securityandtrust.lu

On the Formalisation of GeKo: a Generic Aspect Models Weaver

Max E. Kramer[1,2], Jacques Klein[2], Jim R. H. Steel[3], Brice Morin[5], Jörg Kienzle[6], Olivier Barais[4], and Jean-Marc Jézéquel[4]

[1] SnT, University of Luxembourg, Luxembourg
[2] Karlsruhe Institute of Technology, Karlsruhe, Germany
[3] The University of Queensland, Brisbane, Australia
[4] IRISA-INRIA, Triskell, Rennes, France
[5] SINTEF ICT, Oslo, Norway
[6] McGill University, Montréal, Canada

19 March 2012

978-2-87971-110-2

On the Formalisation of GeKo: a Generic Aspect Models Weaver

Max E. Kramer[12], Jacques Klein[2], Jim R. H. Steel[3], Brice Morin[5], Jörg Kienzle[6], Olivier Barais[4], and Jean-Marc Jézéquel[4]

[1] Karlsruhe Institute of Technology, Karlsruhe
[2] University of Luxembourg, Luxembourg
[3] The University of Queensland, Brisbane
[4] IRISA-INRIA, Triskell, Rennes
[5] SINTEF ICT, Oslo
[6] McGill University, Montréal

Abstract. This technical report presents the formalisation of the composition operator of GeKo, a Generic Aspect Models Weaver.

1 Introduction

The aspect-oriented paradigm has gained attention in the earlier steps of the software development life-cycle leading to the creation of numerous adhoc Aspect-Oriented Modeling (AOM) approaches. These approaches mainly focus on architecture diagrams, class diagrams, state-charts, scenarios or requirements, and generally propose domain specific composition mechanisms. Recently, some generic AOM approaches proposed to extend the notion of aspect to any domain specific modeling language or metamodel. In this trend, we present our generic weaver: GeKo. GeKo is a generic aspect-oriented model composition and weaving approach easily adaptable to any metamodel with no need to modify the domain metamodel or to generate domain specific frameworks. It is a tool-supported approach with a clear semantics of the different operators used to define the weaving. The formalisation of GeKo allows clearly identifying the sets of removed, added and altered elements. Based on this formalisation, GeKo yields non-ambiguous woven models.

After a brief presentation in Section 2 of GeKo in practice, Section 3 details the formalisation of the composition operator of GeKo.

2 Illustrating GeKo by Example

In this section, we introduce GeKo through an example of state chart weaving, but GeKo can be used to weave other models, such as class diagrams, sequence diagrams, and feature diagrams. An open-source prototype of GeKo and other examples can be found online[7]

[7] code.google.com/a/eclipselabs.org/p/geko-model-weaver

The statecharts presented in this section model two aspects of AspectOptima, an aspect-oriented framework implementing run-time support for different transaction models. AspectOptima has been proposed in [3,2] as an independent case study to evaluate aspect-oriented software development approaches, in particular aspect-oriented modeling techniques.

The goal of this technical report is not to present AspectOptima but to detail the formalisation of GeKo using examples of appropriate complexity. Therefore, we concentrate on the states of the *Context* entity. As shown in the base part of Fig. 1, a *Context* enters the `Ready` state when it is created. Upon reception of an `enterContext` message (sent by a process wishing to enter the context), a transition leads to the `Active` state. When the process leaves the context with a `leaveContext` message, the context transitions to the `Completed` state.

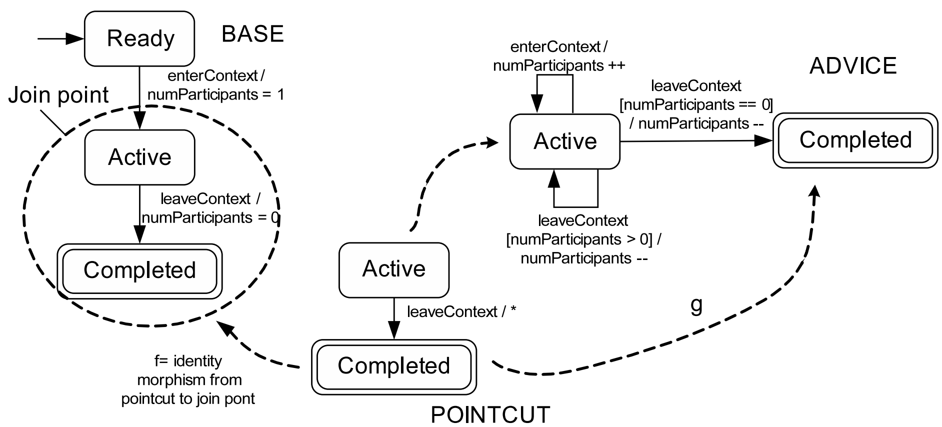

Fig. 1. Applying the Collaborative Aspect to a Context.

In order to implement transactions, contexts have to provide additional functionality. In AspectOptima, additional functionality is encapsulated in aspects. *Collaborative*, for example, is an aspect that can be applied to a context in order to allow the participation of *multiple* processes. As shown in the advice part of Fig. 1, *Collaborative* does not add any states to a context. Instead, it allows contexts to accept multiple `enterContext` messages by adding a transition labeled `enterContext` to and from the `Active` state. As a result, multiple processes are allowed to enter the context, instead of only a single one. The number of entering processes is counted in the local variable `numParticipants`. Likewise, a collaborative context also accepts multiple `leaveContext` messages, and only transitions to the `Completed` state when the last participating process has left the context.

The result of the weaving of the advice state diagram of *Collaborative* into the state diagram of the base *Context* is shown in Fig. 2.

The weaving process is two-phased:

1) The first step consists in the detection of the join points. This detection step uses the business logic integration platform Drools[8] and yields a mapping from the pointcut model to the base model for each detected join point. In Fig. 1, the detection yields a mapping f from the state *Active*, the state *Completed* and the transition *leaveContext* of the pointcut model to the state *Active*, the state *Completed* and the transition *leaveContext* of the base model.

2) The second step consists in the composition of the advice model with the base model at the level of the join points previously detected (for each join point the advice model is composed). The composition is based on the definition of a mapping between the pointcut and the base model (automatically obtained from the detection step), and a mapping between the pointcut and the advice model (specified by the user or automatically detected in unambiguous situations). These mappings are defined over the concrete syntax of models by linking model elements. These links are fully generic and do not use any domain-specific knowledge, so that we can define mappings for any domain metamodel. We simply check that the bound elements are compatible (same type). In Fig. 1, we specified a mapping g from the state *Active* and the state *Completed* of the pointcut model to respectively the state *Active* and the state *Completed* of the advice model. These mappings allow the identification of several sub-sets of objects in the base and advice models characterizing the objects of the base model which have to be kept, to be removed and to be replaced with those of the advice model. The formalisation of the composition is detailed in Section 3.

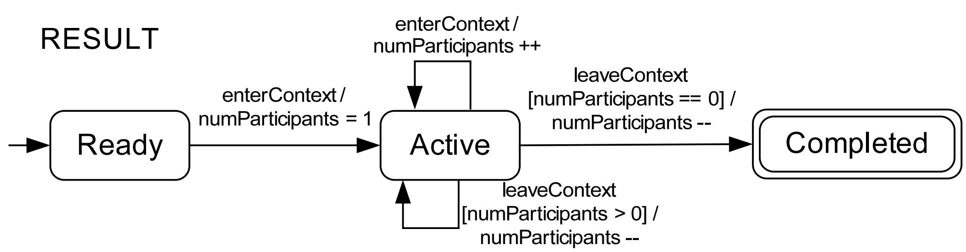

Fig. 2. Result of weaving the Collaborative Aspect into the Context Model.

The second aspect of AspectOptima that we present here for illustration purpose is *OutcomeAware*. An outcome-aware context is a context that has a *boolean* outcome associated with it, i.e., it can end in either *success* or *failure*. The essence of *OutcomeAware* is shown in the advice part of Fig. 3. While active, an outcome-aware context accepts `setOutcome` messages that allow processes to set the outcome of the context to either success or failure. In addition, the `Completed` state is replaced by two different final states: `Success` and `Failure`.

[8] jboss.org/drools

When the last process leaves the context, the `Success` or `Failure` state is entered depending on the current state of the local variable `outcome`.

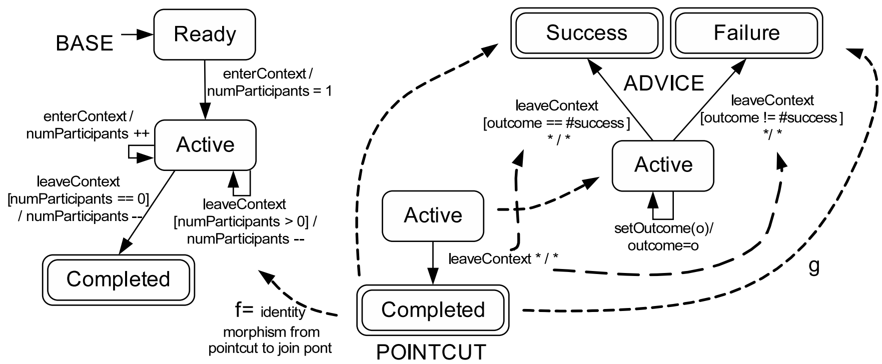

Fig. 3. Applying the OutcomeAware Aspect to a Collaborative Context

The result of the weaving of the *OutcomeAware* aspect model with a Collaborative Context (Base of Fig. 3) is shown in Fig. 4. The mapping (or morphism) g specified by the user associates the state *Active* of the pointcut to the state *Active* of the advice, and the state *Completed* of the pointcut with *both* states *Success* and *Failure* of the advice. That means the state *Completed* of the base will be replaced with both states *Success* and *Failure*. Note that the replacement implies that if a transition $t1$ leaves the state *Completed* of the base, after the weaving this transition leaves both states *Success* and *Failure*. Roughly speaking, the properties of the states *Success* and *Failure* are complemented by those of *Completed*. More details are given in Section 3.

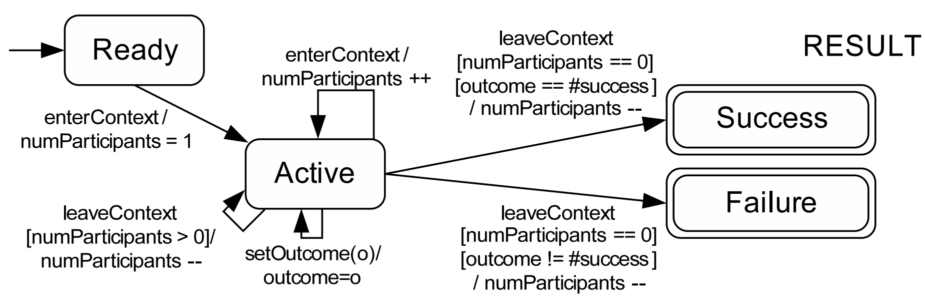

Fig. 4. Woven result of the OutcomeAware Aspect with a Collaborative Context.

3 Composition Formalisation

This section details and formalises GeKo, our generic weaver. This formalisation is based on the OMG standard EMOF and is implemented in Java using the Eclipse Modeling Framework (EMF). Both the formalisation and the implementation are independent from any particular domain metamodel. In other words, it is possible to apply our tool-supported approach to arbitrary models that conform to a well-defined metamodel conforming to EMOF respectively Ecore.

3.1 EMOF: Essential MetaObject Facilities

This subsection introduces Essential Meta-Object Facilities (EMOF) which is the basis for understanding the composition formalisation detailed in the next subsection. EMOF 2.0 is a metamodeling language designed to specify metamodels. It is a subset of the OMG standard MOF [5] providing the set of elements required to model object-oriented systems. The minimal set of EMOF constructs required for GeKo is presented in Fig. 5.

EMOF introduces the notion of *Object*, which is central to our formalisation. Every object has a class which describes its properties and operations. The *getMetaClass()* operation returns the Class that describes the object. The *container()* operation returns the containing parent object. It returns *void* if there is no parent object. The *equals(element)* operation determines if the element (an instance of the Element class) is equal to this Element instance. The *set(property, element)* operation sets the value of the property of the element. The *get(property)* operation returns the value of a property. It can be a List or a single value, depending on the multiplicity of the property.

The *isComposite* attribute under class *Property* returns true if the object is contained by a parent object (called *container*). Cyclic containment is not possible, *i.e.*, an object cannot contain one of its (possibly indirect) containers. Moreover, an object cannot be contained by more than one other object. To remove an object from a model, the object is removed from its container. The *getAllProperties()* operation (not shown in the figure) of the Class returns all the properties of instances of this Class along with the inherited properties. The attributes, *upper* and *lower*, of class *MultiplicityElement*, represent the multiplicities of the associations at the metamodel level. For example, "0..1" represents a lower bound "0" and an upper bound "1". If the upper bound is less than or equal to "1", then the property value is null or a single object; otherwise it is a collection of objects.

Note that a model conforming to a given metamodel that itself conforms to EMOF has a unique root element containing either directly or via composite properties all the elements of the model.

In this technical report, we assume that all the metamodels we use conform to EMOF. However, it is possible to adapt our formalisation to other M3 level meta-metamodels such as KM3 [1].

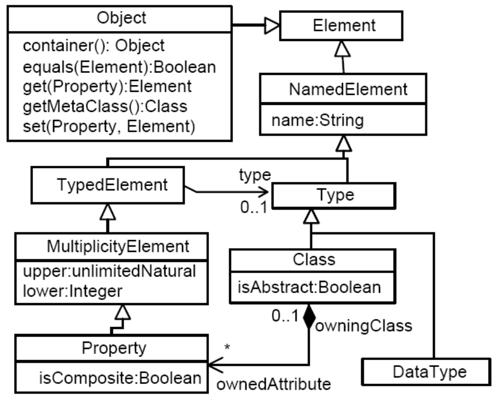

Fig. 5. Fragment of EMOF Classes Required for Composition.

3.2 Composition Formalisation

The main idea of our generic composition of two models *base* and *advice* at the level of a join point is the use of a third model called *pointcut* and two morphisms allowing the identification of the objects of *base* which have to be kept, to be removed and to be replaced with those of *advice*.

Definitions:
Let *base*, *pointcut* and *advice* be three models (defined by a set of objects). Let f and g be two morphisms such that:

1. f is a surjective morphism from *pointcut* to a subset $jp \subseteq base$ (jp being a join point), obtained from the pattern matching engine presented earlier. More precisely, f represents one match or one binding, i.e., one place where the aspect may be woven. Note that for each join point, the pattern matching engine yields a new morphism f.
2. g is a morphism from *pointcut* to *advice*.

The two morphisms partition the models *base* and *advice* into five sets:

i) The set $\mathcal{B}_{keep}$ representing the set of objects of *base* which have to be kept, i.e., which will appear in the target model unchanged. An object *obj* of *base* is in $\mathcal{B}_{keep}$ if there is no object obj' in *pointcut* such as $f(obj') = obj$. More formally,
$$\mathcal{B}_{keep} = \{obj \in base \mid \nexists\, obj' \in pointcut, f(obj') = obj\}.$$
ii) The set $\mathcal{B}_{-}$ representing the set of objects of *base* which have to be removed. An object *obj* of *base* is in $\mathcal{B}_{-}$ if there exists $obj' \in pointcut$ such that f maps obj' on obj *and* if there is no $obj'' \in advice$ such that g maps obj' on obj''. More formally,
$$\mathcal{B}_{-} = \{obj \in base \mid \exists\, obj' \in pointcut, \nexists\, obj'' \in advice, \\ f(obj') = obj \;\wedge\; g(obj') = obj''\}.$$

iii) The set $\mathcal{B}_\pm$ representing the set of objects of *base* which have to be replaced with elements of *advice*. An element *obj* of *base* is in $\mathcal{B}_\pm$ if there exists $obj' \in pointcut$ and $obj'' \in advice$ such that f maps obj' on obj and g maps obj' on obj''. More formally,

$$\mathcal{B}_\pm = \{obj \in base \mid \exists\, obj' \in pointcut, \exists\, obj'' \in advice,$$
$$f(obj') = obj \ \wedge \ g(obj') = obj''\}.$$

iv) In the same way, we define the set $\mathcal{A}_\pm$ representing the objects of *advice* which replace the objects of $\mathcal{B}_\pm$. An object $obj'' \in \mathcal{A}_\pm$ replaces an object $obj \in \mathcal{B}_\pm$ if and only if there exists an object *obj'* in *pointcut* such that $f(obj') = obj$ and $g(obj') = obj''$. Formally,

$$\mathcal{A}_\pm = \{obj \in advice \mid \exists\, obj' \in pointcut, \exists\, obj'' \in base,$$
$$g(obj') = obj \ \wedge \ f(obj') = obj''\}.$$

v) The set $\mathcal{A}_+$ representing the set of objects of *advice* which have to be added to *base*. An object *obj* of *advice* is in $\mathcal{A}_+$ if there is no $obj' \in pointcut$ such that g maps obj' on obj. More formally,

$$\mathcal{A}_+ = \{obj \in advice \mid \nexists\, obj' \in pointcut, g(obj') = obj\}.$$

Both morphisms also allow the definition of two sets in the *pointcut* model:

i) The set $\mathcal{P}_\pm$ containing the elements of the *pointcut* which 'correspond to' the common elements of both *base* and *advice*. Formally,

$$\mathcal{P}_\pm = \{obj \in pointcut \mid \exists\, obj' \in advice, \exists\, obj'' \in base,$$
$$f(obj) = obj'' \ \wedge \ g(obj) = obj'\}.$$

ii) The set $\mathcal{P}_-$ containing the elements of the *pointcut* which 'correspond to' the removed elements of the *base*. Formally,

$$\mathcal{P}_- = \{obj \in pointcut \mid \nexists\, obj' \in advice, g(obj) = obj'\}.$$

Note that $f(\mathcal{P}_\pm) = \mathcal{B}_\pm$, $f(\mathcal{P}_-) = \mathcal{B}_-$ and that $g(\mathcal{P}_\pm) = \mathcal{A}_\pm$, $g(\mathcal{P}_-) = \emptyset$

Finally, as explained in [4] for cases with multiple join points, GeKo distincts between advice elements that have to be reused for all join points (i.e., for all compositions), and advice elements that have to re-instantiated for each join point (i.e., for each composition). In practice, the elements labeled by "x^1" have to be reused for all join points. In this way, we can partition the sets $\mathcal{A}_\pm$ and $\mathcal{A}_+$ of elements of the advice as follows:
$\mathcal{A}_\pm = \mathcal{A}^1_\pm \cup \mathcal{A}^n_\pm$, where $\mathcal{A}^1_\pm$ represents objects that are reused for all join points whereas $\mathcal{A}^n_\pm$ represents objects that are re-instantiated for each join point; and $\mathcal{A}_+ = \mathcal{A}^1_+ \cup \mathcal{A}^n_+$, where $\mathcal{A}^1_+$ represents objects that are reused for all join points whereas $\mathcal{A}^n_+$ represents the objects that are instantiated for each join point.

We recall that the sets $\mathcal{A}^1_\pm$ and $\mathcal{A}^n_\pm$, and the sets $\mathcal{A}^1_+$ and $\mathcal{A}^n_+$ are disjoint, i.e., $\mathcal{A}^1_\pm \cap \mathcal{A}^n_\pm = \emptyset$, and $\mathcal{A}^1_+ \cap \mathcal{A}^n_+ = \emptyset$.

An example illustrating the different partitions of the base, pointcut and advice models is shown in Fig. 6. The letters $a, b, \ldots, l$ represent the base model element, the letters m, n, o, p represent the pointcut model elements, and the letters q, r, s, v, u represent the advice model elements. Let us suppose that the

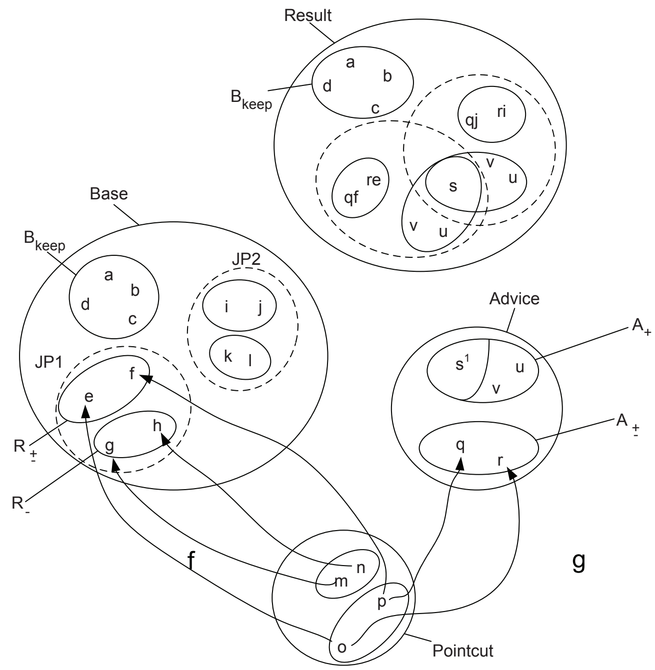

Fig. 6. Illustration of the Partitioning of the Base, Pointcut and Advice Models.

pointcut matches both join points $JP1$ and $JP2$. The morphism f from the pointcut to the second join point $JP2$ is not depicted, but we can easily infer that o and p are linked to i and j, and that m and n are linked to k and l. The result of the composition of the advice with the base for both join points is represented by the *Result* model.

Definition of the Composition:

We can now define the composition of two models at the level of a join point:

Definition 1 (Generic Composition). *Let base, pointcut and advice be three models. Let f and g be two morphisms as defined previously which partition the base and advice models. The composition of base with advice at the level of a join point is three-phased:*

1) result $= \mathcal{B}_{keep} \cup \mathcal{A}^1_+ \cup \mathcal{A}^1_\pm \uplus \mathcal{A}^n_+ \uplus \mathcal{A}^n_\pm$
2) The properties of the objects of result are updated;
3) The properties of the objects of result are cleaned.

First Phase:

In the first phase we keep the objects of $\mathcal{B}_{keep}$. Then, we add all the objects of the advice model: elements of the advice which are simply added ($\mathcal{A}^1_+$ and $\mathcal{A}^n_+$), and elements of the advice which replace existing base model elements ($\mathcal{A}^1_\pm$ and $\mathcal{A}^n_\pm$). However, the elements of $\mathcal{A}^1_+$ and $\mathcal{A}^1_\pm$ are added by using the traditional *union* operator $\cup$. This means that if an element e of $\mathcal{A}^1_+$ or $\mathcal{A}^1_\pm$ is already present in *result*, the element e is not added. The elements of $\mathcal{A}^n_+$ and $\mathcal{A}^n_\pm$ are added by using the disjoint union operator $\uplus$[9]. This means that if an element e of $\mathcal{A}^n_+$ or $\mathcal{A}^n_\pm$ is already present in *result*, the element e is duplicated.

Second Phase:

During the second phase the properties of the objects of *result* are updated.

I) For each $obj' \in \mathcal{A}_\pm$ (obj' is an object that replaces the object $obj \in \mathcal{B}_\pm$), the properties of obj' are modified according to those of obj as follows: let p be a property of obj':

I-1) if $p.upper > 1$ then p is complemented by the corresponding property of obj, i.e., $obj'.get(p) = obj'.get(p) \cup obj.get(p)$.

I-2) If $p.upper = 1$, a preliminary dialog step proposes to resolve the conflicts in the cases where the priority is given to the *base* and where several elements from the *pointcut* are mapped to a single element from the *advice*, i.e., when several elements of the *base* are replaced by a single element of the *advice*. An example illustrating this case is when both states a and b of the *base* FSM are replaced by a state c of the *advice* FSM. The property *name* of the class *State* is unique. If the priority is given to the *base*, the property *name* of the state c has to be updated. There are two possible solutions: a or b. The preliminary dialog step allows to choose between a or b, i.e., it is possible to define one of the *base* element as the priority element. In this case, all the unique properties (with an upper bound equal to one) of this priority element will be kept. It is also possible to define, for each unique priority, the object that has the priority.
In the remainder, we assume that *obj* is the priority *base* element chosen by the user, for the property p, in the case of a N to 1 mapping from *pointcut* to *advice*.
For $p.upper = 1$, let us denote $obj_b = obj.get(p)$ the object targeted[10] by the property p of obj, and $obj_a = obj'.get(p)$ the object targeted by the property p of obj'. Fig. 7 illustrates these notations. There are several possibilities:

I-2-1) If $obj_b == void$ or if $obj_b \in \mathcal{B}_-$, then the property p of obj' is unchanged (i.e., $obj'.get(p) = obj_a$);

I-2-2) If $obj_b \in \mathcal{B}_{keep}$, then

a) If $obj_a == void$ then the property p of obj' is updated with the property of obj, i.e., $obj'.get(p) = obj_b$.

[9] $\uplus$ is the disjoint union of two sets, i.e., an usual union operation where common elements of both sets are duplicated (cloned).

[10] A property p of a object obj targets an object obj'' if $obj'' \in obj.get(p)$.

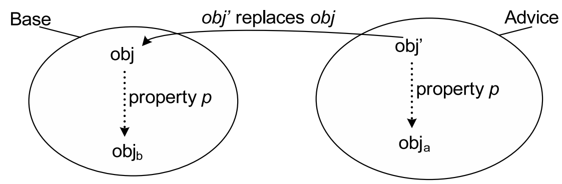

Fig. 7. Illustration of the notations when an object obj' of the advice replaces an object obj of the base.

b) If $obj_a \neq void$, there are two possible values for the property p of obj': either obj_a or obj_b. If the priority is given to the *advice* model, the property p is unchanged, i.e., $obj'.get(p) = obj_a$. If the priority is given to the *base* model, the property p is updated by the property of obj, i.e., $obj'.get(p) = obj_b$.

I-2-3) If $obj_b \in \mathcal{B}_{\pm}$, let us denote obj'_a the object of the *advice* model which replaces obj_b. Fig. 8 illustrates the notations used.

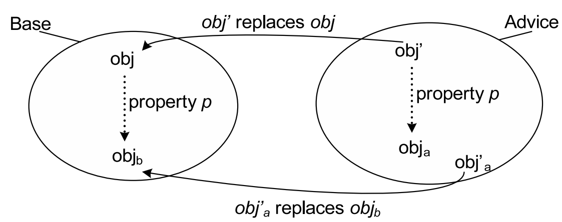

Fig. 8. Illustration of the notations used when an object obj' of the advice replaces an object obj of the base, and when the object targeted by a property p is replaced by an object obj'_a of the advice.

a) If $obj'_a == obj_a$ then the property p of obj' is unchanged.
b) If $obj_a == void$, then the property p of obj' is updated with obj'_a
c) If $obj_a \neq obj'_a$ and $obj_a \neq void$, we do not perform the weaving. Indeed, as illustrated by Fig. 8, the property p (with an upper bound equals to one) of the object obj in the base model is set twice: i) in the advice, obj' refers to obj_a via the property p and obj' replaces obj and ii) obj'_a replaces obj_b. In this case, we raise an exception and ask the designer to refactor the advice. He can either set obj_a to void or keep obj_a and not replace obj_b by obj'_a to avoid setting the property p twice.

II) Next, for each $obj' \in \mathcal{A}_{\pm}$ (obj' is an object that replaces the object $obj \in \mathcal{B}_{\pm}$), all the properties that targeted obj are updated. These properties should now target the corresponding element in $\mathcal{A}_{\pm}$, i.e., obj'. As a result, for every property p of an object obj'' that targeted obj, if $p.upper > 1$ then $obj''.get(p) = obj''.get(p) \cup obj' \setminus obj$, else $obj''.get(p) = obj'$.

III) It is not necessary to update the properties of an object obj' in $\mathcal{A}_+$ because it doesn't replace elements of the *base* model. It is simply added without link to the elements of the *base* model. Nevertheless, we have to consider each property p of an object obj that targets an element obj' of $\mathcal{A}_+$. In this case, the property p is updated according to the "nature" of obj. The model element obj necessary comes from the *advice* model, but:
   - III-1) If $obj \in \mathcal{A}_+$, the property p of obj which targets obj' is unchanged, i.e., $obj.get(p) = obj'$;
   - III-2) If $obj \in \mathcal{A}_\pm$, it is the same case as the item I).

Third Phase:
The third cleaning phase consists in the deletion of the references to objects removed in the first phase (objects of $\mathcal{B}_-$). Let us consider an object obj removed, an object $obj' \in result$ and a property p such as $obj \in obj'.get(p)$. Then, if $p.upper > 1$, we remove obj from the list $obj'.get(p)$. If the cardinality of p is 0..1, we remove obj from $obj'.get(p)$. Finally, if the cardinality of p is 1..1, we remove obj' from *result* to avoid the creation of a non-consistent model. We recursively apply the clean operation on *result* as long as there exist objects which have to be removed from *result*.

3.3 Composition Example Details

Let us illustrate this definition of generic composition by the simple example in Fig. 10 where we compose the Finite State Machines (FSM) *base* and *advice*. In this example, the priority is given to the *advice*, i.e., for a property with a cardinality 1..1 or 0..1, it is the value of the property of the advice which will be chosen instead of the value of the property of the base. The FSMs conform to the metamodel described in Fig. 9[11]. It shows that a FSM con-

[11] To simplify the example, we omitted to specify the property *output* of the transitions.

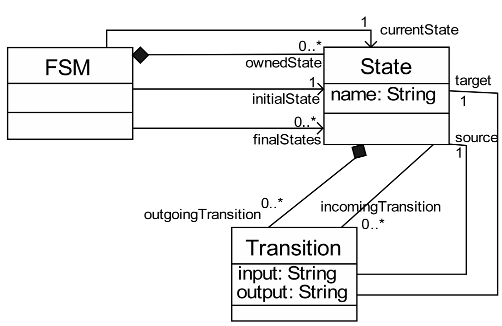

Fig. 9. Meta-model for Finite State Machines (FSM).

sists of named states that contain transitions from source to target states having an input and an output string. Furthermore, it displays that a FSM has exactly one current state, one initial state, and an arbitrary number of final states. The *base* FSM contains the objects[12]: $\{FSM : base$, $State : a^b, State : b, State : c, State : d, State : e, Transition : t_1, Transition : t_2, Transition : t_3, Transition : t_4, Transition : t_5, Transition : t_6\}$. The *advice* FSM contains the objects: $\{FSM : advice, State : a^{ad}, State : f$, $Transition : t_{12}\}$. The morphism f is the identity morphism. The morphism g associates respectively *State a*, *State b*, and *State c* of *pointcut* to $State : a^{ad}$, *State f*, and *State f* of *advice*.

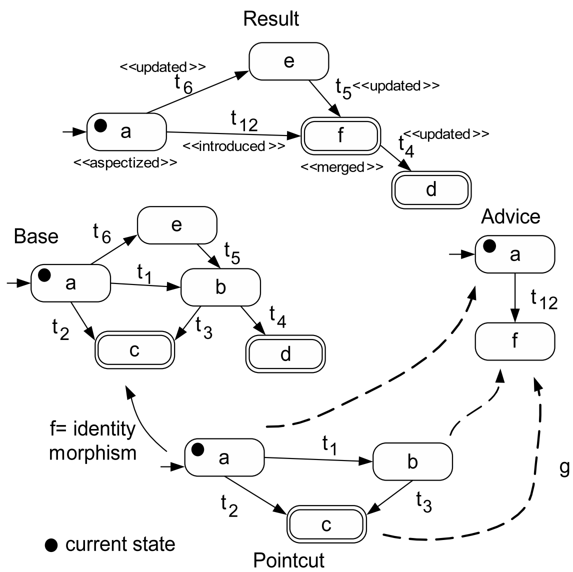

Fig. 10. Example of FSM composition

The morphisms allow the identification of the following sets:

- $\mathcal{B}_{keep} = \{State : d, State : e, Transition : t_4, Transition : t_5, Transition : t_6, FSM : base\}$.
- $\mathcal{B}_{-} = \{Transition : t_1, Transition : t_2, Transition : t_3\}$.
- $\mathcal{B}_{\pm} = \{State : a^b, State : b, State : c\}$.
- $\mathcal{A}_{\pm} = \{State : a^{ad}, State : f\}$.
- $\mathcal{A}_{+} = \{Transition : t_{12}\}$

Consequently, the result of the composition of *base* and *advice* is equal to $result = \{State : d, State : e, Transition : t_4, Transition : t_5, Transition : t_6, State : a^{ad}, State : f, Transition : t_{12}, FSM : base\}$, where the properties

[12] We use $.^b$ and $.^{ad}$ to distinguish the object $State : a^b$ from the *base* and the object $State : a^{ad}$ from the *advice*.

of $State : a^{ad}$ and $State : f$ have been updated, but also the properties of objects which target $State : a^{ad}$ and $State : f$. According to the FSM metamodel, the class $State$ is characterized by three properties: $outgoingTransition[0..*]$, $incomingTransition[0..*]$ and $name[1..1]$. The priority being given to the advice, for the property $name$, the name of the states $State : a^{ad}$ and $State : f$ are unchanged. For the properties with a cardinality higher than 1, we have:

$$\begin{aligned}
State : a.outgoingTransition &= State : a^{ad}.get(outgoingTransition) \cup \\
&\quad State : a^{b}.get(outgoingTransition)^{13} \\
State : a.outgoingTransition &= \{Transition : t_{12}\} \cup \{Transition : t_{6}\}
\end{aligned}$$

$$\begin{aligned}
State : a.incomingTransition &= State : a^{ad}.get(incomingTransition) \cup \\
&\quad State : a^{b}.get(incomingTransition) \\
State : a.incomingTransition &= \emptyset \cup \emptyset
\end{aligned}$$

$$\begin{aligned}
State : f.outgoingTransition &= State : b.get(outgoingTransition) \cup \\
&\quad State : c.get(outgoingTransition) \cup \\
&\quad State : f.get(outgoingTransition) \\
State : f.outgoingTransition &= \{Transition : t_{4}\} \cup \emptyset \cup \emptyset
\end{aligned}$$

$$\begin{aligned}
State : f.incomingTransition &= State : b.get(incomingTransition) \cup \\
&\quad State : c.get(incomingTransition) \cup \\
&\quad State : f.get(incomingTransition) \\
State : f.incomingTransition &= \{Transition : t_{5}\} \cup \emptyset \cup \{Transition : t_{12}\}
\end{aligned}$$

Furthermore, let us consider the properties of the objects which targeted the objects which have been replaced, i.e., $State : a$, $State : b$ and $State : c$:

Replaced Objects	Properties That Target Them
$State : a^{b}$	$FSM : base.\{initialState, currentState, ownedState\}$, $Transition : t_{6}.source$
$State : b$	$FSM : base.ownedState, Transition : t_{5}.target$
$State : c$	$FSM : base.\{finalState, ownedState\}$

After the composition, these properties target $State : a^{ad}$ instead of $State : a^{b}$, and $State : f$ instead of $State : b$ and $State : c$ (for instance, now $State : f$ is a $finalState$ instead of $State : c$).

In this example, the *clean* operation does not remove additional objects from properties because the three removed objects $Transition : t_{1}$, $Transition : t_{2}$, and $Transition : t_{3}$ were only targeted by properties of objects already removed ($State : a$, $State : b$, and $State : c$).

References

1. Frédéric Jouault and Jean Bézivin. Km3: A dsl for metamodel specification. In Roberto Gorrieri and Heike Wehrheim, editors, *Formal Methods for Open Object-Based Distributed Systems*, volume 4037 of *Lecture Notes in Computer Science*, pages 171–185. Springer Berlin / Heidelberg, 2006.
2. J. Kienzle, W. Al Abed, and J. Klein. Aspect-oriented multi-view modeling. In ACM, editor, *8th International Conference on Aspect Oriented Software Development (AOSD.09)*, Charlotteville, Virginia, USA, March 2009.
3. J. Kienzle and S. Gélineau. AO Challenge: Implementing the ACID Properties for Transactional Objects. In *Proceedings of the 5th International Conference on Aspect-Oriented Software Development - AOSD 2006, March 20 - 24, 2006*, pages 202 – 213. ACM Press, March 2006.
4. Brice Morin, Jacques Klein, Jörg Kienzle, and Jean-Marc Jézéquel. Flexible model element introduction policies for aspect-oriented modeling. In *Proceedings of the 13th international conference on Model driven engineering languages and systems: Part II*, MODELS'10, pages 63–77, Berlin, Heidelberg, 2010. Springer-Verlag.
5. OMG. Mof core specification , v2.0. OMG Document number formal/2006-01-01, 2006.