Towards a Meta-tool for Change-Centric Method Engineering: A Typology of Generic Operators

Jolita Ralyté[1], Colette Rolland[2], and Rébecca Deneckère[2]

[1] CUI, University of Geneva, Rue de Général Dufour, 24, CH-1211 Genève 4, Switzerland
ralyte@cui.unige.ch
[2] CRI, University of Paris 1 – Sorbonne, 90 Rue de Tolbiac, 75013 Paris, France
{rolland,denecker}@univ-paris1.fr

Abstract. The work presented in this paper considers how Method Engineering (ME) helps in method changes that are required by Information Systems (IS) changes. In fact, ME provides different approaches allowing to construct situation-specific methods by adapting, extending, improving existing methods or by assembling method components. All these approaches use a set of operations to realize these method changes. Our objective in this paper is to provide a meta-tool for change-centric ME which takes the form of a typology of generic ME operators. The operators for each specific ME approach are instantiated from the generic ones. The paper illustrates and discusses the instantiation of the generic typology for two assembly-based ME approaches.

1 Introduction

In order to survive in a more and more competitive environment, *organizations undergo frequent changes* that imply to adapt, enhance, reform, evolve, merge, interoperate their supporting Information Systems (IS). There are many different kinds of *IS change* and many circumstances for change such as business change, technology progress, ERP installation, company merge/take-over, globalization, standardization of practices across branches of a company etc. Each of them requires specific methods to handle change. Thus, as a consequence of the need for IS change, engineering methods shall themselves adapt to new circumstances of IS evolution. In other words, IS change implies *method change*.

The position taken in this paper is that *Method Engineering* (ME) can help to respond to this need. ME is the 'discipline to study engineering techniques for constructing, assessing, evaluating and managing methods and to study educational techniques for teaching and training method users' [27]. We see ME at two levels of abstraction: the *method level* and the *method meta-level*. The *former* refers to the construction of new change-centric methods using ME techniques. This leads to the development of methods supporting different kinds of evolution such as IS improvement, expansion, transformation as well as methods tailored to emerging application domains such as e-commerce, web services, mobile IS, etc. The *latter* seeks to provide generic ME tools and techniques to support ME at the method level. Processes/Algorithms to support 'on the fly' construction of situation-specific methods is an example of such meta-technique. Typologies of ME approaches or techniques are another example.

Our concern in this paper is to provide a meta-tool for change-centric method engineering which takes the form of a set of *ME operators*. Indeed, at the core of every ME process there is a set of operations to be carried out on the components of the method(s) involved in the ME activity. As these components are mainly Product and Process Models (PPMs), these operations can be formalized as *operators* applicable to the various elements of PPMs. These operators are generic in the sense that they are not dependent of a specific ME approach. On the contrary, they can be *instantiated* in every specific ME approach.

In order to ease the use of generic operators in a given ME approach, we provide in the paper a framework, the *operator-driven ME framework,* to deal with the meta and method levels and to generate operators for any specific ME approach from the generic ones. The usefulness of such a typology of operators is manifolds. It offers a means to easily generate a complete set of operators for a specific ME approach and to base the ME approach on a theoretically sound ground. Such a formalisation is especially required in the case of a corresponding CAME tool creation. Moreover, it offers a possibility to develop mixed ME approaches to deal with a combination of ME situations. This will be achieved by combining operators of different ME types.

The remainder of this paper is as follows: section 2 proposes a typology of ME approaches, which is used for the definition of the generic ME operators presented in section 3. In section 4 we illustrate how the generic typology of ME operators can be instantiated in order to obtain operators for a specific ME approach. Finally, section 5 draws some conclusions and discussions about our future work.

2 Typology of Method Engineering Approaches

A large number of Method Engineering approaches have been proposed in the literature. These approaches provide guidance for the creation of a new method [11, 20, 21] and for the adaptation of an existing method to some conditions of change [30] or to a specific project situation [6, 7, 10]. A literature survey [4, 5, 8, 25, 29] complemented by our own experience [18] leads us to classify these approaches according to four types of method engineering that we referred to as *Ad-Hoc*, *Paradigm-Based*, *Extension-Based* and *Assembly-Based*, respectively.

Ad-Hoc approaches deal with the construction of a new method ‘from scratch’. There are different reasons that can initiate a decision to construct a new method. The appearance of a new application domain that is not yet supported by a specific method is one example, experience capitalisation serving as the start point for a new method construction is another example.

Paradigm-Based ME [17] uses some initial paradigm model or meta-model as a baseline As-Is model which is instantiated [5], abstracted [18] or adapted [30] according to the current ME objective to develop the new To-Be model.

Extension-Based ME proposes different kinds of extension that can be realised on an existing method. Their objective is to enhance a method with new concepts and properties [3]. For example, a static method such the one for construction E/R schemas can be extended to deal more systematically with the representation of time through a calendar of time points, intervals etc. and temporal aspects such as the histories of entities.

Assembly-Based ME proposes to construct new methods or to enhance existing ones by reusing parts of other methods. The core concept in these approaches is one of reusable *method component* [28] also called *method chunk* [16, 22], *method fragment* [2, 7, 14, 26] or *method block* [12]. An Assembly-Based method construction consists in defining method requirements for a current situation, selecting method components satisfying this situation and assembling them. Association and integration are two kinds of assembly that can be applied on the selected method components [15]. Association concerns assembly of method components with different purposes and objectives, whereas integration deals with overlapping method components having the same or similar objective but providing different manners to satisfy it.

3 Towards Generic Operators for Method Engineering

3.1 Role of Operators in Method Engineering

Each ME approach proposes a specific method engineering process which uses a specific set of method construction operators. The objective of this work is to propose a generic typology of ME operators which should ease the definition of a set of specific operators for every specific ME approach while guaranteeing their completeness and correctness. Fig. 1 presents our *operator-driven ME framework* where specific ME operators are generated from the generic ones.

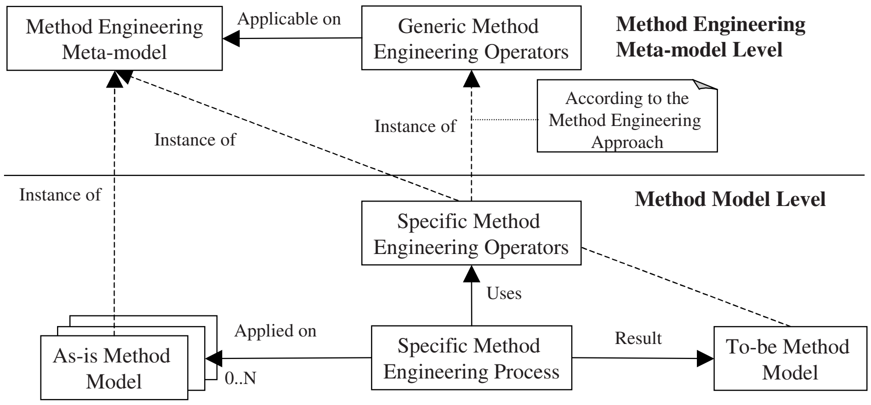
Method Engineering Meta-model
Applicable on
Generic Method Engineering Operators
Method Engineering Meta-model Level
According to the Method Engineering Approach
Instance of
Instance of
Instance of
Method Model Level
Specific Method Engineering Operators
Uses
Applied on
As-is Method Model
0..N
Specific Method Engineering Process
Result
To-be Method Model

Fig. 1. Operator-driven Method Engineering Framework

The generic ME operators are applicable to generic elements that compose any model involved in a ME activity. To achieve this, it is necessary to abstract from the specificity of a given model and generalize model elements, their relationships as well as relationships between different models. Meta-modelling is known as a means to do so. Thus, in order to build the typology of generic ME operators, we first developed a *meta-model* for ME, i.e. a model of models. This meta-model is presented in the following section.

Following our framework (Fig. 1), a set of specific operators is instantiated from the generic ones according to the selected ME approach. These operators are then applied by the specific ME process to transform one or several As-Is model(s) into a To-Be model. In Paradigm-Based ME there is only one To-Be model whereas in Assembly-Based and Extension-Based ME two or more As-Is models are used to produce the resulting To-Be model. Ad-Hoc method construction starts with no As-Is model at all. To sum up, there are some advantages of using a generic typology of ME operators:

1. The generic typology serves as a guide to define the specific typology: the latter is just an instance of the former;
2. The completeness of the specific typology is subsumed by the completeness of the generic typology;
3. Specific typologies are consistent with each other as they are generated from the same mould: this is important when several sub-typologies are used in the same ME approach or several different ME approaches are combined together [17].

3.2 The Meta-model for Defining Generic ME Operators

In Fig. 2 we propose a meta-model which has been designed to highlight characteristics of models involved in a ME activity and therefore to permit to identify the fundamental construction and transformation operations which can be executed on a model. As shown is this figure, every model is made of *Elements*. Every element has a *Name* and is characterised by a set of *Properties*. In the E/R model for example, *Entity type, Attribute* and *Relationship type* as well as the *Is-A* relationship are *elements*. *Domain* is a *property* of *Attribute.* Fig. 2 shows that an element *is-a* another element, i.e. might inherit some of its properties from another element.

Elements are classified into *Simple* and *Compound* ones. *Compound elements* are composed from fine-grained ones whereas *Simple Elements* are not decomposable. For example, in the E/R model an *Entity type* is a compound element made of *Attributes*, which are simple elements.

As the same element can be part of different models, the concept of *ModelElement* represents the link of an element and the model it belongs to. For example, the concept of *Scenario* exists in the *L'Ecritoire* model [23] and the *Use Case* model [9]. In the integration process of these two models [19] we need to know the origin of the *Scenario* that we are manipulating. The concept of *ModelElement* is also necessary to model the relationships between elements of different models. These relationships are represented in Fig. 2: an element from one model can represent an *Abstraction-of* an element in another model; the link *Instance-of* represents the fact that an element can be obtained by instantiating an element of another model; moreover, elements from different models can be connected in order to assemble or extend models. Three connection types are defined in the meta-model: *Association*, *Composition* and *Is-a*. Finally, any model is a compound element which can be reduced to the *root* element.

3.3 The Typology of the Generic ME Operators

The meta-model (Fig. 2) identifies elements (*Element*) in models and relationships between elements *(ModelElement)* belonging to different models. Both of them can be subject to change in a method engineering activity. This allowed us to identify a

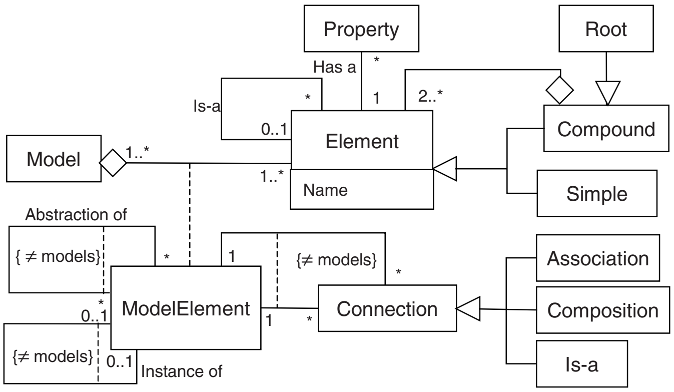

Fig. 2. Meta-model for Method Engineering

set of *ME operators*, which are listed and briefly described in Table 1. In synthesis, we can say that ME operators cover three major types of change: *naming* changes, *element* changes and *structural* changes.

- *Naming changes* are defined with the *Rename* operator. Naming is dealing with hyponyms, synonyms and the like.
- *Element changes* affect elements and are circumscribed to the elements themselves: adding an *attribute* to an *entity type* is an example of such localised change. Table 1 proposes three operators to specify element changes, namely *Modify*, *Give* and *Withdraw*.
- *Structural changes* are the most important as they correspond to a modification of the set of elements that compose the model. There are two types of structural changes:
  – *Inner changes* which affect elements of one single given model: there are eleven operators to specify structural changes (Table 1): *Add, Remove, Merge*, *Split*, *Replace*, *Retype*, *Generalise*, *Specialise*, *AddComponent*, *MoveComponent* and *RemoveComponent*. For example, merging two steps of an As-Is process model in the To-Be process model is an example of such inner structural change.
  – *Inter-model changes* which consists in establishing connections between elements of different models. Table 1 identify six of them: *ConnectViaSpecialisation*, *ConnectViaGeneralisation*, *ConnectViaMerge*, *ConnectViaComposition*, *ConnectViaDecomposition* and *ConnectVia-Association.* For example, defining a set of *Ordered Requirements Chunks* of the *L'Ecritoire* RE method [23] as a specialization of the *Use Case* Concept of the *Use case Model* [9] is an example of inter-model connection.

4 Instantiation of the Generic ME Operators for Assembly-Based ME

This section illustrates the framework and the use of our generic operators to define the collection of operators relevant for a specific ME case. We consider two assem-

Table 1. Generic ME operators

Object	Operator	Description
Element	*Rename*	Change the name of an element.
	Add	Add a new element in the model.
	Remove	Remove an element from the model.
	Merge	Two separate elements become one element.
	Split	An element is decomposed into two elements.
	Replace	An element is replaced by a different one.
	Generalize	An element is created as a generalization of two elements.
	Specialize	Specialise an element into two sub-elements.
Compound	AddComponent	Add a component into an element.
	RemoveComponent	Remove a component from a compound element.
	MoveComponent	A component is repositioned in the structure of a compound.
Property	*Give*	Add a property to an element.
	Withdraw	Remove a property from an element.
	Modify	Change a property in an element.
	Retype	Change the type of an element.
Model Element	Instantiate	Instantiate an As-Is model element into To-Be model element.
	Abstract	Create a To-Be model element as an abstraction of an As-Is model element.
	ConnectVia Specialization	Define an element from one model as a specialization of an element from another model. An is-a link is created between these two elements.
	ConnectVia Generalization	Generalize two elements from different As-Is models into a super-element in the To-Be model.
	ConnectVia Composition	Create a compound element in the To-Be model containing as components elements from two different As-Is models.
	ConnectVia Decomposition	Define an element as a component of an element from another model.
	ConnectVia Association	Add an association link in the To-Be model between two elements from different As-Is models.
	ConnectViaMerge	Two similar elements from different As-Is models become one element in the To-Be model.

bly-based ME approaches: (a) the *assembly by association* proposed by Brinkkemper et al. in [2] and (b) the *assembly by integration* proposed by Ralyté et al. in [19]. In both cases product and process models of the selected method chunks/fragments must be assembled. Therefore, we will first, define the corresponding meta-models and then, instantiate the generic typology of operators in line with the elements of these meta-models and illustrate them with examples.

4.1 Operators for Product Models Assembly

In both ME examples considered in this section, the product models of the method fragments/chunks to assemble are expressed by using class diagrams. Fig. 3 presents the meta-model of the *Object Model* as instance of the ME meta-model (Fig. 2). As shown in this figure, the *Object Model* is composed of *Classes*, which are *compound elements* composed of *Attributes*. A *Class* is connected with one or several other classes via *Association*, *Composition* or *Is-a* links. As an *Association* can have attributes, it is also a compound element whereas the *Composition* or *Is-a* links are *simple* ones. An *Attribute* has a *property* named *Domain* and an *Association* has two properties *SourceMultiplicity* and *TargetMultiplicity*.

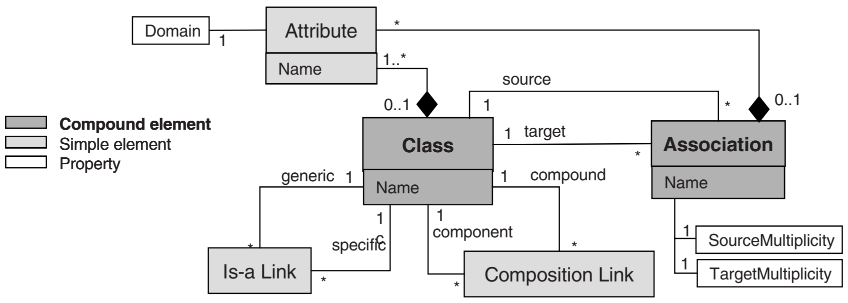

Fig. 3. Meta-model of the Object Model

Table 2 summarizes the operators that are relevant for each of the elements defined in the meta-model (Fig. 3). The name of a specific operator is obtained by concatenation of the name of the generic operator and the name of the corresponding element. Due to space constraint, some names have been shortened. 'N/A' means not applicable.

Table 2. Operators for the Object Models Assembly

Generic Operator	**Operators for object models assembly**				
	Class	Attribute	Association	Composition	Is-a
Rename	RenameClass	RenameAttribute	RenameAssoc	N/A	N/A
Add	AddClass	N/A	AddAssociation	AddComp	AddIsa
Remove	RemoveClass	N/A	RemoveAssoc	RemoveComp	ReIsa
Merge	MergeClass	MergeAttribute	N/A	N/A	N/A
Split	SplitClass	SplitAttribute	N/A	N/A	N/A
Replace	ReplaceClass	ReplaceAttribute	N/A	N/A	N/A
Generalize	GeneralClass	N/A	N/A	N/A	N/A
Specialize	SpecializeClass	N/A	N/A	N/A	N/A
AddComponent	N/A	AddClassAttr, *	N/A	N/A	N/A
RemoveComponent	N/A	RemoveClAtt, *	N/A	N/A	N/A
MoveComponent	N/A	MoveClAttr, *	N/A	N/A	N/A
Give	N/A	GiveDomain	GiveMultiplicity	N/A	N/A
Withdraw	N/A	With.Domain	WithdrawMultipl	N/A	N/A
Modify	N/A	ModifyDomain	ModifyMultipl	N/A	N/A
Retype	RetypeClass	RetypeAttribute	RetypeAssociation	RetypeComp	RtIsa
***CVSpecialization*	CVSpecClass	N/A	N/A	N/A	N/A
CVGeneralization	CVGenerClass	N/A	N/A	N/A	N/A
CVComposition	CVCompClass	N/A	N/A	N/A	N/A
CVDecomposition	CVDecompClass	N/A	N/A	N/A	N/A
CVAssociation	CVAssocClass	N/A	N/A	N/A	N/A
CVMerge	CVMergeClass	N/A	N/A	N/A	N/A
*: AddAssociationAttribute, RemoveAssociationAttribute, MoveAssociationAttribute, ***CV: ConnectVia,*					

Let us briefly comment the table before entering in the detailed analysis of both ME approaches. According to [15, 19], the *integration* of two object models is based on establishing connections between similar classes. Two similar classes from different As-Is models can be *merged* into a new one in the To-Be model. They can also *be connected* via is-a or composition link and finally, a new *generalised class can be created* in the To-Be model in order to relate them. Therefore, the operators which serve for the integration of two object models are *ConnectViaMerge*, *Connect-ViaSpecialisation*, *ConnectViaGeneralisation*, *ConnectViaComposition*, *ConnectVia-Decomposition*. The simple association between similar classes is not applicable here. On the contrary, the operator *ConnectViaAssociation* is the core connection operator in the assembly by association [2]. The *Instantiate* and *Abstract* operators are not applicable in the assembly-based ME as the As-Is and To-Be models are at the same levels of abstraction.

Product Models Assembly by Association. According to [2], the assembly by association of two product fragments is based on the three following operations: (1) Addition of new objects, (2) Addition of new associations, (3) Addition of new attributes. Besides, this approach provides a set of rules that should be satisfied during the assembly process. For instance, at least one concept and/or association should connect two method fragments to be assembled, is an example of a rule. There are no concepts which have the same name and which have different occurrences in a method description, is another example of a rule.

It can be seen that the required operations can be realised by applying the operators *AddClass*, *ConnectViaAssociationOfClasses, AddClassAttribute* and *AddAssociation-Attribute*. These operators are formalised as follows:

ConnectViaAssociationOfClasses. This operator connects two classes from different As-Is models with a new association in the To-Be model.

CnnectViaAssociationOfClasses: Class$^2 \rightarrow$ Association
CnnectViaAssociationOfClasses(C_1, C_2) = A.source (C_1) $\wedge$ A.target (C_2) | A$\in$ Association, $C_1 \subset OM_1$, $C_2 \subset OM_2$, (OM_1, OM_2) $\in$ Object Model

Let us exemplify the association of the two following method fragments *Statechart* and *Object model* as considered in [2]. The behaviour of each *Class* is specified by a set of *States*. An association *Has* is added between these two classes in order to connect them:
CnnectViaAssociationOfClasses(Class, State) = Has.source(Class) $\wedge$ Has. target(State) | State $\subset$ Statechart, Class $\subset$ Object Model.

AddClass. This operator can be applied to add a new class in the To-Be model to make possible the connection between As-Is models.
AddClass: Object Model $\rightarrow$ Class
AddClass(OM) = C | C $\subset$ OM, C $\in$ Class

For example, the *Transaction* element in the *Statechart* fragment has a post condition that refers to an *Attribute* which is an element of the *Object Model* fragment. As a consequence, a new class *PostCondition* should be added in the To-Be model in order to connect the *Transition* and *Attribute* classes:
AddClass(Objectchart) = PostCondition | PostCondition $\subset$ Oojectchart.

AddClassAttribute and ***AddAssociationAttribute*** allow to add attributes in the To-Be method fragment to the classes and associations created as connectors of the As-Is method fragments.

AddClassAttribute: Class $\rightarrow$ Class.Attribute
AddClassAttribute(C) = C.At | At$\in$ Attribute
AddAssociationAttribute: Association $\rightarrow$ Association.Attribute
AddAssociationAttribute(A) = A.At | At$\in$ Attribute

According to the example shown in [2], the attribute *Is-hidden* should be added to the association *Is-annotated-with*:
AddAssociationAttribute(Is-annotated-with) = Is-annotated-with.Is-hidden
As shown in Table 2, there are other operators that support methods assembly in addition to the four presented above. It seems to us that these operators are relevant, in particular to tailor some As-Is fragments to the special needs of the assembly process and also to refine the obtained To-Be fragment if necessary. As an illustration let us consider again the previous *Postcondition* case: we could directly associate these two classes using the *ConnectviaAssociationofClasses* and then, retype this association into the class *Postcondition* by applying the operator *RetypeAssociation.*

RetypeAssociation. This operator transforms an association A, connecting two classes C_1 et C_2, into a new class C. Besides, two new associations are added in order to connect the class C with the classes C_1 et C_2.

RetypeAssociation: Association, $\text{Class}^2 \rightarrow$ Class, Association^2
RetypeAssociation(A, C_1, C_2) = C $\wedge$ A_1.source(C_1) $\wedge$ A_1.target (C) $\wedge$ A_2.source(C) $\wedge$ A_2.target (C_2) | (A_1, A_2) $\in$ Association, C $\in$ Class

In order to satisfy the assembly rules mentioned above, it might be necessary to modify the names of some classes before the assembly of the method fragments.

RenameClass. The operator *RenameClass* allows to give a new name to a class:

RenameClass: Class $\rightarrow$ String
RenameClass(C) = C.name(N) | N$\in$ String

For sake of space it is not possible to illustrate the use of each of the operators for method assembly proposed in Table 2, but a systematic study convinced us that they are useful in method assembly by association.

Product Models Assembly by Integration. To illustrate operators for the assembly by integration, we consider the approach proposed in [19]. According to this approach, the assembly process consists in identifying common elements in product and process models of some selected method chunks and in merging and/or connecting them. This might require some terminology adjustments of model elements before their integration. Elements of product and process models of the selected methods need to be unified based on their similarities, abstracting away their differences and eliminating ambiguities. The integration of two product models which we consider in this section requires to identify similar classes. For example, [15] illustrates the integration of the *Use case* model [9] and the *L'Ecritoire* model [23]. The class *Actor* in the *Use case* model and the class *Agent* in the *L'Ecritoire* model have the same semantic. Therefore, one of these two classes must be renamed prior their merge. In our example *Actor* is renamed into *Agent*.

RenameClass(Actor) = Name(Agent)

MergeClass. When two classes C_1 and C_2 from different As-is models are merged into a new class *C* in the To-Be model, the class *C* replaces C_1 and C_2 in any association having initially C_1 or C_2 as source class or target class.

MergeClass: Class^2, $\{\text{Association}\}^2 \rightarrow$ Class

MergeClass(C_1, C_2, $\{A^s_i, A^t_j\}$) = [$\forall$ i, A^s_i.source(C)] $\wedge$ [$\forall$ j, A^t_j.target(C)] | $C_1 \subset OM_1$, $C_2 \subset OM_2$, $C \subset OM$, $C \in$ Class, (OM, OM_1, OM_2) $\in$ Object Model

Therefore, the *Actor* (*$Actor_{UC}$*) from the *Use case* model and *Agent* (renamed into *Actor*) (*$Actor_E$*) from *L'Ecritoire* are merged into a new class *Actor*. Two associations, *Executes* and *Supports*, between the classes *$Actor_{UC}$* and *Use case* are preserved by replacing *$Actor_{UC}$* by *Actor*. Similarly, in the associations *From* and *To* between the classes *Action* and *$Actor_E$* in the *L'Ecritoire*, the *$Actor_E$* is replaced by *Actor:*

MergeClass($Actor_{UC}$, $Actor_E$, Executes.source($Actor_{UC}$), Supports.source($Actor_{UC}$), From.target($Actor_E$), To.target($Actor_E$)) = Actor $\wedge$ Executes.source(Actor) $\wedge$ Supports.source(Actor) $\wedge$ From.target(Actor) $\wedge$ To.target(Actor)

The operators as *ConnectViaSpecializationClass*, *ConnectViaGeneralizationClass*, *ConnectViaCompositionClass*, *ConnectViaDecompositionClass* are also useful in the assembly by integration. They allow to connect classes that have a similar semantic but different structures and cannot be directly merged. For example, the *Goal* concept exists in both the *Use Case* and *L'Ecritoire* models, but it is defined as an attribute named *Objective* in the class *Use Case* of the first model and as a class in the second one. In order to connect these two concepts we need to transform first the attribute *Objective* into a class in the *Use Case* model. The original approach [19] uses the *Objectify* operator to do that. This operator is formalised here by the *RetypeAttribute* operator.

RetypeAttribute. An attribute of a class C_1 is transformed into a new class C_2 which is associated to the class C_1 with new association.

RetypeAttribute: Class.Attribute $\rightarrow$ Class, Association
RetypeAttribute(C_1.At) = $C_2 \wedge$ A.source(C_1) $\wedge$ A.target(C_2) | $C_2 \in$ Class, A $\in$ Association

RetypeAttribute(Use Case. Objective) = Goal $\wedge$ Has.source(Use Case) $\wedge$ Has.target(Goal).

Even after retyping, the merge is not possible because the *Goal* of the *L'Ecritoire* has a specific structure whereas the goal of the *Use Case* model is an informal statement. The solution is to rename (a) the *Goal* of the *Use Case* model into *Informal Goal* and (b) the *Goal* of the *L'Ecritoire* into *Formal Goal* and to connect them via generalisation into the class *Goal*.

ConnectViaGeneralizationClass. Two classes from different As-Is models are generalized into a generic class in the To-Be model. Two is-a links are created between the specific classes and the generic one.

ConnectViaGeneralisationClass: $\text{Class}^2 \rightarrow \text{Class}, \text{Is-a}^2$
ConnectViaGeneralisationClass(C_1, C_2) = $C \wedge$ [Is-a.generic(C) $\wedge$ Is-a.specific(C_1)] $\wedge$ [Is-a.generic(C) $\wedge$ Is-a.specific(C_2)] $\mid C \in$ Class

ConnectViaGeneralisationClass(Informal Goal, Formal Goal) = Goal $\wedge$ *[Is-a.generic(Goal)* $\wedge$ *Is-a.specific(Informal Goal)]* $\wedge$ *[Is-a.generic(Goal)* $\wedge$ *Is-a.specific(Formal)].*

4.2 Operators for Process Models Assembly

In this section we use the generic typology of operators to generate specific ME operators to assemble process models in (a) the case of assembly by association and (b) the case of assembly by integration. Different kinds of process models can be used to express the process dimension of a method fragment/chunk. It can be a simple ordered list of operations, a more structured activity diagram or a complex multi-strategy model expressed through a directed graph structure. The definition of operators for process models assembly depends on the type of the process models used by the As-Is methods. For example, the approach for assembly by association [2] uses an activity diagram to model process fragments whereas the approach for assembly by integration proposes to integrate process maps [24] (directed graphs of intentions and strategies). In both cases, before generating operators we need to define first the corresponding meta-models.

Process Models Assembly by Association. Fig. 4 presents the meta-model for the *Activity diagram* as instance of the ME meta-model (Fig. 2).

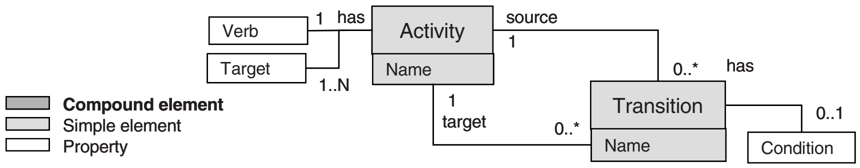

Fig. 4. Meta-model for an Activity Diagram

As shown in this figure, an activity diagram is represented by set of *Activities*, which are *simple elements*. *Transitions* define in which order activities are realised by specifying for each transition the source activity and the target one. Each *Activity* has two *Properties*: a *Verb*, which represents the operation to realise, and a *Target*, which represents the resulting product elements. *Condition* is a property of a *Transition*. Table 3 shows the operators related to the core Activity diagram elements.

In the example dealing with the assembly of *Statechart* and *Object Model* fragments [2], the authors use two core operations: (1) addition of new transitions and (2) addition of new activities. A new transition can be added only to connect the activities from different fragments as well as a new activity can be added only if a new class was added during the corresponding product fragments assembly. These two operations can be formalised with the operators *AddActivity*, *AddTransition* and *Connect-ViaAssociationOfActivities*.

Table 3. Operators for the Activity-driven Process Models Assembly

Generic Operator	Operators for activity-driven process models assembly	
	Activity	Transition
Rename	RenameActivity	RenameTransition
Add	AddActivity	AddTransition
Remove	RemoveActivity	RemoveTransition
Merge	MergeActivity	MergeTransition
Split	SplitActivity	SplitTransition
Replace	ReplaceActivity	ReplaceTransition
Give	GiveVerb, GiveTarget	GiveCondition
Withdraw	WithdrawVerb, WithdrawTarget	WithdrawCondition
Modify	ModifyVerb, ModifyTarget	ModifyCondition
ConnectViaAssociation	ConnectViaAssociationOfActivities	N/A
ConnectViaMerge	ConnectViaMergeOfActivities	N/A

ConnectViaAssociationOfActivities. A new transition connects two activities from form different As-Is models. The source activity must produce the product element(s) required as input product by the target activity.

ConnectViaAssociationOfActivities: $\text{Activity}^2 \rightarrow \text{Transition}$
ConnectViaAssociationOfActivities $(A_1, A_2) = \text{T.source}(A_1) \wedge \text{T.target}(A_2) \mid T \in$ Transition, $A_1 \subset AD_1$, $A_2 \subset AD_2$, $(AD_1, AD_2) \in$ Activity Diagram

For example, the list of classes obtained by executing the *Object Model* construction activity *O1: Identify Objects and Classes* provides an input for the *Statechart* construction activity *S1: Identify States*. Therefore, these two activities can be connected with a new transition called *Input*:

ConnectViaAssociationOfActivities (O1, S1) = Input.source(O1) $\wedge$ *Input.target(S1).*

Again, this example illustrates only partially the use of operators listed in Table 3. However, operators such as *Merge, Split, Replace, Remove* applied to both *Activity* and *Transition* are obviously useful. Similarly, the need for renaming an activity of the As-Is fragment in the To-Be fragment is meaningful. Finally, *Give, Withdraw* and *Modify* make sense to change the properties of an As-Is *Activity* or *Transition* in the corresponding To-Be fragment. For instance, a condition for the transition between two activities can be modified and the verb designating an activity can be different in the To-Be fragment compared to what it was in the As-Is model.

Process Models Assembly by Integration. According to the assembly by integration proposed in [19, 15], the process models integration consists in integrating process maps [24]. Fig. 5 represents the map meta-model as instance of the ME meta-model (Fig. 2).

As shown in Fig. 5, a *Map* is a collection of *Sections*. A section is a *compound element* aggregating two types of *Intentions*, the *Source Intention* and the *Target Intention*, and a *Strategy*.

An *Intention* is a goal that can be achieved by the performance of an activity (automated/semi-automated or manual). For example, *Elicit a goal* is an intention in the *L'Ecritoire* requirements elicitation process; *Write a scenario* is another intention. There are two special intentions *Start* and *Stop* that allow to begin and to end the progression in the map, respectively. An intention is a *simple element* expressed following a linguistic approach proposed by [13] as a clause with a *verb* and a *target*. It can also have several *parameters*, where each parameter plays a different role with respect to the verb.

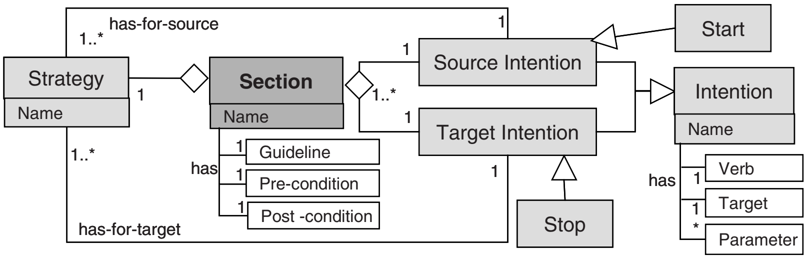

Fig. 5. Map meta-model

A *Strategy* is an approach, a manner to achieve an intention. For example, *By using goal template* is a strategy to achieve the intention *Elicit a goal* proposed in the *L'Ecritoire* approach. A strategy is a *simple element.*

A *Section* is a triplet <*Source Intention, Target Intention, Strategy*>. The arrangement of he sections in a map forms a labelled directed graph with *Intentions* as nodes and *Strategies* as edges. Pre- and Post-conditions of each section specify the progression flows in the map. Each section provides a *Guideline* indicating how to achieve the target intention following the strategy given the source intention has been achieved.

Table 4 proposes the list of operators for maps integration. Only 11 operators have been instantiated from 20 potential ones (Table 5). In fact, some of generic operators do not make sense in the maps integration process. For example, the *Generalize* and *Specialize* operators cannot be instantiated as there are no is-a relationships between intentions, strategies or sections in the map. The three missing operators, namely *AddComponent*, *RemoveComponent* and *MoveComponent* have not been introduced, as it does not make sense to apply them to the *Section* element the structure of which is immutable. The integration of two maps can be done only by merging similar intentions or sections. As a consequence, only the *ConnectViaMerge* operator was instantiated. It is impossible to merge two strategies belonging to different maps.

The example of integration [19] of the *Use Case* and *L'Ecritoire* maps starts with the identification of similar intentions and their merge. The intention I_{UC1}*: Elicit Use Case* in the Use Case model and the intention I_{E1}*: Elicit Goal* in L'Ecritoire have the same semantic: in both cases it means 'to elicit a users goal'. Moreover, the *Goal* concept was defined in the Use Case model during the product models integration illustrated above and allows us to unify the terminology of the two maps. In [19] this was done intuitively by renaming the intention I_{UC1}*: Elicit Use Case* into I_{UC1}*: Elicit Goal*. The generic typology of ME operators allows us to formalise this kind of change in a more precise way: each property of an intention has a proper *Modify* operator: *ModifyVerb*, *ModifyTarget* and *ModifyParametter*. In this example, we need to apply the *ModifyTarget* operator.

ModifyTarget. The value of the intention property *Target* is replaced by a new one. This new value must represent an element of the corresponding product model.

ModifyTarget: Intention $\rightarrow$ String
ModifyTarget(I) = I.target(N) $\mid$ N$\in$ String

ModifyTarget(I_{UC1}:Elicit Use Case) = I_{UC1}.target(Goal)

Table 4. Operators for Maps Integration

Generic Operator	Operators for Maps integration		
	Intention	Strategy	Section
Rename	RenameIntention	RenameStrategy	RenameSection
Add	AddIntention	AddStrategy	AddSection
Remove	RemoveIntention	RemoveStrategy	RemoveSection
Merge	MergeIntention	MergeStrategy	MergeSection
Split	SplitIntention	SplitStrategy	SplitSection
Replace	ReplaceIntention	N/A	N/A
Give	GiveVerb, GiveTarget GiveParameters	N/A	GivePreCond, GivePostCond, GiveGuideline
Withdraw	WithdrawVerb, WdrTarget WithdrawParamerters	N/A	WithdrawPreCondition WdrPostCond, WdrGuideline
Modify	ModifyVerb, ModifyTarget ModifyParameters	N/A	ModifyPreCond, ModifPost-Cond ModifyGuideline
Retype	RetypeIntention	RetypeStrategy	N/A
ConnectViaMerge	CVMergeIntentions	N/A	ConnectViaMergeSections

ConnectViaMergeIntentions. This operator allows to integrate two maps by merging their similar intentions. When two intentions I_1 and I_2 are merged, the intention *I* replaces I_1 and I_2 in any section having initially I_1 or I_2 as source intention or target intention.

ConnectViaMergeIntentions: Intention^2, {Strategy}^{2} $\rightarrow$ Intention

ConnectViaMergeIntentions(I_1, I_2, {St^s_i}, {St^t_j}) = [$\forall$ i, St^s_i.has-for-source(I_r)] $\wedge$ [$\forall$ j, St^t_j.has-for-target(I_r)] | $I_r \in$ Intention, $I_1 \subset M_1$, $I_2 \subset M_2$, (M_1, M_2) $\in$ Map

In order to merge the intentions I_{UC1}*: Elicit a Goal* and I_{E1}*: Elicit a Goal* we must know in which sections of their corresponding maps they are involved. As shown in Fig. 6, there are three sections in the Use Case map containing the intention I_{UC1} whereas the intention I_{E1} is involved in four sections in the L'Ecritoire map. The intention I_1*:Elicit a Goal* replaces I_{UC1} and I_{E1} in all these sections.

ConnectViaMergeIntentions(I_{UC1}, I_{E1}, St_{UC1}, St_{UC2}, St_{UC3}, St_{E1}, St_{E2}, St_{E3}, St_{E4}) = St_{UC1}.target(I_1) $\wedge$ St_{UC2}.source(I_1) $\wedge$ St_{UC3}.source(I_1) $\wedge$ St_{E1}.target(I_1) $\wedge$ St_{E2}.source(I_1) $\wedge$ St_{E2}.target(I_1) $\wedge$ St_{E3}.source(I_1) $\wedge$ St_{E4}.source(I_1).

In the same manner the *Start* and *Stop* intentions of both maps are merged. Other operators such as *AddStrategy* and *RemoveStrategy* are needed in order to improve the final *To-Be* map. For example, the integration of the Use Case and L'Ecritoire maps allows to improve the scenario writing process which is rather poor in the first model by rich guidelines provided by second one. It appears that the original Use Case strategy supporting scenario writing became obsolete and should be removed from the final integrated map.

To conclude on the assembly by integration, a systematic comparison of operators identified in [19] and those generated from the typology of generic ones shows that (a) we missed useful operators in the former and (b) the systematic definition provided by the latter avoid 'ad-hoc' and not fundamentally justified assembly types. The so-called *Objectify* operator mentioned above is an example of (b); *Give, Withdraw and Modify* applied to intention and section are examples of (a).

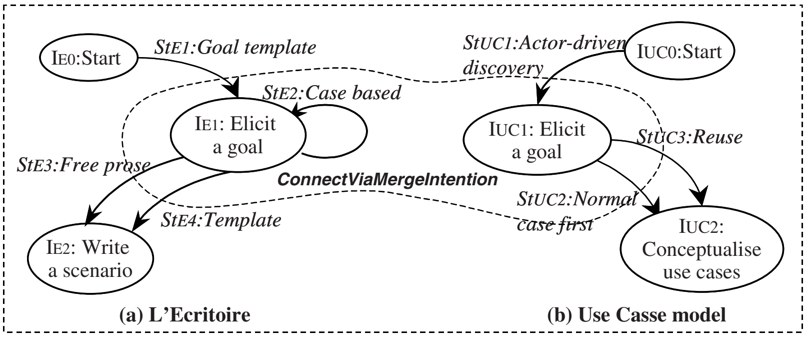

Fig. 6. Example of integration of L'Ecritoire and Use Case maps

5 Conclusion

In this paper we provided a formal ground for tool-supported ME in the form of a set of generic ME operators. The production of these operators is based on a ME meta-model that was especially defined for this purpose and a classification of ME approaches issued from a literature survey. The set of operators allows to understand in a cohesive and consistent way which operations constitute the basis of method construction and method transformation. The set of generic operators considerably eases the generation of the specific set of operators required in a given ME approach.

Our future preoccupation is to facilitate even more the process of generating specific operators from the generic ones by introducing sub-typologies, each being relevant for a ME class of approaches. Table 5 shows our first view on this. For sake of space the figure shows the operators which are different depending of the class of approaches. It can be noticed that the differences relate to *ModelElements*. We indeed think that all operators related to *Element* are relevant irrespective of the ME class.

The generic typology seems to capture all interesting types of method engineering operations. However, the problem to consider next is the validation of its completeness and correctness. According to Banerjee [1], a set of operators is considered to be complete if it subsumes every possible schema evolution; it is correct if the execution of any operator does not result in an incorrect schema. By analogy, in order to prove the completeness of the generic ME operators typology we need to identify a minimal set of operators whereas the correctness of a set of specific operators required to define a set of model invariants. For example, to ensure the correctness of the operators for maps integration, we need to define what a *correct* map is. This is achieved by

Table 5. Sub-typologies of ME operators

Object	**Operator**	**From Scr.**	**Paradigm-Based**			**Exten-sion**	**Assembly**	
			Instan.	Abstr.	Adapt.		Assoc.	Integr.
Model Ele-ment	*Instantiate*		+					
	Abstract			+				
	ConnectViaSpecialisation					+		+
	ConnectViaGeneralisation.					+		+
	ConnectViaComposition.					+		+
	ConnectViaDecomositionp					+		+
	ConnectViaAssociation					+	+	
	ConnectViaMerge							+

adding a set of conditions called invariants to the structural definition of a map. An invariant must hold in any quiescent state of a map, that is, before and after any execution of an operator to one or several As-Is map(s) resulting in a new state of the To-Be map.

Finally, the generic ME operators will serve to the development of a CAME tool supporting different ME approaches.

References

1. Banerjee, J., Kim, W., Kim, H.-J., Korth, H. F. Semantics and Implementation of Schema Evolution in Object Oriented Databases. *Proceedings. of the ACM-SIGMOD Annual Conference*, pp. 311--322, San Francisco, CA, 1987.
2. Brinkkemper S., Saeki, M., Harmsen, F. Assembly Techniques for Method Engineering. *Proceedings of the 10th International Conference CAiSE'98.* Pisa, Italy, 1998.
3. Deneckere, R. Using Meta-patterns to Construct Patterns. *Proc. of the Conference on Object-Oriented Information Systems, OOIS'2002*, Springer, France, 2002.
4. Grundy, J.C., Venable, J.R. Towards an Integrated Environment for Method Engineering. In *Challenges and Strategies for Research in Systems Development*. W.W. Cotterman and J.A. Senn (Eds.). John Wiley & Sons. Chichester. pp. 45-62, 1996.
5. Gupta, D., Prakash, N. Engineering Methods from Method Requirements Specifications. *Requirements Engineering Journal*, Vol.6, pp.135-160, 2001.
6. Harmsen A.F., Brinkkemper, S., Oei, H. Situational Method Engineering for Information System Projects. *In Olle T.W. and A.A. Verrijn Stuart (Eds.), Mathods and Associated Tools for the Information Systems Life Cycle,* Proceedings of the IFIP WG8.1 Working Conference CRIS'94, pp. 169-194, North-Holland, Amsterdam, 1994.
7. Harmsen, A.F. *Situational Method Engineering. Moret Ernst & Young,* 1997.
8. Heym, M., Osterle, H. Computer-aided methodology engineering. *Information and Software Technology*, Vol. 35 (6/7), June/July, pp. 345-354, 1993.
9. Jacobson I., Christenson, M., Jonsson, P., Oevergaard, G. Object Oriented Software Engineering: a Use Case Driven Approach. *Addison-Wesley*, 1992.
10. Kumar, K., Welke, R.J. Method Engineering, A Proposal for Situation-specific Methodology Construction. *In Systems Analysis and Design: A Research Agenda*, Cotterman and Senn (eds), Wiley, pp257-268, 1992.
11. Prakash, N., Bhatia, M. P. S. Generic Models for Engineering Methods of Diverse Domains. *Proceedings of CAISE'02,* Toronto, Canada, LNCS 2348, pp. 612., 2002.
12. Prakash, N. Towards a formal definition of methods. *RE Journal*, 2 (1), 1997.

13. Prat, N. Goal formalisation and classification for requirements engineering. In: Proceedings of the REFSQ'97, Barcelona, 1997.
14. Punter H.T., Lemmen, K. The MEMA model : Towards a new approach for Method Engineering. *Information and Software Technology*, 38(4), pp.295-305, 1996.
15. Ralyté, J., Rolland, C. An Assembly Process Model for Method Engineering. *Proceedings of the 13th CAISE'01*, Interlaken, Switzerland, 2001.
16. Ralyté, J., Rolland, C. An approach for method reengineering. *Proceedings of the 20th ER2001*, Yokohama, Japan, LNCS 2224, Springer, pp.471-484, 2001.
17. Ralyté, J., Deneckère, R., Rolland, C. Towards a Generic Model for Situational Method Engineering. *Proceedings of the 15th International Conference CAISE'03*, Klagenfurt/Velden, Austria, LNCS 2681, Springer, pp. 95-110, 2003.
18. Ralyté, J., Rolland, C., Ben Ayed, M. An Approach for Evolution Driven Method Engineering. To be published in *Information Modeling Methods and Methodologies*. J. Krogstie, T. Halpin, K. Siau (Eds.), Idea Group, Inc., USA, 2003.
19. Ralyté J., Rolland, C., Plihon, V. Method Enhancement by Scenario Based Techniques. *Proc. of the 11th Conference CAISE'99*, Heidelberg, Germany, 1999.
20. Rolland, C., Plihon, V. Using generic chunks to generate process models fragments. *Proceedings of 2nd IEEE International Conference on Requirements Engineering*, ICRE'96, Colorado Spring, 1996.
21. Rolland, C., Prakash, N. A proposal for context-specific method engineering. *Proceedings of the IFIP WG 8.1 Conference on Method Engineering,* Chapman and Hall, pp 191-208, Atlanta, Gerorgie, USA, 1996.
22. Rolland, C., Plihon, V., Ralyté, J. Specifying the reuse context of Scenario Method Chunks. *Proceedings of the 10th International Conference CAISE'98*, Pisa, Italy, 1998.
23. Rolland, C., Souveyet, C., Salinesi, C. Guiding Goal Modelling using Scenarios, IEEE Transactions on Software Engineering, 24(12): 1055-1071, 1998.
24. Rolland, C., Prakash, N., Benjamen, A. A Multi-Model View of Process Modelling. *Requirements Engineering Journal*, Vol. 4 (4), pp169-187, 1999.
25. Rossi, M., Tolvanen, J-P., Ramesh, B., Lyytinen, K., Kaipala, J. Method Rationale in Method Engineering. Proceedings of the 33rd Hawaii International Conference on Systems Sciences, 2000.
26. Saeki, M. Embeding metrics into Information Systems Development methods: An Application of method Engineering Technique. *Proceedings of the 15th International Conference CAISE'03, Velden, Austria, LNCS 2681, Springer,* pp. 374-389, 2003.
27. Saeki, M. Toward Automated Method Engineering: Supporting Method Assembly in CAME. Invited talk in the Int. Workshop on Engineering Methods to Support Information Systems Evolution (EMSISE'03), http://cui.unige.ch/db-research/EMSISE03/ , 2003.
28. Song, X. Systematic Integration of Design Methods. *IEEE Software*, 1997.
29. Tolvanen, J-P., Rossi, M. & Liu H., Method Engineering : Current research directions and implications for future research. In *Method Engineering: Principles of method construction and tool support.* S. Brinkkemper, K. Lyytinen, R.J. Welke (Eds.), Proceedings of the IFIP TC8 WG8.1/8.2. Atlanta, USA, pp. 296-317, 1996.
30. Tolvanen, J.-P. Incremental Mehtod Engineering with Modeling Tools: Theoretical Principles and Empirical Evidence. *PhD Dissertation. University of Jyväskylä,* Finland, 1998.