Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 304 (2014) 81–96

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Towards a $\mathbb{K}$ Semantics for OCL

Andrei Arusoaie Dorel Lucanu

Department of Computer Science
Alexandru Ioan Cuza University
Iaşi, Romania

Vlad Rusu

Inria Lille Nord Europe
Villeneuve d'Ascq, France

Abstract

We give a formal definition to a significant subset of the Object Constraint Language (OCL) in the $\mathbb{K}$ framework. The chosen subset includes the usual arithmetical, Boolean (including quantifiers), and string expressions; collection expressions (including iterators and navigation); and pre/post conditions for methods. Being executable, our definition provides us, for free, with an interpreter for the chosen subset of OCL. It can be used for free in $\mathbb{K}$ definitions of languages having OCL as a component We illustrate some of the advantages of $\mathbb{K}$ by comparing our semantical definition of OCL with the official semantics from the language's standard. We also report on a tool implementing our definition that users can try online.

Keywords: Object constraint language, Formal executable semantics, $\mathbb{K}$ semantic framework

1 Introduction

The Object Constraint Language (OCL) is a textual language for writing constraints over UML models. It has been designed at IBM in the mid nineties, and has been incorporated in the UML standard starting from UML version 1.1.

OCL is intended to be a formal specification language. It is used for adding precision that UML models lack. Its OMG standard [15] defines the syntax of the language, and, to some extent, its semantics. However, despite many academic works on OCL (some of which are referenced in the Related Works section) there is curently no commmercial tool that offers full support for it.

[1] Email: andrei.arusoaie@gmail.com
[2] Email: dlucanu@info.uaic.ro
[3] Email:Vlad.Rusu@inria.fr

http://dx.doi.org/10.1016/j.entcs.2014.05.004

Part of the problem is the standard, which is imprecise, incomplete, and flawed in some places. This is common to many languages for which the semantics is informally described in manuals declared as standards (see, e.g., the definition of C [9]). Some of these problems include the nondeterministic cast operations from unordered to ordered sets and bags, the question of whether a singleton set $\{a\}$ and the set element a should be distinct or not (answered differently in different pages of the standard [15]), and issues due to the presence of an *invalid* value.

In this paper we report on work in progress towards a formal semantics of OCL in the $\mathbb{K}$ framework [22]. $\mathbb{K}$ is a semantical framework mainly intended for defining formal operational semantics for programming languages. $\mathbb{K}$ semantics is executable, meaning that programs in the defined languages can be executed, tested, and in the near future, formally verified [4].

We have defined both *static* parts of OCL (corresponding to well-formedness constraints and queries on UML models) and *dynamic* parts (corresponding to pre/post conditions of methods). Moreover, we allow for evaluating OCL expressions on *symbolic* models, i.e., models which attributes may have symbolic values [5]. This allows us, for example, to compute the condition (on the variables) under which a concrete instance of a symbolic model, satisfiying given OCL constraints, exists; and to compute the conditions under which a given sequence of method calls is feasible. These conditions can then be passed to constraint solvers to check whether they are satisfiable. A negative answer is a useful feedback to users: they need to revise their models and OCL expressions in order to make them consistent with each other.

Regarding the static part, the defined OCL fragment consists of the usual arithmetical, Boolean (including quantifiers), string operations, and collection operations (including navigation via UML associations and iterators), along with let-in and if-then-else constructs. Expressions may have a scalar type (integer, Boolean, string, or UML classes) or a collection type (bags or sets). This is a significant fragment of OCL, allowing users to write most of the useful well-formedness constraints on UML models, and which we may extend in the future when new OCL standards decide on some open issues and fix erroneous ones (such as "ordered sets" with an unkown order). Regarding the dynamic part, all pre/post condition constructions using the defined static part are also defined, as well as the specific constructions for postconditions (for referring to values in the pre-condition and to returned values).

The main advantages of $\mathbb{K}$ formal semantics definitions are their expressiveness, executability, and modularity. Each OCL construction is typically defined using one or two $\mathbb{K}$ rules. We have benefited a lot from $\mathbb{K}$ features such as the syntactical substitution, the automatic generation of $\mathbb{K}$ rules for evaluating argument of *strict* operations, and the automatic context transformers, which require users to only provide $\mathbb{K}$ rules with minimal information for matching and rewriting.

We illustrate these advantages by comparing our $\mathbb{K}$ semantics of the let-in OCL operation with its official semantics from the *Annex A : formal semantics (informative)* of the OCL standard [15]. We also show how the symbolic OCL evaluation is obtained

[4] The language-independent *matching logic* has been defined [20] and is being implemented.

[5] To our best knowledge ours is the only approach dealing with symbolic evaluation of OCL.

in a modular way, just by extending the OCL data types with symbolic values, without changing anything in the $\mathbb{K}$ semantic rules of OCL. Moreover, the $\mathbb{K}$ modules implementing symbolic evalation that we developed OCL can be used for other language definitions as well.

The rest of the paper is composed as follows: Section 2 provides some background on UML, OCL, and the $\mathbb{K}$ framework. In Section 3 we present our $\mathbb{K}$ definition of OCL and illustrate it on examples. Section 4 concludes and discusses related and future work . An online tool implementing our definition is available [1].

2 Background

2.1 UML and OCL

In this section we recall a few notions regarding UML and OCL. We consider a minimal notion of UML *class diagrams*, consisting of: a set of classes with attributes and method declarations (i.e., method names with parameter and return value types); of unidirectional references (or associations) between classes, whose roles (or ends) have $[0..*]$ multiplicities; of a generalisation partial-order between classes; and of a set of enumerations. We assume that these concepts are known; more information can be found in [17]. Other features of class diagrams (bidirectional referencs, roles with multiplicities other than $[0..*]$, composition and aggregation associations. . .) are not considered since they do not add any expressiveness - they can be equivalently encoded using the basic constructions and OCL constraints as shown in [10].

We require that attributes have a basic type (*Integer*, *Real*, *Boolean*, *String*, *Enumeration*), and that associations have collections-of-objects types only. This excludes the possibility for an attribute to be, e.g., a list of integers; however, this is not a limitation, since basic types can be wrapped in classes and attributes can equivalently be transformed to references to those classes. We also exclude the "ordered" qualification allowed by UML class diagrams for association ends, because its semantics is underspecified (it generates "ordered sets" with an unspecified order).

UML *object diagrams* are instances of class diagrams. They consist of objects (instances of classes, with values for the attributes) connected by links (instances of associations). We also assume that these concepts are known.

Figure 1 shows an example of a class diagram and of an object diagram that instantiates it. The class diagram is meant to model the situation where *papers* are authored by *researchers* (some of which are *professors*, and others are *PhD students*), and papers may be submitted to *journals*. Researchers and journals have string-valued *names*, while papers have string-valued *titles* and Boolean *isSubmitted* flags indicate whether they are submitted or not. Papers also have a *submit()* method, with an empty list of parameters and void return type. The pre- and post-conditions of this method are written in a comment next to the *Paper* class.

Professors have an attribute *post*: a value from the *Position* enumeration containing the three usual academic ranks. PhD students are in a *year* of study, a (theoretically, unbounded) integer. There are also several associations (*manuscript*,

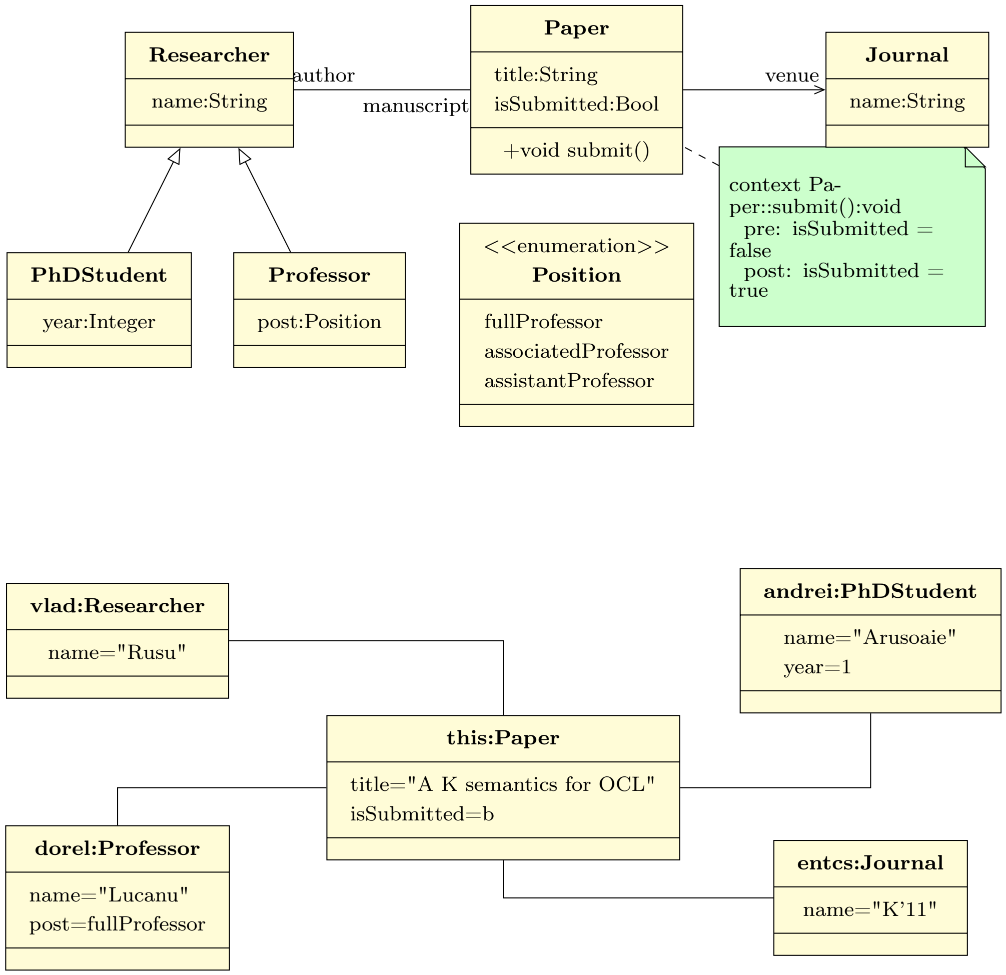

Fig. 1. Examples of: class diagram (top) and object diagram (bottom).

author, *venue*) between the classes of the diagram, some of which are bidirectional and are syntactical sugar for unidirectional ones with OCL constraints (shown below).

The object diagram at the bottom of Fig. 1 is an instance of the class diagram at the top. Note that the Boolean attribute *isSubmitted* has a symbolic value, denoted by the variable b. That is, our object diagram is *symbolic*, and it denotes all (concrete) object diagrams obtained by instantiating the variables by type-consistent values.

2.1.1 OCL Constraints and Queries

OCL constraints and queries are defined over UML class diagrams and are evaluated over UML object diagrams of those class diagrams. The following constraints can be defined on the UML class diagram in Figure 1:

- every paper has at least one author and is linked to at most one venue: $Paper.allInstances() \rightarrow forAll(p \,|\, p.author \rightarrow size() \geq 1 \wedge p.venue \rightarrow size() \leq 1)$;
- if a paper is submitted, then it is submitted to exactly one venue:

$Paper.allInstances() \rightarrow forAll(p \mid p.isSubmitted\ implies\ p.venue \rightarrow size() = 1)$;

- the *author* and *manuscript* associations form a bidirectional association. This means that for each paper, by taking each of its authors and then computing that author's manuscripts, one obtains a set of papers that includes the paper we started with; and a similar constraint holds for researchers:

$$\begin{aligned}&Paper.allInstances() \rightarrow\ forAll(p \mid p.author \rightarrow forAll(r \mid \\ &\qquad r.manuscript \rightarrow includes(p))) \qquad \wedge \\ &Researcher.allInstances() \rightarrow forAll(r \mid r.manuscript \rightarrow forAll(p \mid \\ &\qquad p.author \rightarrow includes(r))).\mathrm{w}\end{aligned}$$

These constraints are satisfied by the object diagram at the bottom of Figure 1.

2.1.2 OCL Pre and Post Conditions for Methods

OCL also offers the possibility of annotating methods with pre/post conditions. As usual, a method's pre-condition is required to be satisfied before the method is called, and its postcondition is guaranteed to hold when the method completes. Pre-conditions can be any static OCL constraints, and refer to the state of an object diagram before the method is called. Postconditions can also be any static OCL constraints, and refer to the state of an object diagram after the method is called, with the exception of class fields prefixed by the annotation *@pre*, which refer to the pre-state. The keyword *return* refers to the return value of the method, if any.

The pre and post conditions for the method *submit()* are respectively *isSubmitted = false* and *isSubmitted = true*. Thus, on the symbolic model shown at the bottom of Figure 1, it should not be possible to call *submit()* twice on *this:Paper*. We use this simple example for demonstrating symbolic evaluation and pre/post condition execution, and show in Section 3 how these mechanisms are specified in $\mathbb{K}$.

2.2 The $\mathbb{K}$ framework

$\mathbb{K}$ [22] is a framework mainly intended for defining and analysing the semantics of programming languages. The main characteristics of $\mathbb{K}$ include:

- *expressiveness*: all programming language paradigms and all their specific features are described in a simple and uniform way;
- *modularity*: each language feature is described once and for all, hence the definitions are scalable;
- *executability*: the definitions are directly executable in order to be experimented with and analysed;
- *support for program reasoning*: the framework servea as a program logic with which the programs can be verified and analysed (see, e.g., [20]).

A $\mathbb{K}$ definition for a language has the following main ingredients: a *syntax*, which is a context-free annotated grammar of the language and from which the *computation tasks* are extracted; the definition of the *values* over which the language computes; a *configuration*, which is a structure of nested cells abstracting the state structure of the machine on which the programs are executed; and $\mathbb{K}$ *rules*, which describe the execution steps using minimal information (only what is needed for matching and rewriting).

The $\mathbb{K}$ tool [13], the current prototype of $\mathbb{K}$, takes as input a $\mathbb{K}$ definition and translates it, using several complex algorithms, into a Maude [6] program. Then, the Maude program can be used for automatically execute and/or model check $\mathbb{K}$ definitions [14].

3 A $\mathbb{K}$ definition for OCL

In this section we show how OCL is defined in $\mathbb{K}$, by taking the following steps:

- defining the syntax of OCL using an annotated context-free grammar;
- defining the values that OCL expressions can evaluate to;
- defining the $\mathbb{K}$ configuration for OCL;
- defining the $\mathbb{K}$ rules for the semantics of OCL.

3.1 Syntax

The annotated grammar of our $\mathbb{K}$ definition of OCL is shown in Appendix A. We currently support the usual integer, real, and Boolean expressions (including Boolean expressions with quantifiers) ; strings and concatenation; collection expressions, including navigation, and iterators; and other expressions (e.g., let-in, if-then-else). Since the $\mathbb{K}$ definitions are modular, we may add new expressions without changing the old ones. Note that OCL is an evolving language and the $\mathbb{K}$ framework can help in experimenting the new constructs introduced in the language.

The annotation strict following some productions specifies which arguments must be evaluated before the evaluation of the defined operator (see [13] for details). We shall see that this mechanism is very useful, as it relieves the user from writing $\mathbb{K}$ rules for evaluating a construction's arguments: those rules are automatically generated by the $\mathbb{K}$ tool from the strictness annotations.

3.2 OCL Standard Library

The OCL Standard library [15] includes (see Figure 2):

- a set of primitive types: *Bool*, *Integer*, *Real*, and *String*;
- collection types, which are parametric in the type T: $Bag\langle T\rangle$, $Set\langle T\rangle$, $OrderedSet\langle T\rangle$, and $Sequence\langle T\rangle$;
- a super-type *OclAny* such that any UML type or collection is a subtype of it;
- a "minimal" type *OclVoid* that is a subtype of any UML type or collection excepting *OclInvalid* and that has a unique value *null*;
- a "minimal" type *OclInvalid* that is a subtype of any UML type and collection excepting *OclVoid* and that has a unique value *invalid*.

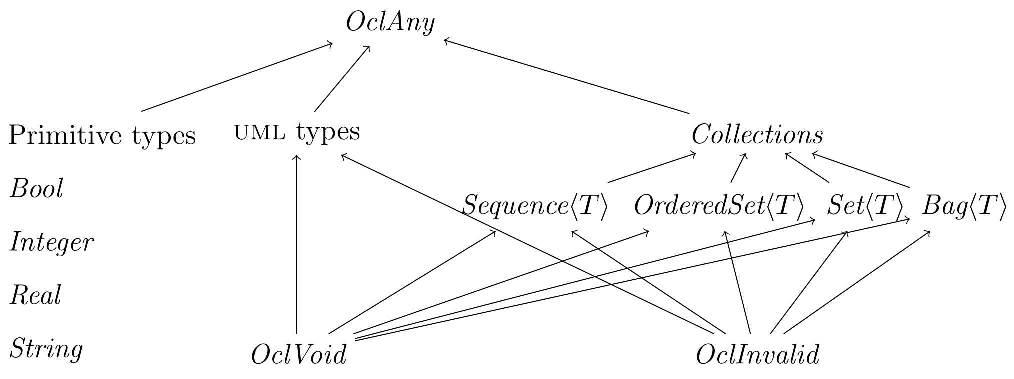

Fig. 2. OCL types

The OCL primitive types are implemented using the 𝕂 built-in types (see Figure 3) *Bool*, *Int*, *Float*, and *String*. The values of these types are called *scalars*. We also use the 𝕂 built-in type *Id* as support for the enumerative OCL types. We have extended all the OCL primitive types with *symbolic values*. In our setting, symbolic values have the form *symBool(x)*, *symInt(x)*, . . . , where the argument x is of sort *Id*.

This simple mechanism enables the evaluation of OCL expressions on symbolic models, without changing the 𝕂 semantical rules of OCL. This is due to the fact that 𝕂 semantics is modular and the same definition can be executed on various data domains with the same interface.

We also defined a 𝕂 module that extends the above built-in types with the following specific data types:

- a type *Object* as a super-type for all UML types. The values of this type are pairs *typedElt*(o, C), where o is an object reference and C is a class name such that the object referenced by o is an instance of *type(C)* (= the type of class C).
- a type *Magma* as a super-type for collections, e.g., *Bag*$\langle T\rangle$ and *Set*$\langle T\rangle$. Union is an operation denoted by _; _ : $Magma \times Magma \to Magma$.

  Objects are elements of an object diagram, and collections are, in our present definition, lists of scalars having one common (super)type. This means that, in our running example, collections of professors and researchers are allowed because *Professor* is a subtype of *Researcher*, but not collections containing of papers and journals because these belong to different types.
- a type *Void* with a singleton value *null*, used to implement *OclVoid*.

We do not implement the type *OclInvalid*. Considering *invalid* as a value leads to complications since it requires specific rules for dealing with it. In our 𝕂 definition, *invalid* is *not* a value: rather, it is defined by the fact that an OCL expression does not reduce to a value by the semantical rules. Thus, the implementation of the operation *oclIsUndefined* is unnecessary. An example for this is given in Section 3.4.

We do not directly support the ordered collections *Sequence*$\langle T\rangle$ and *OrderedSet*$\langle T\rangle$. These collections can be modelled by class diagrams such as that given in Figure 4. The steoreotype ⟨⟨singleton⟩⟩ says that the class *EmptySequence* has just one instance

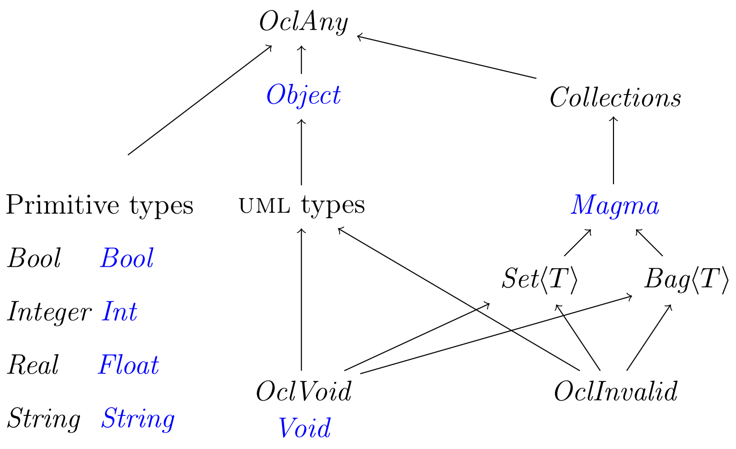

Fig. 3. 𝕂 implementation of the OCL types. The 𝕂 types are shown in color.

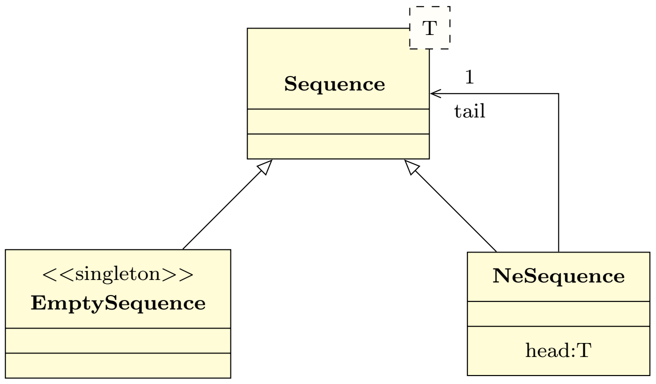

Fig. 4. A diagram for *Sequence*⟨*T*⟩

representing the empty sequence. This is actually a more precise modelling, since, unlike ordered collections, our diagrams specify an order. If needed, instances of such diagrams can be automaticaly generated from user-defined models containing ordered collections by choosing an arbitrary order.

3.3 Configuration

The 𝕂 configuration for OCL consists of five main cells:

- A cell $\langle \_ \rangle_{\mathsf{k}}$ containing the computation tasks sequence; for instance, it includes the computation tasks derived when the OCL expressions are evaluated.
- A cell $\langle \_ \rangle_{\mathsf{metamodel}}$ storing the information about the metamodel: name, enumerative types and classes. For instance, the class Paper described on Page 4 is represented by a cell structure as follows:

```
<class>
  <classname> Paper </classname>
  <constr> . </constr>
  <extends> . </extends>
  <methods>
   <method>
     <methodname> submit </methodname>
     <modifies> isSubmitted  </modifies>
     <post> self.isSubmitted = true </post>
     <pre>  self.isSubmitted = false </pre>
     <returntype> void </returntype>
   </method>
  </methods>
  <attributes>
    isSubmitted |-> bool
    title |-> string
    authors |-> collection
  </attributes>
  <references> . </references>
</class>
```

Note that association relations are represented by attributes (e.g., authors).

- A cell $\langle \_ \rangle_{\mathsf{constraints}}$ storing the static semantics of the metamodel represented as a list of OCL constraints.
- A cell $\langle \_ \rangle_{\mathsf{model}}$ containing the $\mathbb{K}$ representation of the object diagram on which the OCL expressions in the $\mathbb{K}$ cell will be evaluated. An instance of class Paper is represented as follows:

```
<instance>
  <instName> paper </instName>
  <ofClass> Paper  </ofClass>
  <fields>
    isSubmitted |-> typedElt( symBool(b) , bool )
    title |-> typedElt( "A K semantics for OCL" , string )
    authors |-> typedElt(vlad, Researcher) ;
                typedElt(dorel, Professor) ;
                typedElt(andrei, PhDStudent)
  </fields>
</instance>
```

  where the attribute isSubmitted has a symbolic value, b.

- A cell $\langle \_ \rangle_{\mathsf{output}}$, which is linked to the standard output and is used for displaying results or error messages.

3.4 Semantics

Evaluating OCL expressions on object diagrams is done according to the semantics of OCL, which we now define as a set of $\mathbb{K}$ semantical rules. We start with simple rules for evaluating the basic arithmetic and Boolean operations.

The rule for division, shown below, reads as follows: if the top of the k cell is a division expression $A\,/\,B$, where A and B are of sort Int, then that expression is rewritten in-place with the value obtained by taking the $\mathbb{K}$ builtin integer division $A\,/_{Int}\,B$, provided that the rule's condition $\mathsf{not}\,(B =_{Bool} 0)$ holds:

$$\langle \underset{A\,/_{Int}\,B}{\underline{A\,/\,B}} \;\cdots\rangle_{\mathsf{k}} \;\; \mathsf{if\ not}\,(B =_{Bool} 0)$$

Hence, expressions of the form A / 0 are not reduced by the above rule to a legitimate OCL value (here, an integer). Such expressions model the *invalid* value of the OCL standard [15]. The advantage of our definition with respect to the standard is that we do not need to give explicit rules saying how the invalid value is propagated: expressions that contain an undefined value (i.e., a subexpression that is not reduced to a value) are also undefined, because they are not reduced to a value.

Most remaining arithmetic and Boolean operations (Appendix A) have similar rules. Below we show the rules for the forAll quantifier; the rule for the exists quantifier is obtained by duality. The $\mathbb{K}$ semantical rules for forAll are:

$$\langle \underset{\mathsf{true}}{\underline{(\,\cdot\,)\to\mathsf{forAll}(\,\mathit{Var} \mid \mathit{Exp})}} \;\cdots\rangle_{\mathsf{k}} \tag{1}$$

$$\langle \underset{\mathsf{if}\ \mathit{Exp}[\mathit{Elt}/\mathit{Var}]\ \mathsf{then}\ \mathit{Rest}\to\mathsf{forAll}(\,\mathit{Var} \mid \mathit{Exp})\ \mathsf{else\ false}}{\underline{(\mathit{Elt}, \mathit{Rest})\to\mathsf{forAll}(\,\mathit{Var} \mid \mathit{Exp})}} \;\cdots\rangle_{\mathsf{k}} \tag{2}$$

The first rule describes the base case, when the first argument is an empty collection. The second rule describes the inductive step: when the first argument is a nonempty collection of the form (Elt, Rest), the result depends on the value of the expression Exp on the element Elt. This is computed by applying the $\mathbb{K}$ library's *substitution operation* _[_/_] to Exp, Elt, and Var. If the value is true, then the overall result is that of the forAll operation, recursively evaluated on the collection Rest; otherwise, the result is false.

This relatively simple definition is possible due to several powerful mechanisms of $\mathbb{K}$. One of them is the substitution operator, which is actually implemented in $\mathbb{K}$ itself using a *visitor pattern* [22]. We shall come back to the substitution operator later in this section when we define the semantics of the let-in expression and compare it with the corresponding semantics in the OCL standard [15]. The other $\mathbb{K}$ mechanism we use extensively here is the automatic generation of rules that evaluate arguments of strict operators.

Remember (cf. Appendix A) that forAll is strict in its first argument. This means that this argument will be evaluated before the forAll expression is evaluated. In order to do this, $\mathbb{K}$ automatically generates rules of of the form

$$\frac{E_1\to\mathsf{forAll}(V \mid E_2)}{E_1 \curvearrowright \square\to\mathsf{forAll}(V \mid E_2)}$$

by which the first computation task becomes the evaluation of the expression E_1; the "hole" $\square$ is put in place to receive the value of E_1. This can generate further tasks using the same mechanism, depending on the structure of E_1.

When E_1 is evaluated to a proper collection, say, L, $\mathbb{K}$ rules of the form

$$\frac{L \curvearrowright \square \to \mathsf{forAll}(V \mid E_2)}{L \to \mathsf{forAll}(V \mid E_2)}$$

take over and fill the "hole" left by E_1. Eventually, one of the rules (1–2) finishes the evaluation of the **forAll** expression and reduces it to a Boolean.

Let us now turn to the rules for collections. The **select** and **collect** operations are quite similar to the rules (1–2), so we shall not detail them.

Instead, we show the $\mathbb{K}$ rule giving semantics to the query **allInstances()**:

$$\Big\langle \frac{\mathsf{allInstances}(Cls)}{\mathsf{collectAllInstanceNames}(Cls\ ,\ \mathsf{children}(Cls, M)\ ,\ M)} \cdots\Big\rangle_{\mathsf{k}} \langle M \rangle_{\mathsf{model}}$$

The rule says that, in order to compute the instances of a given class Cls in an object diagram M, a helper function **collectAllInstanceNames()** is called with three parameters: the class Cls, its children $\mathsf{children}(Cls)$, and M itself.

A $\mathbb{K}$ rule may have several local rewrites in various configuration cells, and it can use other cells for providing context for the rewrite. For example, in the above rule, the cell $\langle \_ \rangle_{\mathsf{model}}$ provides the diagram M. The function **children()** computes the children of a given class by traversing M, and **collectAllInstanceNames()** traverses the $\langle \_ \rangle_{\mathsf{instance}}$ cells of M, collects their names, and returns them in a bag.

Another collection expression whose semantics extensively uses the $\langle \_ \rangle_{\mathsf{instance}}$ cells is the navigation operation. Its semantic rules are:

$$\Big\langle \frac{(\,\cdot\,).atrr}{\cdot} \cdots\Big\rangle_{\mathsf{k}}$$

$$\Big\langle \frac{(Elt, Rest).attr}{AttrValue\ \mathsf{union}\ (Rest.attr)} \cdots\Big\rangle_{\mathsf{k}} \langle \langle Elt \rangle_{\mathsf{instName}} \langle attr \mapsto AttrValue \cdots\rangle_{\mathsf{fields}} \cdots\rangle_{\mathsf{instance}}$$

The first rule is the base case: it computes the navigation via an attribute $attr$ from an empty collection; the result is empty. The second rule is the inductive step. For a nonempty collection $(Elt, Rest)$, the result is the union between $AttrValue$ and the result of recursively applying the operation to the collection $Rest$. Here, $AttrValue$ is the value of the $attr$ attribute, from the $\langle \_ \rangle_{\mathsf{fields}}$ cell of the $\langle \_ \rangle_{\mathsf{instance}}$ cell whose $\langle \_ \rangle_{\mathsf{instName}}$ cell contains Elt.

Note that only the minimal information required from the cells was given: e.g., we only give a partial matching for $\langle \_ \rangle_{\mathsf{instance}}$ cells, and do not have to say anything about the surrounding cells of the configuration. That information is automatically synthesized by the $\mathbb{K}$ context transformer mechanism [22].

We now turn to the **let-in** operation, and compare its $\mathbb{K}$ semantics with the official semantics from the $\mathbb{K}$ standard [15]. The $\mathbb{K}$ semantics is simple:

$$\langle \underbrace{\mathsf{let}\ \mathit{Var} = \mathit{Val}\ \mathsf{in}\ \mathit{Exp}}_{\mathit{Exp}[\,\mathit{Val}\,/\,\mathit{Var}]}\ \cdots\rangle_{\mathsf{k}} \tag{3}$$

that is, the let-in operation is just syntactic sugar for $\mathbb{K}$ substitution _[_/_].

The semantics of let-in in the current standard [15] (Page 209) is given as follows:

$$I[\![\mathsf{let}\ v = e_1\ \mathsf{in}\ e_2]\!](r) = I[\![e_2]\!](\sigma, \beta\{v/[\![e_1]\!](r)\})$$

The left-hand side is the interpretation $I[\![\mathsf{let}\ v = e_1\ \mathsf{in}\ e_2]\!](r)$ of an expression $\mathsf{let}\ v = e_1\ \mathsf{in}\ e_2$ in an environment r. The latter is a pair (σ, β) consisting of the model σ where the evaluation is performed and of an environment β - a function mapping the free variables in e_1, e_2 to values. The right-hand side is the interpretation of e_2 in an environment $(\sigma, \beta\{v/[\![e_1]\!](r)\})$, where $\beta\{v/[\![e_1]\!](r)\}$ is a notation for the β component of r, modified such that v is bound to the interpretation of e_1.

Note the additional complexity introduced by the (σ, β) pair and by the interpretation of e_1. In our case, we do not need to specify the model σ since $\mathbb{K}$ context transformation automatically infers it, and we do not need to use an explicit variable environment β since the builtin $\mathbb{K}$ syntactical substitution replaces variables with their values. Moreover, we do not need to refer to the interpretation of e_1, since the strictness annotations for the let-in construct in the second argument ensure that this argument is a value - denoted by Val in (3) - when the let-in is evaluated.

Finally we show how to handle method calls and illustrate them on an example. For the sake of simplicity we show only the case of methods without parameters:

$$\langle \underbrace{O.M()}_{\mathsf{evaluate}(\mathit{Pre}) \curvearrowright \mathsf{update}(O,C,F) \curvearrowright \mathsf{evaluate}(\mathit{Post})}\ \cdots\rangle_{\mathsf{k}}$$
$$\langle\cdots\ \langle O\rangle_{\mathsf{instName}}\langle C\rangle_{\mathsf{ofClass}}\langle F\rangle_{\mathsf{fields}}\ \cdots\rangle_{\mathsf{instance}}$$
$$\langle\cdots\ \langle C\rangle_{\mathsf{className}}\langle\cdots\ \langle M\rangle_{\mathsf{methodName}}\langle \mathit{Pre}\rangle_{\mathsf{pre}}\langle \mathit{Post}\rangle_{\mathsf{post}}\ \cdots\rangle_{\mathsf{method}}\ \cdots\rangle_{\mathsf{class}}$$

The rule consists of evaluating the precondition, updating the model, and evaluating the postcondition. Note that the evaluate construct is strict: its argument is already evaluated to a (possibly, symbolic) Boolean value when the following rule is called:

$$\langle \underbrace{\mathsf{evaluate}(B)}_{\cdot}\ \cdots\rangle_{\mathsf{k}}\langle \underbrace{B'}_{B'\ \mathsf{andBool}\ B}\rangle_{\mathsf{constraints}}$$

There are also rules for updating model, not shown here due to lack of space.

The following simple example illustrates the use of method calls and symbolic evaluation. The example consists in calling the *submit* method twice on the *this:Paper* object (cf. Figure 1). The expected result is that this sequence of method calls is not feasible. After the first call the contents of the constraints cell is:

symBool(b) = false andBool symBool(var0) = true where the first equality is the precondition and the second one is the postcodition; here, symBool(var0) is the symbolic value assigned the attribute isSubmitted by the model-updating rules. After the second call the contents of the cell becomes

symBool(b) = false andBool symBool(var0) = true andBool

symBool(var0) = false andBool symBool(var1) = true

Since the condition evaluates to false, the sequence of method calls is not feasible.

We have implemented a satisfiability checker for such conditons in Maude, which handles the most common cases. For more complex cases, satisfiability can be checked by calling specialised constraint solvers. We get the following output:

```
$ krun test
Not feasible; the constraints are simplified to false.
```

4 Conclusion, Related Work, and Future Work

The initial design goals for the Object Constraint Language were to provide UML with a clear, easy to understand, and formal specification language to complement the less formal diagrams. Unfortunately, the realisation of the language in the current standard lacks much of the initially envisioned qualities.

In this paper we have focused on the better understood parts of OCL and have provided them with a formal semantics in the $\mathbb{K}$ framework. Our semantics compares favorably, in terms of simplicity, with the corresponding mathematical/formal definitions from the informative Annex of the OCL standard [15]. A salient feature of our approach is symbolic evaluation of OCL expressions, which we illustrate via an example showing that a sequence of method calls is not feasible, due to insatisfiable OCL constraints generated from the pre and post conditions of the methods.

We have implemented a translator from a textual format of class diagrams, object diagrams, and OCL expressions into $\mathbb{K}$ code. The code is then compiled used together with the $\mathbb{K}$ OCL semantics. The resulting tool is available online [1].

The definition of OCL is evolving, and we believe that using frameworks such as $\mathbb{K}$ for language standards would much improve quality of the future versions.

Related Work We have already mentioned the informative Annex of the OCL standard [15]. Despite its shortcomings, mainly due to numerous typos, the Annex is based on solid formal ground - set-theoretical foundations [19]. We do not know of formal tools implementing the set-theoretical semantics of the OCL standard.

Closest to our approach are several ones based on and implemented in the Maude rewrite-based system: [21] which, according to the authors, implements the full OCL standard; [3] which uses OCL in a metamodelling framework; and [7] that also implements procedures, a feature that makes OCL very close to a programming language. The authors used the experience gained while formalising their OCL semantics to build an efficient implementation of an OCL evaluator written in a conventional programming language [8]. Another academic tool that is close to implementing the full standard, also written in a conventional programming languguage, but not based on a formal semantics, is the OCL evaluator [5]. Various other implementations of OCL exist in model transformation languages such as the QVT standard [16], and the ATL [11], and KerMeta [18] languages.

OCL semantics have also been defined in interactive theorem provers: [2] in KeY, [12] in PVS, [4] in Isabelle/HOL. These tools require human interaction for evaluating OCL constraints, unlike ours and all of the above-cited ones.

Our main current contribution with respect to these works is symbolic evaluation of OCL expressions, which, to our best knowledge, is new in this framework.

Future Work This is part of our ongoing research project for defining 𝕂-based operational semantics for domain-specific modeling languages [23]. The symbolic evaluation of OCL expressions will be used checking the feasability of scenarios and performing formal verification. From a more practical point of view, we are planning to implement tools for desugaring "real" UML models (e.g., with composition and aggregation operations, and using ordered collection types) into our minimal notion of constrained models. The goal is to make our work available to external users.

Acknowledgment

The authors would like to thank the anonymous reviewers for their valuable and constructive comments and suggestions.

The research presented in this paper was supported in part by the DAK project Contract 161/15.06.2010, SMIS-CSNR 602-12516.

References

[1] ***, *ML^K - Online Tool* (2011). URL https://fmse.info.uaic.ro/tools/mlk/

[2] Beckert, B., U. Keller and P. H. Schmitt, *Translating the Object Constraint Language into First-order Predicate Logic*, in: *Proceedings, VERIFY, Workshop at Federated Logic Conferences (FLoC), Copenhagen, Denmark*, 2002, interactive/deductive verification in KeY.

[3] Boronat, A. and J. Meseguer, *Algebraic Semantics of OCL-Constrained Metamodel Specifications*, in: *TOOLS (47)*, Lecture Notes in Business Information Processing **33** (2009), pp. 96–115.

[4] Brucker, A. D. and B. Wolff, *HOL-OCL: A Formal Proof Environment for UML/OCL*, in: J. L. Fiadeiro and P. Inverardi, editors, *FASE*, Lecture Notes in Computer Science **4961** (2008), pp. 97–100.

[5] Chiorean, D., M. Pasca, A. Cârcu, C. Botiza and S. Moldovan, *Ensuring UML Models Consistency Using the OCL Environment*, Electr. Notes Theor. Comput. Sci. **102** (2004), pp. 99–110, one of the few attempts at implementing the full OCL standard.

[6] Clavel, M., F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer and C. L. Talcott., "All About Maude, A High-Performance Logical Framework," Lecture Notes in Computer Science **4350**, Springer, 2007.

[7] Clavel, M. and M. Egea, *ITP/OCL: A Rewriting-Based Validation Tool for UML+OCL Static Class Diagrams*, in: *AMAST*, Lecture Notes in Computer Science **4019** (2006), pp. 368–373.

[8] Clavel, M., M. Egea and M. A. G. de Dios, *Building an Efficient Component for OCL Evaluation*, ECEASST **15** (2008), a more efficient implementation of the Maude OCL semantics.

[9] Ellison, C. and G. Roşu, *An executable formal semantics of C with applications*, in: *Proceedings of the 39th Symposium on Principles of Programming Languages (POPL'12)* (2012), pp. 533–544.

[10] Gogolla, M. and M. Richters, *Transformation rules for* UML *class diagrams*, in: J. Bézivin and P.-A. Muller, editors, *UML*, Lecture Notes in Computer Science **1618** (1998), pp. 92–106.

[11] Jouault, F., F. Allilaire, J. Bézivin and I. Kurtev, *ATL: A model transformation tool*, Sci. Comput. Program. **72** (2008), pp. 31–39.

[12] Kyas, M., H. Fecher, F. S. de Boer, J. Jacob, J. Hooman, M. van der Zwaag, T. Arons and H. Kugler, *Formalizing UML Models and OCL Constraints in PVS*, Electr. Notes Theor. Comput. Sci. **115** (2005), pp. 39–47.

[13] Lazar, D., A. Arusoaie, T. F. Serbanuta, C. Ellison, R. Mereuta, D. Lucanu and G. Roşu, *Executing formal semantics with the K tool*, in: *Proceedings of the 18th International Symposium on Formal Methods (FM'12)*, LNCS (2012), to appear.

[14] Lucanu, D., T. F. Şerbănuţă and G. Roşu, *K Framework distilled*, in: *9th International Workshop on Rewriting Logic and its Applications*, 2012, p. 9, invited talk.

[15] The Object Management Group, *The Object Constraint Language, Version 2.2*, Technical report (2010), `http://www.omg.org/spec/OCL/2.2/`.

[16] The Object Management Group, *Meta Object Facility (MOF) 2.0 Query/View/ Transformation Specification*, Technical report (2011), `http://www.omg.org/spec/QVT/1.1/PDF/`.

[17] The Objet Management Group, *Unified Modeling Language*, Technical report, `http://www.uml.org/`.

[18] Muller, P.-A., F. Fleurey and J.-M. Jézéquel, *Weaving Executability into Object-Oriented Meta-languages*, in: *MoDELS*, Lecture Notes in Computer Science **3713** (2005), pp. 264–278.

[19] Richters, M., "OCL Constraints," Ph.D. thesis, Universität Bremen (2002), formalisation of OCL using set theory, which formed the basis of the standard's informative Annex.

[20] Roşu, G., C. Ellison and W. Schulte, *Matching Logic: An Alternative to Hoare/Floyd Logic*, in: M. Johnson and D. Pavlovic, editors, *Proceedings of the 13th International Conference on Algebraic Methodology And Software Technology (AMAST '10)*, LNCS **6486** (2010), pp. 142–162.

[21] Roldán, M. and F. Durán, *Dynamic Validation of OCL Constraints with mOdCL*, ECEASST **44** (2011), maude implementation of OCL invariant, pre, and post condition evaluation.

[22] Roşu, G. and T.-F. Şerbănuţă, *An Overview of the K Semantic Framework*, Journal of Logic and Algebraic Programming **79** (2010), pp. 397–434.

[23] Rusu, V. and D. Lucanu, *A K-Based Formal Framework for Domain-Specific Modeling Languages*, in: *2nd International Conference on Formal Verification of Object-Oriented Software*, 2011, pp. 306–323, to appear in LNCS. URL `http://foveoos2011.cost-ic0701.org/`

A OCL Syntax in $\mathbb{K}$

The current version of the $\mathbb{K}$ definition of OCL includes only a kernel of the OMG standard. Due to the modularity of $\mathbb{K}$, new instructions can be incrementally added without changing the semantics of existing ones.

A.1 Arithmetical Expressions

$$
\begin{array}{lll}
\textsc{syntax} & \mathit{Exp} ::= & \#\mathit{Id} \\
& \mid & \#\mathit{Int} \\
& \mid & \mathit{Exp}\ \mathtt{+}\ \mathit{Exp}\ [\text{strict}] \\
& \mid & \mathit{Exp}\ \mathtt{*}\ \mathit{Exp}\ [\text{strict}] \\
& \mid & \mathit{Exp}\ \mathtt{/}\ \mathit{Exp}\ [\text{strict}] \\
& \mid & \mathtt{-}\ \mathit{Exp}\ [\text{strict}] \\
& \mid & \mathit{Exp}\mathtt{.size()}\ [\text{strict}]
\end{array}
$$

A.2 Boolean Expressions

```
SYNTAX   Exp ::= #Bool
               | Exp < Exp [strict]
               | Exp <= Exp [strict]
               | Exp > Exp [strict]
               | Exp >= Exp [strict]
               | Exp ( Exps )  [strict(1)]
               | Exp and Exp [strict(1)]
               | Exp or Exp [strict(1)]
               | Exp implies Exp [strict]
               | not Exp [strict]
               | Exp = Exp [strict]
               | Exp includes( Exp ) [strict]
               | Exp includesAll( Exp ) [strict]
               | Exp excludes(Exp ) [strict]
               | Exp excludesAll(Exp ) [strict]
               | isEmpty( Exp ) [strict]
               | Exp ->forAll( #Id | Exp ) [strict(1)]
               | Exp ->exists( #Id | Exp ) [strict(1)]
```

A.3 String Expressions

```
SYNTAX   Exp ::= #String
               | Exp ++ Exp [strict]
```

A.4 Methods

The syntax of methods together with pre- and postconditions:

```
SYNTAX   Method ::= pre: Exp post: Exp method Id : Id(Params)
SYNTAX   Params ::= List{Param,“”}
SYNTAX   Param ::= Id  Id;
```

A.5 Collection Expressions

We consider only a single kind of collections. The right type of a collection is computed by an auxiliary operator oclType() (this holds in fact for all expressions).

```
SYNTAX   Exp ::= empty
               | allInstances(#Id)
               | Exp . #Id [strict(1)]
               | Exp union Exp [strict]
               | Exp ->select( #Id | Exp ) [strict(1)]
               | Exp ->collect( #Id | Exp ) [strict(1)]
               | let #Id = Exp in Exp endlet [binder strict(2)]
               | if Exp then Exp else Exp endif [strict(1)]
```