Refactoring UML models

Gerson Sunyé, Damien Pollet, Yves Le Traon, Jean-Marc Jézéquel

Refactoring UML Models

Gerson Sunyé, Damien Pollet, Yves Le Traon, and Jean-Marc Jézéquel

IRISA, Campus de Beaulieu, F-35042 Rennes Cedex, France
email: sunye,dpollet,yletraon,jezequel@irisa.fr

Abstract Software developers spend most of their time modifying and maintaining existing products. This is because systems, and consequently their design, are in perpetual evolution before they die. Nevertheless, dealing with this evolution is a complex task. Before evolving a system, structural modifications are often required. The goal of this kind of modification is to make certain elements more extensible, permitting the addition of new features. However, designers are seldom able to evaluate the impact, on the whole model, of a single modification. That is, they cannot precisely verify if a change modifies the behavior of the modeled system. A possible solution for this problem is to provide designers with a set of basic transformations, which can ensure behavior preservation. These transformations, also known as refactorings, can then be used, step by step, to improve the design of the system. In this paper we present a set of refactorings and explain how they can be designed so as to preserve the behavior of a UML model. Some of these refactorings are illustrated with examples.

1 Introduction

The activity of software design is not limited to the creation of new applications from scratch. Very often software designers start from an existing application and have to modify its behavior and functionality. In recent years, it has been widely acknowledged as a good practice to divide this evolution into two distinct steps:

1. Without introducing any new behavior on the conceptual level, re-structure the software design to improve quality factors such as maintenability, efficiency, etc.
2. Taking advantage of this "better" design, modify the software behavior.

This first step has been called *refactoring* [**?**], and is now seen as an essential activity during software development and maintenance.

By definition, refactorings should be behavior-preserving transformations of an application. But one of the problems faced by designers is that it is often hard to measure the actual impact of modifications on the various design views, as well as on the implementation code.

This is particularly true for the Unified Modeling Language, with its various structural and dynamic views, which can share many modeling elements. For

instance, when a method is removed from a class diagram, it is often difficult to establish, at first glance, what is the impact on sequence and activities diagrams, collaborations, statecharts, OCL constraints, etc.

Still, the UML also has a primordial advantage in comparison with other design languages: its syntax is precisely defined by a metamodel, where the integration of the different views is given meaning. Therefore, the metamodel can be used to control the impact of a modification, which is essential when it should preserve the initial behavior an application.

The contribution of this paper is to show that refactorings can be defined for UML in such a way that their behavior-preserving properties are guaranteed, based on OCL constraints at the meta-model level. In Section 2 we first recall the motivation for such behavior-preserving transformations for the UML, and then give two concrete examples of refactorings, along with an empirical justification of their behavior-preserving properties. We then try to go further by formalizing refactorings using the OCL at the meta-model level to specify behavior-preserving transformations. For the sake of conciseness, we restrict the scope of this article to the refactoring of class diagrams (Sect. 3) and statecharts (Sect. 4). Finally, we conclude on the perspectives of this approach, most notably tool support that is prototyped in the context of our UML general purpose transformation framework called Umlaut.

2 Refactoring in a Nutshell

2.1 Motivation

Brant and Roberts [**?**] present refactorings as an essential tool for handling software evolution. They point out that "traditional" development methods, based on the waterfall life cycle, consider the maintenance of a software as the last phase of its life cycle and do not take into account the evolution of software. They also remark that some other methods, usually based on the spiral life cycle, such as Rapid Prototyping, Joint Application Development and more recently Extreme Programming [**?**], have better support for software evolution, and therefore for refactorings. These methods encourage the use of fourth generation languages and integrated developing environments, and thus are more appropriate for refactorings. Since UML seems to be closer to the first family of methods and tools than to the second one, one could expect the integration of refactorings in UML not to be worthwhile.

Despite this apparent methodological incompatibility, we still believe that refactorings can be integrated into UML tools. Methods have changed since the first observations of Fred Brooks [**?**], and the boundary between these two families of methods and tools is now less distinct. Recent methods, e.g. Catalysis [**?**] which uses UML as a notation language, take into account software evolution and thus design evolution. Additionally, since some UML tools, e.g. Rose, have some facilities for creating design models from application source code, refactorings could be used to modify this code and improve the design of existing applications.

The forthcoming Action Semantics [**?**] (AS) is an important issue for the integration of refactorings into UML tools. More precisely, the AS will allow UML to fully represent application behavior. Once UML tools could control the impact of modifications, they could propose a set of small behavior-preserving transformations, which could be combined to accomplish important design refactorings, as for instance, apply design patterns [**?**,**?**,**?**]. These transformations could be performed inside a *behavior-preservation* mode, where designers could graphically perform design improvements without generating unexpected results on the various UML views. One may accurately argue that OCL constraints may also be used to specify behavior of applications. Whilst this is true, the use the OCL is also more complex, since the integration of the OCL syntax into the UML metamodel is not yet precisely defined.

Before presenting these transformations in details and to better explain our motivation, we introduce 2 examples where refactorings are used to improve the design of existing applications.

2.2 Class Diagram Example

The class diagram given in Fig. 1 is a simple model of a graphical hierarchy for a vector graphics program. Graphics are constituted of geometric Primitives and subgraphs; they have a method to be displayed. Primitives have a matrix attribute representing how they are scaled, rotated or translated in the global coordinate system.

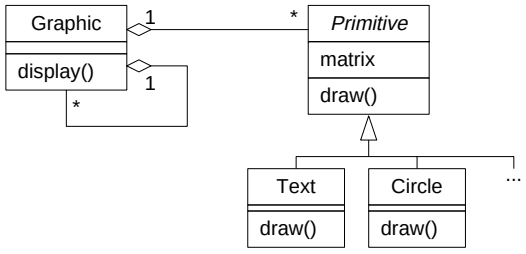

Figure 1. Initial class diagram.

This model has some design flaws; for instance, as Primitives have no inheritance relation with Graphics, they must be treated differently, thus making the code unnecessarily complex. Fortunately, the *Composite* design pattern adresses this type of problem, where a structure is composed of basic objects that can be recursively grouped in a part-whole hierarchy. We will therefore introduce this pattern in the model through the following steps, leading to the diagram presented in Fig. 2:

1. Renaming the Graphic class to Group;

2. Adding an abstract superclass named Graphic to Group.
3. Making the class Primitive a subclass of Graphic.
4. Merging the Group-Group and Group-Primitive aggregations into Group-Graphic.
5. Finally, we can move relevant methods and attributes up to Graphic.

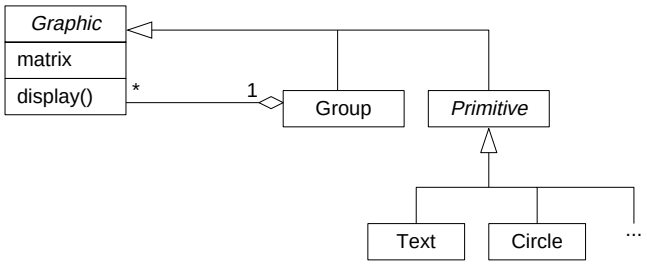

Figure 2. Restructured class diagram.

We need to justify why the behavior preservation condition holds for these model transformations:

- Renaming of a model element does not change anything to the model behavior, provided the new name is legal (i.e. it does not already exist in the model).
- The added abstract superclass has no attributes or methods. It is an "empty" model element; its addition has no effect on the model.
- Creating a generalization between two classes does not introduce new behavior, provided no conflict (due to multiple inheritance, for instance) is introduced; in our case, Primitive had no superclass and Graphic is empty.
- Merging two associations is only allowed when these two associations are disjoint (they do not own the same objects), when the methods invoked through these associations have the same signature, and when the invocation through an association is always followed by an invocation through the other.
- Finally, moved methods or attributes to the superclass will simply be inherited afterwards (overriding is not modelled).

While most of these transformations – namely element renaming and the addition of a superclass – do not have an impact on other views, the merge of two associations may require changes on collaborations and object diagrams.

2.3 Statechart Example

Refactorings can also be used to improve the design of statecharts. However, as state diagrams do not simply model the system structure but its behavior, their transformation raises some difficulties. Figure 3 shows a state diagram for

a simple telephone object. It is quite messy: since one can hang up at any time during communication, transitions have been drawn to the Idle state from every other state in the diagram.

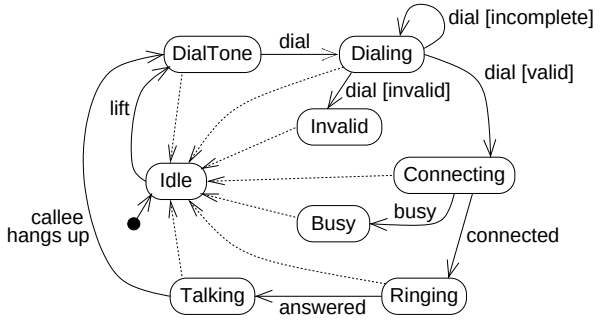

Figure 3. Initial phone state diagram (dotted transitions are triggered when the caller hangs up).

In order to improve understandability, we group the states modeling the behavior of the phone when it is in use into a composite state, thus segregating the Idle state and allowing the use of high-level transitions.

To obtain the result shown in Fig. 4, four refactoring steps are needed:

1. Create a composite superstate, named Active, surrounding the whole current diagram.
2. Move Idle and the initial pseudostate out of Active.
3. Merge the "hang up" transitions into a transition leaving the boundary of Active.
4. Finally, split the "lift" transition into a transition from Idle to the boundary of Active and a default pseudostate/transition targeting DialTone.

These are the justifications for the previous transformations:

- Creating a surrounding state is trivially behavior-preserving.
- Moving the Idle state out is legal here: Active has no entry or exit actions, and so the execution order of existing actions is unchanged.
- Transitions exiting Active can be folded to a toplevel transition since they are equivalent (they hold the same label and target the same state).
- The replacement of the "lift" transition by a toplevel one is possible given that there is no other toplevel transition entering Active.

3 Refactoring Class Diagrams

The refactorings presented by W. Opdyke in his PhD thesis [**?**], which were later perfected and implemented by D. Roberts [**?**], as well as the restructuring

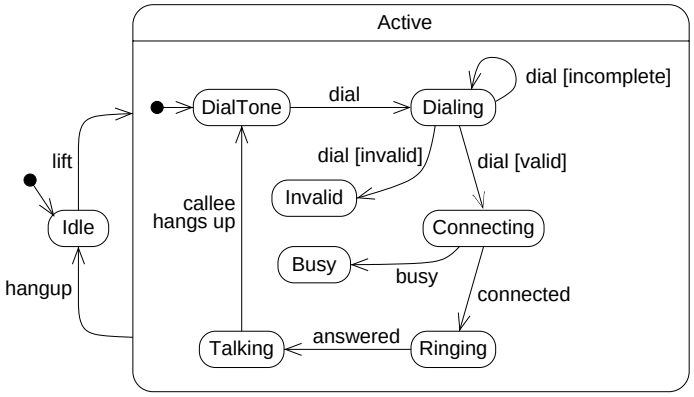

Figure 4. Final phone state diagram.

transformations presented by other research efforts [**?**,**?**,**?**,**?**] apply essentially to three concepts: class, method and variable. Therefore, when we started the transpositions of existing refactorings to UML, we began with class diagrams.

The refactorings presented here can be summarized in five basic operations: *addition, removal, move, generalization* and *specialization* of modeling elements. The two last actions use the generalization relationship to transfer elements up and down a class hierarchy.

Most part of the modeling elements composing the class diagram may have a direct connection to the elements of other views. Therefore, some of the refactorings that apply to class diagrams may have an impact on different UML views.

3.1 Add, Remove and Move

The *Addition* of features (attributes and methods) and associations to a class can be done when the new feature (or association) does not have the same signature as any other features owned by the class or by its parents. The *Removal* of associations and features can only be done when these elements are not referenced in the whole model. A method, for instance, may be referenced inside an interaction diagram (linked to messages and stimuli) and statecharts (linked to actions and events).

Adding and removing classes can be particularly interesting when the inheritance hierarchy is taken into account. The *Insert Generalizable Element* refactoring replaces a generalization between two elements with two other generalizations, having a new element between them. The inserted element must have the same type as the two initial elements and must not introduce new behavior. The *Remove Generalizable Element* does the contrary, it removes an element whithout defined behavior and links its subclasses directly to its superclasses. The element must not be referenced directly and indirectly (by the way of instances, features, etc.) in other diagrams.

The *Move* is used to transfer a method from a class to another, and create a forwarder method in the former. The constraints required by this transformation are rather complex. Initially, it implies the existence of an association, possibly inherited, between both classes. This association must be binary and its association ends must be both navigable, instance-level and have a multiplicity of 1. These constraints are different from those defined by D. Roberts [**?**] for a similar refactoring, where the association was not needed (and could not be identified since Smalltalk is dynamically typed). In his transformation the the transfered method gets an additional parameter, an instance of the original classifier. However, this additional parameter is not needed for a 1:1 multiplicity.

Although this transfer could be applied to any operation, some other constraints must be specified, in order to keep it coherent. The body of the concerned operation must not make references to attributes and only navigate through an association to the target classifier. After the transformation, messages that are sent to *self* are replaced by messages sent through an association. The references to the target classifier are replaced by references to self. This transformation requires the use of the Action Semantics, which can be used to find out which attributes and methods are used inside the body of the operation.

3.2 Generalization and Specialization

The *Generalization* refactoring can be applied to elements owned by classes, such as attributes, methods, operations, association ends and statecharts. It consists in the integration of two or more elements into a single one which is transfered to a common superclass. Since private features are not accessible within the subclasses, they can not be moved.

This transformation implies that all direct subclasses of the superclass own an equivalent element. Whilst the equivalence of attributes, association ends and operations can be verified by a structural comparison, the one of methods and statecharts is rather complex.

The *Specialization* refactoring is the exact opposite of *Generalization*, it consists in sending an element to all direct subclasses of its owner. In an informal way, the behavior is preserved if its owner class is not its *reference context*. The reference context is the class of the object to which a message or an attribute read/write is sent. In a general manner the reference context is the class, or any of its subclasses, that owns the attribute or the method. For instance, the reference context for the display() method (2) is potentially instances of Graphic or of any of its subclasses. If somewhere in the whole model a message calling this method is sent to an instance of Graphic, then the present refactoring can not be applied.

The reference context of attributes is obtained inside object diagrams (if the body of methods is expressed with the Action Semantics, it can be obtained by an analysis of read and write attribute actions). Object diagrams and collaborations can be used to obtain the reference context of association ends. The reference context of methods may be obtained inside interaction diagrams and statecharts.

Since multiple-inheritance is allowed in UML, we must verify whether the classes that would receive the feature do not have common subclasses (i.e. if a repeated-inheritance exists). If a common subclass exists, this means that after the transformation it would inherit two equivalent features, which would be a conceptual error.

4 Refactoring Statecharts

Statecharts make the behavior of classifiers and methods explicit and provide an interesting context for refactorings. Since this kind of refactorings is not considered in the research efforts previously cited, we will detail the constraints that must be satisfied before and after each transformation to ensure behavior preservation. Since our approach concerns the UML, we use the OCL [**?**], at the metamodel level, to specify these constraints

For the sake of simplicity, we will not enter into the details of how each refactoring accomplishes its intent (by the creation of objects and links), but only describe with the OCL what should be verified before and after the transformation. Indeed, the understanding of these meta-level OCL constraints requires some knowledge of the UML metamodel.

Most of the complexity encountered when defining these transformations comes from the activation of actions attached to states, such as *do*, *entry* and *exit* actions. The first one is executed while its state is active. The *entry* action is executed when a state is activated. In the particular case of a composite, its entry action is executed before the entry action of its substates. However, this action is only executed when a transition crosses the border of the composite. The *exit* action is executed when a state is exited. In the particular case of a composite, its exit action is executed after the exit action of its substates.

4.1 State

Fold Incoming/Outgoing Actions These transformations replace a set of actions attached to Transitions, leaving from or leading to a state, by an exit or entry action attached to this state. They imply that an equivalent action is attached to all incoming or outgoing Transitions. Moreover, the source and the target state of each transition must have the same container, i.e. the transition must not cross the boundary of a composite, which could fire an entry or an exit action. Essentially, two actions are equivalent when they call the same operation, instantiate the same class, send the same signal, etc. In the case of action sequences, they are equivalent when they are composed of a basic equivalent actions.

Moreover, the concerned state must not own an entry or an exit action. The pre and post conditions of the *Fold Incoming Actions* transformation are presented below. Since the *Fold Outgoing Actions* is quite similar, it is not presented here.

Fold Incoming Actions

```
State :: foldIncoming
pre:
    self .entry→isEmpty() and
    self .incoming. effect →forAll(a,b:Action|
       a <>b implies a.isEquivalentTo(b)) and
    self .incoming.source→forAll(s:State| s . container = self .container)
post:
    self .entry→notEmpty() and self.incoming.effect→isEmpty() and
    self .incoming. effect @pre→forAll(a :Action| a.isEquivalentTo(self.entry))
```

OCL pre and post conditions of subsequent transformations are listed in the appendices of the paper, starting at p. 11.

Unfold Entry/Exit Action These transformations are symmetrical to the previous ones. They replace an entry or an exit action attached to a state by a set of actions attached to all transitions incoming from or outgoing to this state. The concerned transitions must have no actions attached to them and must not cross the boundary of a composite.

These transformations, as well as those presented above, could be performed on transitions between states having different containers. In these cases, all composite states that are crossed by the transition should not have an exit action (if the transition leaves the composite) or an entry action (if the transition enters the composite).

Group States Groups states into a new composite state. The transformation applies to a set of at least one state, belonging to the same *container*. It takes a name as parameter. The *container* is always a composite state, since according to the UML well-formedness rules for statecharts, the top of any state machine is always a single composite state. The container should not contain a state having the same name as the new composite.

Once this transformation is performed, the state machine contains a new composite state, which contains all states of the collection. This new state must not have incoming, outgoing nor internal transitions, nor any do, entry, or exit actions. This is the refactoring we used during the first step in the phone example (Sect. 2.3).

4.2 Composite State

Fold Outgoing Transitions This transformations replaces a set of transitions, leaving from components of a composite state (one transition for each component) and going to the same state, by a single transition leaving from the composite state and going to this state, as was done in step 3 of the phone example. The actions attached to the transitions must be equivalent. The concerned target state must be specified.

Unfold Outgoing Transition Replaces a transition, leaving from a composite state, by a set of transitions, leaving from all substates (one transition for each substate) going to the same target. All these transitions must own a equivalent action and event. In any case, the order of (entry/exit/transition) actions execution is not changed.

Move State into Composite The insertion of a state into a composite state is a rather complex transformation. Several constraints must be verified before and after its execution. Since the transformation must not add new transitions to the state, for each outgoing transition leaving the composite, the state must have an equivalent one. transitions incoming from other states are indirectly bound to a substate, and thus do not affect the state. The transformation must ensure that the state will not have two equivalent transitions leaving from it. If the composite has a *do* action, then the state must have an equivalent one. After the transformation, the action contained by the state must be removed.

If some of the incoming transition to the state comes from the composite outside, the composite must not have an entry action. But, if the composite has an entry action, then the transitions of the state going to the sub-states must not have an attached action. After the transformation, these transitions receive a copy of the entry action.

If a target of a transition coming from the state is not a substate, then the composite must not have an exit action. If the composite has an exit action, then the transitions of the state coming from sub-states must not have an attached action. After the transformation, these transitions receive a copy of the exit action.

Move State out of Composite Moving a substate out of its composite is also a complex task that is worth some clarification. This refactoring was used in a simple situation with the Idle in the phone example. The substate may have *inner* and *outer* transitions, i.e. transitions with states that are inside or outside the same composite, respectively.

Inner transitions are a problem when the composite has exit and entry actions, since these actions are not activated by this kind of transition and will be activated when the substate is extracted from the composite. In these cases, the extraction can only be done if the inner transitions own an action, which is equivalent to the exit (for incoming) or to the entry action (for outgoing). After the transformation, the actions attached to these transitions must be removed.

The existence of entry and exit actions is also a problem for outer transitions, which cross the composite border and activate these actions, because they will no longer occur after the transformation. The solution used in these cases is simple, a new action is attached to each outer transition. These actions are equivalent to the entry or exit actions, for incoming and outgoing transitions, respectively.

If the substate is linked to the initial pseudostate of the composite — which means that the composite incoming transitions are actually incoming transitions

of the substate — then, after the transformation, these transitions must be transferred to the extracted state.

5 Conclusion

We have presented an initial set of design refactorings, which are transformations used to improve the design of object-oriented applications, without adding new functionalities. Adapting code refactorings to the design level as expressed in UML has proved itself a very interesting endeavor, far more complex than we thought initially. The search for some UML specific refactorings has been somehow frustrating, specially when we wanted transformations to have an impact on different UML views.

For instance, we wanted the activity diagram to be used as the starting place for the *Move Operation* refactoring. Indeed, this diagram, which is used to represent the behavior of a particular functionality, can be split into several *swimlanes* which seem to represent different classes. Thus, moving an activity from a swimlane to another could be interpreted as a *Move Operation*. Unfortunately, this is not possible, since the current syntax does not allow swimlanes to be directly linked to classes: swimlanes are just labels in the underlying model.

Moreover, the abstract syntax of the OCL is not yet precisely specified. Consequently, we are not able to define some OCL-based refactorings, neither to analyze the contents of a constraint and use this information to improve the definition of some refactorings. This might change in the future since an abstract syntax is curently proposed for normalization at the OMG.

As a perspective to this work, we foresee an extensive use of the action Semantics to make design models more precise, which would pave the way for more secure (i.e. proven) refactorings, that would also be, too a large extent, programming language independent. Our initial set of refactorings could then be widely expanded, and directly supported in standard UML tools.

References

1. Updated joint initial submission against the action semantics for uml rfp.
2. Kent Beck. *Extreme Programming Explained: Embracing Change*. Addison-Wesley, 1999.
3. Paul Bergstein. Maintainance of object-oriented systems during structural schema evolution. *TAPOS*, 3(3):185–212, 1997.
4. John Brant and Don Roberts. Refactoring techniques and tools (Plenary talk). In *Smalltalk Solutions*, New York, NY, 1999.
5. F. P. Brooks. *The Mitical Man-Month: Essays on Software Engineering*. Addison-Wesley, Reading, Mass, 1982.
6. Eduardo Casais. *Managing Evolution in Object Oriented Environments: An Algorithmic Approach*. Phd thesis, University of Geneva, 1991.
7. Mel Cinnéide and Paddy Nixon. A methodology for the automated introduction of design patterns. In *International Conference on Software Maintenance*, Oxford, 1999.

8. Desmond D'Souza and Alan Wills. *Objects, Components and Frameworks With UML: The Catalysis Approach*. Addison-Wesley, 1998.
9. W. Griswold. Program restructuring as an aid to software maintenance, 1991.
10. Walter Hursch. *Maintaining Consistency and Behavior of Object-Oriented Systems during Evolution*. Phd thesis, Northeastern University, June 1995.
11. Anneke Kleppe, Jos Warmer, and Steve Cook. Informal formality? the Object Constraint Language and its application in the UML metamodel. In Jean Bézivin and Pierre-Alain Muller, editors, *The Unified Modeling Language, UML'98 - Beyond the Notation. First International Workshop, Mulhouse, France, June 1998*, pages 127–136, 1998.
12. William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois, Urbana-Champaign, 1992. Tech. Report UIUCDCS-R-92-1759.
13. Donald Roberts. *Practical Analysis for Refactoring*. PhD thesis, University of Illinois, 1999.
14. Donald Roberts, J. Brant, and Ralph Johnson. A refactoring tool for Smalltalk. *Theory and Practice of Object Systems*, 3(4), 1997.

A Appendix: Statechart Refactorings

A.1 Unfold Exit Action

State :: unfoldExit
pre:
self . exit→**notEmpty**() **and** self.outgoing.effect→**isEmpty**()
self .outgoing.target→**forAll**(s:State| s . container = self . container)
post:
self . exit→**isEmpty**() **and**
self .outgoing. effect →**forAll**(a:Action| a.isEquivalentTo(self . exit **@pre**))

The *Unfold Entry Action* refactoring is very similar to this one, so its OCL constraints are not detailed here.

A.2 Group States

Collection→groupStates(**name** : Name)
pre:
self →**notEmpty**() **and**
self →**forAll**(each | each.**oclIsKindOf**(State)) **and**
let coll = self . container→**asSet**() in coll→**size** = 1 **and**
coll →**first** (). subvertex→**select**(each:State|each.**name**=**name**)→**isEmpty**()
post:
let coll = self . container→**asSet**() in (coll →**size** = 1 **and**
let compositeState = coll→**first** () in (compositeState.**oclIsNew and**
-- *no internal actions*
compositeState.exit→**isEmpty**() **and** compositeState.entry→**isEmpty**() **and**
compositeState.do→**isEmpty**() **and**
-- *no transitions*

```
    compositeState.internal→isEmpty() and compositeState.incoming→isEmpty() and
    compositeState.outgoing→isEmpty() and compositeState.subVertex = self and
    -- the container is the same
    self .container@pre→forAll(each:CompositeState| each = compositeState.container)))
```

A.3 Fold Outgoing Transition

```
CompositeState::foldOutgoing
pre:
    let possible = self .subvertex.outgoing→select(a,b:Transition|
        a <>b implies (a.target = b.target and
            a. effect .isEquivalentTo(b. effect ) and
            a. trigger .isEquivalentTo(b. trigger ))) in (
    self .subvertex→forAll(s:State| s .outgoing→intersection(possible)→size() = 1)
post:
    possible→isEmpty() and
    self .outgoing→select(t:Transition | possible@pre→forAll(each:Transition|
        each.target = t.target and
            each. effect .isEquivalentTo(t. effect ) and
            each. trigger .isEquivalentTo(t. trigger )))→size() = 1
```

For the sake of clarity, we have used the *possible* expression, defined in the preconditions, inside the postcondition. Actually, this is not possible, the *let* expression should be rewritten.

A.4 Unfold Outgoing Transition

```
Transition :: unfoldOutgoing
pre:
    let cs = self .source in (cs .oclIsKindOf(CompositeState) and
    cs .subvertex→notEmpty())
post:
    let cs = self@pre.source in (cs.subvertex→forAll(s:State| s .outgoing→
        select(t:Transition | t .target = self@pre.target and
            t . effect .isEquivalentTo( self@pre.effect) and
            t . trigger .isEquivalentTo( self@pre.trigger))→size() = 1) and
    -- the transition is no longer referenced:
    self .source→isEmpty and self.target→isEmpty and
    self . trigger →isEmpty and self.effect→isEmpty
```

A.5 Move State into Composite

```
State :: moveInto(cs:CompositeState)
pre:
    let substates = cs.subvertex in
```

```
    substates→excludes(s) and
    cs.container = self.container and
    not cs.isConcurrent and
    -- outgoing transitions
    cs.outgoing→forAll(each:Transition | self.outgoing→exists(t:Transition|
        t.sameLabel(each) and t.target = each.target)) and
    -- do action
    cs.do→notEmpty implies cs.do.sameLabel(self.do) and
    -- entry action
    cs.entry→notEmpty implies (substates→includesAll(self.incoming.source) and
        self.outgoing→select(t:Transition | substates→ includes(t.target))→
            collect( effect )→isEmpty()) and
    -- exit action
    cs.exit→notEmpty implies (substates→includesAll(self.outgoing.target) and
        self.incoming→select(t:Transition | substates→includes(t.source))→
            collect( effect )→isEmpty())
post:
    let substates = cs.subvertex in
    substates→includes(s) and
    cs.outgoing→forAll(each:Transition | self.outgoing→select(t:Transition|
        t.sameLabel(each) and t.target = each.target))→isEmpty() and
    cs.do→notEmpty implies cs.do→isEmpty() and
    cs.entry→notEmpty implies self.outgoing→ select(t:Transition | substates→
        includes(t.target))→forAll(t:Transition | cs.entry.sameLabel(t.effect )) and
    cs.exit→notEmpty implies self.incoming→ select(t:Transition | substates→
        includes(t.source))→forAll(t:Transition | cs.exit.sameLabel(t.effect ))
```

To compare two transitions, an operation named *Same Label* was defined and is presented p. 14.

A.6 Move State out of Composite

```
State :: moveOutOf
pre:
    -- inner transitions
    self.container.exit→notEmpty() implies
        self.incoming→select(t:Transition| t.source <> self and
            self.container.allSubvertex()→includes(t.source))→
        forAll(t:Transition|t. effect .isEquivalentTo( self.container.exit )) and
    self.container.entry→notEmpty() implies
        self.outgoing→select(t:Transition| t.target <> self and
            self.container.allSubvertex()→includes(t.target))→
        forAll(t:Transition|t. effect .isEquivalentTo( self.container.entry )) and
    -- outer transitions
    self.container.exit→notEmpty() implies self.outgoing→select(t:Transition|
        self.container.allSubVertex()→excludes(t.target))→forAll(t:Transition|
            t. effect →isEmpty) and
    self.container.entry→notEmpty() implies self.incoming→select(t:Transition|
        self.container.allSubVertex()→excludes(t.source))→forAll(t:Transition|
```

```
        t . effect →isEmpty) and
    self .container.do→notEmpty() implies self.do→isEmpty()
post:
    let cs = self .container→select(s:State| s = self@pre.container)→first() in (
    cs→notEmpty and
    -- composite outgoing transitions
    cs.outgoing→forAll(t:Transition | self .outgoing→
        exists(ot:Transition| ot.target = t.target and ot.sameLabel(t))) and
    -- initial pseudo substate
    self@pre.incoming.source→exists(s:State| s.oclIsKind(Pseudostate) and
        s.kind = #initial) implies (cs.incoming→isEmpty() and not cs.subvertex→
            exists(s:State| s.oclIsKind(Pseudostate) and s.kind = #initial) and
                self@pre.container.incoming→forAll(t:Transition| self.incoming→
                    exists(ot:Transition| ot.source = t.source and ot.sameLabel(t)))) and
    -- ex inner incoming/outgoing
    cs. exit →notEmpty() implies self.incoming→select(t:Transition| cs.allSubVertex→
        includes(t.source)). effect →isEmpty() and
    cs.entry→notEmpty() implies self.outgoing→select(t:Transition| cs.allSubVertex→
        includes(t.target)). effect →isEmpty() and
    -- ex outer incoming/outgoing
    cs. exit →notEmpty() implies self.outgoing→select(t:Transition| cs.allSubVertex→
        exludes(t.target)). effect →forAll(a:Action| a.isEquivalentTo(cs.exit)) and
    cs.entry→notEmpty() implies self.incoming→select(t:Transition| cs.allSubVertex→
        exludes(t.source)). effect →forAll(a:Action| a.isEquivalentTo(cs.entry)) and
    cs.do→notEmpty() implies self.do.isEquivalentTo(cs.do))
```

A.7 Same Label

```
Transition :: sameLabel(t:Transition)
post:
result = self . effect .isEquivalentTo(t. effect ) and
    self . trigger .isEquivalentTo(t. trigger ) and
    self .guard.isEquivalentTo(t.guard)
```