DOI 10.1007/s00165-016-0354-6
BCS © 2016
Formal Aspects of Computing (2016) 28: 409–440

Formal Aspects of Computing

Correct-by-construction model driven engineering composition operators

Mounira Kezadri Hamiaz[1], Marc Pantel[2], Xavier Thirioux[2] and Benoit Combemale[3]

[1] College of Computer Science and Engineering (CCSE), Taibah University, Al Madinah, Saudi Arabia
[2] IRIT, Université de Toulouse, Toulouse, France
[3] IRISA, Université de Rennes 1, Rennes, France

Abstract. Model composition is a crucial activity in Model Driven Engineering both to reuse validated and verified model elements and to handle separately the various aspects in a complex system and then weave them while preserving their properties. Many research activities target this compositional validation and verification (V & V) strategy: allow the independent assessment of components and minimize the residual V & V activities at assembly time. However, there is a continuous and increasing need for the definition of new composition operators that allow the reconciliation of existing models to build new systems according to various requirements. These ones are usually built from scratch and must be systematically verified to assess that they preserve the properties of the assembled elements. This verification is usually tedious but is mandatory to avoid verifying the composite system for each use of the operators. Our work addresses these issues, we first target the use of proof assistants for specifying and verifying compositional verification frameworks relying on formal verification techniques instead of testing and proofreading. Then, using a divide and conquer approach, we focus on the development of elementary composition operators that are easy to verify and can be used to further define complex composition operators. In our approach, proofs for the complex operators are then obtained by assembling the proofs of the basic operators. To illustrate our proposal, we use the Coq proof assistant to formalize the language-independent elementary composition operators Union and Substitution and the proof that the conformance of models with respect to metamodels is preserved during composition. We show that more sophisticated composition operators that share parts of the implementation and have several properties in common (especially: aspect oriented modeling composition approach, invasive software composition, and package merge) can then be built from the basic ones, and that the proof of conformance preservation can also be built from the proofs of basic operators.

Keywords: Formal verification, Composition, Proof assistant, MDE, MOF

Correspondence and offprint requests to: M. K. Hamiaz. e-mail: mhamiaz@taibahu.edu.sa

1. Introduction

Model composition is a crucial activity in Model Driven Engineering (MDE) both to reuse validated and verified model elements and to handle separately the various aspects in a complex system and then weave them while preserving their properties in a correct by construction manner. Many research activities target compositional validation and verification which allow for assessing components independently from one another and then minimizing the residual validation and verification activities at assembly time. However, contributions are still needed in the area of correct by construction models composition. Providing formal specifications for component models and composition operations allows for assessing the correctness of these strategies and thus relying on them for the correctness of composite systems.

In order to improve the efficiency of system development, there is a continuous and increasing need for the definition of new composition operators that allow the reconciliation of existing models to build new systems according to various requirements. An example is the reconciliation of models developed following aspect-oriented modeling and viewpoint-oriented modeling paradigms and the modification or addition of features for existing models. These newly defined complex composition operators are usually built from scratch and must be systematically verified to assess that they preserve the properties of the assembled elements.

Our work addresses these issues. First, in order to ease the integration of formal specification and verification technologies, [TCCG07] proposed a formal deep embedding of some key aspects of MDE (models, metamodels, conformance and promotion) in Set Theory which was then implemented using the Calculus of Inductive Construction (CIC) and the Coq [1] proof-assistant using its elementary constructs. The purpose of this framework called Coq4MDE is to provide sound mathematical foundations for the study of MDE technologies. The choice of constructive logic and type theory as formal specification language allows for the extraction of prototype tools from the executable specification which can then be used to validate the specification itself with respect to external tools implementing MDE (for example, in the Eclipse Modeling Project). We extended the framework Coq4MDE in [KPCT11] to support the specification of the Invasive Software Composition (ISC) [Aßm03] operators and the proof of the well-foundedness and termination of these operators. The first version was limited to the verification of the metamodel structural conformance relation. It relied on the model and metamodel concepts from Coq4MDE to formalize the notion of fragment proposed by ISC as well as various other aspects assisting with the proof that composition preserves well typedness. This work was extended in [KHPCT14a] to support the proof of the preservation of semantic properties for a sub-set of Meta Object Facility (MOF) [Obj13a] which covers the hierarchy, the abstract classes, the multiplicities, the opposite and the composite references. These formalization and verification activities were conducted from scratch without reusing more elementary composition operators. This paper will extend that work to illustrate how the ISC operators can be formalized and verified relying on common elementary operators.

To ease the verification of other composition operators, we followed a divide and conquer approach by focusing on the development of elementary composition operators that are more easily verifiable and which can then be used in the implementation of more complex composition operators. The proof of the sophisticated operators are then obtained by assembling the proofs of the basic ones. We presented in [KHPCT14b] a first experiment which used two elementary composition operators Union and Substitution for the formalization and the proof of properties for the MOF Package Merge operator [Obj13b]. In this paper, we give more details about our formalization and proof strategies as well as other applications. The composition operators that are supported by our work rely on two principal activities: matching (ensured by the Substitution operator) and assembly (ensured by the Union operator) of the input model elements. The applications of our work include a variety of high level model composition approaches covering for example: product line evolution, model parametrization, and ontology composition.

The main contributions of this paper are: to extend the [KHPCT14b] paper with a more detailed account of the formalization and proofs for the elementary operators, then to show how the ISC operators [Aßm03] presented in [KPCT11] and [KHPCT14a] can be formalized and verified using more elementary operators, and to present a new application of the formalized elementary operators from an aspect-oriented modeling (AOM) composition approach [FR07].

[1] http://coq.inria.fr.

The remainder of this paper is organized as follows: In Sect. 2, we give the Background by introducing MDE, the Coq4MDE framework, the MOF standard and some composition operators used to validate our proposal. In Sect. 3, we show the expected property for the composition and our verification strategy. In Sect. 4, we present the formalization of the primitive operators and the verification of their properties. In Sect. 5, we rely on the use cases to illustrate our proposal: formalize complex composition operators relying on proved elementary operators. Section 6 discusses related work and Sect. 7 concludes and provides insights on future works.

2. Background

In this section, we summarize the principles behind MDE, we then highlight the concepts of the Coq4MDE framework, and present the principles of the MOF standard. Lastly, we detail several composition operators that are used in the use cases section.

2.1. Model driven engineering

The core principle of MDE is *"everything is a model"* [Béz04]. It emphasizes the role of models as primary artifacts in software and even system development and, in particular, argues that models should be precise enough to support automated model transformations between lifecycle phases [WHR14]. Models are defined using modeling languages. Metamodels are models of modeling languages defined using metamodeling languages. A model M conforms to a metamodel MM if MM models the language used to define M. Metamodels, like data types, define the structure common to all its conforming models, but it can also give semantic properties like dependent types do. The Coq4MDE framework described in the next section aims to formalize the principles of MDE through the concepts of model, metamodel, and the conformance relation.

2.2. COQ4MDE

This section gives in a nutshell the framework Coq4MDE [2], derived from [TCCG07]. It separates the model level (value or object) from the metamodel level (type or class), and describes them with different data structures, and hence different types in Coq. A model (M) is at the instance level while a metamodel (MM), as a modeling language used to define models, is thus at the type level. In order to avoid constraints on mixing inductive and co-inductive types in Coq needed for a natural encoding of the model graph structures, we decided to rely on a deep embedding where both the instance and type level are encoded as data in Coq. A MM also specifies semantic properties common to its conforming models. These properties are expressed as predicates on the data types that model both model (instance) and metamodel (type) levels. Both concepts are formally defined in the following way. Let us consider two sets of labels: `Classes`, and `References`, representing the set of all possible classes and reference labels. Let us also consider instances of such classes, the set `Objects` of object labels. `References` also includes a specific inh label used to specify the inheritance relation at the model level. In the next sections, we will elide the word label and directly talk about classes, references and objects.

Definition 1 (*Model*) Let $\mathscr{C} \subseteq$ `Classes` be a set of classes. Let $\mathscr{R} \subseteq \{\langle c_1, r, c_2\rangle \mid c_1, c_2 \in \mathscr{C}, r \in$ `References`$\}$[3] be a set of references between classes.

A `Model` over $\mathscr{C}$ and $\mathscr{R}$, written $\langle MV, ME\rangle \in Model(\mathscr{C}, \mathscr{R})$ is a multigraph built over a finite set MV of typed object vertices and a finite set ME[4] of reference edges such that:

$$MV \subseteq \{\langle o, c\rangle \mid o \in \texttt{Objects}, c \in \mathscr{C}\}$$
$$ME \subseteq \left\{ \langle\langle o_1, c_1\rangle, r, \langle o_2, c_2\rangle\rangle \middle| \begin{array}{l}\langle o_1, c_1\rangle, \langle o_2, c_2\rangle \in MV,\\ \langle c_1, r, c_2\rangle \in \mathscr{R}\end{array} \right\}$$

We thus have three distinct layers in the encoding: set, graph and model.

[2] http://coq4mde.enseeiht.fr/FormalMDE.

[3] $\langle c_1, c_2, r\rangle$ in the Coq code is denoted here for simplification as: $\langle c_1, r, c_2\rangle$.

[4] $\langle\langle o_1, c_1\rangle, r, \langle o_2, c_2\rangle\rangle$ is denoted in the Coq code as: $\langle\langle o_1, c_1\rangle, \langle o_2, c_2\rangle, r\rangle\rangle$.

Note that, in case of inheritance, the same object label will be used several times in the same model graph. It will be associated to the different classes in the inheritance hierarchy going from one of the roots (multiple inheritance can lead to several distinct roots for the inheritance graph) to the class used to create the object (the constructor class). This label reuse encodes the inheritance polymorphism, a key aspect of most OO languages. Inheritance is represented in the models with a special reference called inh. Accordingly, we first define an auxiliary predicate stating that an object o of type c_1 has a downcast duplicate of type c_2.

$$\begin{aligned}&hasSub(o \in \texttt{Objects}, c_1, c_2 \in \texttt{Classes}, \langle MV, ME\rangle) \triangleq \\ &\quad c_1 = c_2 \vee \exists\, c_3 \in \texttt{Classes}, \langle\langle o, c_2\rangle, inh, \langle o, c_3\rangle\rangle \in ME \\ &\quad \wedge\, hasSub(o, c_1, c_3, \langle MV, ME\rangle)\end{aligned}$$

Then, we define the notion of standard inheritance. The first part of the conjunction states that the inheritance relation only relates duplicate objects. The second part states that every couple of duplicates has a common downcast element (a common subclass) using twice the $hasSub$ predicate.

$$\begin{aligned}&standardInheritance(\langle MV, ME\rangle) \triangleq \\ &\quad (\ \ \forall\langle\langle o_1, c_1\rangle, inh, \langle o_2, c_2\rangle\rangle \in ME \rightarrow o_1 = o_2) \\ &\quad \wedge\, (\forall\langle o_1, c_1\rangle, \langle o_2, c_2\rangle \in MV, o_1 = o_2 \rightarrow \exists c \in \texttt{Classes}, \\ &\qquad hasSub(o_1, c_1, c, \langle MV, ME\rangle) \\ &\qquad \wedge\, hasSub(o_2, c_2, c, \langle MV, ME\rangle))\end{aligned}$$

The following property states that c_2 is a direct subclass of c_1.

$$\begin{aligned}&subClass(c_1, c_2 \in \texttt{Classes}, \langle MV, ME\rangle) \triangleq \forall\, o \in \texttt{Objects}, \\ &\quad \langle o, c_2\rangle \in MV \rightarrow \langle\langle o, c_2\rangle, inh, \langle o, c_1\rangle\rangle \in ME\end{aligned}$$

The `isConstructorClass` property states that a class c_1 is a constructor class for an object o_1.

$$\begin{aligned}&isConstructorClass(o_1 \in \texttt{Objects}, c_1 \in \texttt{Classes}, \langle MV, ME\rangle) \triangleq \forall\langle o_2, c_2\rangle \in MV, \\ &\quad o_2 = o_1 \rightarrow hasSub(o_1, c_2, c_1)\end{aligned}$$

Abstract Classes that are specified in a metamodel using the `isAbstract` attribute are not suitable for instantiation. They are used to represent abstract concepts or entities. If an object can be of an abstract class type, it must have a downcast duplicate from a concrete class.

$$\begin{aligned}&isAbstract(c_1 \in \texttt{Classes}, \langle MV, ME\rangle) \triangleq \forall\, o \in \texttt{Objects}, \\ &\langle o, c_1\rangle \in MV \rightarrow \exists c_2 \in \texttt{Classes}, \langle\langle o, c_2\rangle, inh, \langle o, c_1\rangle\rangle \in ME\end{aligned}$$

Definition 2 (*MetaModel*) A `MetaModel` is a multigraph representing classes as vertices and references as edges as well as semantic properties over instantiation of classes and references. It is represented as a pair (MMV, MME) built over a finite set MMV of vertices, a finite set MME of edges, and a predicate over models representing the semantic properties $conformsTo$.

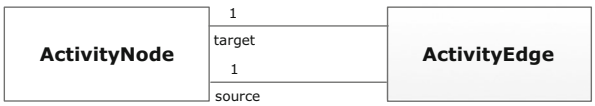
A `MetaModel` is a pair $\langle(MMV, MME), conformsTo\rangle$ such that:

$$\begin{aligned}&MMV \subseteq \texttt{Classes} \\ &MME \subseteq \{\langle c_1, r, c_2\rangle \mid c_1, c_2 \in MMV, r \in \texttt{References}\} \\ &conformsTo : Model \times (MMV, MME) \rightarrow Bool\end{aligned}$$

Given one `Model` M and one `MetaModel` MM, we can check its conformance using the $conformsTo$ predicate embedded in MM. It identifies the set of valid models with respect to a metamodel.

In our framework, the conformance checks for the model M that:

1. every object o in M is the instance of a class C in MM;
2. every link between two objects is such that there exists, in MM, a reference between the two classes typing the two elements. In the following we will say that these links are instances of the reference between classes in MM ;
3. finally, every semantic property defined in MM is satisfied in M. For instance, the multiplicity defined on references between concepts denotes a range of possible links between objects of these classes (i.e. concepts). Moreover, structural properties expressed on the metamodel can also be taken into account.



$$\begin{aligned}&\langle MMV \triangleq \{ActivityNode, ActivityEdge\},\\ &\ MME \triangleq \{(ActivityEdge, target, ActivityNode),\\ &\qquad (ActivityEdge, source, ActivityNode)\},\\ &conformsTo \triangleq lower(ActivityEdge, source, 1)\\ &\qquad\qquad \wedge\ upper(ActivityEdge, source, 1)\\ &\qquad\qquad \wedge\ lower(ActivityEdge, target, 1)\\ &\qquad\qquad \wedge\ upper(ActivityEdge, target, 1)\rangle\end{aligned}$$

Fig. 1. A simplified activity diagram metamodel in the CoQ4MDE notation

The semantic properties associated to the metamodel are encoded in the *conformsTo* predicate.

Figure 1 shows an example of a simplified activity diagram metamodel [Obj13b] on the left and its CoQ4MDE representation on the right hand side. The *conformsTo* predicate describes the properties associated with a metamodel which must be satisfied by the conforming models. It is defined here as a conjunction of four predicates. The first two predicates, $lower(ActivityEdge, source, 1)$ and $lower(ActivityEdge, target, 1)$, set the lower bound of a particular *source* and *target* relation for an instance of *ActivityEdge*. The last two predicates are: $upper(ActivityEdge, source, 1)$ and $upper(ActivityEdge, target, 1)$ setting the maximum number of a *source* and *target* relations for an instance of *ActivityEdge*. The formal description of these properties is given later in the paper.

Figure 2 shows an example of a model that conforms to the metamodel given in Fig. 1. This example is inspired from [HHJZ09] and is reused in Sect. 5. Part (c) of Fig. 2 shows the CoQ4MDE textual representation corresponding to this model. All the structural and semantic properties of the metamodel in Fig. 1 are respected in this model. More specifically, every object is an instance of the class *ActivityNode* or *ActivityEdge* and every object instance of *ActivityEdge* is linked with exactly one relation *source* and one relation *target* to an object instance of *ActivityNode*.

2.3. Meta-object facility (MOF)

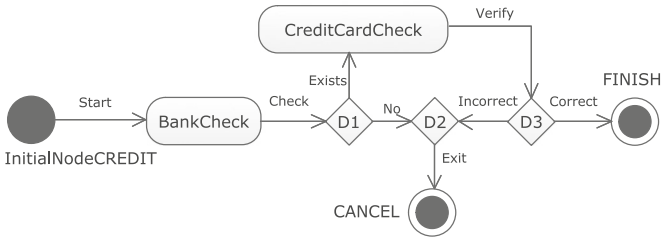
The Object Management Group (OMG) has standardized the MOF, a reflexive metamodeling language (i.e. MOF is defined as a model in the MOF language). MOF is used for the specification of the OMG modeling language standards like MOF itself, UML [Obj13b], OCL [Obj14], SysML [HP08] and many others. The relation between MOF and the metamodels is the same as the one between a metamodel and its conforming models.

Since the MOF version 2 released in 2006 [Obj06], a kernel named EMOF was extracted from the complete version of MOF (CMOF). EMOF provides a minimal set of elements required to model languages. Figure 3 gives the key concepts of EMOF specified as an UML class diagram where the class names written in italics indicate the abstract classes. We informally describe the EMOF and also define a set of generic properties representing its core concepts. Each property, once instantiated, is meant to yield the *conformsTo* predicate.

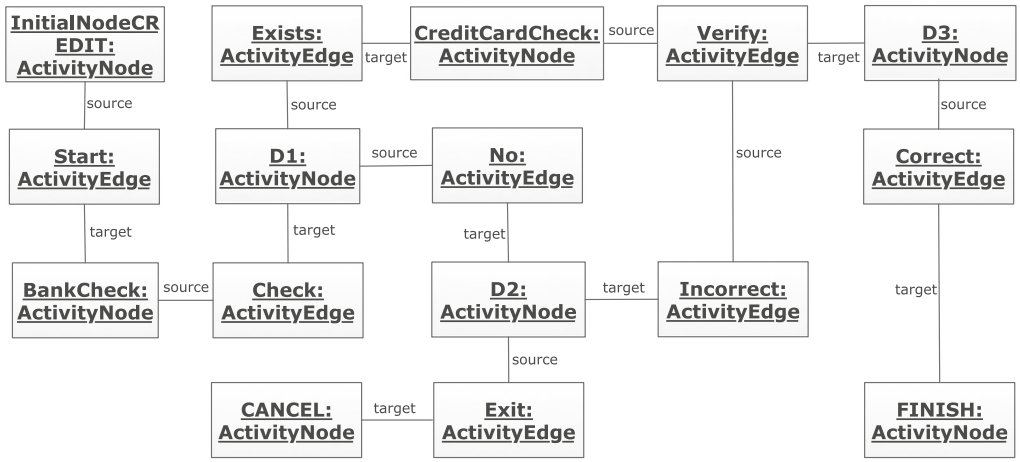
The principal concept is `Class` to define classes (usually called metaclasses) that represent concepts in a modeling language. Classes allow the creation of objects in models. The type of an object is the class that was used to create it. Classes are composed of an arbitrary number of `Properties` (we will call them reference and attribute in order to avoid ambiguities with the model property we want to assess). References allow the creation of relations between the objects in the models. Classes can inherit references and attributes from other classes. Inheritance is expressed using the `superClass` reference from `Class` and introduces a subtyping relation between the types associated to the classes. Classes can be abstract (`isAbstract`): no object can have the type associated to an abstract class as smallest type according to the subtyping relation. `Property` has a `lower` and an `upper` attributes that restrict the number of objects contained in a given reference. Two references can be `opposite`, and build a bidirectional relation between objects in a model.

Figure 4 shows the simplified activity diagram metamodel presented in Fig. 1 as a model conforming to EMOF. In this model, *ActivityEdge* and *ActivityNode* are instances of `Class` and the references *source* and *target* are instances of `Property`.

In order to construct a formal framework for model composition, we extend the existing MDE framework in a manner that allows formalizing and proving the preservation of properties for composition operators. Our ultimate goal is to formalize compositional verification activities but we must first define our targeted composition operators which are principally based on the substitution and union of models.



(a) Activity diagram for credit card check



(b) Class diagram for credit card check

$\langle MV \triangleq \{\langle InitialNodeCREDIT, ActivityNode\rangle, \langle BankCheck, ActivityNode\rangle, \langle D1, ActivityNode\rangle,$
$\langle CreditCardCheck, ActivityNode\rangle, \langle D2, ActivityNode\rangle, \langle D3, ActivityNode\rangle, \langle CANCEL, ActivityNode\rangle,$
$\langle FINISH, ActivityNode\rangle, \langle Start, ActivityEdge\rangle, \langle Check, ActivityEdge\rangle, \langle Exists, ActivityEdge\rangle,$
$\langle No, ActivityEdge\rangle, \langle Verify, ActivityEdge\rangle, \langle Correct, ActivityEdge\rangle,$
$\langle Incorrect, ActivityEdge\rangle, \langle Exit, ActivityEdge\rangle\},$
$ME \triangleq \{\langle\langle Start, ActivityEdge\rangle, source, \langle InitialNodeCREDIT, ActivityNode\rangle\rangle,$
$\langle\langle Start, ActivityEdge\rangle, target, \langle BankCheck, ActivityNode\rangle\rangle,$
$\langle\langle Check, ActivityEdge\rangle, source, \langle BankCheck, ActivityNode\rangle\rangle,$
$\langle\langle Check, ActivityEdge\rangle, target, \langle D1, ActivityNode\rangle\rangle,$
$\langle\langle No, ActivityEdge\rangle, source, \langle D1, ActivityNode\rangle\rangle,$
$\langle\langle Exists, ActivityEdge\rangle, source, \langle D1, ActivityNode\rangle\rangle,$
$\langle\langle Exists, ActivityEdge\rangle, target, \langle CreditCardCheck, ActivityNode\rangle\rangle,$
$\langle\langle Verify, ActivityEdge\rangle, source, \langle CreditCardCheck, ActivityNode\rangle\rangle,$
$\langle\langle Verify, ActivityEdge\rangle, target, \langle D3, ActivityNode\rangle\rangle,$
$\langle\langle Correct, ActivityEdge\rangle, source, \langle D3, ActivityNode\rangle\rangle,$
$\langle\langle Incorrect, ActivityEdge\rangle, source, \langle D3, ActivityNode\rangle\rangle,$
$\langle\langle No, ActivityEdge\rangle, target, \langle D2, ActivityNode\rangle\rangle,$
$\langle\langle Incorrect, ActivityEdge\rangle, target, \langle D2, ActivityNode\rangle\rangle,$
$\langle\langle Exit, ActivityEdge\rangle, source, \langle D2, ActivityNode\rangle\rangle,$
$\langle\langle Exit, ActivityEdge\rangle, target, \langle CANCEL, ActivityNode\rangle\rangle,$
$\langle\langle Correct, ActivityEdge\rangle, target, \langle FINISH, ActivityNode\rangle\rangle\}\rangle$

(c) The Coq4MDE textual representation for credit card check

Fig. 2. An activity diagram model in the Coq4MDE notation

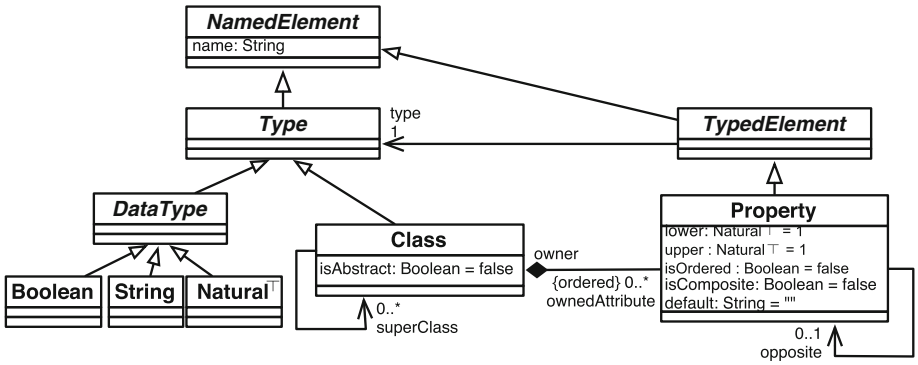


Fig. 3. The basic concepts of EMOF

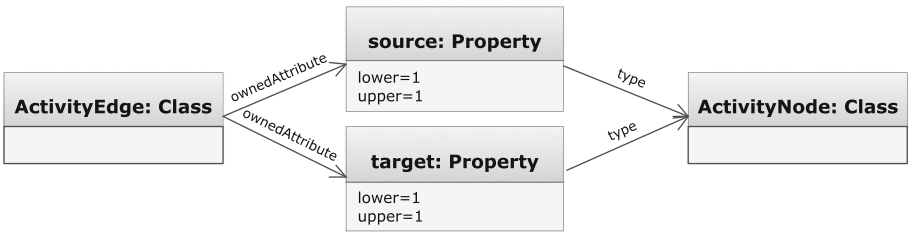


Fig. 4. Simplified activity diagram metamodel as a model conforms to EMOF

2.4. Composition operators

We present in this section several composition operators from the literature that are used later in the paper to illustrate our proposal. These operators share a significant part of their implementation and they can be formalized as compositions of elementary composition operators.

2.4.1. Composition in the aspect-oriented modeling approach

The concept of Separation of Concerns (SoC) is not a new one [Dij76, Par72], its key idea is the identification of the different concerns in a system and separate them by encapsulating them in appropriate modules that constitute independent parts of the developed software with explicit interfaces. The AOM [SSK+07] is an approach that adopts this idea and the principle behind, i.e., to allow designing in isolation the different aspects that make up an integrated system.

Aiming to automate the support for composing aspects models, France et al. in [FRGG04, FFR+07] proposed a model composition technique that allows for the conceptualization of logical dependability solution in isolation leading to so called aspect models. An integrated view of the system is produced by composing aspects and primary models. The main motivation for their work is to formulate a technique that uses rules for syntactically matching elements across models, which allow for a fully automation of the composition. The matching rules use syntactic deterministic properties to define the similarities between model elements. For instance, a matching rule can state that all the classes having the same name represent the same concept and allows in this case for merging classes having the same name and different attributes as provided by the OMG *PackageMerge* operator.

In the version presented in [FRGG04], an aspect-oriented architecture model of an application consists of (1) a base model, (2) one or several aspect models together with the bindings used for their instantiation in the application context, and (3) composition directives that define how the composition of the instantiated aspect models with the primary model should be performed to produce the composed model. Figure 5 [FR07] illustrates how an aspect-oriented architecture model consisting of two aspect models (aspect model 1 and aspect model 2) and a base model is composed. The aspect models are instantiated by binding template models parameters to specific values for the application. Composition of context-specific aspects and primary models produces the composite model.

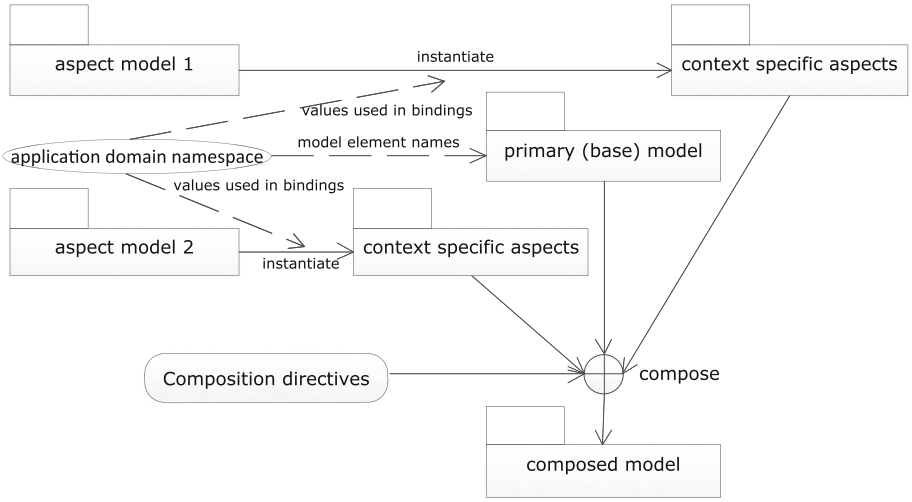


Fig. 5. An overview of composition in the AOM approach [FR07]

This AOM approach thus provides a basic composition procedure that can be altered in restricted ways with the help of composition directives. For example, a composition directive can (1) specify how to override conflicting properties in aspect models, (2) specify the modifications of the model elements during composition, and (3) determine the order in which the aspect models are composed. In this method, as explained in [FFR+07], a signature type is defined as a set of properties associated with a specific element type. A particular model element signature is composed of the values given to the properties associated with the element type.

In this method, the model elements are composed if they are of the same type or in other words if they are instances of the same meta model class. An aspect model may also contain a concept that is not present in a base model, and vice versa. In these cases, the model elements are added directly to the composed model.

An example is given in Fig. 6 (inspired from [RGF+06] and [KAAK09]) using the Kompose tool.[5] For the class diagrams, we follow the standard notation.[6] More precisely, the arrows specify the direction of the association while a solid line connecting two classes without arrow represents a binary association. In this example, a modeller creates a target model in which an instance of Writer (an output producer) sends outputs directly to the output device to which it is linked and that is an instance of FileStream. The modeller then decides to introduce a buffering feature into the model by composing a buffering aspect model. The aspect model describes how entities that produce outputs (represented by instantiations of Buffer) are decoupled from output devices through the use of buffers. The result of the composition of the base and aspect models is the model named Composed Class Diagram where the model elements with the same name have been combined.

Note that France et al. also proposed in [FFR+07] a language of directives to modify the models before and after the composition step. This language is useful to adapt a generic aspect model to a specific target model or to improve the composed model. We are interested in the pre-merge directives which specify simple model modifications that are to be made before the models are merged. We present a detailed example in the use cases section.

The aim of our work is to ensure that the composition is carried out in a safe manner (i.e., the model it produces has the intended properties), and for this there is a need to take into consideration the semantic properties when matching and merging model elements.

2.4.2. *ISC*

ISC is a generic technology for extending a Domain Specific Modeling Language (DSML) with model composition facilities. Its first version was defined for composing Java programs. A universal extension called U-ISC was proposed in [Hen09] that deals with textual components first described using context-free grammars and further

[5] http://www.kermeta.org/kompose.
[6] http://www.uml-diagrams.org/class-reference.html.

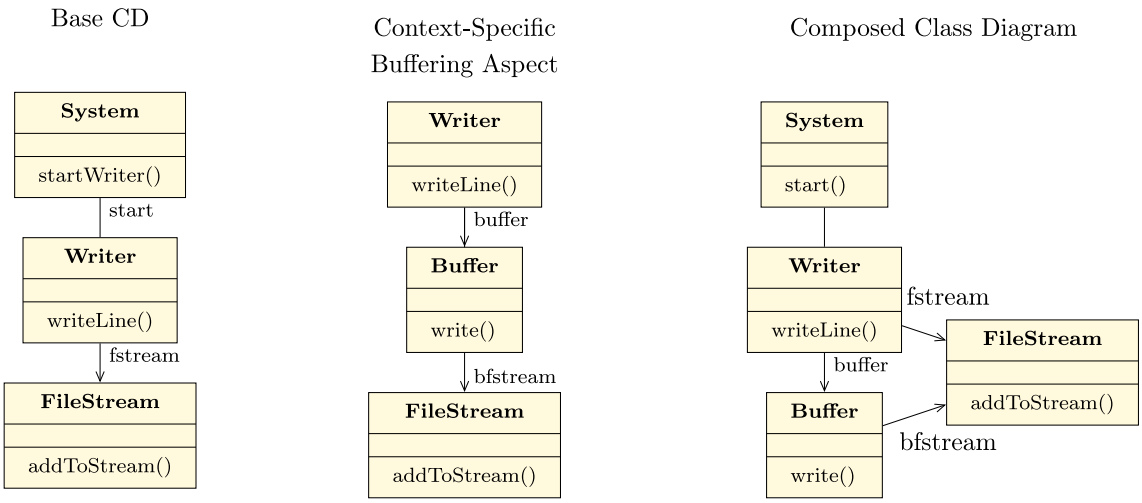


Fig. 6. Merging class diagrams with Kompose

as trees. The method uses tree merging to implement composition. In order to deal with graphical languages the method was extended to support typed graphs in [Jen11], and implemented in the ReuseWare framework. This implementation is consistent with the description of models as graphs in our Coq4MDE framework.

ISC introduces the fragment box structure to group model or source code fragments. The fragment box provides a composition interface with concepts and associated tools to assist with the composition. The composition interface for a fragment box is a set of addressable points. Two types of addressable points are defined, the variation points which are elements inside the fragment box that can be used as a receptor for other elements and reference points which are used to address some parts inside a fragment box. We formalize in our work one type of correspondence (variation/reference) points, the pair (hook/prototype). As described in [HHJZ09] a hook is a variation point that behaves as a place-holder for a fragment referenced by a prototype reference point.

We proposed in [KPCT11] and [KHPCT14a] an extension of the Coq4MDE framework to support ISC concepts and to define a sound basis for ensuring the correctness by construction for this composition style. These first formal embedding of ISC were done from scratch without trying to factorize or reuse elements. Here, we provide an alternate specification that encode the ISC basic operators using more elementary operators provided by our framework.

2.4.3. Package merge

Package Merge is an operator for composing packages defined at the metamodeling level. It "*can be used to quickly define new modeling languages, either by extending existing metamodels with new features through package merge or by building a new metamodel using existing packages as a base*" [Zit06]. Package Merge is a directed relationship between two packages (see Fig. 7), and indicates that the contents of the two packages are to be combined. It is very similar to Generalization in the sense that the source element conceptually adds the characteristics of the target element to its own characteristics resulting in an element that combines the characteristics of both [Obj13b]. The basic merge procedure is simple: any elements (for example, classes, associations, operations) in the target package with no corresponding element having the same name in the source package are simply included in the resulting package. If several elements have the same name, they are combined into one element and their features are combined recursively [Zit06].

We proposed in [KHPCT14b] a formalization of the Package Merge with fully verified properties using some elementary operators that we present in the next section. The proposed formalization takes into account the conflicts between the models in relation with abstract classes and the multiplicities (the lower and upper bounds of attributes and relations). The conflicts are resolved according to the UML specification [Obj13b]. In this paper, we present a more general picture of our framework and illustrate our proposal with more generic composition methods like the composition in AOM and ISC.

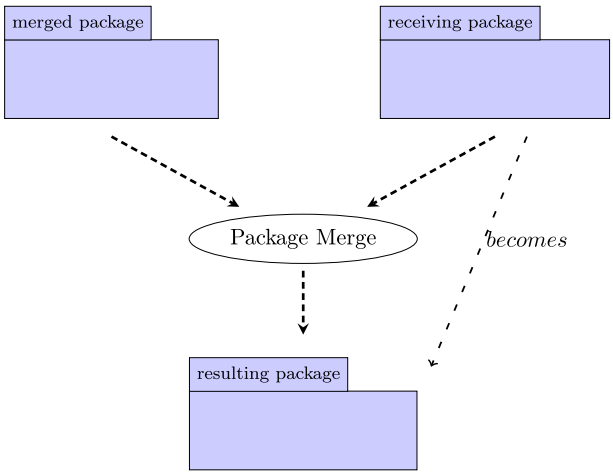


Fig. 7. Conceptual view of the package merge [Obj13b]

3. Expected property and verification strategy

3.1. Expected property

The expected property is the preservation of the metamodel conformance during composition. For a model M and a metamodel MM, this property checks that: (1) every object o in M was created from a class C in MM. (2) every relation between two objects in M is such that there exists, in MM, a reference between the two classes used for creating the two objects. (3) every semantic property defined in MM is satisfied in M. The semantic properties from `EMOF` (see Fig. 3) are: Inheritance (`subClass` and `classicInheritance`), Abstract classes (`isAbstract`), Multiplicities (`lower`, `upper`), Opposite (`isOpposite`) and Composite (`areComposite`) references.

As verifying these properties directly for the AOM composition approach, `ISC` composition operators, or `MOF` Package Merge operator is complex and contains many common aspects, we have selected a divide and conquer approach to capture these commonalities. This approach resulted from experimenting with the other composition approaches after early work with `ISC`. Common aspects occurred in the various formalization that were factorized thereafter leading to the full proposal described in this paper.

3.2. Verification strategy

We advocate the use of generic primitive composition operators that can be used to specify and prove more complex ones. We aim for a pragmatic compositional verification: minimize the residual verification that must be conducted on the result of the composition of correct models. We rely on a simple methodology to design the contract (pre and post conditions) for the composition operators. If Φ is the expected property for a model built using composition operators, then Φ must be, on the one hand, the postcondition on the model resulting from the application of each operator; and, on the other hand, the precondition of the parameters of each of the operators involved. These preconditions are eventually consolidated using an additional glueing property Ψ which depends on the value of all the parameters of the operator. Ψ is the residual property that must be checked for each composition. This approach is common for compositional verification strategies.

Definition 3 (*Correct composition operator*) For a set of models $m_1, \ldots, m_n$ and an n-ary composition operator f over models, we say that f is correct (or property preserving) with respect to a property Φ and a glueing condition Ψ if:

$$\bigwedge_{1 \leq i \leq n} \Phi_{m_i} \wedge \Psi_{m_1,\ldots,m_n} \Rightarrow \Phi_{f(m_1,\ldots,m_n)}$$

The formalization of the composition operators and the correctness verifications are carried out using the Coq proof assistant. Details are given in the next section.

4. Formal primitive composition operators

The primitive operators need to be addressed at all three layers of the Coq4MDE encoding of models i.e., Sets layer, the graph layer, and the model layer.

Set structure A finite set is a structure that can store elements, without any order or repetition of values. Sets in type theory, in particular the vertex and edge sets in our framework are represented by their characteristic or indicator function which ensure by definition that each element is unique. The characteristic function of a set A, denoted as $\chi_A(x)$, is implemented in the library Uniset[7] from the Coq standard library as follows. $\chi_A(x)$ takes the value $true$ if x is a member of the set A, and takes the value $false$ otherwise. The sets of vertices and edges are structured into two modules E and V respectively (the module structure is implemented in Coq since its version 7.4 [Chr03]). These generic modules can be parametrized using different types and include the definition of the function add that adds an element to a set of vertices (V.add) or a set of edges (E.add), the function $image$ which will be defined in 4.1.1, together with their proved properties.[8]

The graph structure A graph is defined in Coq as an inductive dependent type with three constructors: `Nil`[9], `AddV` and `AddA`.

$$\frac{}{graph(EmptySet,\ EmptySet)}\,Nil$$

$$\frac{graph(vs,\ es) \wedge (v \notin vs)}{(graph((V.add\ v\ vs),\ es))}\,AddV$$

$$\frac{graph(vs,\ es) \wedge (v_1 \in vs) \wedge (v_2 \in vs) \wedge (\langle v_1,\ e,\ v_2\rangle \notin es)}{(graph(vs,\ (E.add\ \langle v_1,\ e,\ v_2\rangle\ es)))}\,AddA$$

The model structure As discussed in Sect. 2.2, a model is denoted as $\langle vs,\ es\rangle$ where vs is a finite set of typed objects and es is a set of reference edges is constructed from the sets vs and es associated with the proof that these two sets build a multigraph.

4.1. Definition of the primitive operators

We are interested in two basic operations which were, in our experiments, sufficient to implement complex composition operators.

4.1.1. Substitution

The model elements are typed objects. For example $\langle x,\ c\rangle$ is a model element whose type is c and name is x. The `Substitution` operator replaces a model element name by another name. This operation as well as the `Union` operator need to be described at all three hierarchical layers of the encoding of models.

Set layer Substituting a model element name o_1 by a model element name o_2 in the vertices and edges sets is done using three operations: $mapv$, $mapa$ and $mape$.

$mapv$ is used to map o_1 to o_2 in the set of vertices. It is defined as follows:

$$mapv\ \langle o_1,\ c_1\rangle\ \langle o_2,\ c_2\rangle\ \langle x,\ c\rangle = \begin{cases} \langle o_2,\ c\rangle\ if\ x = o_1 \\ \langle x,\ c\rangle\ otherwise \end{cases}$$

An edge is defined as having two ends (two model elements) and an edge label. For example $\langle\langle v_1,\ v_2\rangle,\ a\rangle$ is the edge between v_1 and v_2 whose label is a. The function $mapa$ defines the edges labels image by the `Substitution`

[7] http://coq.inria.fr/stdlib/Coq.Sets.Uniset.html.
[8] coq4mde.enseeiht.fr/FormalMDE/Subst_Verif.html.
[9] EmptySet in the definition represents the empty set.

operator, and it is defined in our case as an identity function $mapa\ a = a$.[10] The function $mape$ maps the model element names in the edges such as:

$$mape\ \langle\langle v_1, v_2\rangle, a\rangle = \langle\langle mapv\ o_1\ o_2\ v_1, mapv\ o_1\ o_2\ v_2\rangle, mapa\ a\rangle$$

The graph layer The image of the sets of vertices or edges of a graph g with vs the set of vertices and es the set of edges using $mapv$ or $mape$ is given, using the functions $image$ encoded for the vertices and edges respectively in the modules V and E previously mentioned,[11] by $(V.image\ mapv\ (vs, es)\ g)$ or $(E.image\ mapv\ mapa\ (vs, es)\ g)$. Coding sets as characteristic functions allows for deciding efficiently whether an element belongs or not to a set. It is difficult, however, to iterate over the elements of a set, e.g., to define the $V.image$ that applies $mapv$ on all elements in the set of vertices. A special *fold* function is used to implement $V.image$ and $E.image$ (the function that applies $mape$ on all the edges). The *fold* function applies $mapv$ or $mape$ on every vertex or edge of the graph and constitutes the image of the vertices set and the image of the edges set. The image of a graph is constructed from images of its vertices and edges. This property is formalized in Theorem 4.1 which shows that such graph always exists; this theorem is proved by induction on the structure of the graph[12] and allows for the definition of the structure of the resulting graph from the substitution.

Theorem 4.1 (graph substitution)
$\forall\ vs \in Classes,\ es \in References, g \in graph(vs, es),$
$graph\ (V.image\ mapv\ (vs, es)\ g)\ (E.image\ mapv\ mapa\ (vs, es)\ g)$

The model layer Substituting a model element o_1 by another o_2 in the vertex or edge sets in addition to the graph image of substitution (defined by Theorem 4.1) constitute the substituted model. The operator $Substitution : (Objects \times Classes) \times (Objects \times Classes) \times Model \rightarrow Model$ is defined as:

$$\begin{aligned} &Substitution\ o_1\ o_2\ \langle MV, ME\rangle = \langle SubstV\ o_1\ o_2\ MV, SubstE\ o_1\ o_2\ ME\rangle \\ &\quad where\ (SubstV\ o_1\ o_2\ MV)\ substitutes\ o_1\ by\ o_2\ in\ MV \\ &\quad and\ (SubstE\ o_1\ o_2\ ME)\ substitutes\ o_1\ by\ o_2\ in\ ME. \end{aligned}$$

4.1.2. Union

Set layer We also make use of the `Union` operator $\cup$, for characteristic functions, which is implemented in Uniset as well. The set union is implemented using the boolean disjunction such that, x is a member of $A_1 \cup A_2$ if and only if the value of $\chi_{A_1}(x) \vee \chi_{A_2}(x)$ is $true$.

$$\chi_{A_1 \cup A_2}(x) = \chi_{A_1}(x) \vee \chi_{A_2}(x)$$

The graph layer The union of two graphs is also a graph, the set of vertices is the union of vertex sets and the set of edges is the union of the edge sets of the two models. The graph union is formalized in Theorem 4.2, the proof that the two sets make a graph is by induction on the structure of the second graph $(graph(vs_2, es_2))$.[13]

Theorem 4.2 (graph union)
$\forall\ vs_1\ vs_2 \in \mathscr{C}, es_1\ es_2 \in \mathscr{R}$
$graph(vs_1, es_1) \wedge graph(vs_2, es_2) \rightarrow graph(vs_1\ \cup vs_2, es_1 \cup es_2)$

The model layer The `Union` of two models is the union of their vertex and edge sets in addition to the proof of the graph union (Theorem 4.2). The operator $Union : Model \times Model \rightarrow Model$ is defined as:

$$Union\ \langle MV_1, ME_1\rangle\ \langle MV_2, ME_2\rangle = \langle MV_1 \cup MV_2, ME_1 \cup ME_2\rangle$$

[10] This function can allow for the definition of the image of the a label for other operators.
[11] coq4mde.enseeiht.fr/FormalMDE/Subst_Verif.html.
[12] http://coq4mde.enseeiht.fr/FormalMDE/Subst_Verif.html#elements.
[13] http://coq4mde.enseeiht.fr/FormalMDE/Graph.html#elements.

4.2. Properties preservation for the primitive composition operators

The `Substitution` and `Union` operators are defined in order to enforce both the well typedness and `MOF` semantic properties. These two operators like all the concepts, theorems and proofs presented in this paper are encoded in the Coq proof assistant and are available on the Formal MDE web page.[14] The aim of this formalization is to help with checking some properties on the composite models and also provide a formal basis for the specification and proof of correctness of compositional verification technologies. The first property considered is the well typedness property. This property is related to the conformance defined in Sect. 2.2. It checks for a model M and a metamodel MM that every object in M is the instance of a class in MM and that every link in M is an instance of a relation in MM. To prove that this verification is compositional, we need to prove that the composition of two model instances of the same metamodel is also an instance of the same metamodel. We define the first validity criterion for any composition function. This criterion is defined as a higher order predicate that checks the well typedness of some operator. The function `InstanceOf` is used for this purpose, it checks that all objects and links of a model are instances of classes and references in a metamodel. The function $InstanceOf : Model \times MetaModel \rightarrow Bool$ is defined as:

$$InstanceOf(\langle MV, ME\rangle, \langle (MMV, MME), conformsTo\rangle) \triangleq$$
$$\forall \langle o, c\rangle \in MV, c \in MMV \wedge \forall \langle \langle o, c\rangle, r, \langle o', c'\rangle\rangle \in ME, \langle c, r, c'\rangle \in MME$$

Then, this predicate is used to verify that the result of applying the `Substitution` function to an instance of MM is also an instance of MM. This is described in Theorem 4.3.[15]

Theorem 4.3 (ValidSubstitution)
$$\forall\, M \in Model, MM \in MetaModel,\ o_1\ o_2 \in (Objects \times Classes), InstanceOf\ (M, MM)$$
$$\rightarrow InstanceOf\ ((Substitution\ o_1\ o_2\ M), MM)$$

Proof Sketch of Theorem 4.3 Let M be an instance of metamodel MM. We prove that the model obtained by applying the `Substitution` operator to this model using any two model elements o_1 and o_2 is also an instance of the metamodel MM. We show that the `Substitution` operator does not change the types of the vertices and edges and so preserves the conformance to the metamodel. □

Many frameworks for model specification such as the Unified Modeling Language (`UML`) and the Eclipse Modeling Framework (EMF) are based on the `MOF` standard. We present an elegant way to take into account the `MOF` metametamodel constraints and to verify that the basic operators preserve the conformity with respect to the metamodel's semantic properties (other than the type safety). This approach avoids extracting the properties from the metamodel which is not easily accomplished. The solution is to check that each elementary property verified by the initial models is also verified on the model resulting from the application of a basic operator. If the initial models are conforming to some metamodel, the resulting model is also conforming to the same metamodel. The elementary semantic properties are: the hierarchy (`subClass` and `standardInheritance`), the abstract classes (`isAbstract`), the multiplicities (`lower`, `upper`), the opposite references (`isOpposite`) and the composite references (`areComposite`).

In the following, we present for every elementary semantic property its formalization, the lemma that formalizes the preservation of the property for the `Substitution` basic operator, a proof sketch, and the link to the complete proof in Coq.

subClass The `subClass` property formalization is given in Sect. 2. Theorem 4.4[16] states that this property is preserved by the `Substitution` operator. For any classes c_1 c_2 and for any model elements o_1 o_2, if c_1 is a subclass of c_2 in a model M, then c_1 is also a subclass of c_2 in $(Substitution\ o_1\ o_2\ M)$.

Theorem 4.4 (SubstSubClassPreserved)
$$\forall\, M \in Model,\ c_1\ c_2 \in Classes,\ o_1\ o_2 \in (Objects \times Classes),$$
$$subClass\ c_1\ c_2\ M \rightarrow subClass\ c_1\ c_2\ (Substitution\ o_1\ o_2\ M)$$

[14] http://coq4mde.enseeiht.fr/FormalMDE.
[15] http://coq4mde.enseeiht.fr/FormalMDE/Subst_Verif.html#ValidSubst.
[16] http://coq4mde.enseeiht.fr/FormalMDE/Subst_Verif.html#SubstSCP.

Proof Sketch of Theorem 4.4 We prove that any two classes c_1 and c_2 linked by the inh relation in a model M, are also linked by the inh relation in the model obtained from the `Substitution` of any two model elements o_1 and o_2 in the model M. We suppose that we have a relation inh between any two model elements with types c_1 and c_2 in the model M. We show that the `Substitution` operator does not change the types of model elements and the types of relations and, thus in the resulted model we always have an inh relation between any model elements with types c_1 and c_2. The Coq proof is long but straightforward and considers all the cases of equality between the name of any model element typed by c_1 and the names of the model elements o_1 and o_2 and shows in all cases that the existing inh relations are preserved. □

standardInheritance The `standardInheritance` property formalization is given in Section 2. Theorem 4.5 states that this property is preserved by the application of the `Substitution` operator for a model M if for every couple of objects o_1 and o_2 both in M, and for every couple c_1 and c_2 of constructor classes of o_1 respectively o_2, either $\langle o_1, c_1\rangle$ or $\langle o_2, c_2\rangle$ is in M.

Theorem 4.5 (SubstStandardInheritancePreserved)
$\forall\ M \in Model,\ o_1\ o_2 \in Objects, c_1\ c_2 \in Classes,$
$condStandardInhSubst\ M\ o_1\ o_2 = true \wedge standardInheritance\ M$
$\rightarrow standardInheritance\ (Substitution\ (o_1, c_1)\ (o_2, c_2)\ M)$

The gluing condition is given by $condStandardInhSubst$:

$condStandardInhSubst(\langle MV, ME\rangle \in Model,\ o_1 \in \texttt{Objects},\ o_2 \in \texttt{Objects}) \triangleq$
$\forall\ c_1\ c_2 \in \texttt{Classes}, isConstructorClass(o_1, c_1, \langle MV, ME\rangle) \wedge isConstructorClass(o_2, c_2, \langle MV, ME\rangle)$
$\rightarrow \langle o_1, c_1\rangle \in MV \vee \langle o_2, c_2\rangle \in MV$

isAbstract The `isAbstract` property is also formalized in the Sect. 2. The preservation of this property by the `Substitution` basic operator is proved using Theorem `SubstIsAbstractPreserved`.[17] This theorem states that all the abstract classes in some model are also abstract in the model after substitution.

Theorem 4.6 (SubstIsAbstractPreserved)
$\forall\ M \in Model,\ c \in Classes,\ o_1\ o_2 \in (Objects \times Classes),$
$isAbstract\ c\ M \rightarrow isAbstract\ c\ (Substitution\ o_1\ o_2\ M)$

Proof Sketch of Theorem 4.6 We prove that any abstract class c in any model M, is also abstract in the model obtained by the `Substitution` of any two model elements o_1 and o_2 using in the model M. We suppose that the class c is abstract in the model M. We show that the `Substitution` operator does not introduce a concrete instance for the element typed by c and that this class stays abstract in the model after substitution. □

Lower and upper For both attributes and references, a minimum and maximum number of instances of the target concept can be defined using the `lower` and `upper` attributes. This pair is usually referred to as *multiplicity*. In order to ease the manipulation of this data-type, we introduce the type $\text{Natural}^\top = \mathbb{N} \cup \{\top\}$. Using both attributes, it is used to represent a range of possible numbers of instances. Unbounded ranges can be modelled using the $\top$ value for the `upper` attribute.

$lower(c_1 \in MMV, r_1 \in MME, n \in Natural^\top, \langle MV, ME\rangle) \triangleq \forall \langle o, c\rangle \in MV,$
$c = c_1 \rightarrow |\{m_2 \in MV \mid \langle\langle o, c_1\rangle, r_1, m_2\rangle \in ME\}| \geq n$

An analogous formalization is defined for the `upper` property replacing $\geq$ by $\leq$. Theorem 4.7[18] states that starting from any two models conforming to the same metamodel; the preservation of the `lower` property is not trivial and requires some gluing condition to be satisfied in the two models. The condition in this case requires the `Substitution` operator to be injective, i.e., not allowing for two distinct elements to be mapped to the same element, thus preserving the uniqueness in the resulting model. This is ensured if the object o_2 is not in the substituted model, and therefore, the `Substitution` operator does not add an element that is already in the model. The `lower` bounds are thus preserved. This verification is generic and can be applied for any metamodel where the `lower` property is specified independently about the considered models.

[17] http://coq4mde.enseeiht.fr/FormalMDE/Subst_Verif.html#SubstIAP.
[18] http://coq4mde.enseeiht.fr/FormalMDE/Subst_Verif.html#SubstLP.

Theorem 4.7 (SubstLowerPreserved)
$\forall\ \langle MV, ME\rangle \in Model,\ c \in Classes,\ r \in References,\ n \in Natural^{\top},$
$\langle o_1, c_1\rangle\ \langle o_2, c_2\rangle \in (Objects \times Classes),$
$c_1 = c_2 \wedge (\nexists \langle o, c\rangle \in MV \wedge o = o_2) \wedge Injective\ Substitution \wedge (lower\ c\ r\ n\ \langle MV, ME\rangle)$
$\rightarrow (lower\ c\ r\ n\ (Substitution\ \langle o_1, c_1\rangle\ \langle o_2, c_2\rangle\ \langle MV, ME\rangle)).$

Proof Sketch of Theorem 4.7 Let us assume that, for a model $\langle MV, ME\rangle$, a lower bound n is satisfied for the class c in relation with the reference r (maximum n model elements are related by the relation r to the same instance of the class c). We prove that this lower bound n is also satisfied in the model obtained from the `Substitution` of any two model's elements o_1 and o_2 using the model $\langle MV, ME\rangle$. We show that the `Substitution` operator does not change the types of the model elements and does not reduce the lower bound in the resulting model because the `Substitution` operator is injective and so does not introduce new model elements duplications. The Coq proof is long and uses intermediate lemmas to simplify iterations and calculations of the links and the model elements. □

The `upper` property preservation is described using Theorem `SubstUpperPreserved`[19] which is similar to the previous theorem for the `lower` property. The model must not contain an element whose name is o_2.

isOpposite A reference can be associated to an *opposite* reference. It implies that, in valid models, for each such a link between two object instances, there must exist a link in the opposite direction between these two same objects.

$$isOpposite(r_1, r_2 \in MME, \langle MV, ME\rangle) \triangleq \forall\ m_1, m_2 \in MV, \langle m_1, r_1, m_2\rangle \in ME \leftrightarrow \langle m_2, r_2, m_1\rangle \in ME$$

Theorem `SubstIsOppositePreserved`[20] states that any two references opposite in some model remain also opposite in the model after applying the `Substitution` operator. So, the `isOpposite` property is preserved.

Theorem 4.8 (SubstIsOppositePreserved)
$\forall\ M \in Model,\ r_1\ r_2 \in References,\ o_1\ o_2 \in (Objects \times Classes), (isOpposite\ r_1\ r_2\ M)$
$\rightarrow (isOpposite\ r_1\ r_2\ (Substitution\ o_1\ o_2\ M)).$

Proof Sketch of Theorem 4.8 We prove that any two references r_1 and r_2 that are opposite in a model M, are also opposite in the model obtained from the `Substitution` of any two model elements o_1 and o_2 in the model M. We show that the `Substitution` operator does not change the references and so we can find all the opposite references from the initial models. □

areComposite A reference can be *composite* and as a matter of fact, defining a set of references as a whole to be composite, instead of a single one, is closer to the intended meaning. In such a case, instances of the target concept belong to a single instance of source concepts.

$$areComposite(c_1 \in MMV, R \subseteq MME, \langle MV, ME\rangle) \triangleq \forall\ \langle o, c\rangle \in MV,$$
$$c = c_1 \rightarrow\ \ |\{m_1 \in MV \mid \langle m_1, r, \langle o, c\rangle\rangle \in ME, r \in R\}| \leq 1$$

Theorem 4.9[21] states that, a set of references which are composite in some model are also composite in the model obtained after applying the `Substitution` operator. This theorem also assumes that the `Substitution` operator is injective and requires that the substituted model does not contain an element named o_2.

Theorem 4.9 (SubstAreCompositeSubsPreserved)
$\forall\ \langle MV, ME\rangle \in Model,\ c \in Classes,\ r \subset References,\ \langle o_1, c_1\rangle\ \langle o_2, c_2\rangle \in (Objects \times Classes),$
$c_1 = c_2 \wedge (\nexists \langle o, c\rangle \in MV \wedge o = o_2) \wedge (\texttt{Injective}\ Substitution) \wedge\ (areComposite\ c\ r\ \langle MV, ME\rangle)$
$\rightarrow (areComposite\ c\ r\ (Substitution\ \langle o_1, c_1\rangle\ \langle o_2, c_2\rangle\ \langle MV, ME\rangle))$

[19] http://coq4mde.enseeiht.fr/FormalMDE/Subst_Verif.html#SubstUP.
[20] http://coq4mde.enseeiht.fr/FormalMDE/Subst_Verif.html#SubstIOP
[21] http://coq4mde.enseeiht.fr/FormalMDE/Subst_Verif.html#SubstACP.

Proof Sketch of Theorem 4.9 Let us assume that, for any model $\langle MV, ME\rangle$, and for any instance of a class c in this model, at most one ancestor is linked with a composite reference. We verify that this property is also satisfied in the model obtained from the substitution of any two model elements o_1 and o_2 using this model. We show that the `Substitution` operator does not change the types of the model elements and does not increase the number of composite references for any model element. We prove this using the fact that the `Substitution` operator is injective and thus it does not introduce new model elements duplications. The Coq proof is long and uses intermediate lemmas to simplify iterations and calculations of the references and models elements. The proof considers all the cases of equality between the name of any instance of c in both models and the names of the model elements o_1 and o_2 and shows in all cases that the limit for the number of composite relation for any model element is preserved. □

The same properties are verified for the `Union` operator. The `Union` operator given in Sect. 4.1.2 requires other assumptions to ensure that all the properties are satisfied.[22]

To preserve the property `standardInheritance`, the two models M_1 and M_2 must satisfy the condition `condStandardInhUnion` that can be explained as: for every object o both in M_1 and M_2 and for every couple c_1 and c_2 of constructor classes of o respectively in M_1 and M_2, either $\langle o, c_1\rangle$ is in M_1 or $\langle o, c_2\rangle$ is in M_2.

$$\begin{aligned}&condStandardInhUnion(\langle MV_1, ME_1\rangle, \langle MV_2, ME_2\rangle \in Model) \triangleq\\ &\forall \langle o, c\rangle \in MV_1, \langle o, c'\rangle \in MV_2, c_1\ c_2 \in Classes,\\ &isConstructorClass(o, c_1, \langle MV_1, ME_1\rangle) \wedge isConstructorClass(o, c_2, \langle MV_2, ME_2\rangle)\\ &\quad \rightarrow \langle o, c_1\rangle \in MV_1 \vee \langle o, c_2\rangle \in MV_2\end{aligned}$$

To preserve the `lower` property, i.e. to prove $(lower\ c\ r\ n)$ for the model obtained from the `Union` of the two models $\langle MV_1, ME_1\rangle$ and $\langle MV_2, ME_2\rangle$ that satisfy the `lower` property, the cardinality n of the object o typed by the class c in relation with the reference r must satisfy the following condition:

$$\begin{aligned}&lowerCond(\langle o, c\rangle \in (Objects \times Classes), n \in Natural^{\top}, \langle MV_1, ME_1\rangle\ \langle MV_2, ME_2\rangle \in Model) \triangleq\\ &\quad n \geq |\{o_2 \in (MV_1 \cap MV_2) \mid \langle\langle o, c\rangle, r, o_2\rangle \in (ME_1 \cap ME_2)\}|\end{aligned}$$

It follows thus, that the number of links r from an element of type c in the intersection of the two models must be lower or equal to its cardinality n from the metamodel.

To preserve the `upper` property in the model resulting from the `Union` of the two models $\langle MV_1, ME_1\rangle$ and $\langle MV_2, ME_2\rangle$ must satisfy the `upper` property. The multiplicity n must satisfy the `upperCond` condition:

$$\begin{aligned}&upperCond(\langle o, c\rangle \in Objects, n \in Natural^{\top}, \langle MV_1, ME_1\rangle\ \langle MV_2, ME_2\rangle \in Model) \triangleq\\ &\quad n > |\{o_2 \in MV_1 \mid \langle\langle o, c\rangle, r, o_2\rangle \in ME_1\}| + |\{o_2 \in MV_2 \mid \langle\langle o, c\rangle, r, o_2\rangle \in ME_2\}|\\ &\qquad -|\{o_2 \in (MV_1 \cap MV_2) \mid \langle\langle o, c\rangle, r, o_2\rangle \in (ME_1 \cap ME_2)\}|\end{aligned}$$

It follows, thus, that the cardinality n of a link r in relation with an element whose type is c must be higher then the number of links r from an element of type c in both the first and the second models.

A similar condition to the `upperCond` which replaces n by the value 1 is necessary to verify the property `areComposite`. Thus, for a composite reference instance, the target concept belongs to a single instance of the source concept which must be the same in both models.

5. Formalization of high level composition operators

In this section, we present the usage of the two elementary operators `Union` and `Substitution` for describing the high level composition operators presented in Sect. 2.4.

5.1. Aspect oriented model composition

A composition of models corresponding to design views following the AOM approach [FRGG04] is structured into two major phases (1) the matching phase where model elements that describe different views of the same concept are identified, (2) the merging phase where the matched model elements are merged to create the integrated

[22] http://coq4mde.enseeiht.fr/FormalMDE/Union_Verif.html.

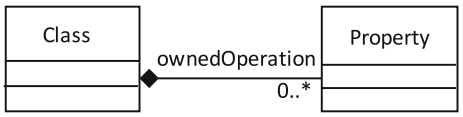


Fig. 8. An extension of EMOF concepts presented in Fig. 3

views. We show in the first example that the basic match is supported by the `Union` operator. In the second example, we show that more sophisticated matches can be given manually (represented as directives) and unification can be done using the `Substitution` operator. The following examples require the representation of operations associated with classes, and to support this property we extend the concepts of `EMOF` presented in Fig. 3 via a relation *ownedOperation* as presented in Fig. 8.

5.1.1. Example 1

We illustrate first how a composition can be carried out using the small example in Fig. 6, the Base CD model is represented in Coq4MDE using the set of vertices MV_{BaseCD} and the set of edges ME_{BaseCD} as follows:

$$\begin{aligned} MV_{BaseCD} \triangleq\ & \{\, \langle System, Class\rangle, \langle Writer, Class\rangle, \langle FileStream, Class\rangle, \\ & \langle start, Property\rangle, \langle fstream, Property\rangle, \langle startWriter, Property\rangle, \\ & \langle writeLine, Property\rangle, \langle addToStream, Property\rangle \,\} \end{aligned}$$

$$\begin{aligned} ME_{BaseCD} \triangleq\ & \{\, \langle\langle System, Class\rangle, \texttt{ownedOperation}, \langle startWriter, Property\rangle\rangle, \\ & \langle\langle System, Class\rangle, \texttt{ownedAttribute}, \langle start, Property\rangle\rangle, \\ & \langle\langle start, Property\rangle, \texttt{type}, \langle Writer, Class\rangle\rangle, \\ & \langle\langle Writer, Class\rangle, \texttt{ownedOperation}, \langle writeLine, Property\rangle\rangle, \\ & \langle\langle Writer, Class\rangle, \texttt{ownedOperation}, \langle fstream, Property\rangle\rangle, \\ & \langle\langle fstream, Property\rangle, \texttt{type}, \langle FileStream, Class\rangle\rangle, \\ & \langle\langle FileStream, Class\rangle, \texttt{ownedOperation}, \langle addToStream, Property\rangle\rangle \,\} \end{aligned}$$

The Context-Specific Buffering Aspect is represented in Coq4MDE as follows:

$$\begin{aligned} MV_{CSBA} \triangleq\ & \{\, \langle Writer, Class\rangle, \langle Buffer, Class\rangle, \langle FileStream, Class\rangle, \\ & \langle buffer, Property\rangle, \langle bfstream, Property\rangle, \langle writeLine, Property\rangle, \\ & \langle write, Property\rangle, \langle addToStream, Property\rangle \,\} \end{aligned}$$

$$\begin{aligned} ME_{CSBA} \triangleq\ & \{\, \langle\langle Writer, Class\rangle, \texttt{ownedOperation}, \langle writeLine, Property\rangle\rangle, \\ & \langle\langle Writer, Class\rangle, \texttt{ownedAttribute}, \langle buffer, Property\rangle\rangle, \\ & \langle\langle buffer, Property\rangle, \texttt{type}, \langle Buffer, Class\rangle\rangle, \\ & \langle\langle Buffer, Class\rangle, \texttt{ownedOperation}, \langle write, Property\rangle\rangle, \\ & \langle\langle Buffer, Class\rangle, \texttt{ownedAttribute}, \langle bfstream, Property\rangle\rangle, \\ & \langle\langle bfstream, Property\rangle, \texttt{type}, \langle FileStream, Class\rangle\rangle, \\ & \langle\langle FileStream, Class\rangle, \texttt{ownedOperation}, \langle addToStream, Property\rangle\rangle \,\} \end{aligned}$$

The union of the two models *BaseCD* and *CSBA* corresponds exactly to the intended model in Fig. 6. The sets of vertices and edges of the obtained model are represented in Coq4MDE as follows:

$$\begin{aligned} MV_{Union\ BaseCD\ CSBA} \triangleq\ & \{\, \langle System, Class\rangle, \langle Writer, Class\rangle, \langle FileStream, Class\rangle, \\ & \langle start, Property\rangle, \langle fstream, Property\rangle, \langle startWriter, Property\rangle, \\ & \langle writeLine, Property\rangle, \langle addToStream, Property\rangle, \langle Buffer, Class\rangle, \\ & \langle buffer, Property\rangle, \langle bfstream, Property\rangle, \langle write, Property\rangle \,\} \end{aligned}$$

$$ME_{Union\ BaseCD\ CSBA} \triangleq \{\ \langle\langle System, Class\rangle, \texttt{ownedOperation}, \langle startWriter, Property\rangle\rangle,$$
$$\langle\langle System, Class\rangle, \texttt{ownedAttribute}, \langle start, Property\rangle\rangle,$$
$$\langle\langle start, Property\rangle, \texttt{type}, \langle Writer, Class\rangle\rangle,$$
$$\langle\langle Writer, Class\rangle, \texttt{ownedOperation}, \langle writeLine, Property\rangle\rangle,$$
$$\langle\langle Writer, Class\rangle, \texttt{ownedAttribute}, \langle fstream, Property\rangle\rangle,$$
$$\langle\langle fstream, Property\rangle, \texttt{type}, \langle FileStream, Class\rangle\rangle,$$
$$\langle\langle FileStream, Class\rangle, \texttt{ownedOperation}, \langle addToStream, Property\rangle\rangle,$$
$$\langle\langle buffer, Property\rangle, \texttt{type}, \langle Buffer, Class\rangle\rangle,$$
$$\langle\langle Writer, Class\rangle, \texttt{ownedAttribute}, \langle buffer, Property\rangle\rangle,$$
$$\langle\langle Buffer, Class\rangle, \texttt{ownedOperation}, \langle write, Property\rangle\rangle,$$
$$\langle\langle Buffer, Class\rangle, \texttt{ownedAttribute}, \langle bfstream, Property\rangle\rangle,$$
$$\langle\langle bfstream, Property\rangle, \texttt{type}, \langle FileStream, Class\rangle\rangle\ \}$$

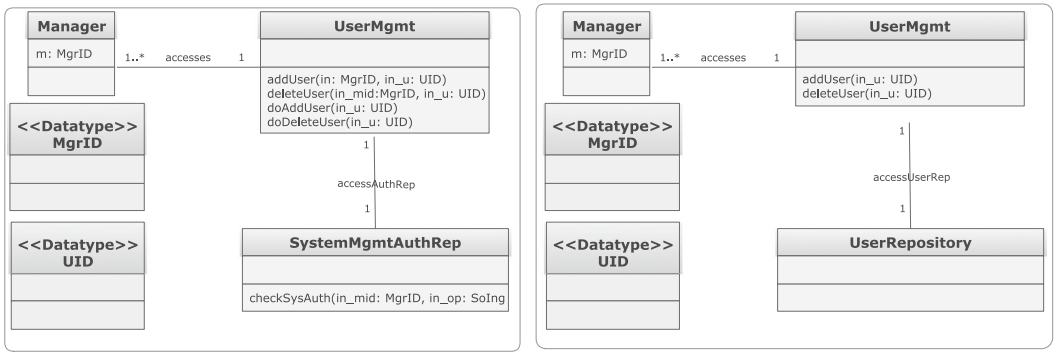
5.1.2. Example 2

We use here an example from [FRGG04]. Figure 9b which shows a primary model that describes a user management system in which Manager objects are linked to a UserMgmt object that controls access to a repository of user information (a UserRepository object). The UserMgmt class defines operations for adding a user to the repository (addUser) and for deleting a user from the repository (deleteUser). Access to the addUser and deleteUser operations by Manager objects is unrestricted in the primary model. The corresponding model for the User Management primary class diagram presented in Fig. 9b written manually in the CoQ4MDE syntax is $\langle MV_{UserManagement}, MV_{UserManagement}\rangle$ where the two sets of vertices and edges are completely represented as follows:

$$MV_{UserManagement} \triangleq \{\ \langle Manager, Class\rangle, \langle UserMgmt, Class\rangle, \langle UserRepository, Class\rangle,$$
$$\langle m, Property\rangle, \langle addUser, Property\rangle, \langle deleteUser, Property\rangle,$$
$$\langle MgrID, Datatype\rangle, \langle UID, Datatype\rangle, \langle in\_u, Property\rangle,$$
$$\langle accesses, Property\rangle, \langle accessUserRep, Property\rangle\ \}$$

$$ME_{UserManagement} \triangleq \{\ \langle\langle Manager, Class\rangle, \texttt{ownedAttribute}, \langle m, Property\rangle\rangle,$$
$$\langle\langle m, Property\rangle, \texttt{type}, \langle MgrID, Datatype\rangle\rangle,$$
$$\langle\langle Manager, Class\rangle, \texttt{ownedAttribute}, \langle accesses, Property\rangle\rangle,$$
$$\langle\langle accesses, Property\rangle, \texttt{type}, \langle UserMgmt, Class\rangle\rangle,$$
$$\langle\langle accesses, Property\rangle, \texttt{ownedAttribute}, \langle 1, lower_{UserMgmt}\rangle\rangle,$$
$$\langle\langle accesses, Property\rangle, \texttt{ownedAttribute}, \langle 1, upper_{UserMgmt}\rangle,$$
$$\langle\langle UserMgmt, Class\rangle, \texttt{ownedAttribute}, \langle accesses, Property\rangle\rangle,$$
$$\langle\langle accesses, Property\rangle, \texttt{type}, \langle Manager, Class\rangle\rangle,$$
$$\langle\langle accesses, Property\rangle, \texttt{ownedAttribute}, \langle 1, lower_{Manager}\rangle\rangle,$$
$$\langle\langle accesses, Property\rangle, \texttt{ownedAttribute}, \langle \top, upper_{Manager}\rangle,$$
$$\langle\langle UserMgmt, Class\rangle, \texttt{ownedOperation}, \langle addUser, Property\rangle\rangle,$$
$$\langle\langle addUser, Property\rangle, \texttt{ownedAttribute}, \langle in\_u, Property\rangle\rangle,$$
$$\langle\langle in\_u, Property\rangle, \texttt{type}, \langle UID, Datatype\rangle\rangle,$$
$$\langle\langle UserMgmt, Class\rangle, \texttt{ownedOperation}, \langle deleteUser, Property\rangle\rangle,$$
$$\langle\langle deleteUser, Property\rangle, \texttt{ownedAttribute}, \langle in\_u, Property\rangle\rangle,$$
$$\langle\langle UserMgmt, Class\rangle, \texttt{ownedAttribute}, \langle accessUserRep, Property\rangle\rangle,$$
$$\langle\langle accessUserRep, Property\rangle, \texttt{type}, \langle UserRepository, Class\rangle\rangle,$$
$$\langle\langle accessUserRep, Property\rangle, \texttt{ownedAttribute}, \langle 1, lower_{UserRepository}\rangle\rangle,$$
$$\langle\langle accessUserRep, Property\rangle, \texttt{ownedAttribute}, \langle 1, upper_{UserRepository}\rangle\rangle,$$
$$\langle\langle UserRepository, Class\rangle, \texttt{ownedAttribute}, \langle accessUserRep, Property\rangle\rangle,$$
$$\langle\langle accessUserRep, Property\rangle, \texttt{type}, \langle UserMgmt, Class\rangle\rangle,$$
$$\langle\langle accessUserRep, Property\rangle, \texttt{ownedAttribute}, \langle 1, lower_{UserMgmt}\rangle\rangle,$$
$$\langle\langle accessUserRep, Property\rangle, \texttt{ownedAttribute}, \langle 1, upper_{UserMgmt}\rangle\rangle\ \}$$

To restrict access to these operations the instantiated Authentication aspect model shown in Fig. 9a is composed with the primary model to obtain the composed model shown in Fig. 10.



(a) Authentication context-specific aspect class diagram **(b)** User Management primary class diagram

Composition Directives

$Rename\ Primary :: UserMgmt :: addUser()\ to\ doAddUser()$

$Rename\ Primary :: UserMgmt :: deleteUser()\ to\ doDeleteUser()$

Fig. 9. Integrated aspect and primary view

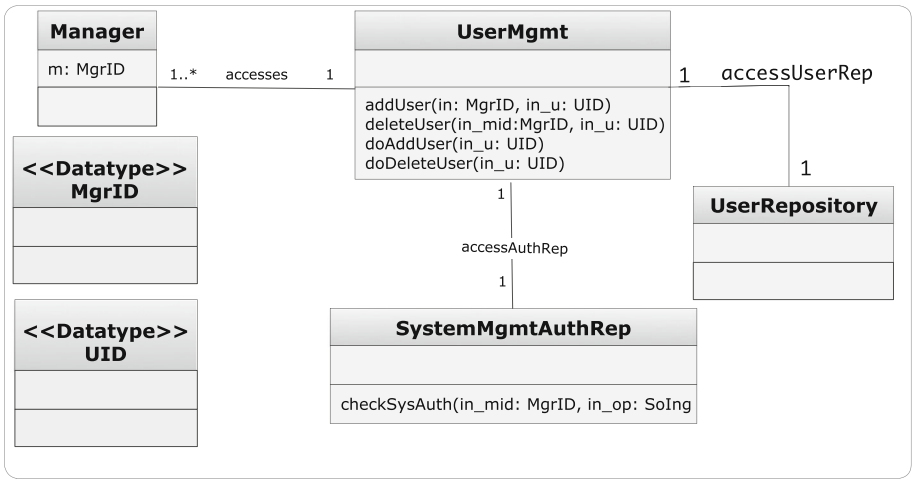


Fig. 10. Composed class diagram

The context-specific aspect model in Fig. 9a is obtained by instantiating the Authentication aspect model using bindings that define the values that are to be substituted for parameters in the authentication diagram templates. A binding relates an aspect model element to a model element and can be expressed as a pair (aspect element name, model element name). The model element name can be the name of a primary model element or the name of an application-specific element that is to be added to the composed model during composition.

The set of vertices and the set of edges corresponding to the authentication context-specific aspect model are represented manually using the CoQ4MDE syntax by: $MV_{AuthContext}$ and $ME_{AuthContext}$. In this definition, an operation is linked to its parameters via a relation `ownedAttribute`, and a parameter is linked to its type via a relation `type`.

$$
\begin{aligned}
MV_{AuthContext} \triangleq \{\ & \langle Manager, Class\rangle, \langle UserMgmt, Class\rangle, \langle SystemMgmtAuthRep, Class\rangle, \\
& \langle m, Property\rangle, \langle addUser, Property\rangle, \langle deleteUser, Property\rangle, \\
& \langle doAddUser, Property\rangle, \langle doDeleteUser, Property\rangle, \langle checkSysAuth, Property\rangle, \\
& \langle in, Property\rangle, \langle in\_u, Property\rangle, \langle in\_mid, Property\rangle, \\
& \langle in\_op, Property\rangle, \langle accesses, Property\rangle, \langle accessAuthRep, Property\rangle \\
& \langle MgrID, Datatype\rangle, \langle UID, Datatype\rangle\ \}
\end{aligned}
$$

$$
\begin{aligned}
ME_{AuthContext} \triangleq \{\ & \langle\langle Manager, Class\rangle, \texttt{ownedAttribute}, \langle m, Property\rangle\rangle,\\
& \langle\langle m, Property\rangle, \texttt{type}, \langle MgrID, Datatype\rangle\rangle,\\
& \langle\langle Manager, Class\rangle, \texttt{ownedAttribute}, \langle accesses, Property\rangle\rangle,\\
& \langle\langle accesses, Property\rangle, \texttt{type}, \langle UserMgmt, Class\rangle\rangle,\\
& \langle\langle accesses, Property\rangle, \texttt{ownedAttribute}, \langle 1, lower_{UserMgmt}\rangle\rangle,\\
& \langle\langle accesses, Property\rangle, \texttt{ownedAttribute}, \langle 1, upper_{UserMgmt}\rangle,\\
& \langle\langle UserMgmt, Class\rangle, \texttt{ownedAttribute}, \langle accesses, Property\rangle\rangle,\\
& \langle\langle accesses, Property\rangle, \texttt{type}, \langle Manager, Class\rangle\rangle,\\
& \langle\langle accesses, Property\rangle, \texttt{ownedAttribute}, \langle 1, lower_{Manager}\rangle\rangle,\\
& \langle\langle accesses, Property\rangle, \texttt{ownedAttribute}, \langle \top, upper_{Manager}\rangle,\\
& \langle\langle UserMgmt, Class\rangle, \texttt{ownedOperation}, \langle addUser, Property\rangle\rangle,\\
& \langle\langle addUser, Property\rangle, \texttt{ownedAttribute}, \langle in, Property\rangle\rangle,\\
& \langle\langle in, Property\rangle, \texttt{type}, \langle UID, Datatype\rangle\rangle,\\
& \langle\langle addUser, Property\rangle, \texttt{ownedAttribute}, \langle in\_u, Property\rangle\rangle,\\
& \langle\langle in\_u, Property\rangle, \texttt{type}, \langle UID, Datatype\rangle\rangle,\\
& \langle\langle UserMgmt, Class\rangle, \texttt{ownedOperation}, \langle deleteUser, Property\rangle\rangle,\\
& \langle\langle deleteUser, Property\rangle, \texttt{ownedAttribute}, \langle in\_mid, Property\rangle\rangle,\\
& \langle\langle in\_mid, Property\rangle, \texttt{type}, \langle UID, Datatype\rangle\rangle,\\
& \langle\langle deleteUser, Property\rangle, \texttt{ownedAttribute}, \langle in\_u, Property\rangle\rangle,\\
& \langle\langle UserMgmt, Class\rangle, \texttt{ownedOperation}, \langle doAddUser, Property\rangle\rangle,\\
& \langle\langle doAddUser, Property\rangle, \texttt{ownedAttribute}, \langle in\_u, Property\rangle\rangle,\\
& \langle\langle doDeleteUser, Property\rangle, \texttt{ownedAttribute}, \langle in\_u, Property\rangle\rangle,\\
& \langle\langle UserMgmt, Class\rangle, \texttt{ownedAttribute}, \langle accessAuthRep, Property\rangle\rangle,\\
& \langle\langle accessAuthRep, Property\rangle, \texttt{type}, \langle SystemMgmtAuthRep, Class\rangle\rangle,\\
& \langle\langle accessAuthRep, Property\rangle, \texttt{ownedAttribute}, \langle 1, lower_{SystemMgmtAuthRep}\rangle\rangle,\\
& \langle\langle accessAuthRep, Property\rangle, \texttt{ownedAttribute}, \langle 1, upper_{SystemMgmtAuthRep}\rangle\rangle,\\
& \langle\langle SystemMgmtAuthRep, Class\rangle, \texttt{ownedAttribute}, \langle accessAuthRep, Property\rangle\rangle,\\
& \langle\langle accessAuthRep, Property\rangle, \texttt{type}, \langle UserMgmt, Class\rangle\rangle,\\
& \langle\langle accessAuthRep, Property\rangle, \texttt{ownedAttribute}, \langle 1, lower_{UserMgmt}\rangle\rangle,\\
& \langle\langle accessAuthRep, Property\rangle, \texttt{ownedAttribute}, \langle 1, upper_{UserMgmt}\rangle\rangle,\\
& \langle\langle SystemMgmtAuthRep, Class\rangle, \texttt{ownedAttribute}, \langle checkSysAuth, Property\rangle\rangle,\\
& \langle\langle checkSysAuth, Property\rangle, \texttt{ownedAttribute}, \langle in\_mid, Property\rangle\rangle,\\
& \langle\langle in\_mid, Property\rangle, \texttt{type}, \langle MgrID, Datatype\rangle\rangle,\\
& \langle\langle checkSysAuth, Property\rangle, \texttt{ownedAttribute}, \langle in\_op, Property\rangle\rangle,\\
& \langle\langle in\_op, Property\rangle, \texttt{type}, \langle UID, Datatype\rangle\rangle\ \}
\end{aligned}
$$

The models obtained from these sets and the proof of the corresponding models are denoted by: $\langle MV_{AuthContext}, ME_{AuthContext}\rangle$ and $\langle MV_{UserManagement}, ME_{UserManagement}\rangle$.
The composition directives can be implemented using the `Substitution` operator, as:

- The directive *Rename Primary :: UserMgmt :: addUser() to doAddUser()* corresponds exactly to: *Substitution addUser doAddUser* $\langle MV_{UserManagement}, ME_{UserManagement}\rangle$
- The directive *Rename Primary :: UserMgmt :: deleteUser() to doDeleteUser()* corresponds exactly to: *Substitution deleteUser doDeleteUser* $\langle MV_{UserManagement}, ME_{UserManagement}\rangle$

Applying the `Union` operator on the model resulting from the two applications of the `Substitution` operator generates directly the intended result for this example thanks to the representation of models in Coq4MDE which ensures that attributes and operations for classes are directly merged in classes having the same class name.

5.2. Formalizing the `ISC` operators

Following the `ISC` method, besides a component, an interface consists of connection points (variation/reference points) that reveal how a component can be interconnected with its reuse context, which items are communicated to and from the environment, and which relations to the outer world must be established. In the following we present first the mechanisms of metamodel extension [KPCT11], followed by the operators for models composition, and finally we present a detailed example.

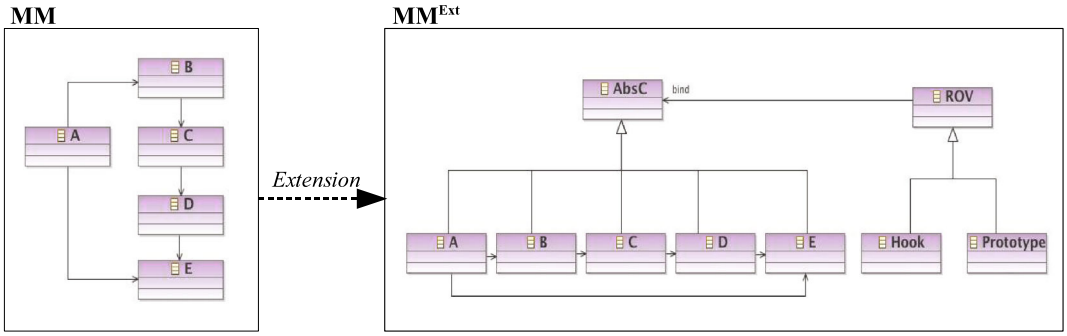


Fig. 11. MetaModel extension

5.2.1. *Extended metamodel with model component*

We must be able to extend any metamodel to support the definition of fragment boxes. This extension adds the definition of a fragment interface constituted from a set of addressable points. We note the extended metamodel for some metamodel MM as MM^{Ext}. We note ROV the abstract class representing the addressable points, the $Hook$ variation point and the $Prototype$ reference point are subclasses of ROV. In MM^{Ext}, every node in the graph representing MM can be referenced by an addressable point. For this purpose, an abstract class called $AbsC$ is added as a super class for all the classes of MM. This class is linked by the reference $bind$ with ROV. The three classes ROV, $Hook$ and $Prototype$ are also automatically imported to the metamodel with appropriate inheritance relations between them.[23]

The following definition represents the extension function implemented in Coq as a graph transformation which is not in the scope of this paper.

Definition 4 Let $MM = \langle\langle MMV, MME\rangle, conformsTo\rangle$ be a metamodel.
Let $ROV, Hook, Prototype, AbsC \in Classes, bind \in References$.
MM^{Ext} is defined as $\langle\langle MMV^{Ext}, MME^{Ext}\rangle, conformsTo^{Ext}\rangle$ such that:

$$
\begin{aligned}
&MMV^{Ext} = MMV \cup \{ROV, Hook, Prototype, AbsC\}\\
&MME^{Ext} = MME \cup \{\langle ROV, bind, AbsC\rangle\}\\
&conformsTo^{Ext}(\langle MV, ME\rangle) \triangleq conformsTo(\langle MV, ME\rangle)\\
&\wedge\ isAbstract(ROV)\\
&\wedge\ subClass(Hook, ROV)\\
&\wedge\ subClass(Prototype, ROV)\\
&\wedge\ isAbstract(AbsC)\\
&\wedge\ \forall\, c \in MMV, subClass(c, AbsC)
\end{aligned}
$$

Figure 11 shows the example of the extension of the MetaModel MM.

5.2.2. *Components composition*

In this section, we present the implementation in our framework of the two basic operators of `ISC` (`bind` and `extend`) presented in [Aßm03, Jen11]. The difference between these operators is that "the `bind` applied to the hook replaces the hook (i.e., it removes the hook from its containing fragment) while `extend` applied on a hook does not modify the hook itself but uses it as a position for extension (i.e., the hook remains in its containing fragment)".

[23] The metamodel extension used in [Jen11] is defined at the third modeling level (metametamodel level) which may use the promotion notion to be defined in the Coq4MDE framework. The extension defined thereafter uses only the second modeling level (metamodel level) which seems to be sufficient.

5.2.3. Bind

The bind operator replaces an object $\mathtt{o}_1$ referenced by a hook variation point from the first model by an object $\mathtt{o}_2$ referenced by a prototype reference point from the second model. The links to (resp. from) the object $\mathtt{o}_1$ are replaced with links to (resp. from) the object $\mathtt{o}_2$. The composed model is obtained by substituting the object $\mathtt{o}_1$ by $\mathtt{o}_2$ in both objects and links sets using the `Substitution` operator.
The operator $bind : Model \times Model \times (Objects \times Classes) \times (Objects \times Classes) \rightarrow Model$ is defined as:

$$
\begin{aligned}
&bind(\langle \mathtt{MV}_1, \mathtt{ME}_1\rangle, \langle \mathtt{MV}_2, \mathtt{ME}_2\rangle, \langle \mathtt{b}, \mathtt{B}\rangle, \langle \mathtt{b}', \mathtt{B}'\rangle) = \langle \mathtt{MV}_3, \mathtt{ME}_3\rangle\\
&where\ \langle \mathtt{b}, \mathtt{B}\rangle \in \mathtt{MV}_1\ and\ \langle \mathtt{b}', \mathtt{B}'\rangle \in \mathtt{MV}_2,\\
&\quad \exists\ \mathtt{h}, \mathtt{p} \in Objects, \langle\langle \mathtt{h}, \mathtt{Hook}\rangle, \mathtt{inh}, \langle \mathtt{h}, \mathtt{ROV}\rangle\rangle \in \mathtt{ME}_1\\
&\quad \wedge\langle\langle \mathtt{h}, \mathtt{ROV}\rangle, \mathtt{bind}, \langle \mathtt{b}, \mathtt{AbsC}\rangle\rangle \in \mathtt{ME}_1\\
&\quad \wedge\langle\langle \mathtt{b}, \mathtt{B}\rangle, \mathtt{inh}, \langle \mathtt{b}, \mathtt{AbsC}\rangle\rangle \in \mathtt{ME}_1\\
&\quad \wedge\langle\langle \mathtt{p}, \mathtt{Prototype}\rangle, \mathtt{inh}, \langle \mathtt{p}, \mathtt{ROV}\rangle\rangle \in \mathtt{ME}_2\\
&\quad \wedge\langle\langle \mathtt{p}, \mathtt{ROV}\rangle, \mathtt{bind}, \langle \mathtt{b}', \mathtt{AbsC}\rangle\rangle \in \mathtt{ME}_2\\
&\quad \wedge\langle\langle \mathtt{b}', \mathtt{B}'\rangle, \mathtt{inh}, \langle \mathtt{b}', \mathtt{AbsC}\rangle\rangle \in \mathtt{ME}_2\\
&\quad \wedge \mathtt{B} = \mathtt{B}'\\
&and\ finally : \langle \mathtt{MV}_3, \mathtt{ME}_3\rangle = Substitution\ b\ b'\ \langle \mathtt{MV}_1, \mathtt{ME}_1\rangle
\end{aligned}
$$

The implemented operator[24] considers also the case where o_1 is referenced by a prototype reference point and o_2 is referenced by a hook variation point and replaces in this situation o_2 by o_1 in the model $\langle \mathtt{MV}_2, \mathtt{ME}_2\rangle$. A more general version of this function is called `bind2MSH`[25] (for binding two models using several hooks). It defines a recursive call for the previous function using a list of correspondence (Variation/Reference) points allowing for the replacement of several objects at the same time by trying for every element of the list to replace the variation point by the reference point. Figure 12 shows an example where the hooks $\langle Closed, State\rangle$, $\langle Opened, State\rangle$, and $\langle open, Transition\rangle$ are replaced, respectively, by the prototypes $\langle Locked, State\rangle$, $\langle Unlocked, State\rangle$, and $\langle unlock, Transition\rangle$.

We proved for the different versions of the `bind` operator all the properties presented in Sect. 4. We successfully reused all the proofs previously done for the `Substitution` and `Union` operators and the improvement can be largely seen in the reduced number of lines needed to perform each proof comparing with performing the proofs without relating on the proofs of the elementary operators. Also we took advantage of the Coq language of tactics (Ltac)[26] that allows us to define proof strategies that try to repeat the application of the theorems proving the properties for the two versions of `bind`.[27]

5.2.4. Extend

This operator allows for the extension of a model $\langle \mathtt{MV}_1, \mathtt{ME}_1\rangle$ (the extension point is an object $\mathtt{o}_1$ addressed as a hook variation point inside the model) by a model $\langle \mathtt{MV}_2, \mathtt{ME}_2\rangle$ at an object $\mathtt{o}_2$ addressed as a prototype reference point. Figure 13 presents an example of the extend operator where the model $\mathtt{M}_1$ is extended by the model $\mathtt{M}_2$ and the position for extension is the object $\langle o_1, c_1\rangle$ that is linked with the object $\langle o_2, c_2\rangle$ from the model $\mathtt{M}_2$.

The function `Extend`[28] is parametrized by a metamodel (to ensure type safety) and a name for the added link between $\mathtt{o}_1$ and $\mathtt{o}_2$. The composed model consists of a multigraph built over the union of all objects of $\langle \mathtt{MV}_1, \mathtt{ME}_1\rangle$ and $\langle \mathtt{MV}_2, \mathtt{ME}_2\rangle$, all links of the two models in addition to a link between the objects $\mathtt{o}_1$ and $\mathtt{o}_2$.

[24] Is called Bind2M in the implementation: http://coq4mde.enseeiht.fr/FormalMDE/Bind2M_Verif.html#Bind2M.
[25] http://coq4mde.enseeiht.fr/FormalMDE/Bind2M_Verif.html#Bind2MSH.
[26] https://coq.inria.fr/refman/Reference-Manual012.html#ltac.
[27] http://coq4mde.enseeiht.fr/FormalMDE/Bind2M_Verif.html.
[28] http://coq4mde.enseeiht.fr/FormalMDE/Extend_Verif.html#compExt.

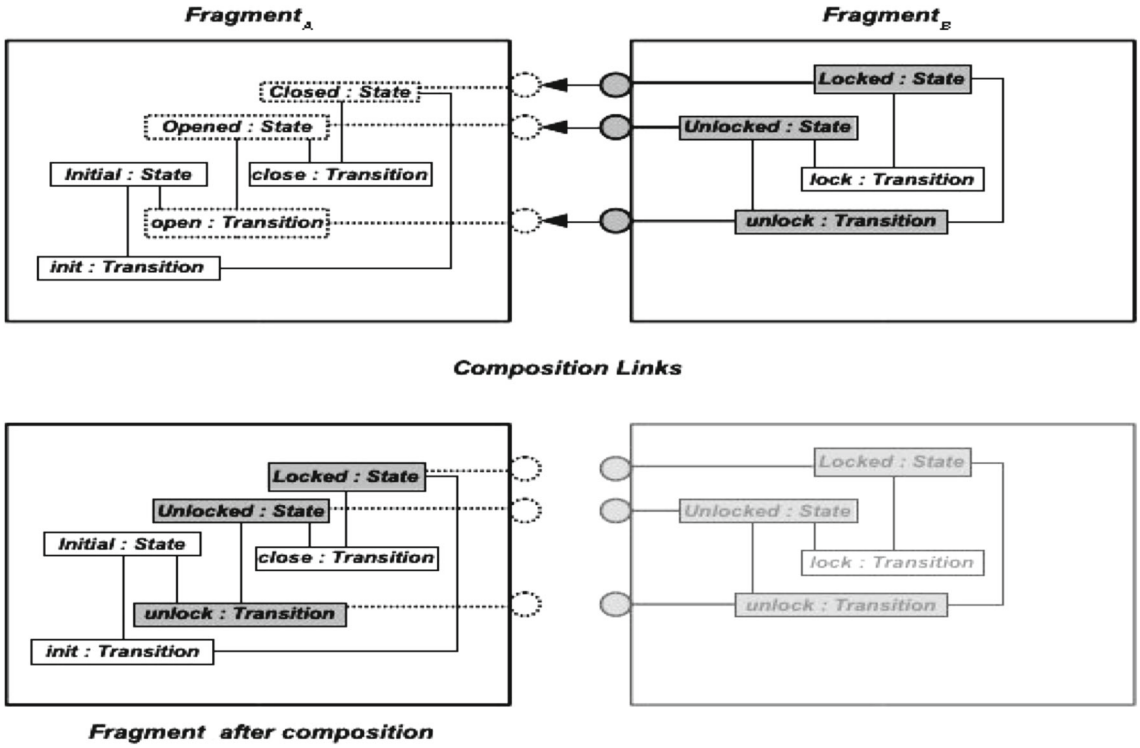


Fig. 12. Bind operator

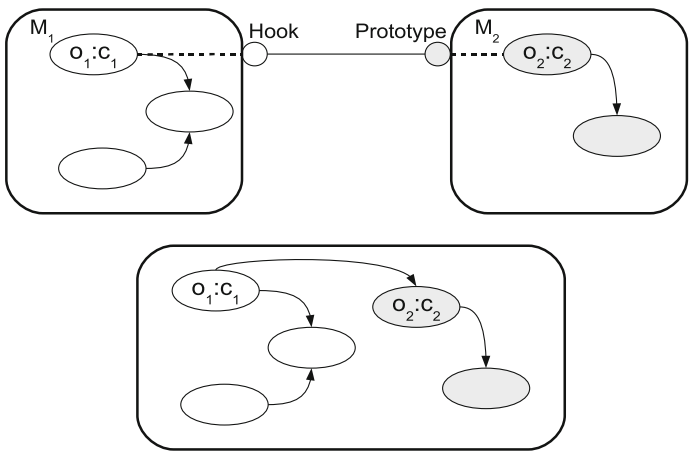


Fig. 13. Extend operator

$extend : Model \times Model \times (Objects \times Classes) \times (Objects \times Classes) \times MetaModel \times References \rightarrow Model$ is defined as:

$extend(\langle \mathtt{MV}_1, \mathtt{ME}_1\rangle, \langle \mathtt{MV}_2, \mathtt{ME}_2\rangle, \langle \mathtt{b}, \mathtt{B}\rangle, \langle \mathtt{b}', \mathtt{B}'\rangle,$
$(\langle \mathtt{MMV}, \mathtt{MME}\rangle, conformsTo), \mathtt{LinkName}) = \langle \mathtt{MV}_3, \mathtt{ME}_3\rangle$
$where\ \exists\ \langle \mathtt{b}, \mathtt{B}\rangle \in \mathtt{MV}_1\ and\ \langle \mathtt{b}', \mathtt{B}'\rangle \in \mathtt{MV}_2,\ we\ have:$
$extensible(\langle \mathtt{MV}_1, \mathtt{ME}_1\rangle, \langle \mathtt{MV}_2, \mathtt{ME}_2\rangle, \langle \mathtt{b}, \mathtt{B}\rangle, \langle \mathtt{b}', \mathtt{B}'\rangle,$
$(\langle \mathtt{MMV}, \mathtt{MME}\rangle, conformsTo), \mathtt{LinkName})\ such\ that:$
$\mathtt{MV}_3 = \mathtt{MV}_1 \cup \mathtt{MV}_2$
$\mathtt{ME}_3 = \mathtt{ME}_1 \cup \mathtt{ME}_2 \cup \{\langle\langle \mathtt{b}, \mathtt{B}\rangle, \mathtt{LinkName}, \langle \mathtt{b}', \mathtt{B}'\rangle\rangle\}$

The predicate *extensible* checks that a model $\langle \mathtt{MV}_1, \mathtt{ME}_1 \rangle$ whose interface is $\langle \mathtt{b}, \mathtt{B} \rangle$ regarding some metamodel can be extended by another model $\langle \mathtt{MV}_2, \mathtt{ME}_2 \rangle$ whose interface is $\langle \mathtt{b}', \mathtt{B}' \rangle$.

$$
\begin{aligned}
&\mathit{extensible}(\langle \mathtt{MV}_1, \mathtt{ME}_1 \rangle, \langle \mathtt{MV}_2, \mathtt{ME}_2 \rangle, \langle \mathtt{b}, \mathtt{B} \rangle, \langle \mathtt{b}', \mathtt{B}' \rangle,\\
&(\langle \mathtt{MMV}, \mathtt{MME} \rangle, \mathit{conformsTo}), \mathtt{LinkName}) \triangleq\\
&\mathit{isExtendedH}(\langle \mathtt{MV}_1, \mathtt{ME}_1 \rangle, \langle \mathtt{b}, \mathtt{B} \rangle)\\
&\wedge \mathit{isExtendedP}(\langle \mathtt{MV}_2, \mathtt{ME}_2 \rangle), \langle \mathtt{b}', \mathtt{B}' \rangle)\\
&\wedge (\mathtt{B}, \mathtt{LinkName}, \mathtt{B}') \in \mathtt{MME}
\end{aligned}
$$

The predicate *isExtendedH* verifies that $\langle \mathtt{b}, \mathtt{B} \rangle$ is a hook in $\langle \mathtt{MV}_1, \mathtt{ME}_1 \rangle$.

$$
\begin{aligned}
&\mathit{isExtendedH}\, \langle \mathtt{MV}_1, \mathtt{ME}_1 \rangle \langle \mathtt{b}, \mathtt{B} \rangle \triangleq\\
&\exists\ \mathtt{h} \in \mathit{Objects}, \langle \langle \mathtt{h}, \mathtt{Hook} \rangle, \mathtt{inh}, \langle \mathtt{h}, \mathtt{ROV} \rangle \rangle \in \mathtt{ME}_1\\
&\wedge \langle \langle \mathtt{h}, \mathtt{ROV} \rangle, \mathtt{bind}, \langle \mathtt{b}, \mathtt{AbsC} \rangle \rangle \in \mathtt{ME}_1\\
&\wedge \langle \langle \mathtt{b}, \mathtt{B} \rangle, \mathtt{inh}, \langle \mathtt{b}, \mathtt{AbsC} \rangle \rangle \in \mathtt{ME}_1
\end{aligned}
$$

The predicate *isExtendedP* verifies that $\langle \mathtt{b}, \mathtt{B} \rangle$ is a prototype in the model.

$$
\begin{aligned}
&\mathit{isExtendedP}\, \langle \mathtt{MV}_2, \mathtt{ME}_2 \rangle \langle \mathtt{b}, \mathtt{B} \rangle \triangleq\\
&\exists\ \mathtt{p}, \langle \langle \mathtt{p}, \mathtt{Prototype} \rangle, \mathtt{inh}, \langle \mathtt{p}, \mathtt{ROV} \rangle \rangle \in \mathtt{ME}_2\\
&\wedge \langle \langle \mathtt{p}, \mathtt{ROV} \rangle, \mathtt{bind}, \langle \mathtt{b}, \mathtt{AbsC} \rangle \rangle \in \mathtt{ME}_2\\
&\wedge \langle \langle \mathtt{b}, \mathtt{B} \rangle, \mathtt{inh}, \langle \mathtt{b}, \mathtt{AbsC} \rangle \rangle \in \mathtt{ME}_2
\end{aligned}
$$

We describe in the following section the use of the previously defined basic operators in an example of model composition following ISC.

5.2.5. *Detailed example*

This example [HHJZ09] is based on UML activity diagrams which outlines a real-world scenario. The principal motivation for this example is that although UML activity diagrams can be modularized into single models, reusing and combining parts of activities modelled separately is not well-supported by UML itself. The ISC method enables us to define general processes with activity diagrams which can be extended with specific activities for concrete application use-cases.

The process contains a checking activity that determines whether certain data (here customer data) is consistent. In order to keep the order processing activity extensible such that additional checks can be inserted in parallel to the customer data check variation points must be defined. To perform the extension, a developer should not need to know the internal implementation of the ordering process, only that check activities can be inserted. The developer needs to know that a check activity has to have one incoming control flow (from the *checkFork* node) and two outgoing flows (to the *checkMerge* and *checkJoin* nodes). With this knowledge, the developer can design additional check activities for instance, the one from Fig. 15 which determines the customer's credit card liquidity. In the ISC method, such extensibility can be realized by thinking about models as components. We consider also the Model described in Fig. 2 with its associated CoQ4MDE notation.

The first step is the definition of the interface for each model. In the model $\mathtt{M}_1$, only the *checkFork*, *checkMerge*, and *checkJoin* nodes, to which the incoming and outgoing flows of additional checks can connect, should be reflected in the composition interface. *CheckFork*, *checkJoin*, and *checkMerge* are defined as prototypes as presented in Fig. 14. In the model $\mathtt{M}_2$, we think of the initial (*InitialNodeCREDIT*) and final nodes (*FINISH* and *CANCEL*) as open ports in the model which need to be manipulated through the composition interface. So, *InitialNodeCREDIT*, *FINISH*, and *CANCEL* are defined as hooks like described in Fig. 15.

We can then easily define and execute a composition of both activities. The application of the function bind on the two fragments using the list of (hook, prototype) composed of (*InitialNodeCREDIT*, *checkFork*), (*CANCEL*, *checkMerge*), and (*FINISH*, *checkJoin*) followed by the elimination of the interface produces the model $\mathtt{M}_{\mathtt{bind}}$ shown in Fig. 16.

Then, *Start*, *Exit*, and *Correct* are defined in $\mathtt{M}_{\mathtt{bind}}$ as hook variations points as shown in Fig. 17 in the class diagram graphical notation.

The execution of the function extend on the two models $\mathtt{M}_1$ in Fig. 14 and $\mathtt{Fragment}_{\mathtt{M}_{\mathtt{bind}}}$ in Fig. 17 after the interface elimination generates the model presented in Fig. 18.

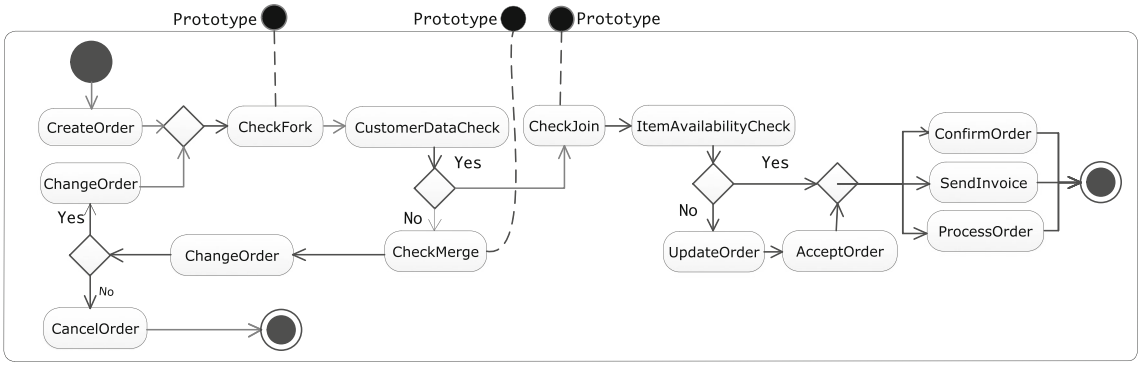


Fig. 14. M_1: an activity diagram for the control flow of an order process

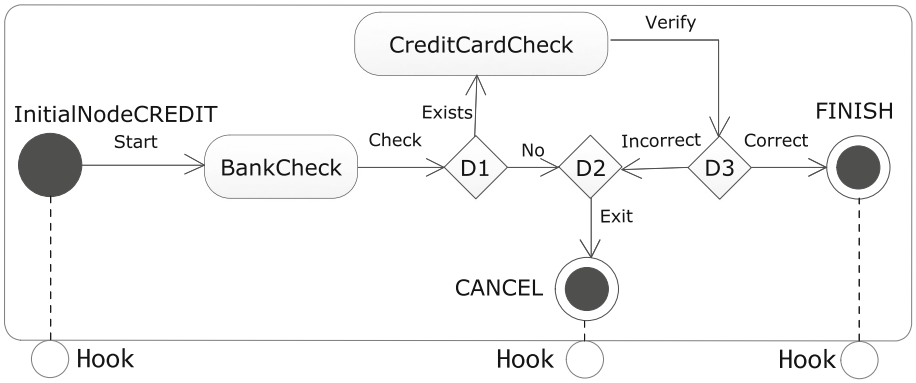


Fig. 15. M_2: an activity diagram for credit-card checks that can extend the order process with additional check

5.3. Formalizing the package merge

To have a more general idea for the usage of our framework, we recall briefly in this section the use case presented in [KHPCT14b] that uses our two assembly operators (`Union` and `Substitution`) to define the Package Merge. The Package Merge implementation as described hereafter is available at: http://coq4mde.enseeiht.fr/PackageMergeCoq/.

To illustrate our methodology, we give an example derived from [Zit06]. The source package (in this case the package `BasicEmployee` mentioned in Fig. 19) is the package `merged`. The package `EmployeeLocation` shown

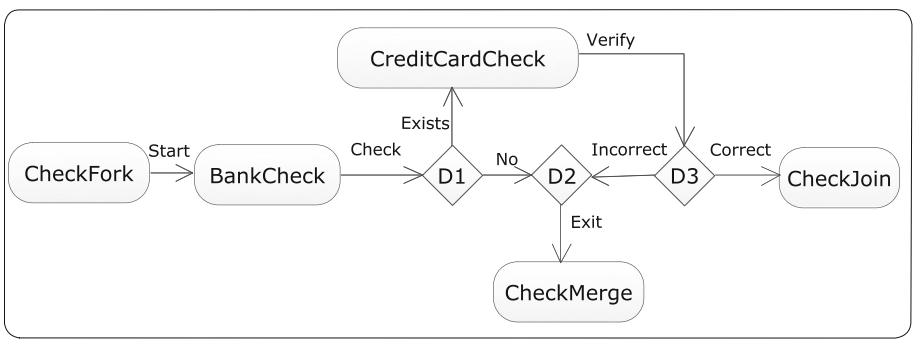


Fig. 16. M_{bind}: Result of the application of the `bind` operator on the two models M_1 and M_2

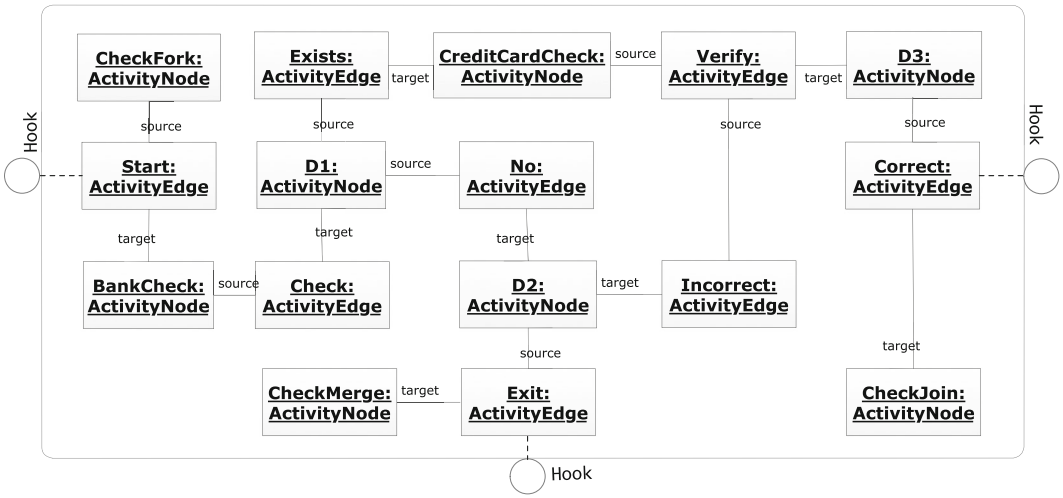


Fig. 17. $Fragment_{M_{bind}}$: fragment box for the model $\mathtt{M_{bind}}$

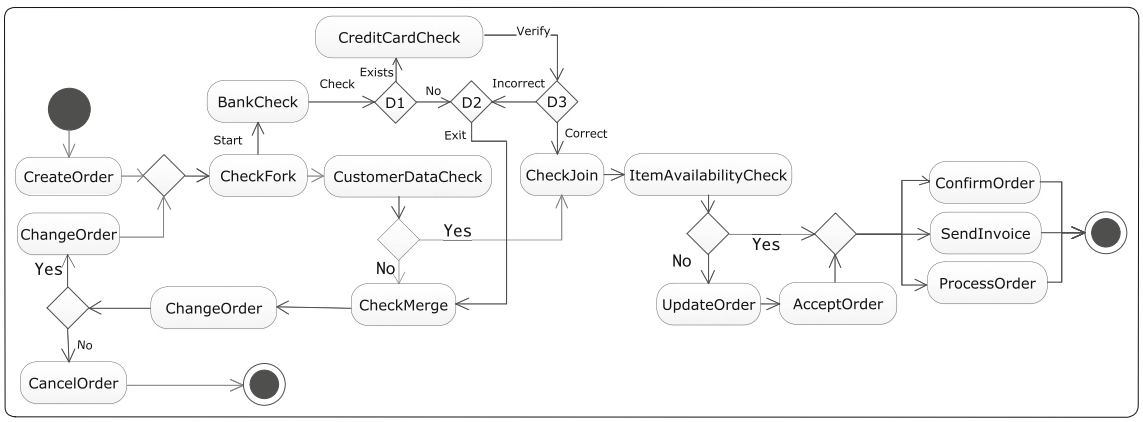


Fig. 18. Final composition result

in Fig. 20 is the package `receiving`. This package contains the additional elements that must be merged with the package `merged`. Two conflicts occur between the models `merged` and `receiving`. The first one is related to the attribute `upper` of *worksAs* [the maximal bound is equal to 2 in the model `BasicEmployee` (Fig. 19) and is equal to 3 in the model `EmployeeLocation` (Fig. 20)]. The second conflict is related to the class *Employee* that is abstract (name in italic) in the `merged package` and concrete in the `receiving package`.

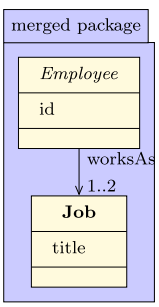


Fig. 19. `BasicEmployee`

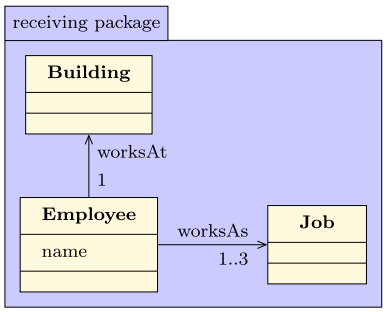


Fig. 20. `EmployeeLocation`

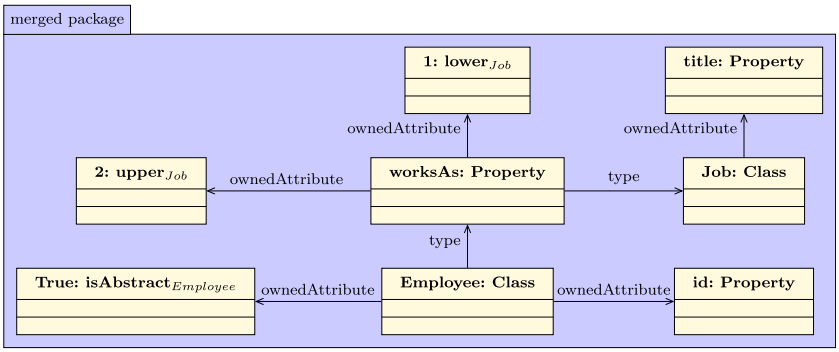


Fig. 21. An excerpt from the `BasicEmployee` model

The resolution of this kind of conflicts is done according to the `UML` specification [Obj13b]. The rule to resolve the conflict for the `upper` attribute is: $upper_{Resulting} = max(upper_{Merged}, upper_{Receiving})$. The rule for the `isAbstract` attribute is: $isAbstract_{Resulting} = isAbstract_{Merged} \wedge isAbstract_{Receiving}$. In this work, we focused on the rules related to the abstract classes and cardinalities, the list of all the possible transformations is available in the specification [Obj13b, p. 255–258]. The rules used for the resolution of conflicts are specific to the Package Merge operator as specified in the `UML` specification and may not be the same for other operators.

Concretely in our abstract syntax, we manipulate the metamodels as models conforming to `MOF`, so the abstraction property for classes is represented with attributes *isAbstract* suffixed with the name of the class (this attribute is equal to True in the model `BasicEmployee` (Fig. 19) and equal to *False* in the model `EmployeeLocation` (Fig. 20)). The same principle is used to represent all the properties linked to `MOF` such as `lower` and `upper`. We show in Fig. 21 the representation of the package `BasicEmployee` as a model conforming to `MOF`. The `EmployeeLocation` package is represented using the same principle, we don't show it here for space reasons.

The first step is to resolve all the conflicts. For this, the `Substitution` operator is applied twice. The first application replaces 2 by 3 for the $upper_{Job}$ attribute in the `merged` model. The second application of the `Substitution` operator replaces *True* by *False* for the $isAbstract_{Employee}$ attribute in the `merged` model.

Once the conflicts are resolved, the final step is the union of the obtained models `merged` and `receiving` (the constraints of the `Union` operator are satisfied in this case). The result is exactly the merge of the two packages `merged` and `receiving` shown in Fig. 22.

In the previous example, the `Package Merge` is expressed using the primitive composition operators `Union` and `Substitution`. Defining the `Package Merge` in this manner ensures that the resulting model is well typed in relation with the packages `merged` and `receiving` and also that it satisfies the semantic properties of the metamodel when the preconditions are satisfied.

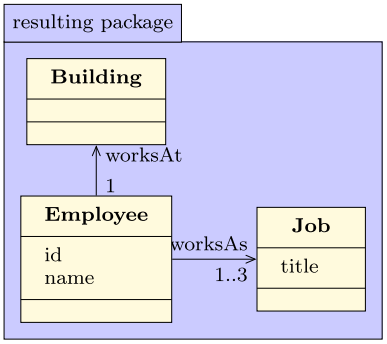


Fig. 22. The resulting metamodel

6. Related work

Our notion of model follows the MDE vision and we are interested in the formalization and the verification by proofs of the composition operators. In that sense, our work is related mainly to two lines of research: (1) formalization of MDE and model transformation and (2) composition approaches and formalization of composition operators. We highlight in this section previous work that is relevant with regards to our approach. First, we discuss some formalizations of the MDE to help position our proposal. We then explain our contributions in the context of several existing composition approaches.

6.1. Formalization of model driven engineering and model transformation

Specification of mathematical foundations for MDE that would allow for the verification of model transformation properties has attracted a lot of attention from researchers. We refer in the following the closest work to our contribution.

Aiming to define an algebraic semantics for MOF, Boronat et al proposed in [BM10] a model management framework based on experiments in formal model transformation and data migration called MoMENT (MOdel manageMENT). This framework provides a set of generic operators to manipulate models and relies on the algebraic formalisms provided by the Maude language [CDE+02]. In MoMENT, the metamodels are represented as algebraic specifications and the operators are defined independently of the metamodel. To be used, the operators must be described in a module called signature which specifies the constructs of the metamodel. The approach was implemented in a tool[29] that also gives an automatic translation from an EMF metamodel to a signature model. Also using Vallecillo et al. have designed and implemented a different embedding of metamodels, models [JRJEFA07] and model transformations [TV10]. Both embeddings are shallow and rely strongly on the MAUDE's object structure in order to define model elements as objects, and on MAUDE's object rewriting semantics for its implementation of model transformations.

The use of the CIC to formalize the concepts of MDE is not a new idea, Poernomo has proposed an encoding of metamodels and models using type theory [Poe06] which allows for the development of correct by construction model transformation using proof assistants like Coq [Poe08], but the proposed formalization was not put in practice. Also using CIC in the context of model transformation, Calegari et al proposed in [CLST11] a framework where models and metamodels are represented as types. Similar to our approach for the formalization of model composition, the specifications of model transformations are universally or existentially quantified logical formula. The proofs of these formulas in CIC are constructive and therefore allow for the automated extraction of programs computing the target model from some specified input. This ensures that the output is correct with respect to the given specification and in this way, it is possible to obtain certified zero-fault model transformations/compositions that can also be enriched with verifiable properties that are not included in the initial specification.

[29] http://moment.dsic.upv.es/.

Some simple experiments have been conducted using Coq mainly on tree-shaped models [PT10] using inductive types. General graph model structure can be encoded using co-inductive types but the encoding, as shown in [PM11] by Picard and Matthes is quite complex. Its complexity is due to Coq enforcing structural constraints when combining inductive and co-inductive types which makes impossible the use of the natural encodings proposed by Giorgino et al. in [GSMP11]. These ones rely on a spanning tree of the graph combined with additional links to overcome these restrictions using the Isabelle proof assistant. The model transformation relies on slightly adapted inductive proofs and which can be followed by the extraction of classical imperative implementations. These embeddings are all shallow as they all rely on similar sophisticated data structure to represent model elements and metamodels (e.g. Coq (co-)inductive data types for model elements and object and (co-)inductive types for metamodel elements).

The work described in this paper is a deep embedding, each model and metamodel concept being encoded using elementary constructs instead of relying on using similar elements from MAUDE, Coq or Isabelle. The purpose of this contribution is, in addition to implement model composition using correct-by-construction tools, to give a kind of denotational semantics for model driven engineering concepts that should provide a deeper understanding and allow for the formal validation of the various implemented technologies.

6.2. Composition approaches

Models are views of the system that must be composed to generate the final system. This is true in a wide variety of modeling domains and several techniques have been proposed to accomplish it. We review here several approaches collected in [Jea08] : Rational Software Architect[30] is designed for UML2[31] and provides a wizard enabling the composition of two models [Let05]. Bernstein's data model [BHP00] treats both models and mapping as first class citizens, where the models are sets of objects, and the mappings capture relationships between models, and match and merge operations are available for model composition. Atlas Model Weaver (AMW),[32] part of the AMMA platform providing the ATL transformation language, also allows for establishing relationships between elements from different models [DFBJ+05]. Epsilon[33] provides a set of languages for the common Model Driven Engineering tasks. Signature-based composition [RFG+05] is an approach that does not use mappings to compose models, the composition being done instead by comparing signatures. Modeling Aspects using a Transformation Approach (MATA) [JWEG07] is an asymmetric approach for model composition, in which one model plays the role of base model and the other is the aspect. Theme/UML extends the UML standard with new modularization and compositional concepts [Cla02]. Prompt [NM00] is an algorithm for merging ontologies, it is implemented in the Protégé[34] toolkit. Easterbrook et al proposed the behavioral merge of state charts [NSC+07] which allows for the comparison and merging of hierarchical Statecharts. Last but not least, the semantic-based weaving of scenarios [KLJM06] proposed by Klein et al. (ModMap)[35] is a method and tool for defining alignment rules between both homogeneous and heterogeneous languages, the mappings representing the relations between the components. The tool allows for manual, semi-automatic and automatic alignment, the user being able to provide additional directives for improving the quality of mapping descriptions.

In the database context, Carvalho et al proposed in [CLA15] a framework based on MDE to integrate database schemas where these ones are viewed as database models. The authors make a correspondence between schema matching and merging and model matching and merging, respectively. This kind of composition can be clearly supported by our operators which also enable us to add semantic and apply formal verification techniques.

The challenge of composition is also present in the context of software product lines and feature-driven product derivation [SS09, ALMK08] where there is an emerging necessity to integrate multiple features into a single system. In this area, recent research is also oriented into the formalization and the correct-by-construction methodologies; for example, [PDL15] is a first step towards the production of a methodology and a platform that help the user to automatically produce correct-by-construction product variants from the related feature modules. This latter contribution is not yet implemented and depends on the definition of the GFML language that allows

[30] http://www-306.ibm.com/software/awdtools/architect/swarchitect/.
[31] http://www.uml.org/.
[32] http://www.eclipse.org/gmt/amw/.
[33] http://www.eclipse.org/gmt/epsilon/.
[34] http://protege.stanford.edu/.
[35] http://www.kermeta.org/mdk/ModMap.

writing the features that are compiled to FoCaLiZe[36] source program that can be analysed and translated into Coq sources for verification.

We were looking first for an approach that supports modularizing models and model composition. The ISC [HHJ+08, HHJZ09] approach supports these two characteristics. The method enables us to extend arbitrary languages to turn them to reuse adding the concepts of fragment and interfaces of fragments. In this method, components can be invasively composed, this can be done by adapting or extending the component at some variation point (fragments or positions, which are subject to change) by transformation. After formalizing ISC, we were searching for more elementary composition operators that can be used for the formalization of ISC and other composition methods. We found that the defined elementary operators can be used in the formalization of composition following AOM approach and especially for MOF and UML Package Merge that allows for the merge of the content of two packages [Obj13b].

7. Conclusion

Although formal verification is an expensive activity, it is the only sound solution in the case of safety critical systems. We report on our attempt to bridge the gap between model composition and verification where the intended level of quality and safety of a composed model is achieved by formalizing and proving the properties of the intended composed model.

To this end, we used the CIC and the Coq proof assistant as a certification framework for model composition. Starting from our formal framework for model and metamodel formal specification Coq4MDE, we have tackled the problem of model composition.

We followed a divide and conquer approach which relies on primitive composition operators that are easy to verify and can be used to build more sophisticated ones. We prove the correctness of the expected properties of the primitive operators introducing mandatory preconditions to help us reach the compositional verification of the targeted properties. The proofs of property preservation for the high level operators combine the proofs of the primitive ones.

We validate our proposal using the MOF model conformance property, the MOF Package Merge operator, a more general AOM composition method, and the ISC approach.

All these notions are also currently reflected in the Coq proof assistant, an embedding which provides correct-by-construction pieces of executable code for the different model operations related to composition. As we target a general purpose MDE-oriented framework, our work applies to any model, modeling language and application and is not restricted to a particular language context.

This proposal is a preliminary mandatory step in the formalization of compositional formal verification technologies. We have tackled the formal composition of models independently of the properties satisfied by the model and the expected properties for the composite model. The developed Coq code is about 18000 lines with more than 200 definition and 90 theorems(http://coq4mde.enseeiht.fr/FormalMDE).

This work can be extended in several directions. First, other elementary operators and more sophisticated composition operators should be supported; the actual use of the elementary operators is promising and allows for the consideration of richer composition operators. Also, for the definition of other elementary operators, we hope to enrich and combine our approach in using proof assistants with automated proofs tools (e.g., SMT solvers like Z3)[37] so that we can generate part of the proofs that can be completed interactively using specialized tools like Why.[38] Some changes to the encoding is required to adapt it to Z3 (first-order logic) syntax but the interactive usage of Coq is still possible using Why.

Also, among the wide variety of properties that can be verified, we focussed on the minimal requirement of the conformance to the initial metamodel. The next step is to improve the compositional verification of models with various constraints, from simple static constraints such as verification of OCL constraints satisfaction to dynamic properties such as deadlock freedom as proposed in the BIP framework [BBS06].

In order to accomplish this, we need to model the behavioral part of each language and we propose to make use of the generic behaviors applicable to several metamodels along the lines presented in [LG13]. We plan also to experiment the behavioral aspect by considering the merging of Statecharts Specifications [NSC+07]. In the long

[36] http://focalize.inria.fr.
[37] https://z3.codeplex.com/.
[38] http://why3.lri.fr.

run, we plan to integrate the work of Garnacho et al. [MJPM13] that provide an embedding into Coq of timed transition systems and to apply our formal proposal in the GEMOC framework in order to prove the correctness of the proposed language composition operators.

Acknowledgements

The first author thanks Taibah university for her research funding. This is an extension of a previous work funded by the french FUI in the TOPCASED project, it was funded by the European Union and the french DGCIS through the ARTEMIS Joint Undertaking inside the CESAR project, the P project funded by the FUI and the région Midi Pyrénées in France as well as the GEMOC ANR project. Thanks to Daniela Rosu for her comments on this paper.

References

[ALMK08] Apel S, Lengauer C, Möller B, Kästner C (2008) An algebra for features and feature composition. In: Algebraic methodology and software technology, Springer, New York, pp 36–50
[Aßm03] Aßmann U (2003) Invasive software composition. Springer, New York
[BBS06] Basu A, Bozga M, Sifakis J (2006) Modeling heterogeneous real-time components in BIP. In: Software engineering and formal methods, 2006. SEFM 2006. 4th IEEE international conference on, pp 3–12
[Béz04] Bézivin J (2004) In search of a basic principle for model driven engineering. Novat J Special Issue 5(2):21–24
[BHP00] Bernstein PA, Halevy AY, Pottinger RA (2000) A vision for management of complex models. ACM Sigmod Rec 29(4):55–63
[BM10] Boronat A, Meseguer J (2010) An algebraic semantics for MOF. Formal Aspects Comput 22(3–4):269–296
[CDE+02] Clavel M, Durán F, Eker S, Lincoln P, Martı-Oliet N, Meseguer J, Quesada JF (2002) Maude: specification and programming in rewriting logic. Theor Comput Sci 285(2):187–243
[Chr03] Chrząszcz J (2003) Implementing modules in the Coq system: theorem proving in higher order logics, pp 270–286
[Cla02] Clarke S (2002) Extending standard UML with model composition semantics. Sci Comput Progr 44(1):71–100
[CLA15] Vinícius Carvalho M, Lopes D, Abdelouahab Z (2015) A framework based on model driven engineering to support schema merging in database systems. In: New trends in networking, computing, e-learning, systems sciences, and engineering, Springer, New York, pp 397–405
[CLST11] Calegari D, Luna C, Szasz N, Tasistro Á (2011) A type-theoretic framework for certified model transformations. In: Formal methods: foundations and applications, Springer, New York, pp 112–127
[DFBJ+05] Didonet Del Fabro M, Bézivin J, Jouault F, Breton E, Gueltas G (2005) AMW: a generic model weaver. In: Proceedings of the 1ères Journées sur l'Ingénierie Dirigée par les Modèles
[Dij76] Dijkstra EW (1976) A discipline of programming, volume 1. Prentice-Hall, Englewood Cliffs
[FFR+07] France R, Fleurey F, Reddy R, Baudry B, Ghosh S (2007) Providing support for model composition in metamodels. In: Enterprise distributed object computing conference, 2007. EDOC 2007. 11th IEEE international, pp 253–253
[FR07] France R, Rumpe B (2007) Model-driven development of complex software: a research roadmap. In: 2007 Future of software engineering, IEEE Computer Society, pp 37–54
[FRGG04] France R, Ray I, Georg G, Ghosh S (2004) Aspect-oriented approach to early design modelling. IEEE Proc Softw 151(4):173–185
[GSMP11] Giorgino M, Strecker M, Matthes R, Pantel M (2011) Verification of the schorr-waite algorithm–from trees to graphs. Logic-based program synthesis and transformation, pp 67–83
[Hen09] Henriksson J (2009) A lightweight framework for universal fragment composition with an application in the semantic web. PhD thesis, TU Dresden
[HHJ+08] Henriksson J, Heidenreich F, Johannes J, Zschaler S, Aßmann U (2008) Extending grammars and metamodels for reuse: the Reuseware approach. Softw IET 2(3):165–184
[HHJZ09] Heidenreich F, Henriksson J, Johannes J, Zschaler S (2009) On language-independent model modularisation. In: Transactions on aspect-oriented software development VI, pp 39–82
[HP08] Holt J, Perry S (2008) SysML for systems engineering, volume 7. IET
[Jea08] Jeanneret C (2007–2008) An analysis of model composition approaches. Master's thesis, Ecole Polytechnique Fédérale de Lausanne
[Jen11] Jendrik J (2011) Component-based model-driven software development. PhD thesis, vorgelegt an der Technischen Universität Dresden Fakultät Informatik
[JRJEFA07] José Raúl R, José Eduardo R, Francisco D, Antonio V (2007) Formal and tool support for model driven engineering with maude. J Object Technol 6(9):187–207
[JWEG07] Jayaraman P, Whittle J, Elkhodary AM, Gomaa H (2007) Model composition in product lines and feature interaction detection using critical pair analysis. In: Model driven engineering languages and systems, pp 151–165
[KAAK09] Kienzle J, Al Abed W, Klein J (2009) Aspect-oriented multi-view modeling. In: Proceedings of the 8th ACM international conference on aspect-oriented software development, pp 87–98, ACM
[KHPCT14a] Kezadri Hamiaz M, Pantel M, Combemale B, Thirioux X (2014) Correct-by-construction model composition: application to the invasive software composition method. In: FESCA, pp 108–122

[KHPCT14b] Kezadri Hamiaz M, Pantel M, Combemale B, Thirioux X (2014) A formal framework to prove the correctness of model driven engineering composition operators. In: ICFEM'14–16th international conference on formal engineering methods, Springer, New York, pp 235–250
[KLJM06] Klein J, Loïc H, Jean-Marc J (2006) Semantic-based weaving of scenarios. In: Proceedings of the 5th international conference on aspect-oriented software development, ACM, pp 27–38
[KPCT11] Kezadri M, Pantel M, Combemale B, Thirioux X (2011) A proof assistant based formalization of components in MDE. In: 8th international symposium on formal aspects of component software (FACS 2011), Springer, Berlin, pp 223–240
[Let05] Letkeman K (2005) Comparing and merging UML models in IBM rational software architect. IBM Rational
[LG13] Lara J, Guerra E (2013) From types to type requirements: genericity for model-driven engineering. Softw Syst Model 12(3):453–474
[MJPM13] Manuel G, Jean-Paul B, Mamoun F-A (2013) A mechanized semantic framework for real-time systems. In: Formal modeling and analysis of timed systems, Springer, New York, pp 106–120
[NM00] Noy NF, Musen MA (2000) Algorithm and tool for automated ontology merging and alignment. In: Proceedings of the 17th national conference on artificial intelligence (AAAI-00). Available as SMI technical report SMI-2000-0831
[NSC+07] Nejati S, Sabetzadeh M, Chechik M, Easterbrook S, Zave P (2007) Matching and merging of statecharts specifications. In: Proceedings of the 29th international conference on software engineering, IEEE Computer Society, pp 54–64
[Obj06] Object Management Group, Inc. (2006) Meta object facility (MOF) 2.0 Core Specification
[Obj13a] Object Management Group (2013) OMG meta object facility (MOF) Core Specification, Version 2.4.1
[Obj13b] Object Management Group (2013) OMG unified modeling language TM (OMG UML) Version 2.5
[Obj14] Object Management Group (2014) Object constraint language, Version 2.4
[Par72] Parnas DL (1972) On the criteria to be used in decomposing systems into modules. Commun ACM 15(12):1053–1058
[PDL15] Pham T-K-Z, Dubois C, Lévy N (2015) Towards correct-by-construction product variants of a software product line: Gfml, a formal language for feature modules. In: Proceedings 6th workshop on formal methods and analysis in SPL engineering, FMSPLE 2015, London, 11 April 2015, pp 44–55
[PM11] Picard C, Matthes R (2011) Coinductive graph representation : the problem of embedded lists. In: Electronic communications of the EASST, special issue graph computation models, GCM'10
[Poe06] Poernomo I (2006) The meta-object facility typed. In Hisham H (ed) SAC, ACM pp 1845–1849
[Poe08] Poernomo I (2008) Proofs-as-model-transformations. In: Antonio V, Jeff G, Alfonso P (eds)ICMT volume 5063 of Lecture Notes in Computer Science, Springer, New York, pp 214–228
[PT10] Poernomo I, Terrell J (2010) Correct-by-construction model transformations from partially ordered specifications in Coq. In: Dong JS, Zhu H (eds) ICFEM, volume 6447 of Lecture Notes in Computer Science, Springer, New York, pp 56–73
[RFG+05] Reddy R, France R, Ghosh S, Fleurey F, Baudry B (2005) Model composition: a signature-based approach. In: Aspect oriented modeling (AOM) Workshop
[RGF+06] Reddy R, Ghosh S, France R, Straw G, Bieman J, McEachen N, Song E, Georg G (2006) Directives for composing aspect-oriented design class models. In: Transactions on aspect-oriented software development I, Springer, New York, pp 75–105
[SS09] Schirmeier H, Spinczyk O (2009) Challenges in software product line composition. In: 42nd Hawaii international conference on system sciences, IEEE, pp 1–7
[SSK+07] Schauerhuber A, Schwinger W, Kapsammer E, Retschitzegger W, Wimmer M, Kappel G (2007) A survey on aspect-oriented modeling approaches. Relatorio tecnico, Vienna University of Technology
[TCCG07] Thirioux X, Combemale B, Crégut X, Garoche P-L (2007) A framework to formalise the MDE foundations. In Richard P, Jean B (eds) International Workshop on Towers of Models (TOWERS), Zurich, pp 14–30
[TV10] Troya J, Vallecillo A (2010) Towards a rewriting logic semantics for ATL. In: Tratt L, Gogolla M (eds) ICMT, volume 6142 of Lecture Notes in Computer Science, Springer, New York pp 230–244
[WHR14] Whittle J, Hutchinson J, Rouncefield M (2014) The state of practice in model-driven engineering. Softw IEEE 31(3):79–85
[Zit06] Zito A (2006) UML's package extension mechanism: taking a closer look at package merge. PhD thesis, Queen's University

Received 4 June 2015
Revised 26 October 2015
Accepted 18 December 2015 by Stephan Merz, Jun Pang, and Jin Song Dong
Published online 27 January 2016