLASM change over time by evaluating expressions and executing statements of the input program. The main idea is to create a rule for each type of language constructs available in OCL. For example OCLASM has rules for iterate, navigation, or variable declaration actions. These rules describe the semantics of the expression and manage dynamic functions.

OCLASM a central rule called *eval*. The rule *eval* is a rather complex rule, it has a switch-case block with many branches, more precisely for each type of language constructs, *eval* has a branch (see sketch of the rule below). It checks the type of the parameter *Phrase* and executes the appropriate branch by calling the rule associated with the *PhraseType*. Therefore, *eval* acts as a mapping function between *Phrase*s and rules of OCLASM. Moreover, *eval* helps calling the rules with the appropriate parameters (it adds sub-expressions as parameters, using the *Child* function). Updating the value of *CurrentPos* is also handled by *eval*.

```
1.  rule eval(Phrase)
2.  {
3.    CurrentPos():=Phrase;
4.    switch(PhraseType(Phrase))
5.    {
6.      case 'VariableDeclaration':
7.        VariableDeclaration(Child(Phrase,1), Child(Phrase,2),
8.                            Child(Phrase,3));
9.      case 'NavigationCall':
10.       Navigate(Child(Phrase,1), Child(Phrase,2);
11.     …
12.   }
13. }
```

The initial position of OCLASM sets the *CurrentPos* to the start position of the outermost constraint expression and sets the value of all other dynamic functions to *undef* for all possible parameters. A run of OCLASM is started by calling *eval* for the start position. When the run of the state machine of OCLASM is finished then the outermost expression holds a single value. If the evaluated constraint was an invariant, then this value shows whether the model was valid.

4.6 Invariants

OCLASM, as presented until this point is useful to simulate the execution of a simple or complex OCL expression, but not a whole constraint, such as an invariant. This definition is the core of OCLASM, but the presented approach can be extended in order to support OCL invariants or pre and post conditions. The current description shows how invariants can be formalized.

```
1.      rule CheckModelInvariants(Invariants)
2.      {
3.        forall(Invariant in Invariant)
4.        {
5.          forall (ID in {∀ID: ModelItem(ID)= true})
6.          {
7.            if (Invariant.Context) = Meta(ID))
8.            {
9.              CurrentPos():=Invariant.StartPosition;
10.             Local():=undef;
11.             if (not eval(CurrentPos())
12.             {
13.               print "The model is not valid.";
```

```
14.            exit;
15.          }
16.        }
17.      }
18.    }
19.  }
```

To have this rule as part of OCLASM, the universe of *Invariants* must be added to the superuniverse in Defintion 5. *Invariants* have a *Context* property and a pointer to the outermost expression in the invariant. The extended OCLASM presented in this section is referred to as $OCLASM_{Inv}$ to differentiate from the original OCLASM definition. Pre and post conditions can be handled similarly resulting a family of OCLASM formalisms.

4.7 Examples

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.

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.

```
1.    rule VariableDeclaration(VarName, VarType, VarInit)
2.    {
3.     Name(CurrentPos)  = Value(VarName)
4.     Type(CurrentPos)  = eval(VarType)
5.     if (VarInit!= undef)
6.        Value(CurrentPos) = eval(VarInit)
7.     else
8.        Value(CurrentPos) = undef
9.     endif
10.   }
```

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 the operation is applied; (ii) the declaration of the iterate variable, (iii) the declaration of the result variable, and (iv) the iteration body.

```
1.  rule iterate(CollectionDef, IteratorDecl, ResultDecl, Iteration)
2.  {
3.     eval(CollectionDef);
4.     eval(IteratorDecl);
5.     eval(ResultDecl);
```

```
6.      Local(Name(ResultDecl)):= new(Pos);
7.      Value(Local(Name(ResultDecl))):= Value(ResultDecl);
8.      Type(Local(Name(ResultDecl))):= Type(ResultDecl);

9.      forall collectionElement in Value(CollectionDef)
10.        Local(Name(IteratorDecl)):= new(Pos);
11.        Value(Local(Name(IteratorDecl)))= collectionElement
12.        Type(Local(Name(IteratorDecl))) = Type(IteratorDecl)
13.        eval(Iteration);
14.     endfor

15.     Local(Name(IteratorDecl))= undef;
16.     Value(CurrentPos) = Value(Local(Name(ResultDecl)));
17.     Type(CurrentPos)  = Type(Local(Name(ResultDecl)));

18.     Local(Name(ResultDecl))= undef;
19.  }
```

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 *To* always returns a list (with the IDs of the destination nodes), in the first case the first element of the result array is used (`To(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.

```
1.   rule Navigate(Origin, DestName)
2.   {
3.     eval(Origin);
4.     if ( Mul(Value(Origin),DestName)[2]==1 and
5.          Mul(Value(Origin),DestName)[3]==1 and
6.          IsModelNode(To(Value(Origin),DestName)[0]))
7.     {
8.        Value(CurrentPos)= To(Value(Origin),DestName)[0]
9.        Type(CurrentPos) = 'ModelItem'
10.    }
11.    else
12.    {
13.       Type(CurrentPos) = 'Set'
14.       Value(CurrentPos) = Set()
15.       forall ModelId in To(Value(Origin),DestName)
16.         Append ModelId in Value(CurrentPos)
17.       endfor
18.    }
19.  }
```

5 Application of the Formalism

5.1 The Relocation Algorithm

This section introduces how OCLASM can be used to prove the correctness of a dynamic algorithm working on OCL constraints. The *RelocateConstraint* algorithm is an optimization algorithm for OCL, it has been presented in [5] and in [6]. The main idea behind the algorithm is to reduce the time-consuming model queries, if it is possible. Therefore, the algorithm tries to relocate OCL invariants defined by the user to another context, where the evaluation requires the least model queries. Using OCLASM, it is possible to give a formal proof of correctness for the algorithm.

The algorithm applies the relocation always between adjacent nodes. In other words, the original and the new context of the constraint are always connected in the metamodel. This limitation does not restrict the optimization capabilities of the algorithm (as shown in [6]). Note that this does not mean that the nodes of the original context and the new context are connected *on the instance level* as well. This is because checking invariants is based on meta level (see algorithm X in section Y) and metamodels can allow zero multiplicity between the elements. Our research has shown [6] that the correctness of relocation is heavily affected by the multiplicities on the source and on the destination side.

The algorithm is defined as a rule, which runs before *eval* is called with the outermost position of the invariant. The algorithm modifies the constraints by updating the *GetPhrase* and *Child* functions. To simplify the rule it is worth defining formulas and helper rules:

```
φA.Dest(Phrase, IDA, Dest)  =  Value(eval(Child(Phrase,0))) = IDA and
                               Value(Child(Phrase,1)) = Dest

φA.x (Phrase, IDA, Dest)  =   Value(eval(Child(Phrase,0))) = IDA and
                              Value(Child(Phrase,1)) <> Dest
```

If the original context is A and the new context is B then $\varphi_{A.Dest}$ expresses that the *Phrase* is a navigation from A to B, while $\varphi_{A.X}$ expresses that the *Phrase* is a navigation from A to anywhere, but not to B. The formulas use the information that the first (0^{th}) child of navigations (to model nodes and attributes) is the origin, while the second (1^{st}) child is the name of the destination.

```
ForallCheck():=  ∃ModelPhrase: GetPhrase(ModelPhrase)<> undef and
                 (φA.x(ModelPhrase,0,D) or (φA.Dest(ModelPhrase,0,D)
                 and Parent(ModelPhrase).PhraseType<>'ForAll') )))
```

This derived helper function returns true, if (i) there is navigation/attribute call from A to x (x<>B), or (ii) there is a navigation from A to B and the outermost phrase in the constraint has a specific type ('ForAll').

$$\text{GetSrc(ID, Dest)} := \begin{cases} \text{Src: if } \exists ID_2\text{: } ID_2 \in \text{To(ID,Dest) and} \\ \qquad\qquad\qquad\qquad \text{ID} \in \text{To}(ID_2, \text{Src}) \\ \text{undef, otherwise} \end{cases}$$

This derived helper function obtains the name of the source side of navigations. For example there is a navigation between A and B, we can navigate from A to B using the destination name "B". This function obtains the reverse direction, namely to source name, which is used in navigation from B to A.

$$\begin{aligned} \text{Top(Phrase)} &:= \text{Phrase, if Parent(Phrase)=undef} \\ &\quad\ \text{Phrase}_{TOP}\text{, if } \exists \text{Phrase}_{TOP} : \\ &\qquad\qquad \text{Phrase}_{TOP} = \text{Top(Parent(Phrase))} \end{aligned}$$

This derived function tries to find the outermost *Phrase* of the constraint by using a recursive method.

```
1.   rule AddChild (Type,Parent,Idx)
2.   {
3.     Child(Parent, Idx):= new(Pos);
4.     GetPhrase(Child(Parent, Idx)):= new (Phrase);
5.     GetPhrase(Child(Parent, Idx)).PhraseType:= Type;
6.   }

1.  rule AddBackNavigation(Phrase, WithForallCheck)
2.  {
3.    if (WithForallCheck and (ForallCheck() and Mul(O,D)[1]>1))
4.    {
5.         AddChild('VariableCall',Phrase,0);
6.         AddChild('OrigSelf', Child(Phrase,0), 0);
7.    }
8.    else
9.    {
10.        AddChild ('NavigationCall', Phrase,0);
11.        AddChild ('SelfReference',Child(Phrase,0),0);
12.        AddChild ( GetSrc(O,D),Child(Phrase,0),1);
13.   }
14. }
```

These helper rules are used to automate the repeatable parts of the main relocation rule. The first rule creates a new *Phrase* with the given type and adds it as a child of the parameter *Phrase*. This rule is used when the algorithm inserts new expressions into the constraint. The rule *AddBackNavigation* inserts a complete set of *Phrase*s as a child of the parameter *Phrase*. The rule has two different modes according to the second parameter, the result of *ForallCheck* function and the multiplicity on the source side. The first mode creates phrases that call a variable with the name 'OrigSelf', while the second mode inserts phrases implementing a navigation call from the new context to the original context. The reason while the modes are differentiated is explained later.

```
1.  rule RelocateConstraint(O, D)

2.  if (Mul(O,D)[2]=0)
3.  {
4.    exit ("Relocation error.")
5.  }

6.  if (Mul(O,D)[0]=0)
7.  {
8.    AddChild ('IfExpr', undef ,0);
9.    AddChild ('IsEmpty', Child(undef,0) ,0);
10.   AddBackNavigation(Child(Child(undef,0), 0), false);
11.   Child(Child(undef,0), 1):= Top(ModelPhrase);
12.   Child(Child(undef,0), 2):= True;
13. }

14. if (Mul(O,D)[1]>1 and ForAllCheck())
15. {
16.   AddChild ('ForAll', undef ,0);
17.   AddBackNavigation(Child(undef,0), false);
18.   AddChild ('IteratorVariable', undef ,1);
19.   Value(Child(undef,1)):='OrigSelf';
20.   Child(Child(undef,0), 2):= Top(ModelPhrase);
21. }

22. forall (ModelPhrase in {∀Phrase: GetPhrase(Phrase)<> undef and
                     (Phrase.PhraseType='AttributeCall' or
                     Phrase.PhraseType = 'NavigationCall')})
23. {
24.     if (φA.Dest(ModelPhrase,0,D))
25.     {
26.        if (Mul(O,D)[3]>1)
27.        {
28.           if (Parent(ModelPhrase).PhraseType='ForAll')
29.           {
30.              GetPhrase(Parent(ModelPhrase)):=
31.              GetPhrase(Child(Parent(ModelPhrase),2));
32.           }
33.           else
34.              AddBackNavigation(ModelPhrase, true);
35.        }
36.        else
37.           ModelPhrase:= Child(ModelPhrase, 0);
38.     }
39.     else
40.        AddBackNavigation(ModelPhrase, true);
41. }
42. endrule
```

The rule is relatively complex; different cases are summarized in Table 1. The table shows the different cases based on the multiplicities on the source side (rows of the table) and on the destination side (columns of the table). The table summarizes the special constructs used in different cases, each cell has the corresponding line numbers for the constructs in the rule. The basic relocation case can be found at line 37 and 40 in the rule. Recall that the explanation why we distinguish these cases can be found in [6].

	<1	**1**	**>1**
<1	Not allowed (#2-5)	Add Filter (#6-13)	Add Filter, ForAll check (#6-13; #28-34)
1	Not allowed (#2-5)	No spec.	ForAll check (#28-34)
>1	Not allowed (#2-5)	Add ForAll (#14-21)	Add ForAll, ForAll check (#14-21; #28-34)

Table 1
Relocation overview

5.2 Proof of Correctness

The algorithm applied by the *RelocateConstraint* rule is correct, if none of the constructs (modification of the constraint) modifies the result of validation, i.e. if the set of valid models remains the same. Therefore, the presented proof shows that the steps of the rule are correct by mapping expressions of the original constraint to expressions in the new constraint and show that they are equivalent. The proof uses the *CheckModel* rule from section 4.6. For sake of simplicity, the condition expression at line #11 in the algorithm is referred to as 'ValidModelCheck'. We refer to sections of the rules as Rule(#From-To), for example RelocateConstraint(#2-5). Note that the aim of the proving method is to show the equivalency of the original and the relocated constraint, therefore, it is not stressed whether the rule RelocateConstraint is optimal.

To distinguish the original and the relocated constraints we use the labels 'old' and 'new', for example C_{old} (that is the original constraint), or $self_{old}$ (self reference of the original constraint) Furthermore, for sake of simplicity we suppose that the original context type was A, while the new context type is B. The instantiations of type A are a_1, $a_{2 \ldots} a_n$ (collected in a set $Inst_A$), the instantiation of type *B* are b_1, $b_{2\ldots} b_n$ (collected in the set $Inst_B$). *B* can be reached from *A* using the name *Dest* (**To(a_i,Dest)=b_j**) while the reverse direction uses *Src* (**To(b_i,Dest)=a_j**).
If the model allows zero multiplicity, then one of the following formulas are true:

$\varphi_{ZeroOnSrc}$: **Mul(A,Dest)[0] = 0**
$\varphi_{ZeroOnDest}$: **Mul(A,Dest)[2] = 0**

We prove that the algorithm is always correct in these cases, then we formalize other possible cases as follows:

$\varphi_{ExactlyOne}$: **Mul(A,Dest)[1]=1 ∧ Mul(A,Dest)[3]=1**
$\varphi_{ManyOnSrc}$: **Mul(A,Dest)[1]>1 ∧ Mul(A,Dest)[3]=1**
$\varphi_{ManyOnDest}$: **Mul(A,Dest)[1]=1 ∧ Mul(A,Dest)[3]>1**
$\varphi_{ManyBoth}$: **Mul(A,Dest)[1]>1 ∧ Mul(A,Dest)[3]>1**

If the constraint does not contain any model query, then the constraints results in a simple Boolean value true, or false. The result does not depend on the underlying model, thus it is *always* true, or false.

If this constant result is true, then it means that all possible models are valid. This means that ValidModelCondition is always satisfied, thus, the inner *forall* expression at CheckModel(#5-17) can be replaced by a constant true value. Therefore C_{new} always evaluates to true as well.

If this constant result of the constraint is false (no model is valid, which contains a model item of the selected type), then $\text{ValidModelCondition}_{Old}$ is not satisfied for instances of *A*, while $\text{ValidModelCondition}_{New}$ is not satisfied for instances of *B*. Therefore, the relocation of the constraint is not correct if Inst_A, or Inst_B is empty.

Inst_B can be empty only if formula $\varphi_{\textbf{ZeroOnDest}}$ holds (zero multiplicity is allowed on the destination side). This case is handled at RelocateConstraint(#2-5): the condition checks whether for each a_i there exists at least one b_i, more generally that there is a b_i in the model, which can be used as a host for the constraint (host nodes of a constraint are nodes, in which the constraint is defined). If not, then the rule throws an error message.

Similarly, Inst_A can be empty if formula $\varphi_{\textbf{ZeroOnSrc}}$ holds. The problem is solved by the condition at RelocateConstraint(#6-13), which ensures that the constraint is evaluated only to those b_js which are connected to at least one a_i (otherwise it returns with a constant true). Thus, if Inst_A is empty, then the constraint is not evaluated.

As result of the conditions at RelocateConstraint(#2-13), the following formulas *always* hold:

$\varphi_{\textbf{AToB}}$: $\forall$i: Meta(i)=A $\rightarrow$ ($\exists$j, $\exists$Dest: Meta(j)=B $\wedge$ j$\in$To(i, Dest))

$\varphi_{\textbf{BToA}}$: $\forall$i: Meta(i)=B $\rightarrow$ ($\exists$j, $\exists$Src: Meta(j)=A $\wedge$ j$\in$To(i, Src))

Note that these formulas ensure also that the relocation of constraints without model queries is always correct.

If the constraint contains model queries, then the result of eval at CheckModel(#11) can be affected by the navigations and attributes of a_i. The relocation is correct if the result of model queries in the original context is the same as the result in the new context. Different cases are indexed by the formulas defined above:

$\varphi_{\textbf{ExactlyOne}}$**:** RelocateConstraint(#37) and RelocateConstraint(#40) are used. RelocateConstraint replaces navigations from A to B with a self reference. Instead of self_{Old}.Dest the new constraint has a self_{New} expression (RelocateConstraint(#40)). According to the rules of OCLASM, the value of self_{Old}.Dest is retrieved by the monitored function call To(a_i,"Dest"), which results b_j, an instance of B. Since formula φ_{AtoB} holds, thus this b_j always exists, therefore

the expression To(a_i,"Dest") can be replaced by the value b_j for this certain navigation expression for this certain a_i. However, the constraint is cheked against all a_is (CheckModel(#7)), thus To(a_i,"Dest") is always replacable by an appropriate b_j. In the relocated constraint this replacement is done by using the new self reference. Note that relocation does not create false host nodes (nodes which could not affect the resul of the original constraint, but could affect the result of the relocated version) for the constraint (due to $\varphi_{BtoA.}$).

RelocateConstraint inserts a navigation expression from B to A (RelocateConstraint(#37)) before any navigations/attribute queries from A to anywhere, but not B. The expression $self_{Old}$.x (where x is not "Dest") is replaced by $self_{New}$.Src.x. Proving is similar to the previous case, but on the reverse direction. Since the formula φ_{BtoA} holds, thus $self_{New}$.Src which results in a To(b_i, "Src") can be replaced by the value a_j for this certain navigation/attribute expression for this certain b_i. Moreover, the new constraint is cheked against all occurences of b_i (CheckModel(#7)), thus To(b_i,"Src") is always replacable by an appropriate a_j. Therefore, the value of $self_{Old}$ always equals with the value of $self_{New}$.Src. Note that relocation does not delete host nodes because of φ_{AtoB}.

$\varphi_{ManyOnDest}$: The section RelocateConstraint(#28-34) and RelocateConstraint(#40) are applied. Correctness of navigations of the form $self_{Old}$.x, where x <> "Dest" is ensured by RelocateConstraint(#40) according to constructs used in $\varphi_{ExactlyOne}$. However, the result of navigations from A to B results in a set of b_is. The condition at RuleConstraint(#28) checks whether the result of the navigation expression is used in a *forall* construct, if so, then the original expression is replaced by the inner expression of *forall*. Thus, $self_{Old}$.Dest $\rightarrow$ *forall*(Exp_{Old}) is transformed into Exp_{New}. In OCL, *forall*(Exp) is a set operation, which is true only if Exp is satisfied for each item in the set. Therefore, the original expression $self_{Old}$.Dest $\rightarrow$ *forall*(Exp) is true for a certain a_i, if Exp is true for each $b_j \in$ To(a_i, Dest). This means that the original expression in a certain a_i can be replaced by Exp_{New} in $b_j \in$To(a_i, Dest). The original constraint is evaluated for each a_i, thus the replacement is correct in each b_j connected to one of the a_is. Moreover, since φ_{BToA} holds, thus, b_js are always connected to an a_i. This means that the relocation is always correct in this case.

If the condition at RelocateConstraint(#28) is not satisfied, then replacement inserts a navigation back to the original constraint and the expression is evaluated there. The expression $self_{Old}$.Dest is replaced by $self_{New}$.Src.Dest. Because of φ_{BToA} this replacement is always correct as shown constructs used in $\varphi_{ExactlyOne}$.

$\varphi_{ManyOnSrc}$: The section RelocateConstraint(#14-21) and RelocateConstraint(#38) are applied. Correctness of navigations of the form $self_{Old}$.Dest is granted by RelocateConstraint(#38) according to constructs used in $\varphi_{ExactlyOne}$. However, navigations from B to A results in a set of a_is. The condition at RelocateConstraint(#14) checks whether such navigation is required (using the rule ForAllCheck). Backward navigation to the original constraint is required in

the case of $self_{Old}$.x expressions (where x <> "Dest"), or if Mul(A,Dest)[1]>1 according to constructs used in $\varphi_{ManyOnDest}$. Here only the first case is possible, which is handled by adding a *forall* expression to the constraint during relocation (RelocateConstraint(#15-21)). The new *forall* expression encapsulates the whole constraint and it simulates the backward navigation using the iteration variable 'OrigSelf', where navigation expressions are replaced by variable calls in AddBackNavigation(#5-6). Therefore, the constraint … $self_{Old}$.x … is transformed into $self_{New}$.Src→*forall*(OrigSelf | … OrigSelf.x …).

For a certain expression and b_j, the replacement is correct for a_is connected to b_j. Since φ_{AtoB} holds, thus, each a_i is connected with at least one b_j and evaluation checkes each b_j of the model, thus, the replacement is correct for all a_is of the model. Note that the outermost *forall* expression ensures that different navigation/attribute calls of the constraint are using the same a_i (since the value of OrigSelf does not change), thus the attributes/navigation of a_is are not mixed up.

$\varphi_{ManyOnBoth}$: RelocateConstraint(#14- 21) and RelocateConstraint(#28-34) are used. Firstly an encapsulating *forall* expression is added if there is a navigation/attribute call of form $self_{Old}$.x (where x<>"Dest") or a $self_{Old}$.Dest expression without *forall*. The construction from $\varphi_{ManyOnSrc}$. Secondly, the *forall* expressions of the original constraint are replaced according to constructs used in $\varphi_{ManyOnDest}$.

The replacement of the expression $self_{Old}$.x (x<>"Dest") in a certain a_i is correct for each $b_j \in To(a_i, Dest)$ according to constructs used in $\varphi_{ManyOnSrc}$ Moreover, the replacement of these type of expressions is also correct in general (because of φ_{AtoB}). However, it is possible that the expression is evaluated in a certain a_i more then once (more precisely once for each element of $To(a_i, Dest)$).

The replacement of the expression $self_{Old}$.Dest →*forall* is correct according to constructs used in $\varphi_{ManyOnDest}$. Multiplicity 'MoreThanOne' on the source side does not affect this correctness, because the updated expressions use navigations only from context B, thus, it is not important how many a_is are connected with the current b_j.

This is not the case with expressions of form $self_{Old}$.Dest->Exp, where Exp is not *forall*. Here navigation back to the original context is mandatory, thus the original expressions is transformed into $self_{New}$.Src.Dest->Exp. Relocation is correct in this case if the value of $self_{Old}$ can be replaced by $self_{New}$.Src. However, the function *ForAllCheck* returns true at the condition at RuleConstraint(#14), which means that the constraint is encapsulated by a new *forall* as in $\varphi_{ManyOnSrc}$. This *forall* expression ensures that for a certain b_j, the constraint is evaluated for all items of the set $self_{New}$.Src seperately (for all a_is connected with b_j). Moreover φ_{BtoA} holds, which means that all a_is are checked by the relocated constraint. This means that the relocation is correct in this case as well.

The possible multiplicity combinations were tested, we have proved that the RelocateConstraint rule is correct in all cases.

Conclusions

Textual constraints are useful in order to extend visual model definitions and create precise models. OCL is one of the most popular textual constraint language, it is used to provide precise, unambiguous definitions in several modeling techniques such as UML, or metamodeling techniques in general. 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 describes the syntax and semantics of OCL and it can prove the completeness of the models using OCL, but it does not contain the definition of constraint evaluation, dynamic behavior of the constraint expressions. Due to this limitation of the OCL formalism, it cannot be used to prove the correctness of dynamic, OCL manipulating algorithms, for example our optimization algorithms.

This paper has presented OCLASM, a new formalism for OCL. The paper has presented the main reasons, why a new formalism was created instead of extending the original formalism, or one of its extensions. The new formalism is based on the popular ASM technology, it can be used to study the dynamic 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 and constraint expression retrievals use monitored (external) functions showing that constraint validation must be independent from the current model and constraint 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. The paper has shown how to use OCLASM in order to define RelocateConstraint algorithm (which is used in constraint optimization) and how to prove its correctness formarly. Future work mainly consists of continuing this work and prove the correctness of other optimization algorithms as well.

Acknowledgement

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

References

[1] L. Lengyel, T. Levendovszky, H. Charaf: Constraint Handling in Feature Models, 5th International Symposium of Hungarian Researchers on Computational Intelligence, 2004

[2] L. Lengyel, T. Levendovszky, G. Mezei, B. Forstner, I. Kelényi, H. Charaf: Model-based System Development for Embedded Mobile Platforms, ECBS, pp. 43-52, 2006

[3] J. Warmer, A. Kleppe: Object Constraint Language, The: Getting Your Models Ready for MDA, Second Edition, Addison Wesley, 2003

[4] G. Mezei, L. Lengyel, T. Levendovszky, H. Charaf: Extending an OCL Compiler for Metamodeling and Model Transformation Systems: Unifying the Twofold Functionality, INES 2006

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

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

[7] VMTS Web Site, http://vmts.aut.bme.hu

[8] E. Börger, R. Stärk: Abstract State Machines: A Method for High-Level System Design and Analysis, Springer-Verlag, 2003

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

[10] S. Flake: Towards the Completion of the Formal Semantics of OCL 2.0 ACSC, pp. 73-82, 2004

[11] S. Flake, W. Müller: An ASM Definition of the Dynamic OCL 2.0 Semantics, UML 226-240, 2004

[12] R. F. Stärk, J. Schmid, E. Börger: Java and the Java Virtual Machine: Definition, Verification, Validation, Springer Verlag, 2001

[13] G. Mezei, T. Levendovszky, H. Charaf: An Attribute Transformation Technique For N-Layer Metamodeling, Microcad, 2007

[14] UML 2.0 Specification http://www.omg.org/uml/

[15] G. Mezei, T. Levendovszky, L. Lengyel, H. Charaf: Multilevel Metamodeling - A Case Study, MicroCAD, March 10-11, 2005, Miskolc, pp. 321-326