On the Formalisation of GeKo: a Generic Aspect Models Weaver

Max E. Kramer¹,², Jacques Klein², Jim R. H. Steel³, Brice Morin⁵, Jörg Kienzle⁶, Olivier Barais⁴, and Jean-Marc Jézéquel⁴

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. Kramer1,2, Jacques Klein2, Jim R. H. Steel3, Brice Morin5, Jörg Kienzle6, Olivier Barais4, and Jean-Marc Jézéquel4

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\footnote{jboss.org/drools} 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.

![RESULT](image)

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

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 $t_1$ 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. 3. Applying the OutcomeAware Aspect to a Collaborative Context

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].
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 $B_{\text{keep}}$ representing the set of objects of base which have to be kept, i.e., which will appear in the target model unchanged. An object $\text{obj}$ of base is in $B_{\text{keep}}$ if there is no object $\text{obj}'$ in pointcut such as $f(\text{obj}') = \text{obj}$. More formally,

$$B_{\text{keep}} = \{ \text{obj} \in \text{base} \mid \nexists \text{obj}' \in \text{pointcut}, f(\text{obj}') = \text{obj} \}.$$

ii) The set $B_{-}$ representing the set of objects of base which have to be removed. An object $\text{obj}$ of base is in $B_{-}$ if there exists $\text{obj}' \in \text{pointcut}$ such that $f$ maps $\text{obj}'$ on $\text{obj}$ and if there is no $\text{obj}'' \in \text{advice}$ such that $g$ maps $\text{obj}'$ on $\text{obj}''$. More formally,

$$B_{-} = \{ \text{obj} \in \text{base} \mid \exists \text{obj}' \in \text{pointcut}, \nexists \text{obj}'' \in \text{advice},
             f(\text{obj}') = \text{obj} \land g(\text{obj}') = \text{obj}'' \}.$$
iii) The set $B_\pm$ representing the set of objects of base which have to be replaced with elements of advice. An element obj of base is in $B_\pm$ 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,
\[
B_\pm = \{ \text{obj} \in \text{base} \mid \exists \text{obj}' \in \text{pointcut}, \exists \text{obj}'' \in \text{advice}, \ f(\text{obj}') = \text{obj} \land g(\text{obj}') = \text{obj}'' \}.
\]

iv) In the same way, we define the set $A_\pm$ representing the objects of advice which replace the objects of $B_\pm$. An object obj” $\in$ $A_\pm$ replaces an object obj $\in$ $B_\pm$ if and only if there exists an object obj’ in pointcut such that f maps obj’ $\in$ obj and g maps obj’ $\in$ obj”. Formally,
\[
A_\pm = \{ \text{obj} \in \text{advice} \mid \exists \text{obj}' \in \text{pointcut}, \exists \text{obj}'' \in \text{base}, \ g(\text{obj}') = \text{obj} \land f(\text{obj}') = \text{obj}'' \}.
\]

v) The set $A_+$ representing the set of objects of advice which have to be added to base. An object obj of advice is in $A_+$ if there is no obj’ $\in$ pointcut such that g maps obj’ on obj. More formally,
\[
A_+ = \{ \text{obj} \in \text{advice} \mid \nexists \text{obj}' \in \text{pointcut}, g(\text{obj}') = \text{obj} \}.
\]

Both morphisms also allow the definition of two sets in the pointcut model:
i) The set $P_\pm$ containing the elements of the pointcut which 'correspond to' the common elements of both base and advice. Formally,
\[
P_\pm = \{ \text{obj} \in \text{pointcut} \mid \exists \text{obj}' \in \text{advice}, \exists \text{obj}'' \in \text{base}, \ f(\text{obj}') = \text{obj}'' \land g(\text{obj}') = \text{obj}' \}.
\]

ii) The set $P_-$ containing the elements of the pointcut which 'correspond to' the removed elements of the base. Formally,
\[
P_- = \{ \text{obj} \in \text{pointcut} \mid \nexists \text{obj}' \in \text{advice}, g(\text{obj}) = \text{obj}' \}.
\]

Note that $f(P_\pm) = B_\pm$, $f(P_-) = B_-$ and that $g(P_\pm) = A_\pm$, $g(P_-) = \emptyset$

Finally, as explained in [4] for cases with multiple join points, GeKo distinguishes 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 “"^1" have to be reused for all join points. In this way, we can partition the sets $A_\pm$ and $A_+$ of elements of the advice as follows:

$A_\pm = A_\pm^1 \cup A_\pm^0$, where $A_\pm^1$ represents objects that are reused for all join points whereas $A_\pm^0$ represents objects that are re-instantiated for each join point; and $A_+ = A_+^1 \cup A_+^0$, where $A_+^1$ represents objects that are reused for all join points whereas $A_+^0$ represents the objects that are instantiated for each join point.

We recall that the sets $A_\pm^1$ and $A_\pm^0$, and the sets $A_+^1$ and $A_+^0$ are disjoint, i.e., $A_\pm^1 \cap A_\pm^0 = \emptyset$, and $A_+^1 \cap A_+^0 = \emptyset$.

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
pointcut matches both join points $JP_1$ and $JP_2$. The morphism $f$ from the pointcut to the second join point $JP_2$ 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) $result = B_{keep} \cup A^1_+ \cup A^1_- \cup A^n_+ \cup A^n_-$
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 $B_{keep}$. Then, we add all the objects of the advice model: elements of the advice which are simply added ($A^1_+$ and $A^n_+$),
and elements of the advice which replace existing base model elements ($A^1_+$ and $A^1_-\)$. However, the elements of $A^1_+$ and $A^1_-$ are added by using the traditional union operator $\cup$. This means that if an element $e$ of $A^1_+$ or $A^1_-$ is already present in $result$, the element $e$ is not added. The elements of $A^n_+$ and $A^n_-$ are added by using the disjoint union operator $\uplus\). This means that if an element $e$ of $A^n_+$ or $A^n_-$ 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.

1) For each $obj' \in A_{\pm}$ ($obj'$ is an object that replaces the object $obj \in B_{\pm}$), the properties of $obj'$ are modified according to those of $obj$ as follows: let $p$ be a property of $obj'$:
   1-1) if $p.upper > 1$ then $p$ is complemented by the corresponding property of $obj$, i.e., $obj'.get(p) = obj'.get(p) \cup obj.get(p)$.
   1-2) 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 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:

   1-2-1) If $obj_b == void$ or if $obj_b \in B_-$, then the property $p$ of $obj'$ is unchanged (i.e., $obj'.get(p) = obj_b$);
   1-2-2) If $obj_b \in B_{keep}$, then
      a) If $obj_a == void$ then the property $p$ of $obj'$ is updated with the property of $obj$, i.e., $obj'.get(p) = obj_b$.

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

A property $p$ of a object $obj$ targets an object $obj''$ if $obj'' \in obj.get(p)$.
b) If \( obj_a \neq \text{void} \), 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 \( obj_b \in B_{\pm} \), let us denote \( obj'_a \) the object of the advice model which replaces \( obj_b \). Fig. 8 illustrates the notations used.

\[ \text{Fig. 7. Illustration of the notations when an object } \text{obj}' \text{ of the advice replaces an object } \text{obj} \text{ of the base.} \]

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

a) If \( obj'_a == obj_a \) then the property \( p \) of \( obj' \) is unchanged.
b) If \( obj_a == \text{void} \), then the property \( p \) of \( obj' \) is updated with \( obj'_a \).
c) If \( obj_a \neq obj'_a \) and \( obj_a \neq \text{void} \), 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 \( obj' \in A_{\pm} \) (\( obj' \) is an object that replaces the object \( obj \in B_{\pm} \)), all the properties that targeted \( obj \) are updated. These properties should now target the corresponding element in \( A_{\pm} \), 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 $A_+$ 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 $A_+$. 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 $obj \in A_+$, the property $p$ of $obj$ which targets $obj'$ is unchanged, i.e., $obj$.get($p$) = $obj'$;

III-2) If $obj \in A_{\pm}$, 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 $B_-$). 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. It shows that a FSM con-

\[\text{To simplify the example, we omitted to specify the property output of the transitions.}\]

\[\text{Fig. 9. Meta-model for Finite State Machines (FSM).}\]
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: \{FSM : base, State : a\textsuperscript{b}, State : b, State : c, State : d, State : e, Transition : \textit{t}_1, Transition : \textit{t}_2, Transition : \textit{t}_3, Transition : \textit{t}_4, Transition : \textit{t}_5, Transition : \textit{t}_6\}. The advice FSM contains the objects: \{FSM : advice, State : a\textsuperscript{ad}, State : f, Transition : \textit{t}_{12}\}. The morphism \(f\) is the identity morphism. The morphism \(g\) associates respectively \textit{State a, State b, and State c of pointcut to State : a\textsuperscript{ad}, State f, and State f of advice.}

Fig. 10. Example of FSM composition

The morphisms allow the identification of the following sets:

- \(B_{\text{keep}} = \{\text{State : d, State : e, Transition : \textit{t}_4, Transition : \textit{t}_5, Transition : \textit{t}_6, FSM : base}\}\.
- \(B_{\text{ex}} = \{\text{Transition : \textit{t}_1, Transition : \textit{t}_2, Transition : \textit{t}_3}\}\.
- \(B_{\text{c}} = \{\text{State : a\textsuperscript{b}, State : b, State : c}\}\.
- \(A_{\text{c}} = \{\text{State : a\textsuperscript{ad}, State : f}\}\.
- \(A_{\text{+}} = \{\text{Transition : \textit{t}_{12}}\}\.

Consequently, the result of the composition of \textit{base} and \textit{advice} is equal to \(\text{result} = \{\text{State : d, State : e, Transition : \textit{t}_4, Transition : \textit{t}_5, Transition : \textit{t}_6, State : a\textsuperscript{ad}, State : f, Transition : \textit{t}_{12}, FSM : base}\}\), where the properties

\[\text{We use } \textit{b and } \textit{ad to distinguish the object State : a\textsuperscript{b} from the base and the object State : a\textsuperscript{ad} from the advice.}\]
of State: :ad and State: :f have been updated, but also the properties of objects which target State: :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: :ad and State: :f are unchanged. For the properties with a cardinality higher than 1, we have:

\[
\text{State: :ad.outgoingTransition} = \text{State: :ad.get(outgoingTransition)} \cup \text{State: :b.get(outgoingTransition)}^1
\]
\[
\text{State: :ad.outgoingTransition} = \{\text{Transition: :t12}\} \cup \{\text{Transition: :t6}\}
\]

\[
\text{State: :ad.incomingTransition} = \text{State: :ad.get(incomingTransition)} \cup \text{State: :b.get(incomingTransition)}
\]
\[
\text{State: :ad.incomingTransition} = \emptyset \cup \emptyset
\]

\[
\text{State: :f.outgoingTransition} = \text{State: :b.get(outgoingTransition)} \cup \text{State: :c.get(outgoingTransition)} \cup \text{State: :f.get(outgoingTransition)}
\]
\[
\text{State: :f.outgoingTransition} = \{\text{Transition: :t4}\} \cup \emptyset \cup \emptyset
\]

\[
\text{State: :f.incomingTransition} = \text{State: :b.get(incomingTransition)} \cup \text{State: :c.get(incomingTransition)} \cup \text{State: :f.get(incomingTransition)}
\]
\[
\text{State: :f.incomingTransition} = \{\text{Transition: :t5}\} \cup \emptyset \cup \{\text{Transition: :t12}\}
\]

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:

<table>
<thead>
<tr>
<th>Replaced Objects</th>
<th>Properties That Target Them</th>
</tr>
</thead>
<tbody>
<tr>
<td>State: :c</td>
<td>FSM: base.{finalState, ownedState}</td>
</tr>
</tbody>
</table>


In this example, the clean operation does not remove additional objects from properties because the three removed objects Transition: :t1, Transition: :t2, and Transition: :t3 were only targeted by properties of objects already removed (State: :a, State: :b, and State: :c).
References


