Algebraic Semantics of OCL-constrained Metamodel Specifications

Artur Boronat[1] and José Meseguer[2]

[1] Department of Computer Science, University of Leicester
aboronat@mcs.le.ac.uk
[2] Department of Computer Science, University of Illinois at Urbana-Champaign
meseguer@uiuc.edu

Abstract. In the definition of domain-specific languages a MOF metamodel is used to define the main types of its abstract syntax, and OCL invariants are used to add semantic constraints. The semantics of a metamodel definition can be given as a model type whose values are well-formed models. A model is said to *conform* to its metamodel when it is a value of the corresponding model type. However, when OCL invariants are involved, the concept of model conformance has not yet been formally defined in the MOF standard. In this work, the concept of OCL-constrained metamodel conformance is formally defined and used for defining style-preserving software architecture configurations. This concept is supported in MOMENT2, an algebraic framework for MOF metamodeling, where OCL constraints can be used for both static and dynamic analysis.

Key words: Membership equational logic, OCL invariants, MOF metamodel, Static and dynamic analysis of models.

1 Introduction

Model-driven development (MDD) constitutes a paradigm for representing software artifacts with models, for manipulating them and for generating code from them in an automated way. A model can be defined with a so-called domain-specific language (DSL), which provides high-level modeling primitives to capture the semantics of a specific application domain, such as business processes, configuration files or web-based languages (see [1] for an annotated bibliography), or it can be defined with a general-purpose modeling language such as the UML. In both cases, the corresponding modeling language is constituted by an abstract syntax and a concrete syntax, which can be either graphical or textual. In this paper, we focus on the formal semantics of a modeling language, by considering its abstract syntax, when it is enriched with additional semantic requirements specified by OCL constraints.

The Meta-Object Facility (MOF) [2] standard provides a UML-based modeling language for defining the abstract syntax of a modeling language as a metamodel $\mathscr{M}$, where types are metarepresented in a UML-like class diagram. A MOF metamodel $\mathscr{M}$ provides the abstract syntax of a modeling language, but not the semantics of the model conformance relation. In [3,4], we identified this problem and provided a formal framework where the notions of *metamodel realization*, of *model type* $[\![\mathscr{M}]\!]$ and of *model conformance* are formally specified. These notions are implemented

in MOMENT2 [5], where a metamodel realization is algebraically characterized by a theory in membership equational logic (MEL) [6] that is automatically generated from a MOF metamodel $\mathscr{M}$. Within this theory, the carrier of a specific sort in the initial algebra of the metamodel realization constitutes a model type $[\![\mathscr{M}]\!]$, which defines the set of well-formed models M that conform to the metamodel $\mathscr{M}$. We call such a relation the *structural conformance* relation, denoted $M : \mathscr{M}$.

The static semantics of a metamodel $\mathscr{M}$ can be enriched by adding constraints such as invariants, where the concepts that are defined in $\mathscr{M}$ can be used. These invariants can be defined with the standard Object Contraint Language (OCL) [7], enhancing the expressiveness of MOF. However, in the MOF and OCL standards, it is not clear how OCL constraints affect the semantics of a model type $[\![\mathscr{M}]\!]$ defined in a metamodel $\mathscr{M}$, and only implementation-oriented solutions are provided. In this paper, we extend the algebraic semantics of MOF metamodels, presented in [4], by considering OCL constraints.

We build on previous experience on encoding OCL expressions as equationally-defined functions [8]. The main new contributions of this work are: (i) the notion of *metamodel specification* as a pair $(\mathscr{M}, \mathscr{C})$, where $\mathscr{M}$ is a MOF metamodel and $\mathscr{C}$ is a set of OCL constraints that are *meaningful* for $\mathscr{M}$; (ii) algebraic semantics for metamodel specifications, so that the structural conformance relation is enriched with the satisfaction of OCL constraints and characterized by equational axioms; (iii) the use of OCL expressions for dynamic analysis using Maude-based verification techniques [9]; and (iv) the implementation of these new concepts in the MOMENT2 framework by enabling the validation of OCL constraints over models in the Eclipse Modeling Framework (EMF) [10].

In the following subsection, we describe the application of OCL constraints to definitions of style-preserving software architecture configurations. We use this as a running example throughout the paper.

1.1 An Example: architectural style preservation

To illustrate our approach we use a basic specification of a software architecture component type, shown in Fig. 1.(a), where a component can be defined as client or server (type attribute) and can be connected to other components. This component type is given then as a MOF metamodel $\mathscr{M}$. As explained above, such a metamodel metarepresents a model type $[\![\mathscr{M}]\!]$, whose terms constitute well-formed software architecture configurations. By using such a model type $[\![\mathscr{M}]\!]$, we can define a specific software architecture configuration where client component instances can connect to other clients or servers, as shown in Fig. 1.(b). A software architecture configuration is a model M that conforms to the model type $[\![\mathscr{M}]\!]$, denoted $M : \mathscr{M}$.

While a given configuration of component instances can be changed to improve communication among components, there may still be some structural constraints that must be preserved. Constraints of this kind are know as *architectural styles*, which are specified by sets of rules indicating which components can be part of the architecture and how they can be legally interconnected [11]. In this paper, we use as an example the client/server architectural style, in which client component instances can only connect to server component instances. The OCL invariant in Fig. 1.(a) represents such an architectural style in the software architecture definition of the example.

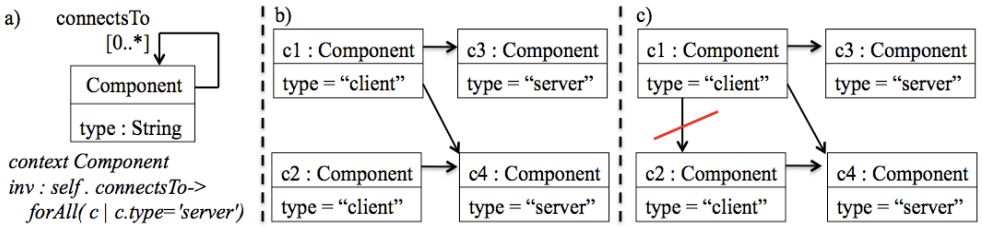

Fig. 1. (a) Metamodel $\mathscr{M}$. (b) Valid configuration. (c) Non style-preserving configuration.

We can therefore view the client/server architecture as a *metamodel specification* $(\mathscr{M}, \mathscr{C})$, where $\mathscr{M}$ is the above-defined metamodel for connected components, and where $\mathscr{C}$ consists of the single OCL constraint just mentioned. A style-conformant configuration for the client/server architecture is then a well-formed architectural model M that satisfies the previous OCL invariant. The configuration provided in Fig. 1.(c) is *not* client/server-conformant due to the marked link between the objects *c1* and *c2*.

In this paper we provide an algebraic, executable semantics for the *constrained conformance relation* to both formalize the use of OCL invariants for the definition of the static semantics of a domain-specific language by means of a metamodel specification $(\mathscr{M}, \mathscr{C})$, and to provide automatic verification of the constrained conformance relation. In Section 2, we summarize preliminary concepts about MEL and Maude. Section 3 gives a summary of the main concepts of the algebraic semantics for MOF: model type $[\![\mathscr{M}]\!]$ and structural conformance between a model M and its metamodel $\mathscr{M}$, i.e., $M : \mathscr{M}$. These concepts are used in Section 4 to define an algebraic, executable semantics for OCL expressions. Section 5 gives the algebraic semantics for metamodel specifications $(\mathscr{M}, \mathscr{C})$, illustrating how OCL can be used for static analysis. Section 6 shows how OCL can be used for dynamic, model checking analysis in MOMENT2. Section 7 compares our approach with other approaches that formalize OCL for different purposes, and Section 8 summarizes the main contributions of the paper.

2 Preliminaries

A membership equational logic (MEL) [6] *signature* is a triple (K, Σ, S) (just Σ in the following), with K a set of *kinds*, $\Sigma = \{\Sigma_{w,k}\}_{(w,k)\in K^* \times K}$ a many-kinded signature and $S = \{S_k\}_{k\in K}$ a K-kinded family of disjoint sets of sorts. The kind of a sort s is denoted by $[s]$. A MEL Σ-algebra A contains a set A_k for each kind $k \in K$, a function $A_f : A_{k_1} \times \cdots \times A_{k_n} \to A_k$ for each operator $f \in \Sigma_{k_1 \cdots k_n, k}$ and a subset $A_s \subseteq A_k$ for each sort $s \in S_k$, with the meaning that the elements in sorts are well-defined, while elements without a sort are *errors*. $T_{\Sigma,k}$ and $T_\Sigma(X)_k$ denote, respectively, the set of ground Σ-terms with kind k and of Σ-terms with kind k over variables in X, where $X = \{x_1 : k_1, \ldots, x_n : k_n\}$ is a set of kinded variables.

Given a MEL signature Σ, *atomic formulae* have either the form $t = t'$ (Σ-equation) or $t : s$ (Σ-membership) with $t, t' \in T_\Sigma(X)_k$ and $s \in S_k$; and *Σ-sentences*

are conditional formulae of the form $(\forall X)\ \varphi\ \ \mathit{if}\ \ \bigwedge_i p_i = q_i\ \wedge\ \bigwedge_j w_j : s_j$, where φ is either a Σ-equation or a Σ-membership, and all the variables in φ, p_i, q_i, and w_j are in X.

A MEL theory is a pair (Σ, E), with Σ a MEL signature and E a set of Σ-sentences. The paper [6] gives a detailed presentation of (Σ, E)-algebras, sound and complete deduction rules, and initial and free algebras. In particular, given a MEL theory (Σ, E), its initial algebra is denoted $T_{(\Sigma/E)}$; its elements are E-equivalence classes of ground terms in T_Σ. Under appropriate executability requirements explained in [9], such as confluence, termination, and sort-decreasingness modulo A, a MEL theory (Σ, E), where $E = E_0 \cup A$, becomes *executable* by rewriting with the equations and memberships E_0 *modulo* some structural axioms A. Furthermore, the initial algebra $T_{(\Sigma/E)}$ then becomes isomorphic to the *canonical term algebra* $Can_{\Sigma/E_0,A}$ whose elements are A-equivalence classes of ground Σ-terms that cannot be further simplified by the equations and memberships in E_0.

A *rewrite theory* [12] is a triple $\mathcal{R} = (\Sigma, E, R)$, where (Σ, E) is a MEL theory, and R is a collection of (possibly conditional) rewrite rules of the form $t \longrightarrow t'\ \ \mathit{if}\ \ C$, where t, t' are Σ-terms of the same kind, and C is the rule's condition. Intuitively, what the rewrite theory $\mathcal{R}$ specifies is a concurrent system, whose states are elements of the algebraic data type $T_{\Sigma/E}$ defined by the MEL theory (Σ, E), and whose concurrent transitions are specified by the rules R. That is, a rule $t \longrightarrow t'\ \ \mathit{if}\ \ C$ specifies transitions in which a fragment of the current state matches the pattern t, and then in the resulting state that state fragment is transformed by the corresponding instance of t', provided that the condition C is satisfied.

Maude [9] is a declarative language where programs are *rewrite theories*, and where Maude computations are *logical deductions* using the axioms specified in the theory/program. Maude provides several verification techniques, including model checking, for systems specified as rewrite theories. We show how these model checking facilities are applied to model analysis with OCL invariants in Section 6.

3 Algebraic Semantics of MOF Metamodels

In this section, notions of the MOF standard and their formalization are presented. These notions are specified in the MOMENT2 framework and constitute the basis for the rest of the paper. MOF is a semiformal approach to define modeling languages by defining their abstract syntax in a so-called metamodel $\mathscr{M}$, which metarepresents a model type $[\![\mathscr{M}]\!]$. What this metamodel describes is, of course, a *set* of models. We call this the *extensional* semantics of $\mathscr{M}$, and denote this semantics by $[\![\mathscr{M}]\!]$, which can be informally defined as follows:

$$[\![\mathscr{M}]\!] = \{M \mid M : \mathscr{M}\}.$$

In [4], we presented a formalization of the MOF framework in which the informal MOF semantics just described is made mathematically precise in terms of the *initial algebra semantics* of MEL. Let $[\![\mathrm{MOF}]\!]$ denote the set of all MOF metamodels $\mathscr{M}$, and let *SpecMEL* denote the set of all MEL specifications. Our algebraic semantics is then defined as a function

$$\mathbb{A} : [\![\mathrm{MOF}]\!] \longrightarrow \mathit{SpecMEL}$$

that associates to each MOF metamodel $\mathscr{M}$ a corresponding MEL specification $\mathbb{A}(\mathscr{M})$, which constitutes the metamodel realization. Recall that any MEL signature Σ has an associated set S of sorts. Therefore, in the initial algebra $T_{(\Sigma,E)}$ each sort $s \in S$ has an associated set of elements $T_{(\Sigma,E),s}$. The key point is that in any MEL specification of the form $\mathbb{A}(\mathscr{M})$, there is always a sort called *Model*, whose data elements in the initial algebra are precisely the data representations of those models that conform to $\mathscr{M}$. That is, the sort *Model* syntactically represents the *model type* $[\![\mathscr{M}]\!]$ associated to a metamodel $\mathscr{M}$. The *structural conformance* relation between a model and its metamodel is then defined mathematically by the equivalence

$$M : \mathscr{M} \quad \Leftrightarrow \quad M \in T_{\mathbb{A}(\mathscr{M}),Model}.$$

Therefore, we can give a precise mathematical semantics to our informal MOF extensional semantics by means of the defining equation

$$[\![\mathscr{M}]\!] = T_{\mathbb{A}(\mathscr{M}),Model}.$$

Note that this algebraic semantics gives a precise mathematical meaning to the entities lacking such a precise meaning in the informal semantics, namely, the notions of: (i) model type $[\![\mathscr{M}]\!]$, (ii) metamodel realization $\mathbb{A}(\mathscr{M})$, and (iii) conformance relation $M : \mathscr{M}$.

For the metamodel $\mathscr{M}$ in Fig. 1.(a), the model depicted in Fig. 1.(c) can be defined as a term of sort *Model* in the $\mathbb{A}(\mathscr{M})$ theory as follows:

```
<< < 'c1 : Component | type = "client",
    connectsTo = 'c2 'c3 'c4 >
  < 'c2 : Component | type = "client", connectsTo = 'c4  >
  < 'c3 : Component | type = "server", connectsTo >
  < 'c4 : Component | type = "server", connectsTo > >>,
```

where each tuple `< Oid : ClassName | Properties >` represents an *object* that is typed with a specific object type of the corresponding metamodel. Objects are defined with properties of two kinds: attributes, typed with simple data types, and references, typed with object identifier types. Each property is defined by a pair `(name = value)`. All the constructors that are used in the previous term are defined in the signature of the $\mathbb{A}(\mathscr{M})$ theory. Note that a term of this kind represents an attributed typed graph with labelled unidirectional edges where the type graph is the metamodel. This representation of models (graphs) as algebraic terms is automatically generated by MOMENT2 from EMF models.

4 Algebraic Semantics of OCL Expressions

In this section, we introduce the algebraic, executable specification of OCL available in MOMENT2, which is based on [8,3]. OCL permits defining expressions that are useful to perform queries upon models M that conform to a given metamodel $\mathscr{M}$. OCL expressions are parameterized with user-defined types, such as classes or enumerations, that are provided in a metamodel $\mathscr{M}$. In MOMENT2, OCL expressions acquire an algebraic semantics that can be used for many purposes. One of them

is the executable formalization of the constrained conformance relation between a model M and a metamodel specification $(\mathscr{M}, \mathscr{C})$ in the following section.

Let *OclExpression* denote the set of well-formed OCL expressions. Not all OCL expressions in *OclExpression* make sense for a given metamodel. We say that a specific OCL expression e, such that $e \in$ *OclExpression*, is *meaningful* for a metamodel $\mathscr{M}$, denoted by $(\mathscr{M}, e)$, iff all user-defined types, if any, that are used in the expression e are metarepresented in $\mathscr{M}$.

An OCL expression is evaluated over a root object in a model M from which other objects in the model can be traversed. In the example in Fig. 1.c), we can traverse connected components through the `connectsTo` reference. In particular, those objects that are reachable through references can be traversed. In the OCL expression, the object type of the root object is indicated in the so-called *context* of the OCL expression. In this setting, the object type is called *contextual type* and the root object, to which an OCL expression is applied, is called *contextual instance*. In an OCL expression, the contextual instance can be explicitly referred to by means of the *self* keyword.

In MOMENT2, we represent the concrete syntax of OCL as a membership equational theory *OCLGrammar* having a sort `OclExpressionCS` whose terms represent syntactically well-formed OCL expressions[3]. Therefore, the following OCL expression is a valid term of sort `OclExpressionCS`:

```
'self . 'connectsTo -> forAll( 'c | 'c . 'type == # "server")
```

and it is a meaningful OCL expression for the metamodel $\mathscr{M}$ of the example if we take into account the object type *Component* as contextual type of the expression.

4.1 Algebraic Executable Semantics of Meaningful OCL Expressions

OCL expressions can be defined by using generic OCL types, such as collections, and user-defined types. The generic OCL types can be split in two groups: basic datatypes and collection types. OCL provides a set of operators that can be applied on values of these types. In particular, among collection operators, we can distinguish between regular operators and loop operators. On the one hand, regular operators provide common functionality such as the size of a collection or the inclusion of elements within a collection. On the other hand, loop operators constitute second-order operators that receive a user-defined OCL expression as argument, normally called *loop body expression*, and that apply it to the elements of a collection depending on the nature of the operator. For example, the `forAll` operator receives a boolean loop body expression and checks that all the elements in the collection satisfy the predicate.

The algebraic semantics of a meaningful OCL expression $(\mathscr{M}, e)$ is given by the partial function

$$\mathbb{C} : [\![\mathrm{MOF}]\!] \times \mathit{OclExpression} \rightharpoonup \mathit{SpecMEL},$$

[3] There are a few minor syntactic differences with the concrete syntax of OCL: identifiers are preceded by a quote, literal values are wrapped by the `#` operator, and the equals operator symbol `=` is `==`.

which is defined for all meaningful OCL expressions $(\mathscr{M}, e)$ and maps $(\mathscr{M}, e)$ to a MEL theory $\mathbb{C}(\mathscr{M}, e)$ such that $\mathbb{A}(\mathscr{M}) \subseteq \mathbb{C}(\mathscr{M}, e)$. In addition, the $\mathbb{C}(\mathscr{M}, e)$ theory provides an algebraic specification for OCL basic datatypes and OCL collection types, where their operators are provided as equationally-defined functions[4].

The $\mathbb{C}$ function can be viewed as an internal compiler from OCL to MEL. The $\mathbb{C}$ function traverses the term-based abstract syntax tree of the input OCL expression e by generating equationally-defined functions for user-defined OCL expressions, such as loop body expressions, in a top-down manner. The outmost OCL expression is represented by an equationally-defined function of the form

$$exp : Object \times Model \longrightarrow \langle OclType \rangle,$$

where *exp* is a symbol that is generated for the corresponding OCL expression e, *Object* represents the set of objects that can be defined with user-defined types from a metamodel $\mathscr{M}$, *Model* is the model type that corresponds to $\mathscr{M}$, and $\langle OclType \rangle$ represents a valid OCL type depending on the type of e. In our running example, we obtain

```
exp : Object Model -> Bool.
```

OCL expressions can then be *executed* over a contextual instance o in a model definition M by means of a term of the form $exp(o, M)$, whose return type depends on the user-defined expression. For the expression

```
'self . 'connectsTo -> forAll( 'c | 'c . 'type == # "server"),
```

the *exp* function is defined by the equation:

```
exp(c , M) = c . 'connectsTo(M) -> forAll ( body ; empty-env ; M ),
```

where `body` is a constant that identifies the function that represents the loop body expression, `c . 'connectsTo(M)` projects all the objects in the model `M` that can be traversed from the object `c` through the reference `connectsTo`, `empty-env` is a list of environment variables (used when there are variables that are bound in an outer OCL expression), and `M` is a variable that represents the model M. If we denote the model in Fig. 1.(c) by `model2` and the objects with identifiers `c1` and `c2` by `o1` and `o2`, repectively, we obtain the following results with the above OCL expression: `exp(o1, model2) = false` and `exp(o2, model2) = true`.

Therefore, the $\mathbb{C}$ function provides an algebraic executable semantics of meaningful OCL expressions $(\mathscr{M}, e)$ by means of an executable MEL theory where all OCL datatypes are algebraically defined, and where user-defined OCL expressions are available as equationally-defined functions. By considering the executability requirements for MEL theories, the presented OCL formalization can be used for evaluating OCL queries within a MOF-like modeling environment, such as the Eclipse Modeling Framework through MOMENT2. In addition, and even more importantly, this algebraic semantics for OCL expressions can be used at a theoretical level to provide a formal semantics to concepts that are not yet sufficiently precise in the MOF standard, such as the constrained conformance relation, and, at a more practical level, to enhance static and dynamic model analysis with OCL invariants.

[4] See [3] for a detailed explanation of the complete algebraic specification of OCL.

5 Algebraic Executable Semantics of Metamodel Specifications

OCL permits imposing constraints upon specific object types in a metamodel $\mathscr{M}$, thus constraining the set of models M that conform to $\mathscr{M}$. We call the resulting conformance relation the *constrained conformance relation*. In this section, we define the concept of *metamodel specification* $(\mathscr{M}, \mathscr{C})$, which is used to attach a set $\mathscr{C}$ of meaningful OCL constraints to a metamodel $\mathscr{M}$. Relying on the aforementioned algebraic semantics of OCL expressions, the algebraic semantics of a metamodel specification $(\mathscr{M}, \mathscr{C})$ is also given, defining how a model type can be semantically enriched with OCL constraints.

5.1 Metamodel Specifications

An OCL *invariant* c is a constraint that is defined using a boolean body expression that evaluates to *true* if the invariant is satisfied. The body expression of an invariant c, denoted $body(c)$, is a well-formed boolean OCL expression, i.e., $body(c) \in OclExpression$. An invariant is also defined with a contextual type by means of the clause `context` as follows:

```
context 'Component inv : <meaningful OCL expression>
```

where `'Component` is the name of an object type, in this case defined in the metamodel $\mathscr{M}$ of the example. An OCL invariant c must hold *true* for any instance of the contextual type at any moment in time. Only when an instance is executing an operation, is c allowed not to evaluate to *true*.

An invariant c is *meaningful* for a metamodel $\mathscr{M}$ iff $(\mathscr{M}, body(c))$ is a meaningful boolean OCL expression. A set of OCL invariants that are meaningful for a metamodel $\mathscr{M}$ may be evaluated over a specific model $M : \mathscr{M}$. More precisely, each OCL invariant $c \in \mathscr{C}$ is evaluated for each contextual instance $o \in M$. We say that a model M *satisfies* a set $\mathscr{C}$ of OCL invariants that are meaningful for a metamodel $\mathscr{M}$ iff all such invariants evaluate to *true* for every contextual instance of the model M. We write $M \vDash \mathscr{C}$ to denote this OCL constraint satisfaction relation.

A metamodel specification $(\mathscr{M}, \mathscr{C})$ is constituted by a metamodel $\mathscr{M}$, such that $\mathscr{M}$: MOF, and a set $\mathscr{C}$ of OCL invariants that are meaningful for $\mathscr{M}$. A metamodel specification $(\mathscr{M}, \mathscr{C})$ defines a model type whose values are models M that both conform to the metamodel $\mathscr{M}$ and satisfy the set $\mathscr{C}$ of meaningful OCL invariants. We define the *extensional semantics* of a metamodel specification $(\mathscr{M}, \mathscr{C})$ by the equality:

$$[\![(\mathscr{M}, \mathscr{C})]\!] = \{M \mid M : \mathscr{M} \ \wedge \ M \vDash \mathscr{C}\}.$$

5.2 Algebraic Executable Semantics of Metamodel Specifications

Let *SpecMOF* denote the set of well-defined metamodel specifications $(\mathscr{M}, \mathscr{C})$. To realize a metamodel specification $(\mathscr{M}, \mathscr{C})$, we define a function

$$\mathbb{A} : SpecMOF \longrightarrow SpecMEL$$

that maps a metamodel specification $(\mathscr{M},\mathscr{C})$ to an executable MEL theory $\mathbb{A}(\mathscr{M},\mathscr{C})$. The resulting $\mathbb{A}(\mathscr{M},\mathscr{C})$ theory, called *metamodel specification realization*, also formalizes the body expressions of all constraints $\mathscr{C}$ involved in the metamodel specification, i.e., we have the MEL theory inclusion

$$\bigcup_{c \in \mathscr{C}} \mathbb{C}(\mathscr{M}, body(c)) \subseteq \mathbb{A}(\mathscr{M},\mathscr{C}).$$

Given a metamodel specification $(\mathscr{M},\mathscr{C})$, the model type $[\![\mathscr{M}]\!]$ is defined in the $\mathbb{A}(\mathscr{M})$ theory. The model type $[\![\mathscr{M}]\!]$ is preserved in the $\mathbb{A}(\mathscr{M},\mathscr{C})$ theory by means of the subtheory inclusion

$$\mathbb{A}(\mathscr{M}) \subseteq \mathbb{A}(\mathscr{M},\mathscr{C}),$$

where the constrained model type $[\![(\mathscr{M},\mathscr{C})]\!]$ is defined as a subset of the model type $[\![\mathscr{M}]\!]$, i.e., $[\![(\mathscr{M},\mathscr{C})]\!] \subseteq [\![\mathscr{M}]\!]$. $[\![(\mathscr{M},\mathscr{C})]\!]$ constitutes the *constrained model type* that is syntactically represented by the sort *CModel* in the theory $\mathbb{A}(\mathscr{M},\mathscr{C})$. This model type inclusion is syntactically defined by the subsort relation $CModel < Model$.

The $\mathbb{A}$ function defines the constrained model type $[\![(\mathscr{M},\mathscr{C})]\!]$, in the $\mathbb{A}(\mathscr{M},\mathscr{C})$ theory, by means of a membership axiom of the form

$$\begin{array}{l} M : CModel \\ \textit{if} \;\; M : Model \;\wedge\; condition_1(M) = true \;\wedge \cdots \wedge\; condition_n(M) = true, \end{array} \qquad (\dagger)$$

where each constraint definition c_i, in $\mathscr{C}$, corresponds to a boolean function $condition_i$ that is generated by means of the $\mathbb{C}$ function and that is evaluated over a model M as explained in Section 4.1. When $\mathscr{C} = \emptyset$, $[\![\mathscr{M}]\!] = [\![(\mathscr{M},\emptyset)]\!]$ and therefore $\mathbb{A}(\mathscr{M}) = \mathbb{A}(\mathscr{M},\emptyset)$. The above membership axiom forces the evaluation of OCL constraints $\mathscr{C}$ over a model M, so that $M : (\mathscr{M},\mathscr{C})$ iff $M \vDash \mathscr{C}$. Therefore, a model definition M, such that $M : \mathscr{M}$, satisfies all the constraints that are defined in $\mathscr{C}$, iff M is a value of the constrained model type $[\![(\mathscr{M},\mathscr{C})]\!]$. Each $condition_i$ function evaluates the boolean body expression of an invariant for all the corresponding contextual instances. In the invariant of our running example we obtain the function:

```
op condition : Model -> Bool .
eq [counterexample] : condition(<<
  < O1 : Component | connectsTo = O2 S, PS1 >
  < O2 : Component | type = "client", PS2 > ObjCol >> ) = false .
eq [invariant-satisfied] : condition(M) = true [owise] .
```

where the equation `counterexample` tries to find a component where the invariant is not satisfied, i.e., is connected to a client component. If no counterexample exists the equation `invariant-satisfied` states that the invariant is satisfied. The resulting membership that is automatically compiled for the OCL constraint of the example is expressed, in Maude notation, as follows

```
cmb M : CModel if condition(M).
```

Using the MEL theory $\mathbb{A}(\mathscr{M},\mathscr{C})$, the semantics of the constrained model type $[\![(\mathscr{M},\mathscr{C})]\!]$ is defined in terms of the initial algebra semantics of $\mathbb{A}(\mathscr{M},\mathscr{C})$ as follows

$$\boxed{[\![(\mathscr{M},\mathscr{C})]\!] = T_{\mathbb{A}(\mathscr{M},\mathscr{C}),CModel}}$$

the OCL constraint satisfaction is defined by the equivalence

$$\boxed{M \vDash \mathscr{C} \Leftrightarrow M \in [\![(\mathscr{M}, \mathscr{C})]\!]}$$

and the constrained conformance relation $M : (\mathscr{M}, \mathscr{C})$ is defined by the equivalence

$$\boxed{M : (\mathscr{M}, \mathscr{C}) \Leftrightarrow M \in [\![(\mathscr{M}, \mathscr{C})]\!].}$$

The $\mathbb{A}(\mathscr{M}, \mathscr{C})$ theory constitutes a formal realization as a theory in MEL of the metamodel specification $(\mathscr{M}, \mathscr{C})$. In addition, $\mathbb{A}(\mathscr{M}, \mathscr{C})$ is *executable*, providing a formal decision procedure for the OCL constraint satisfaction relation. Furthermore, by being a MEL theory with initial algebra semantics, it gives an algebraic semantics for the types that are defined as data in $(\mathscr{M}, \mathscr{C})$.

5.3 MOMENT2-OCL

The presented algebraic semantics for OCL is available in MOMENT2 [5], a formal specification and verification framework for MOF-based software artifacts, specified in Maude and plugged into the EMF. MOMENT2 provides a collection of tools, among which MOMENT2-OCL provides a front-end for defining metamodel specifications $(\mathscr{M}, \mathscr{C})$, where $\mathscr{M}$ is an EMF metamodel and $\mathscr{C}$ is a set of textual OCL invariants. MOMENT2-OCL defines the semantics of a metamodel specification $(\mathscr{M}, \mathscr{C})$ by means of the function $\mathbb{A}$, and offers a mechanism to automatically check the constrained conformance relation between a model M and a constrained model type $[\![(\mathscr{M}, \mathscr{C})]\!]$, i.e., $M : (\mathscr{M}, \mathscr{C})$ by evaluating the membership axiom †.

6 Dynamic Analysis with OCL Invariants

In this section, we show how the aforementioned algebraic semantics for OCL expressions and metamodel specifications $(\mathscr{M}, \mathscr{C})$ can be used for formal dynamic analysis. In our running example, software architectures are configurations of components that may be connected to each other through the *connectsTo* reference. Given a specific initial configuration M of components, we can use model checking in order to verify whether or not all possible reconfigurations of an initial configuration are style-preserving w.r.t. a metamodel specification $(\mathscr{M}, \mathscr{C})$, where the set $\mathscr{C}$ defines the corresponding architectural style to be preserved.

We add a dynamic connection load balancing strategy, so that a server component should not have more than two connections at a time, i.e., when a component has more than two connections, the spare connections are forwarded to other components with less than two incoming connections. We depict the reconfiguration as a graph transformation rule in Fig. 2, where a rule is defined with a left-hand side (LHS) pattern and a right-hand side (RHS) pattern. Each pattern is constituted by nodes that represent *Component* objects in a model (graph) M and edges representing *connectsTo* references between them. A reconfiguration can be applied whenever the LHS pattern of the rule can be matched against a specific configuration M of components, and then the edges are manipulated as follows: an edge in the LHS and not in the RHS is removed from the configuration, an edge not in the LHS but in

the RHS is added, the rest of edges remain unmodified. Marked edges indicate that the references must not exist in order to apply the rule; this is known as a *negative application condition* in the graph transformation community.

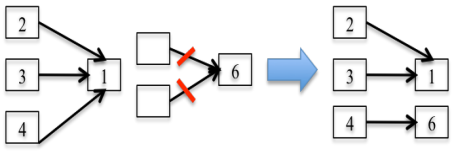

Fig. 2. Reconfiguration rule.

The graph-theoretic nature of models is axiomatized in our algebraic semantics as a set of objects modulo the associativity, commutativity, and identity axioms of set union. The semantics of a reconfiguration can then be naturally expressed as a *rewrite theory* [12] extending the algebraic semantics $\mathbb{A}(\mathscr{M},\mathscr{C})$ of our metamodel specification with *rewrite rules* that are applied *modulo* the equational axioms. In this way, the above graph-transformation rule can be summarized at a high level as follows:

```
op free-reconfiguration : Model -> Model .
crl free-reconfiguration(M) =>
  free-reconfiguration(<<  < O1 : Component | PS1 >
    < O2 : Component | connectsTo = O1 S2, PS2 >
    < O3 : Component | connectsTo = O1 S3, PS3 >
    < O4 : Component | connectsTo = O6 S4, PS4 >
    < O6 : Component | PS6 >  ObjCol >>)
if <<  < O1 : Component | PS1 >
    < O2 : Component | connectsTo = O1 S2, PS2 >
    < O3 : Component | connectsTo = O1 S3, PS3 >
    < O4 : Component | connectsTo = O1 S4, PS4 >
    < O6 : Component | PS6 >
    ObjCol >> := M /\ nac(O6, M) .
```

where the terms with variables that constitute the LHS and RHS of the equation represent the graph patterns defined in Fig. 2, and the condition corresponds to the negative application condition that enables the application of the rule:

```
op nac : Oid Model -> Bool .
eq [satisfiedNAC] : nac(O1, M) = true [owise] .
eq [counterexampleNAC] :
    nac(O1, <<  < O2 : Component | connectsTo = O1 S2, PS2 >
    < O3 : Component | connectsTo = O1 S3, PS3 > ObjCol >>) = false .
```

The metamodel specification realization $\mathbb{A}(\mathscr{M},\mathscr{C})$, corresponding to Fig. 1.(a) and to the singleton set $\mathscr{C}$ of OCL constraints, and the rewriting rule just presented above define a state transition system, where states represent configurations M and M' of components and transitions $M \longrightarrow M'$ represent an application of the reconfiguration rule. However, a reconfiguration of this kind could conceivably produce

configurations M' of components that are not client/server style-preserving, i.e., such that $M' : \mathscr{M}$ but $M' \not\models \mathscr{C}$. Therefore, it is important to formally *verify* whether or not a given reconfiguration, like the one above, is style-preserving.

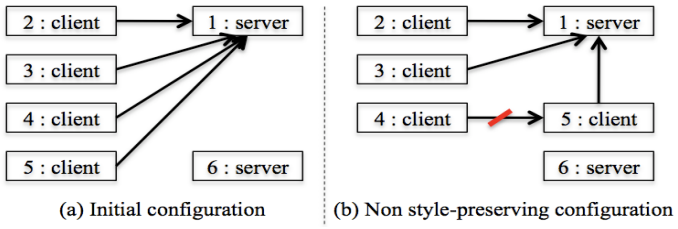

Fig. 3. Initial software architecture configuration (a) and configuration that does not preserve the client/server architectural style (b).

In Maude, the *search* command allows one to exhaustively explore (following a breadth-first strategy) the reachable state space defined by a state transition system as the one above, checking whether an invariant is violated. We can use the *search* command to find out if the reconfiguration `free-reconfiguration` produces such an illegal configuration as follows:

```
search [1] free-reconfiguration(model) =>+
  free-reconfiguration(M:Model)
such that not(M:Model :: CModel) .
```

where `model` is a constant that represents the model M in Fig. 3.a. This command finds a counterexample, shown in Fig. 3.b, where a client component is connected to another client component. An alternative reconfiguration rule can be defined to avoid this problem as shown below, by indicating that the node `6` in the graph patterns of the rule in Fig. 2 is of type *server*. No counterexamples are found when running again the search command with the new reconfiguration rule.

```
op safe-reconfiguration : Model -> Model .
crl safe-reconfiguration(M) =>
  safe-reconfiguration(<<  < O1 : Component | PS1 >
    < O2 : Component | connectsTo = O1 S2, PS2 >
    < O3 : Component | connectsTo = O1 S3, PS3 >
    < O4 : Component | connectsTo = O6 S4, PS4 >
    < O6 : Component | type = "server", PS6 >  ObjCol >>)
if <<  < O1 : Component | PS1 >
    < O2 : Component | connectsTo = O1 S2, PS2 >
    < O3 : Component | connectsTo = O1 S3, PS3 >
    < O4 : Component | connectsTo = O1 S4, PS4 >
    < O6 : Component | type = "server", PS6 >
    ObjCol >> := M /\ nac(O6, M) .
```

7 Related work

The formal semantics of OCL was introduced in [13] and was included in the standard specification [7]. The MOF standard specification [2] provides the semantics of the MOF meta-metamodel and indicates how OCL constraints can be attached to a MOF metamodel from a syntactical point of view.

Clark, Evans and Kent formalized the use of UML and OCL with the MML language in [14]. In their approach the concrete syntax of MML is mapped to the MML Calculus, which provides an operational semantics for both UML modeling constructs and OCL operations. In MOMENT2, we also follow a translational approach to provide semantics to MOF (as presented in [15]) and to OCL. MOF (through EMF) and OCL constitute our concrete syntax and the $\mathbb{C}$ mapping provides the translation of OCL expressions, defined in a metamodel specification, into a MEL theory. Due to the fact that we focus on MOF, we have not considered class methods at this stage, so that OCL pre- and post-conditions are not currently supported.

Our goal in MOMENT2 consists in leveraging the use of rewriting logic and Maude-based formal verification techniques in model-driven development, in particular in the use of OCL in this paper. Therefore, we discuss below other approaches that provide support for formal analysis based on OCL. On the one hand, several tools provide support for static analysis with OCL constraints: USE [16] and MOVA [17] with validation of OCL constraints, and HOL-OCL [18], UML2Alloy [19] and UMLToCSP [20] for verification of UML/OCL models, among others. In particular, HOL/OCL provides an elegant approach to encode OCL iterator operators as higher order constructs. However, we have chosen MEL for our OCL formalization because it is a sublogic of rewriting logic, hence equationally-defined OCL expressions can be used to define properties that can be verified in rewrite systems by means of Maude's facilities for reachability analysis and LTL model checking. MOVA also uses MEL as underlying formalism and focuses on the specification of UML/OCL models, which are represented as MEL theories. In MOMENT2, we focus on metamodel specifications $(\mathscr{M}, \mathscr{C})$, and a model M is defined as a term modulo associativity commutativity and identity, i.e., a graph [4], and not as a MEL theory as in MOVA.

On the other hand, dynamic analysis with OCL is usually supported by mapping UML/OCL models into a given formalism for model checking, such as the Object-Based Temporal Logic (BOTL), a logic based on branching temporal logic CTL and OCL, in [21]. The tool SOCLe [22] provides an extension of OCL (EOCL) with CTL temporal operators and first-order features, inspired in BOTL, that allows model checking EOCL predicates on UML models expressed as abstract state machines. In our case, we automatically map metamodel specifications $(\mathscr{M}, \mathscr{C})$ to MEL theories enabling model checking of OCL invariants in rewriting logic as shown above. Although without OCL, in [23], the authors present how to directly use Maude to provide the structural and dynamic semantics of DSLs, where metamodels are encoded as rewrite theories and models as collections of objects. Therefore, Maude-based formal verification techniques can be applied as described in [9]. Despite the similar use of verification techniques, there are several differences between both approaches. MOMENT2 uses OMG standards, such as MOF and OCL, as interface between industrial environments, such as EMF, and the formalism based on Maude so that the use of the formalism remains hidden to the user. In addition, we have identified and formalized the notions of model type and conformance relation where

OCL constraints can be taken into account. A complete description of these concepts and their formalization is provided in [3].

In the graph transformation field, several analysis techniques have been developed [24], but they usually work with simple type graphs, less expressive than metamodel specifications. [25] shows how graph transformations can be mapped to OCL pre- and post-conditions, so that the aforementioned tools for OCL-based formal verification can be applied. In addition, the authors considered a number of analysis techniques with OCL properties, such as correctness preservation when a transformation rule is applied, among others. Our approach and strategy are just the opposite: we have mapped metamodel specifications $(\mathscr{M}, \mathscr{C})$ into MEL theories so that the $\mathbb{C}$ function is used to include OCL expressions in graph transformations in the tool MOMENT2-GT [5].

As for the running example, Architectural Design Rewriting (ADR) [26] is an approach for hierarchical style-based reconfigurations of software architectures that is based on rewriting logic. ADR allows defining style-preserving reconfigurations while in our approach style preservation should be explicitly verified. However, our approach is not specific to the service-oriented computing domain and relies on OMG standards for formalizing DSLs.

8 Conclusions and Future Work

We have presented several contributions towards the main goal of increasing the formal analysis power of model-based software engineering. Our first contribution has centered on the fact that metamodel specifications frequently include both the metamodel syntax itself and additional semantic constraints, thus making the issue of checking conformance of a model to a metamodel specification a semantic one. To automate the cheking of what we have called *constrained model conformance*, we have given an algebraic, executable semantics of metamodel specifications, embodied in the function $\mathbb{A}$. This provides a *static analysis* feature for models in a modeling language with semantic constraints, and we have illustrated its use with a client-server architectural style example. Our second contribution has been to show how the same $\mathbb{A}$ function can also be used for *dynamic analysis* when models are transformed, so that one can check that the models obtained by transforming a given model using reconfiguration rules satisfy some correctness criteria. We have illustrated this by showing through model checking how a given reconfiguration rule is incorrect, producing models that fail to satisfy the client-server matamodel specification and giving a better rule not making such violations. More generally, the semantics and infrastructure developed here can be used in conjuntion with Maude's LTL model checker to verify any dynamic properties expressed as LTL formulas whose atomic predicates are defined by OCL invariants. Our third and last contribution has been to incorporate support for automatic verification of OCL invariants and of the constrained conformance relation in the latest version MOMENT2 tool [5].

The presented semantics for OCL expressions is integrated in a QVT-based graph transformation language, MOMENT2-GT (also available in the MOMENT2 framework), so that OCL expressions can be used to perform queries and to manipulate data. In MOMENT2-GT, a model transformation is defined as a collection of graph production rules, which are compiled to equations and rewrites as the ones shown in

Section 6. Therefore, Maude-based model checking facilities can be used for model checking model transformations with OCL predicates. In future work, we plan to apply MOMENT2 and Maude-based formal verification techniques to perform formal analysis of real-time embedded systems in the avionics specific domain, by using model-based languages like the Architecture Analysis and Design Language (AADL) [27].

References

1. van Deursen, A.v., Klint, P., Visser, J.: Domain-specific languages: an annotated bibliography. SIGPLAN Not. **35** (2000) 26–36
2. Object Management Group: Meta Object Facility (MOF) 2.0 Core Specification (ptc/06-01-01) (2006) `http://www.omg.org/cgi-bin/doc?formal/2006-01-01`.
3. Boronat, A.: MOMENT: a formal framework for MOdel manageMENT. PhD in Computer Science, Universitat Politènica de València (UPV), Spain (2007) `http://www.cs.le.ac.uk/people/aboronat/papers/2007_thesis_ArturBoronat.pdf`.
4. Boronat, A., Meseguer, J.: An Algebraic Semantics for MOF. In: Fundamental Approaches to Software Engineering, 9th International Conference, FASE 2008, in ETAPS 2008, Budapest, Hungary, 29 March - 6 April, 2008, Proceedings. Lecture Notes in Computer Science, Springer (2008)
5. Boronat, A.: The MOMENT2 web site (2008) `http://www.cs.le.ac.uk/people/aboronat/tools/moment2`.
6. Meseguer, J.: Membership algebra as a logical framework for equational specification. In Parisi-Presicce, F., ed.: Proc. WADT'97, Springer LNCS 1376 (1998) 18–61
7. Object Management Group: OCL 2.0 Specification (2006) `http://www.omg.org/cgi-bin/doc?formal/2006-05-01`.
8. Boronat, A., Oriente, J., Gómez, A., Ramos, I., Carsí, J.A.: An Algebraic Specification of Generic OCL Queries Within the Eclipse Modeling Framework. In Rensink, A., Warmer, J., eds.: ECMDA-FA. Volume 4066 of LNCS., Springer (2006) 316–330
9. Clavel, M., Durán, F., Eker, S., Meseguer, J., Lincoln, P., Martí-Oliet, N., Talcott, C.: All About Maude. Springer LNCS Vol. 4350 (2007)
10. Eclipse Organization: The Eclipse Modeling Framework (2007) `http://www.eclipse.org/emf/`.
11. Shaw, M., Garlan, D.: Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall (1996)
12. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science **96** (1992) 73–155
13. Richters, M.: A Precise Approach to Validating UML Models and OCL Constraints. PhD thesis, Universität Bremen, Logos Verlag, Berlin, BISS Monographs, No. 14 (2002)
14. Clark, T., Evans, A., Kent, S.: The metamodelling language calculus: Foundation semantics for uml. In Hußmann, H., ed.: FASE. Volume 2029 of Lecture Notes in Computer Science., Springer (2001) 17–31
15. Boronat, A., Carsí, J.A., Ramos, I.: Algebraic specification of a model transformation engine. In Baresi, L., Heckel, R., eds.: FASE. Volume 3922 of Lecture Notes in Computer Science., Springer (2006) 262–277
16. Gogolla, M., Büttner, F., Richters, M.: USE: A UML-based specification environment for validating UML and OCL. Sci. Comput. Program. **69** (2007) 27–34
17. Egea, M.: An Executable Formal Semantics for OCL with Applications to Model Analysis and Validation. PhD in Computer Science (to appear), Universidad Complutense de Madrid, Spain (2008)

18. Brucker, A.D., Wolff, B.: The HOL-OCL book. Technical Report 525, ETH Zürich (2006)
19. Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: UML2Alloy: A Challenging Model Transformation. In: MoDELS 2007. Springer (2007) 436–450
20. Cabot, J., Clarisó, R., Riera, D.: UMLtoCSP: a tool for the formal verification of UML/OCL models using constraint programming. In: ASE'07, ACM (2007) 547–548
21. Distefano, D., Katoen, J.P., Rensink, A.: On a temporal logic for object-based systems. In Smith, S.F., Talcott, C.L., eds.: Formal Methods for Open Object-based Distributed Systems, Kluwer Academic Publishers (2000) 305–326 Report version: TR–CTIT–00–06, Faculty of Informatics, University of Twente.
22. Mullins, J., Oarga, R.: Model Checking of Extended OCL Constraints on UML Models in SOCLe. In Bonsangue, M.M., Johnsen, E.B., eds.: FMOODS. Volume 4468 of Lecture Notes in Computer Science., Springer (2007) 59–75
23. Rivera, J.E., Vallecillo, A.: Adding behavioral semantics to models. In: 11th IEEE International Enterprise Distibuted Object Computing Conference. EDOC 2007, 15-19 October 2007 Annapolis, Maryland, USA. Proceedings, Los Alamitos, California, IEEE Computer Society (2007) 169–180
24. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series). Springer (2006)
25. Cabot, J., Clarisó, R., Guerra, E., de Lara, J.: Analysing graph transformation rules through OCL. In: ICMT'08: International Conference on Model Transformation. Volume 5063., Springer LNCS (2008) 225–239
26. Bruni, R., Lluch-Lafuente, A., Montanari, U., Tuosto, E.: Style-Based Architectural Reconfigurations. Bulletin of the EATCS (2008)
27. SAE: AADL (2007) `http://www.aadl.info/`.