UML MODEL REFACTORING USING GRAPH TRANSFORMATION

by

Alessandro Folli

Supervisor: Tom Mens

THESIS SUBMITTED AS PART
OF THE ERASMUS PROGRAM
AT
UNIVERSITÉ DE MONS-HAINAUT
BELGIUM
JUNE 2007

UMH

ACADÉMIE UNIVERSITAIRE
WALLONIE-BRUXELLES

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Contents

Chapter 1

Introduction

Model–driven engineering (MDE) is a software engineering approach that promotes the use of models and transformations as primary artifacts. Its goal is to tackle the complexity of developing, maintaining and evolving complex software systems by raising the level of abstraction from source code to models. The mechanism of *Model transformation* is at the heart of this approach, and it represents the ability to transform and manipulate models. [1]

The transformations between models provide a chain that enables the evolution and the automated generation of an executable system from its corresponding models. Model transformation definition, implementation and execution are critical aspects of this process. Furthermore, model transformations are also models, and therefore an integral part of this model–based approach. Model transformations need specialised support in several aspects in order to realize their full potential for software architects, software developers and tool vendors. The problem goes beyond having languages to represent model transformations because the transformations also need to be reused and they need to be integrated into software development methodologies and development environments that make full use of them.

The term *refactoring* was originally introduced by Opdyke in his seminal PhD dissertation [2] in the context of object–oriented programming. Martin Flower [3] defines this activity as “the process of changing a software system in such a way that it does not alter the external behaviour of the code, yet improves its internal structure”. This research will focus on the problem of Model Refactoring, which aims to apply refactoring techniques at model level. Model Refactoring is a particular kind of model transformation and may also be called *Behaviour–Preserving Model Transformation*.

The *Unified Modeling Language* (UML) [4, 5] can be used to specify, visualize, and document models of software systems, including their structure and design. Since the UML is the generally accepted object–oriented modeling language, it ought to play an essential role in MDE. A software design is typically modeled as a collection of different UML diagrams. Because different aspects of the software system are covered by different types of UML diagrams, there is an inherent risk that the overall specification of the system is inconsistent. Also model transformations, such as (arbitrary) model evolutions, can transform a model into an inconsistent state. *Inconsistency management* has been defined in Finkelstein et al. [6] as "the process by which inconsistencies between software models are handled so as to support the goals of the stakeholders concerned". It is a complex process that includes activities for detecting, diagnosing, and handling the inconsistencies. These activities are extended by Spanoudakis and Zisman [7] to include detection of overlaps, detection of inconsistencies, diagnosis of inconsistencies, handling of inconsistencies, tracking, specification and application of an inconsistency management policy.

Current–day UML CASE modeling environments provide poor support for evolving and managing inconsistencies between UML models. In particular, little research has been done taking into account a wide range of inconsistencies over different kinds of UML diagrams. Inconsistency management in the UML context is quite complicated due to several reasons. The most obvious reasons are the lack of formal semantics for the UML and that the UML is a general purpose language that can be applied to several application domains and in several software development processes. [8]

Therefore, the goal of this dissertation will be to formally explore the refactoring of UML models and the related process of consistency maintenance in order to ensure that the refactoring will not make the UML model inconsistent.

In this dissertation, we will use graphs to represent UML models and graph transformations to specify and apply model transformations. This choice is motivated by the following reasons:

- Graphs are a natural representation of models that are intrinsically graph–based in nature (e.g., class diagrams, state machine diagrams, activity diagrams, sequence diagrams).
- Graph transformation theory provides a formal foundation for the analysis and the automatic and interactive application of model transformations.

In this dissertation, we will propose an initial catalog of model refactorings for different kinds of UML diagrams (e.g., class diagrams, state machine diagrams); each model refactoring will be formalised, explained and motivated using a concrete example. The AGG general–purpose graph transformation tool supplies a suitable instrument to formally define and verify them.

We will also use the graph transformation formalism to check that the consistency between different diagrams is preserved when the model refactorings are applied. Inconsistency detections and their resolutions can be expressed as graph transformation rules, in this way they will integrate the rules defined to formalise the model refactorings specified earlier.

To summarize, the main contributions of this dissertation will be:

- An initial catalog of model refactorings for UML models with a detailed description of the characteristics, the applicability and the problems that must be addressed.
- A study of the use of graph transformations for specification of model refactoring. In particular, to assess the feasibility using a specific graph transformation tool – AGG – and to provide recommendations about AGG may be improved to better support model refactoring.
- A guideline for future work to be carried out in the domain of model refactoring.
- A study of the relation between model refactoring and model consistency in presence of UML models composed of different kinds of diagrams.

Chapter 2

Unified Modeling Language

2.1 Unified Modeling Language

In the field of software engineering, the Unified Modeling Language (UML) is a non–proprietary specification language for object modeling. UML is a general–purpose modeling language that includes a standardized graphical notation used to create an abstract model of a system, referred to as a UML model. [9]

UML is officially defined by the Object Management Group (OMG) [4, 5] by the UML meta-model. UML was designed to specify, visualize, construct, and document software–intensive systems.

Distinction between the UML model and the set of diagrams of a system is important. A diagram is a partial graphical representation of a system's model. The model also contains a "semantic backplane" – documentation such as written use cases that drive the model elements and diagrams. [9]

UML allows to describe the system in three different prominent aspects, each of them using a different set of diagrams. The diagrams could possibly be related in order to add more information to the modeled system.

- Functional Model: Shows the functionality of the system from the user's point of view, therefore it describes the external behaviour of the system. It includes the *Use Case diagrams*.

- Object Model: Shows the structure and substructure of the system using objects, attributes, operations, and associations. It includes *Class diagrams*, *Object diagrams* and *Deployment diagrams*.

- Dynamic Model: Shows the internal behaviour of the system; it describes therefore how the objects evolve and interact. It includes *Sequence diagrams*, *Activity diagrams* and *State Machine diagrams*.

In UML 2.0, there are 13 types of diagrams [4]. To understand them, it is sometimes useful to categorize them hierarchically, as shown in figure 2.1.

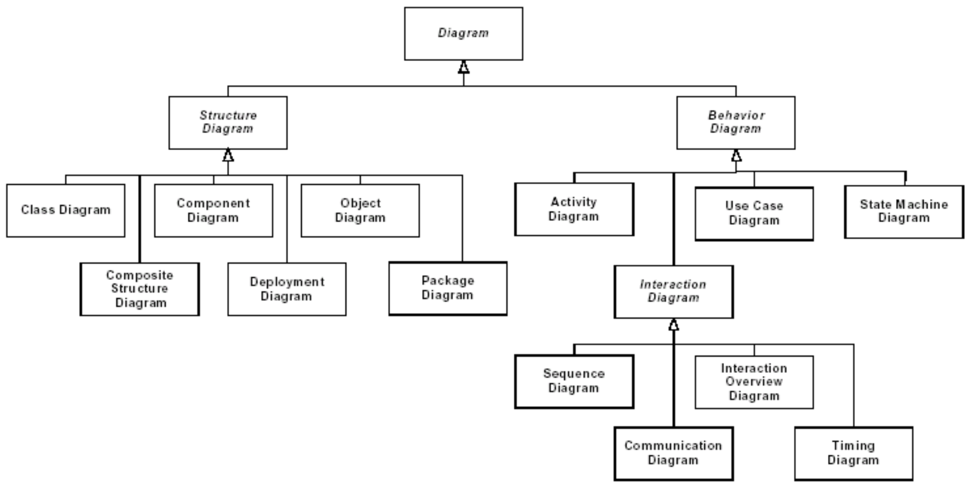


Figure 2.1: Hierarchy of UML 2.0 Diagrams

Structure Diagrams emphasize what must be in the system being modelled:

- Class diagram
- Component diagram
- Composite structure diagram
- Deployment diagram
- Object diagram

- Package diagram

Behaviour Diagrams emphasize what must happen in the system being modelled:

- Activity diagram
- State Machine diagram
- Use case diagram

Interaction Diagrams –a subset of behaviour diagrams– emphasize the flow of control and data among the things in the system being modelled:

- Communication diagram
- Interaction overview diagram (UML 2.0)
- Sequence diagram
- UML Timing Diagram (UML 2.0)

This dissertation will focus on the analysis of Class diagrams and State Machine diagrams as they are among the commonly used diagrams. However, other kinds of diagrams will be used to formally define the results.

2.2 Class diagrams

A *Class diagram* is a graphical representation; it represents the structure of the system and it should be noticed that a Class diagram is a static view of the modeled system.

The purpose of a *Class diagram* is to depict the classes within a model. In an object oriented application, classes have attributes (member variables), properties (member functions) and relationships with other classes. The UML class diagram can depict all these things quite easily. Figure 2.2 shows an example of Class diagram.

2.3 State Machine diagrams

The *State Machine diagrams* –formerly called *State Chart diagrams* in UML 1.x– describe how the instances of the class objects work. They are a specification of the dynamic behaviour of individual class objects.

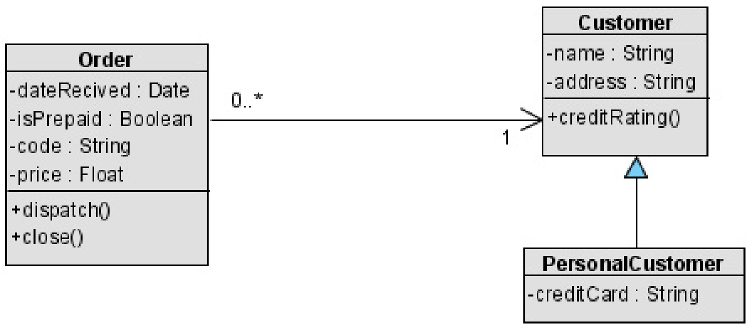


Figure 2.2: Class Diagram Example

State Machine diagrams depict the various states that an object may be in, and the transitions between those states. A state represents a stage in the behaviour pattern of an object; it is possible to have initial states and final states. An initial state –also called a creation state– is the one that an object is in when it is first created, whereas a final state is one in which no transitions lead out of. A transition is a progression from one state to another and will be triggered by an event that is either internal or external to the object. Figure 2.3 shows an example of State Machine diagram.

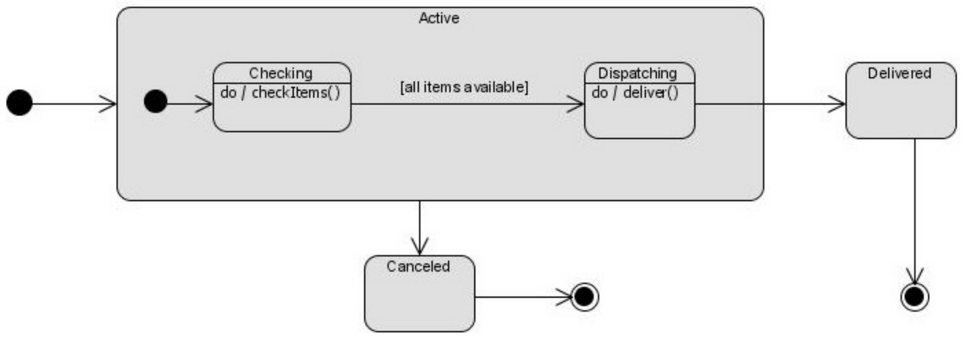


Figure 2.3: State Machine Diagram Example

// Chapter 3

UML Model Refactoring

3.1 Model Transformation

Model refactoring represents only one specific kind of model transformation. This section will give a general high–level overview of model transformation, and will show where model refactoring fits in. A detailed taxonomy of model transformations has been presented by Tom Mens and Pieter Van Gorp [10] during the *Workshop on Graph and Model Transformation* from which we will summarise some important ideas here.

In order to transform models, the latter need to be expressed by some modeling language, the syntax of which being expressed by a *metamodel* [11].

A distinction can be made between *endogenous* and *exogenous* transformations, based on the number of metamodels that are used to express the source and the target models. *Endogenous Transformations* are transformations between models expressed through the same metamodel, contrary to the *Exogenous Transformations* which involve models expressed through different metamodels.

Typical examples of exogenous transformations are:

- *Synthesis* of a higher–level, more abstract, specification (e.g., an analysis or design model) into a lower–level, more concrete one (e.g, a model of a Java program). A typical example of synthesis is *code generation*, where the source code is translated into bytecode (that runs on a virtual machine) or executable code, or where the design models are translated into source code.

- *Reverse engineering* is the inverse of synthesis, and extracts a higher–level specification from a lower–level one.
- *Translation* is the translation of a model from one language to another, all the while keeping the same level of abstraction. For example from the UML to the XMI representation of a model, or from the Class diagram to the E–R diagram representation of a database model.

Typical examples of endogenous transformation are:

- *Optimization*, a transformation aimed to improve certain operational qualities (e.g., performance), while preserving the semantics of the software.
- *Refactoring*, a change to the internal structure of software to improve certain structural qualities (such as understandability, modifiability, reusability, modularity, adaptability) without changing its observable behaviour [3].
- *Simplification* and *normalization*, used to decrease the syntactic complexity. For example, the Simplification could be used to remove redundancy and obsolete code. The normalization is often used in the database modeling process.
- *Component adaptation*, to modify and adapt the code of existing software components, either statically or dynamically (i.e., during component execution), to the user needs.

In the same way, one more distinction can be made between *horizontal* and *vertical* transformations based on the abstraction level which the source and target models reside on. *Horizontal Transformation* are those where the source and the target models belong to the same abstraction level; on the contrary the *Vertical Transformations* involve models at different abstraction levels.

Table 3.1 illustrates that the dimensions “horizontal” versus “vertical” and “endogenous” versus “exogenous” are truly orthogonal, by giving a concrete example of all possible combinations [10].

Others distinctions can be made based on the characteristics of the transformations, however this is outside the scope of this paper.

	Horizontal	**Vertical**
Endogenous	*Refactoring* *Optimization*	*Formal refinement*
Exogenous	*Language migration* *Translation*	*Code generation* *Reverse engineering*

Table 3.1: Orthogonal dimensions of model transformations with examples.

3.2 Model Refactoring

Model refactoring –as shown in the previous section– is a special kind of endogenous, horizontal model transformation which aims to evolve and improve the structure of the model, all the while preserving (certain aspects of) its behaviour. Like the process of source code refactoring [12], the process of model refactoring consists of distinct activities:

1. Identify of where the model should be refactored.
2. Determine which model refactoring(s) should be applied to the places identified.
3. Apply the model refactoring(s).
4. Guarantee that the model refactoring applied preserves its behaviour.
5. Assess the effect of the refactoring on quality characteristics of the model (such as complexity, understandability, maintainability).
6. Maintain consistency between the model refactored and other software artifacts (such as related models, program code, etc.).

A definition of refactoring has been introduced by *Don Bradley Roberts* in his PhD dissertation [13]. He defines refactorings as program transformations containing particular preconditions that must be verified before the transformation can be applied.

The following definition of model refactoring is adapted from Roberts' refactoring definition [13, 14]:

Definition 3.2.1. *A Model refactoring is a pair $R = (pre, T)$ where pre is the set of preconditions that the model must satisfy, and T is the model transformation.*

By using this, when we formally define the model refactoring we will express all the preconditions that must be satisfied before the transformation can be performed, and in chapter 4 we will show how these preconditions may be expressed as part of graph transformations.

In formally defining model refactoring, it is also necessary to ensure three prominent aspects:

- that the transformation does not lead to an ill–formed (i.e. syntactically incorrect) model. In principle, this requirement will be guaranteed in AGG by defining a Type Graph and typed graph transformations.
- that the transformation does not lead to an inconsistent model. Even if the syntax is correct, the model could be inconsistent. The same is true, by the way, for any programming language.
- that the transformation preserves the model behaviour.

A model refactoring is not supposed to change the behaviour specified by the models in question. However, even if the preservation of model behaviour is crucial to model refactoring, it is very difficult to be achieved. Model refactoring is a rather recent research issue and such definitions of behaviour preservation properties have not yet been completely given. There are some proposals about behaviour preservation but, in the context of the UML, such definitions do not exist because there is no consensus on a formal definition of behaviour.

Also for source code refactorings, definitions of behaviour preservation are rarely provided. Opdyke [2] suggests the following definition of behaviour preservation: “for the same set of input values, the resulting set of output values should be the same before and after the refactoring”. To ensure this kind of behaviour preservation, refactoring preconditions and postconditions need to be specified [13]. However, as explained by [12], this kind of behaviour preservation is sometimes insufficient since many other aspects of the behaviour may be relevant as well. This implies the need for a wide range of definitions of behaviour preservation depending on domain–specific, user–specific, or company–specific concerns.

Behaviour preservation can also be dealt with in a more pragmatic way. A first approach is by means of rigorous testing. Another pragmatic approach is to specify a weaker notion of behaviour preservation that is not sufficient enough to guarantee the full program semantics preservation, but focuses on specific issues. For example, we may adopt a notion of call preservation, which guarantees that all method calls are preserved by the refactoring [15].

In this dissertation we will focus more on the problem of inconsistency management as a complete definition of behaviour preservation does not yet exist. In other words, we will consider a model refactoring to be correct if it does not lead to an inconsistent model.

3.3 Refactoring Examples

To better understand what *model refactorings* are and how a transformation could lead to an inconsistent model, we will show some detailed examples. A running example is used throughout the dissertation to explain and illustrate the main concepts.

3.3.1 Pull Up Operation and Push Down Operation

The *Push Down Operation* refactoring is applied on UML Class diagrams and it copies an operation from a superclass to its subclasses, deleting the original. In the example shown in figure 3.1, the *snapshot* operation has been removed from the *Viewer* class and has been added to the subclasses *PhotoPlayer* and *MoviePlayer*.

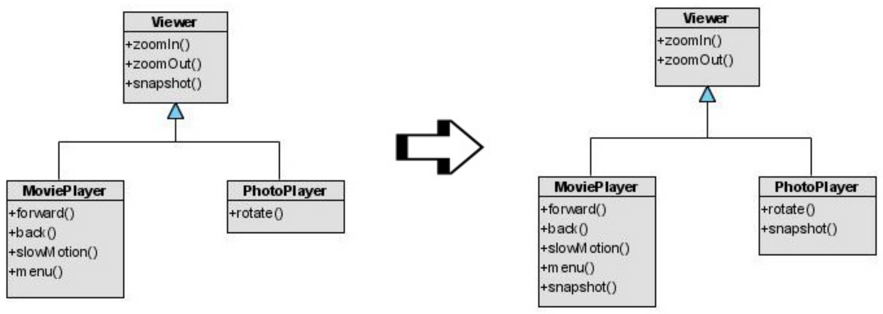


Figure 3.1: Push Down Operation Refactoring

The *Pull Up Operation* refactoring, shown in figure 3.2, is the counterpart of the *Push Down Operation* refactoring; it copies an operation from the subclasses to a superclass, deleting the original. In the example shown in figure 3.2, the *snapshot* operation has been removed from the subclasses *PhotoPlayer* and *MoviePlayer* and has been added the superclass *Viewer*.

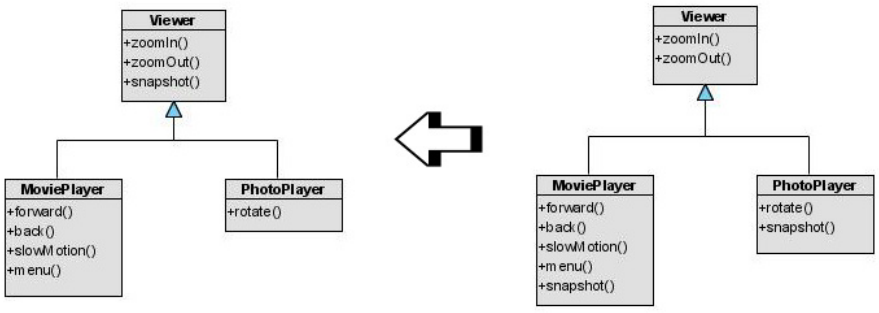


Figure 3.2: Pull Up Operation Refactoring

3.3.2 Extract Class

The *Extract Class* refactoring is applied on UML Class diagrams and, as suggested by the name, it extracts a class from an existing one exporting a set of operations and attributes. The refactoring creates a new class containing the operations and attributes specified and connects it through a new association to the source class from where it was extracted.

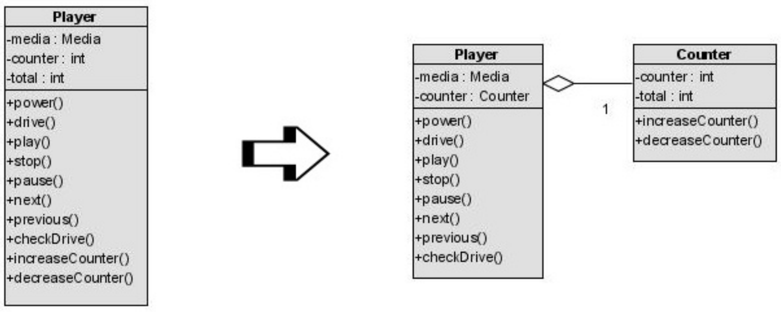


Figure 3.3: Exctract Class Refactoring

In the running example, shown in figure 3.3, the class *Counter* has been extracted from the class *Player*, the operations *increaseCounter* and *decreaseCounter* being moved to the new class. The type of the original attribute *counter* of the class *Player* has been changed and, after the refactoring, it corresponds to the newly created class.

The *Extract Subclass* refactoring, similar to the Extract Class refactoring, creates a new class as well, but inserts it between the source class and its direct subclasses.

Figure 3.4 shows an example of model inconsistency generated by this kind of refactoring,

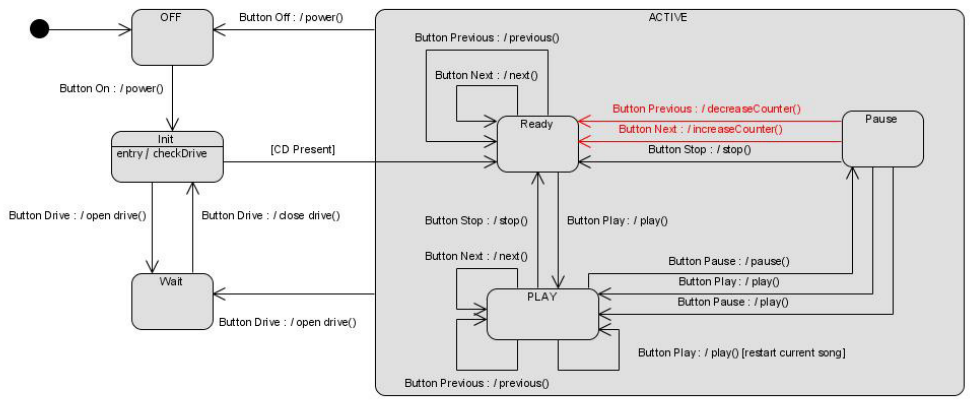


Figure 3.4: Model Inconsistencies

even if the Class diagram modified by the refactoring is well–formed and correct.

The State Machine diagram of the running example, shown in figure 3.4, describes the behaviour of the class *Player*. The diagram refers to the operations *increaseCounter* and *decreaseCounter* contained by the class *Player*. After the refactoring, these operations are not contained by the class anymore and the State Machine diagram is incorrect, the refactoring having generated a model inconsistency.

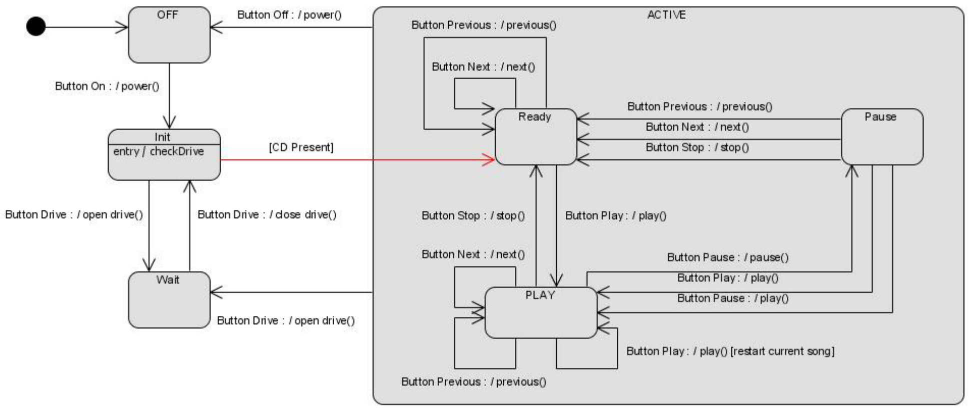
Obviously this kind of inconsistency could be generated only by working with different kinds of UML diagrams.

3.3.3 Introduce/Remove Pseudostates

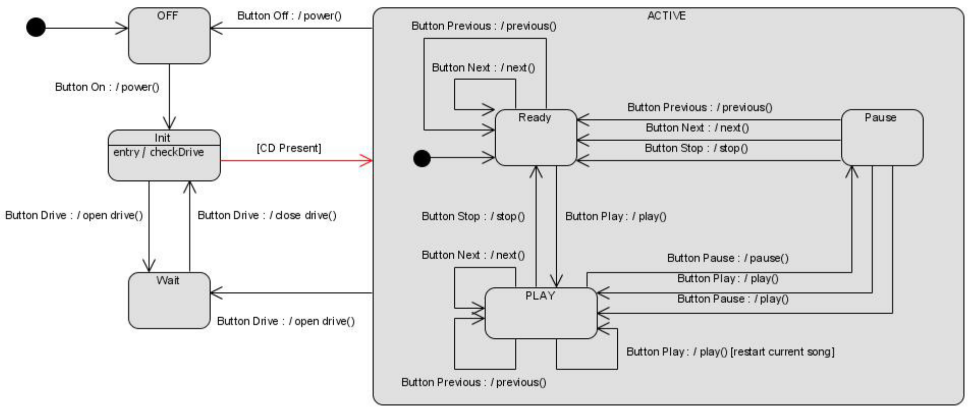
The *Introduce/Remove Pseudostates* refactoring is applied on UML State Machine diagrams and, as suggested by the name, it is used to introduce and remove some kinds of pseudostates (i.e. Initial pseudostate).

Figure 3.5 shows a simple example of the Introduce Initial Pseudostate refactoring. An initial pseudostate has been added to the *ACTIVE* region and the target of the transition –that initially refers to the *Ready* state– has been changed to become the region itself. An automatic transition has been defined between the initial pseudostate and the *Ready* state.

The *Ready* state will become the default initial state of the *ACTIVE* region; a transition whose target is the *ACTIVE* state will lead the State Machine to the *Ready* state.



(a) Without Initial Pseudostate



(b) With Initial Pseudostate

Figure 3.5: Introduce/Remove Initial Pseudostate Refactoring

Chapter 4

Graph Transformations

In this chapter, we will introduce the necessary concepts about graph transformations for the purpose of Model Refactoring, and we will present the *AGG graph transformation tool* used to implement and test model refactorings.

4.1 Graph Theory

Within the last decade, graph transformation has been used as a modeling technique in software engineering and as a meta–language to specify and implement visual modeling techniques like the UML. Different concepts of graph transformation exist and we will focus on the use of *Graph Grammars* in this dissertation.

The graph grammar theory [16, 17] applies formal language theory to the specification of graphs. A formal language for a textual language specifies a grammar for the language, the grammar consisting of a series of rules or productions that specify how valid sentences in the language are constructed. The same way, a graph–grammar consists of a set of productions that can be used to construct valid sentences in a graph.

We can extract a formal definition of graph from the book *‘‘Fundamentals of Algebraic Graph Transformation”*: [18].

Definition 4.1.1. *Labelled graph.*
A labelled graph G consists of a set V of nodes (also called vertices) and a set of edges E, two functions $s, t : E \rightarrow V$ to associate to each edge a source and target vertex, and the functions $lv : V \rightarrow L$ and $le : E \rightarrow L$ to associate a label to every vertex and edge.

The figure 4.1 shows a simple example of graph.

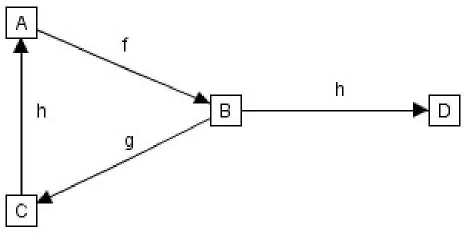


Figure 4.1: Example of graph

Definition 4.1.2. *Graph morphism.*
Let G and H be two graphs. A graph morphism $m : G \rightarrow H$ consists of a pair of partial functions $m_V : V_G \rightarrow V_H$ and $m_E : E_G \rightarrow E_H$ that preserve sources and targets of edges, i.e., $s_H \circ m_E = m_V \circ s_G$ and $t_H \circ m_E = m_V \circ t_G$. It also preserves vertex labels and edge labels, i.e., $l_H \circ m_V = l_G$ and $l_H \circ m_E = l_G$.
A graph morphism $m : G \rightarrow H$ is injective (surjective) if both m_V and m_E are injective (surjective). It is isomorphic if m is injective and surjective. In that case, we write $G \cong H$.

Note that the functions m_V and m_E are partial to allow for vertex deletions and edge deletions. All vertices in $V_G \backslash dom(m_V)$ and all edges in $E_G \backslash dom(m_E)$ are considered to be deleted by m.

A *match* m of graph G_1 to graph G_2 is a total graph morphism that maps the vertices and edges of G_1 to G_2 such that the graphical structure and the labels are preserved.

To determine if a graph is well–formed, it is necessary to check whether it conforms to a so–called Type Graph. The formal definition of typed graphs is taken from [19].

Definition 4.1.3. *Typed graph.*
Let TG be a graph (called the type graph). A typed graph (over TG) is a pair (G, t) such that G is a graph and $t : G \rightarrow TG$ is a graph morphism. A typed graph morphism $(G, t_G) \rightarrow (H, t_H)$ is a graph morphism $m : G \rightarrow H$ that also preserves typing, i.e., $t_H \circ m = t_G$.

Attributed graphs are graphs where each vertex or edge can contain zero or more attributes used to to attach additional information to them. Such an attribute is typically a name–value pair, that allows the specification of a value for each attribute name. These values can be very simple (e.g., a number or a string) or more complex (e.g., a Java expression). The notion of *typed graph* can be extended for the *attributed graphs*.

Definition 4.1.4. *Graph production (rule).*
Let P_l and P_r be labelled graphs. A graph production is a graph morphism $P : P_l \to P_r$.

A *graph transformation* is the result of application of a graph production. A production rule P is a partial morphism between a left–hand side P_l and a right–hand side P_r, which provides information about which elements are preserved, deleted and created in case of an application of the production. A production is applicable to a graph G, if there is a match of P_l to G.

Definition 4.1.5. *Graph transformation.*
A graph transformation $G \Rightarrow_t H$ is a pair $t = (P, m)$ consisting of a graph production $P : P_l \to P_r$ and an injective graph morphism (called match) $m : P_l \to G$.

The mechanism of graph transformation may also be extended with the notion of *application conditions*.

Definition 4.1.6. *Negative application condition.*
Let $P : P_l \to P_r$ be a graph production. A negative application condition for P is a graph morphism $nac : P_l \to \hat{P_l}$. A graph transformation $G \Rightarrow_{(P,m)} H$ satisfies a negative application condition nac if no graph morphism $\hat{m} : \hat{P_l} \to G$ exists such that $\hat{m} \circ nac = m$.

Needless to say, the introduction of this notion of application conditions makes graph transformation considerably more expressive.

Two different approaches are available to deal with a system containing a large number of graph productions: *programmed graph transformation* and *graph grammars*.

The *programmed graph transformation* approach is implemented by tools such as Fujaba [20] and it is possible to specify a control structure that controls the order in which graph productions can be applied.

Graph grammars do not impose any control structure and all available graph productions are applied non–deterministically or at random. As such, a given initial graph G can give rise to a whole range of possible result graphs, which is referred to as $L(G)$, the language generated by the graph grammar. Each word in this language corresponds to a possible sequence of graph transformations that can be applied to G. Graph grammars are used in the AGG graph transformation tool. [21]

4.2 Specifying model refactorings as graph transformations

According to Bézivin and many others [21], a model can be represented using a graph–based structure. Figure 4.2 intuitively shows the correspondence between models and their graph representation. The graph transformation theory offers many theoretical results that can help during analysis of model refactorings.

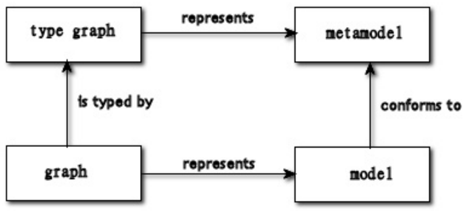


Figure 4.2: Relationship between models and their graph representation.

According to the results of many studies [21, 22], graph transformations are a suitable technology to deal with model transformation. Table 4.1 shows the result of a comparison between graph transformation and model refactoring.

Graph transformation	**Refactoring**
type graph and global graph constraints	well–formedness constraints
negative application conditions	refactoring preconditions
parameterised graph productions with NACs	refactoring transformation
programmed graph transformations	composite refactorings

Table 4.1: Comparison of graph transformation and refactoring concepts.

To emphasize the results mentioned in Table 4.1, we will start by introducing AGG, a general–purpose graph transformation tool.

4.3 The AGG graph transformation tool

AGG [23, 24] is a rule–based visual programming environment supporting an algebraic single–pushout approach [25] to graph transformation. It aims at the specification and prototypical implementation of applications with complex graph–structured data. AGG may be used as a general purpose graph transformation engine in high–level JAVA applications employing graph transformation methods.

AGG program consists of a *graph grammar* attributed by Java objects which may come from user–defined Java classes, and it supports the specification of type graphs with multiplicities and attributes.

Graph grammars consist of one graph –the start graph initializing the system– and a set of rules describing the actions which can be performed. The start graph may also be attributed by Java objects and expressions. Moreover, rules may be equipped by negative application conditions. The way graph rules are applied, directly realises the single–pushout approach to graph transformation presented in [25] .

A graph consists of two disjoint sets containing the nodes and the arcs of the graph, they acting like ordinary Java variables to which a value can be assigned. As a whole, the nodes and arcs are called the objects of the graph. The attribution of nodes and arcs by Java objects and expressions follows the ideas of attributed graph grammars as stated in [26].

4.3.1 Attributed graphs

AGG graphs are directed, and their nodes and arcs may be typed and attributed. A type can be composed from a string and the visual layout. It is possible that the string is empty and the type is determined only by the layout, i.e. different layouts mean different types.

The attributes are specified by a type, a name and a value. Each graph node and arc may have several attributes. All graph objects (nodes and arcs) of one type also share their attribute declaration, i.e. the list of attribute types and names.

The attributes may be typed by any valid Java type. This means that it is not only possible to annotate graph objects by simple types like strings or numbers, but it is also possible to use arbitrary custom classes to gain maximal flexibility in attribution.

4.3.2 Graph Rules

Graph rules are used to describe graph transformation. They consist of a left and a right–hand side L and R and, moreover, a set of negative application conditions. The graphs occurring in a rule are typed and attributed, as mentioned above. However, it depends on the rule side which kinds of attributes are allowed. Left–hand sides do not only contain concrete Java objects, but are also allowed to have variables. They are used to abstract the operation from concrete attribute values.

Moreover, the right-hand side may contain more complex Java expressions to express computations on the attributes. If an attribute is used only to check some value without changing it, it only has to occur on the left-hand side of a rule. If an attribute value is changed independently of the value it had before, it only has to occur on the right–hand side.

The left and the right–hand side of a rule are related by a partial graph morphism L $\Rightarrow$ R. Those graph parts related by this morphism are preserved by the rule; all the other graph objects in the left–hand side are deleted; all others in the right–hand side are newly created. To indicate which objects are mapped to one another in the graphs, the AGG tool use numerical tags preceding an object's type name, separated by a colon.

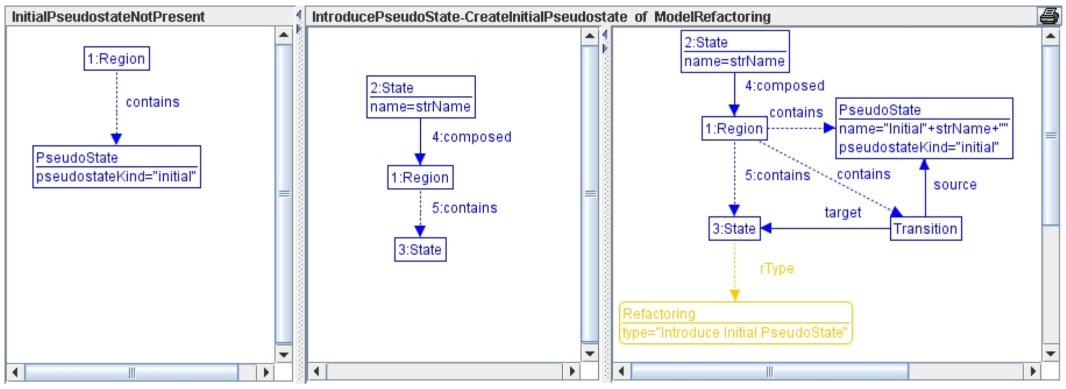

InitialPseudostateNotPresent
1:Region
contains
PseudoState
pseudostateKind="initial"
IntroducePseudoState-CreateInitialPseudostate of ModelRefactoring
2:State
name=strName
4:composed
1:Region
5:contains
3:State
2:State
name=strName
4:composed
contains
1:Region
PseudoState
name="Initial"+strName+""
pseudostateKind="initial"
5:contains
contains
source
target
3:State
Transition
rType
Refactoring
type="Introduce Initial PseudoState"

Figure 4.3: Graph Rule with NAC

Figure 4.3 shows an example of graph rule for which a negative application condition has been defined.

4.4 UML Type Graph

UML models can be represented as a graph–based structure, and graphs must conform to the corresponding *Type Graph*, as well the models must conform to their *Metamodel*.

The *Unified Modeling Language* is officially defined by the UML metamodel [4, 5] as mentioned above. A *Type Graph* corresponding to the *UML metamodel* is required to formally represent the UML models as graphs and to formally define the UML Model Refactoring.

For the purpose of this dissertation, we have chosen to take into account a subset of the concepts defined by the UML metamodel, in particular we have focused on the implementation of *UML Class diagrams* and *UML State Machine diagrams*.

Figure 4.4 shows the Type Graph corresponding to the part of interest of the UML metamodel, implemented using the AGG graph transformation tool.

AGG offers many concepts that are useful to define a type graph very close to the UML metamodel. AGG allows enrichment of the type graph with an inheritance relation between nodes, and each node type can have one or more direct ancestor (parent) from which it inherits the attributes and edges. Moreover, it is also possible to define a node type as an abstract type; it is not possible to create an instance node of an abstract type.

However, using AGG it is more complicated to represent concepts like *Aggregation* and *Composition* used by the UML metamodel. In some cases the Type Graph has been simplified considering the more generic concept of association. Moreover, AGG does not have any notion of Enumeration type. The property *kind* of the *Pseudostate* element has been represented using a String value.

Looking at the UML metamodel [4, 5] we have encountered some poorly–defined concepts, and decisions were required to ensure a correct definition of model refactorings. For example, the UML specification [4, 5] affirms that all transitions are owned by regions but the specification is not clear about which transitions can be owned by which regions. The chosen rule for this dissertation is that a transition between two states is owned by the "closest" region that contains both states. States and regions constitute a containment hierarchy, then the closest region would be the least common ancestor (LCA) of the source and target states of the transition in that hierarchy.

On the other hand, it is also possible to assume that all transitions are owned by the top–most region, considering valid the concepts defined in previous versions of UML [27]: *"All Transitions which are essentially relationships between States, except internal transitions, are owned by the StateMachine"*.

That decision could generate incompatibilities among tools that use a different approach. Anyway, it is always possible to apply some transformations in order to adapt the graph to the solution chosen.

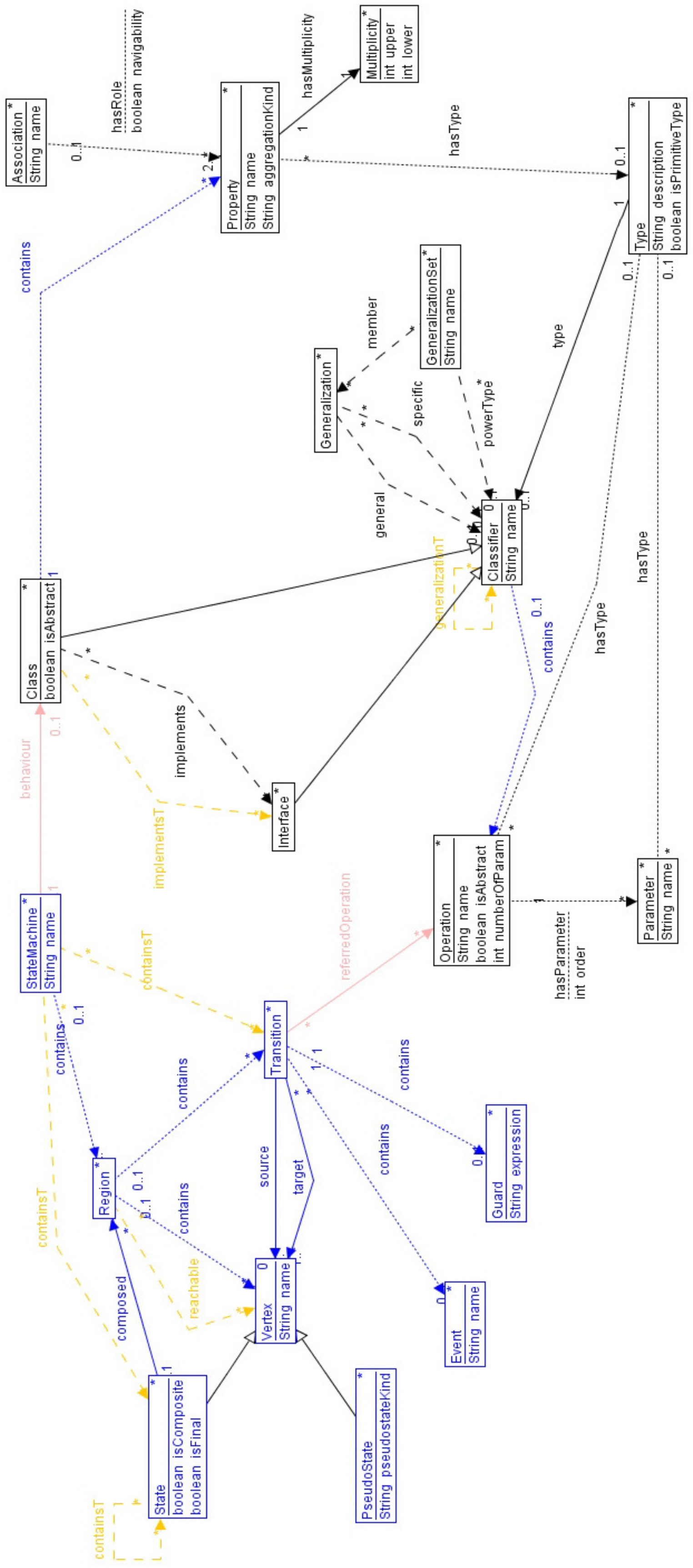


Figure 4.4: UML 2.0 Type Graph

Chapter 5

Model Refactoring Formalisation

This chapter will supply a detailed description of each model refactoring, the characteristics and the applicability, by showing also some concrete examples. An explanation of the general refactoring behaviour precedes the description of each single transformation step that in case are implemented using the AGG general–purpose graph transformation tool.

5.1 List of Model refactorings

The standard catalogue for source code refactorings can be found in [3]. This catalogue describes in detail seventy–two refactorings for the restructuring of object–oriented constructs. With respect to the research of model refactoring at a higher abstract level, Sunyé et al. [28] proposed an initial set of refactorings for UML Class diagrams and State Machine diagrams. Their research provided a fundamental paradigm for model refactoring to improve the design of object-oriented applications; nevertheless, they do not have any concrete implementation of representative tools.

This chapter describes the list of the chosen model refactorings used in our dissertation. The representative list will constitute an initial catalog of model refactorings for UML models.

Implemented refactorings for UML Class diagrams:

- **Pull Up Operation**: The *Pull Up Operation* refactoring pulls an operation out of one or maybe several subclasses up into the super class. This prevents possible duplicated code if a super class has multiple subclasses.

- **Push Down Operation**: The *Push Down Operation* refactoring removes an operation from the super class and pastes it into all its subclasses. It is useful when every subclass needs the operation to behave in its own way and it is not possible to generalize the behaviour in the super class.
- **Extract Class**: The *Extract Class* refactoring splits one class into two separate classes. This is necessary if the functionality provided by one class would have a better logical structure when it was split into two.
- **Generate Subclass**: The *Generate Subclass* refactoring creates a subclass and it copies the operations of the superclass.

Implemented refactorings for UML State Machine diagrams:

- **Introduce Initial Pseudostate**: The *Introduce Initial Pseudostate* refactoring adds an initial pseudostate to a composite state, or region.
- **Introduce Region**: The *Introduce Region* refactoring introduces a region in a State Machine diagram moving to it a set of selected states.
- **Remove Region**: The *Remove Region* refactoring removes a region from a State Machine diagram by moving states and transitions contained by the region to the parent region.
- **Flatten State Transitions**: The *Flatten States Transitions* refactoring replaces an incoming/outgoing transition, connected to a composite state, with the corresponding transitions associated to each sub–states.

The above representative list is inevitable incomplete but gives a clear idea about the diversity of the available model refactorings. Those model refactorings have been chosen in order to show a sufficient variety of graph transformation rules dealing with all the entity types defined in the Type Graph.

Some model refactorings that have not been implemented, present characteristics very similar to the chosen refactorings. For example the *Pull Up Property* refactoring, that pulls a property out of one subclass up into the super class, is functionally equivalent to the *Pull Up Operation* refactoring.

The *Move Operation* refactoring that moves an operation from one class to another and the *Remove Operation* refactoring that removes an operation from a class, are obviously part of the *Extract Class* refactoring.

The *Rename Operation* refactoring –as its name lets assume– renames a chosen operation and, at the source code level, this includes the change of the calls to that operation. This kind of refactoring does not have significant impacts over the chosen graph representation of UML models and could be accomplished using one simple graph transformation rule.

5.2 Refactoring template

In this dissertation, for model refactoring an attempt has been made to find and write down universal descriptions and instructions, which possibly would become an initial model refactoring catalogue.

Each model refactoring has been described with an explanation of when it should be used, how it can be realized, and with a discussion of the list of mechanisms to accomplish the refactoring itself.

The same structure has been adopted and utilized along the dissertation to document each model refactoring. The following describe the purpose of each section:

Scope. This section specifies which type of UML models are affected by the model refactoring. In some cases, the scope of the refactoring has been limited in order to avoid that the transformation leads to an inconsistent model.

Description. It identifies in which case the model refactoring should be applied and gives a general overview of the characteristics, the features, and the behaviour of the model refactoring.

Motivation and Applicability. It gives a detailed explanation of the benefits of the model refactoring and specifies under which conditions the model refactoring can be applied. All these conditions will be formalised in subsequent sections.

Example. The descriptions are illustrated and motivated using a running example, it will serve to demonstrate the applicability of the model refactoring. Of course, such examples can only show certain aspects of the usability of model refactorings, they can not demonstrate their complete functionality and the variety and flexibility of possible applications.

Refactoring Implementation. This section explains the general behaviour of the model refactoring and its implementation. The notation of *UML Interaction Overview diagrams* has been used to formally define the control flow of each model refactoring and to specify in which order the graph transformation rules implemented in AGG must be applied.

Graph Transformation Rules. It gives a detailed description of each single transformation step that are implemented in the case using the AGG general–purpose graph transformation tool.

Consequences. The last section resumes the impacts of the model refactoring. Moreover, it reports some considerations and possible improvements.

5.3 Type Graph extensions

We have extended the Type Graph described in chapter 4.4 in order to better support the model refactorings. Figure 5.1 shows a simplified version of the Type Graph, yellow nodes and edges representing the refactoring entities that have been added for that purpose.

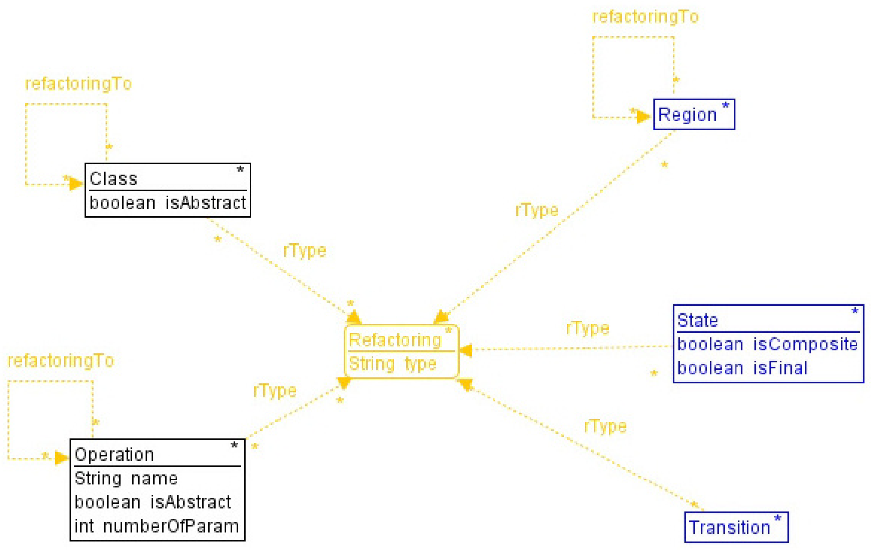


Figure 5.1: Type Graph - Refactoring Entities

The *"Refactoring"* node and the *"rType"* edge specify which type of the refactoring takes place as well as the role of the associated node. The *"refactoringTo"* edge is used to maintain a relationship between different nodes during the execution of different refactoring steps.

An example of their utilization can be seen in figure 5.23: the *"Refactoring"* node states that an *Extract Class* refactoring is going to be applied to the existing class and the *"refactoringTo"* edge specifies which is the extracted class. The refactoring entities are necessary in order to easily match the interested nodes during the subsequent steps of the model refactoring.

The refactoring entities will be removed when the process is completed, the same happens in case of failure.

5.4 Push Down Operation

5.4.1 Scope

The *Push Down Operation* refactoring produces changes on a Class diagram only. Other diagrams are not affected by the transformations, but it is necessary to check that the operation involved in the refactoring is not used by other diagrams.

If the operation involved in the refactoring is used by other diagrams –for example by a State Machine diagram– application of some other complex transformations is necessary to ensure the models being consistent after the refactoring.

5.4.2 Description

The *Push Down Operation* refactoring removes an operation from the super class and pastes it into its subclasses. The refactoring does not give the user the choice whether the operation will be pushed down into all or only a subselection of the subclasses. Another possible solution is to copy the operation only into the subclasses that make use of it.

5.4.3 Motivation and Applicability

This kind of refactoring moves an operation into the subclasses that do not define the operation. This makes sense if every subclass needs the operation to behave in its own way, and the behaviour cannot be generalised in the super class. It is also useful when an operation in a super class is logically better placed as in its deriving classes.

The involved operation must not be referred by any State Machine diagrams otherwise the refactoring will generate an inconsistent model. However, different approaches are possible. In general, an operation is pushed down to specialize its behaviour for each subclass and it does not represent anymore the behaviour of the superclass.

A simplified solution could leave the operation in the superclass in order to preserve the model consistency; this operation will be overridden by the one defined in the subclasses. Another possible solution could interact with the user to determine whether is better to remove the reference to the operation in the State Machine diagram or make use of another operation.

5.4.4 Example

In the example shown in figure 5.2 the *snapshot* operation has been removed from the Viewer class (a) and has been added to the subclasses PhotoPlayer and MoviePlayer (b).

It is reasonable to think that the *snapshot* operation of the MoviePlayer has a different behaviour from the one of the PhotoPlayer. In this case the *Push Down Operation* refactoring is necessary in order to specialize its behaviour for each subclass.

The *snapshot* operation of the PhotoPlayer makes sense only if combined with the *zoomIn* and *rotate* operations, otherwise the snapshot will correspond to the entire picture. If the *snapshot* operation is not necessary for the PhotoPlayer and it is better to define it only in the MoviePlayer, it is possible to apply a *Push Down Operation* refactoring followed by a *Remove Operation* refactoring.

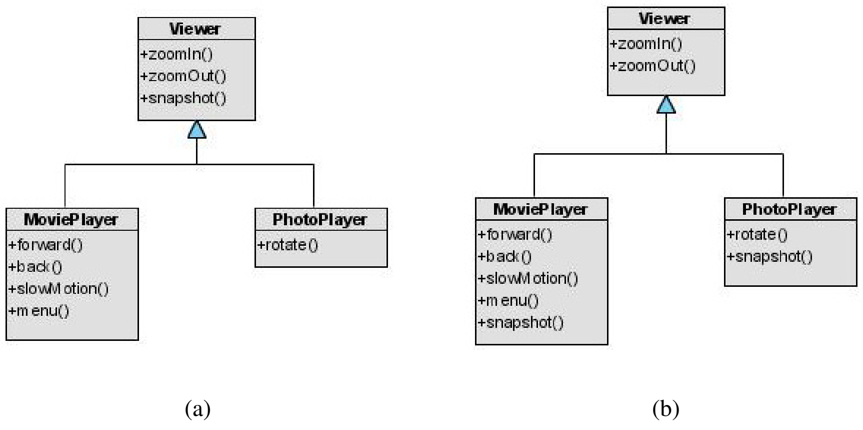


Figure 5.2: UML Class Diagram - Push Down Operation

5.4.5 Refactoring Implementation

Figure 5.3 shows the *Push Down Operation* refactoring as a sequence of primitive refactoring actions. In order to apply the refactoring, it is necessary to provide an input parameter *o* that identifies the operation that has to be pushed down.

The refactoring verifies if it is possible to move the specified operation checking all the necessary preconditions. It verifies that the operation is not referred by a State Machine diagram and is not already defined in the subclasses. If at least one of these preconditions is not respected,

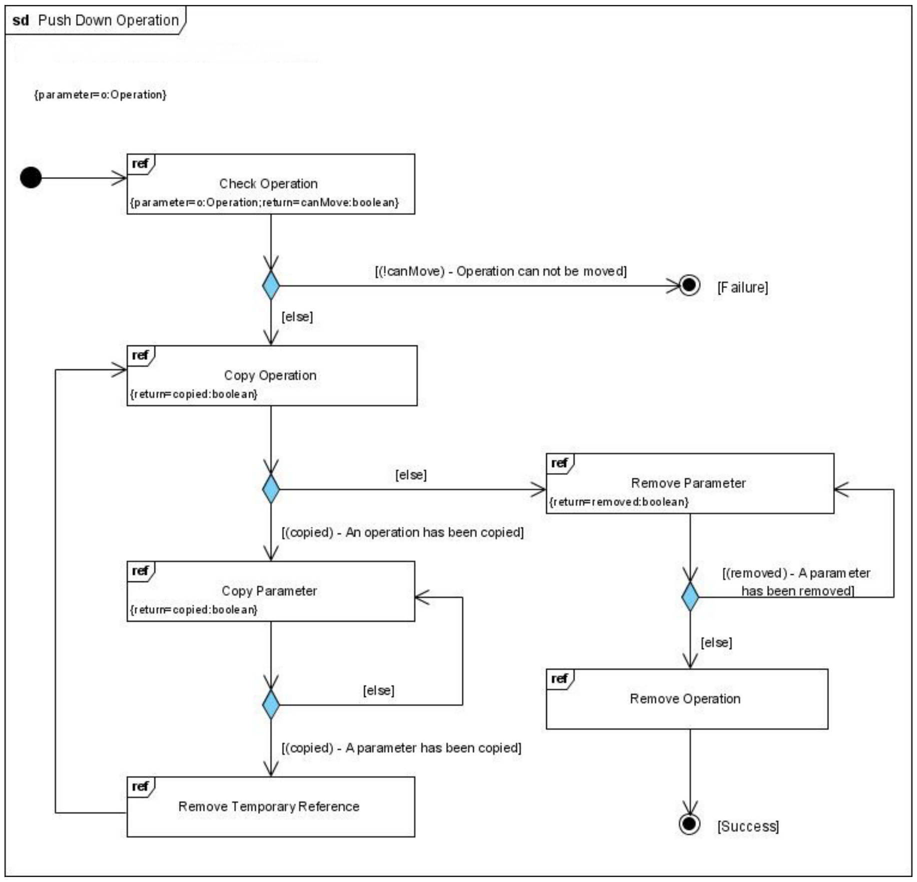


Figure 5.3: Push Down Operation Refactoring

the refactoring will be aborted.

After the verification of all these preconditions the refactoring will copy the operation specified to each subclass; this action can not be performed by executing the step only once but the corresponding graph transformation rules must be repeated for each subclass. Every time when the operation is copied to a subclass, it is also necessary to copy all the defined parameters.

When the operation has been copied to each subclass, it is possible to remove it from the superclass, the parameters defined for it being removed too.

5.4.6 Graph Transformation Rules

The first step named *Check Operation* is shown in figure 5.4. It is composed of two NACs shown in figure 5.5 that verify whether the operation can be pushed down to the subclasses. The operation that is going to be pushed down must be supplied as input parameter to this step.

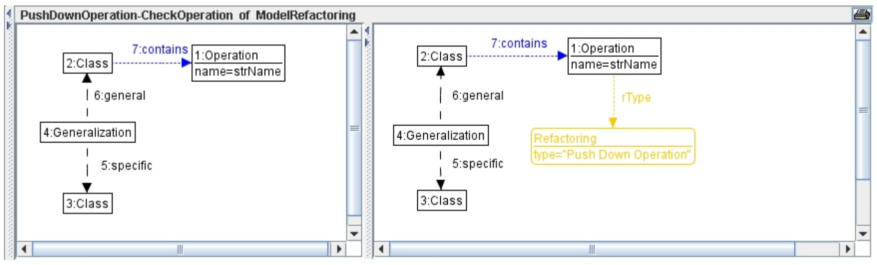


Figure 5.4: Push Down Operation - Check Operation

Input Parameters $o : Operation \Rightarrow 1$

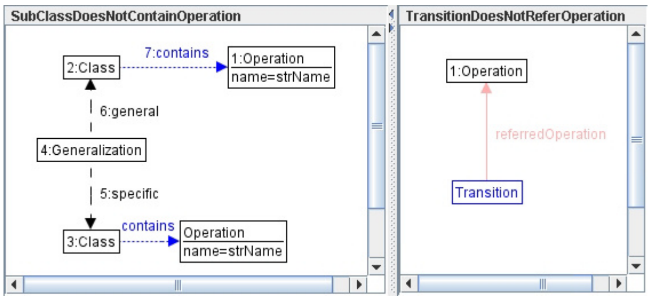


Figure 5.5: Push Down Operation - Check Operation NACs

The NAC *TransitionDoesNotReferOperation* verifies that the operation is not referenced by any transition in the State Machine diagram. The NAC *SubclassDoesNotContainOperation* verifies that the operation is not defined in the subclasses; if at least one of the subclasses contains the operation, the refactoring could not be applied. If the preconditions are respected, the transformation rule marks the operation with a “Refactoring” node in order to recognize it during the execution of the subsequent steps.

The *strName* variable defined in the rule is not an input parameter, but is used by the NACs to search operations with the same name in the subclasses.

The step named *Copy Operation* copies the operation to a subclass. A NAC is defined for this rule in order to avoid that the operation is copied multiple times to same subclass. The copied operation is marked with a “Refactoring” node that identifies it during the next step used to copy the parameters of the operation. This step, together with the one that copies the parameters, must

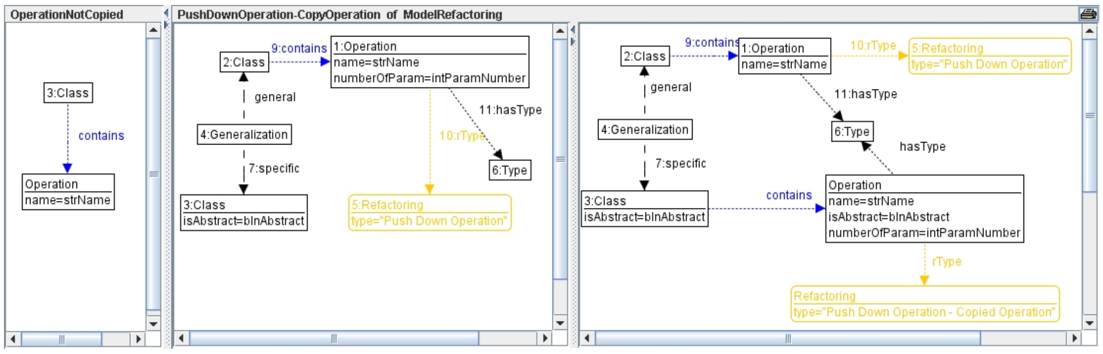


Figure 5.6: Push Down Operation - Copy Operation

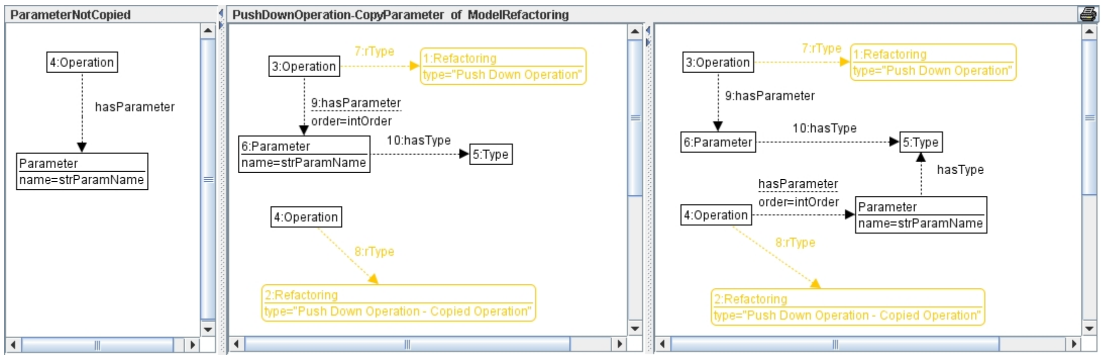


Figure 5.7: Push Down Operation - Copy Parameter

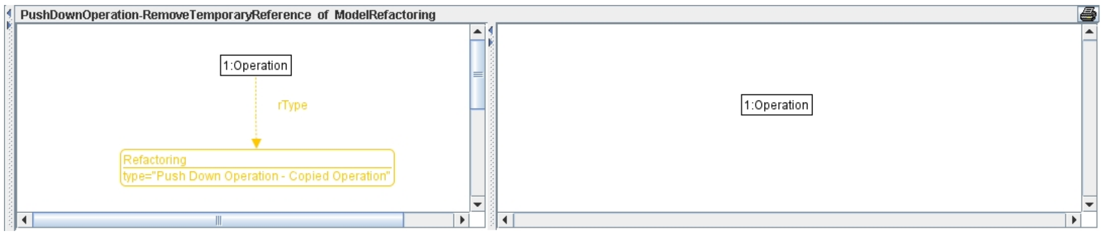

PushDownOperation-RemoveTemporaryReference of ModelRefactoring
1:Operation
rType
Refactoring
type="Push Down Operation - Copied Operation"
1:Operation

Figure 5.8: Push Down Operation - Remove Temporary Reference

be repeated multiple times in order to copy the operation to each subclass.

The transformation rule *Copy Operation* contains some variables that are not defined as input parameters. The *strName* and *intParameterNumber* variables are used respectively to assign the same name and the same number of parameters to the copied operation. The *blnAbstract* variable is used to assign to the operation copied the abstract level defined for the subclass.

The step named *Copy Parameter* copies a parameter from the original operation to the one created in the previous step; a NAC is defined for this rule in order to avoid that it copies multiple times the same parameter. The graph transformation rule must be repeated for each parameter of the original operation.

The transformation rule *Copy Parameter* contains some variables that are not defined as input parameters. The *strParameterName* and *intOrder* variables are used respectively to assign the same name and the same position to the parameter copied.

The step named *Remove Temporary Reference* removes the “Refactoring” node associated to the operation copied. At this point, it is possible to copy the operation to another subclass repeating the same steps.

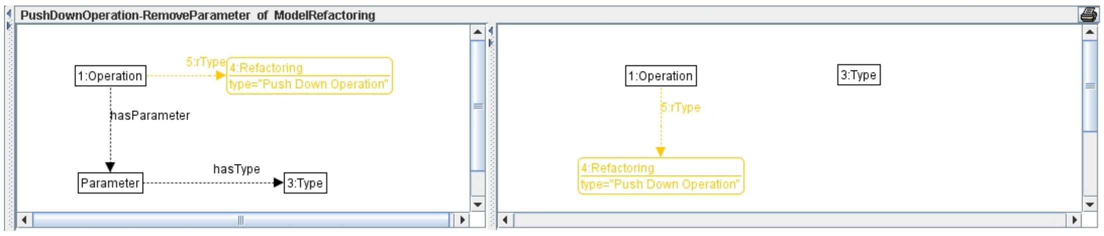

PushDownOperation-RemoveParameter of ModelRefactoring
1:Operation
5:rType
4:Refactoring
type="Push Down Operation"
hasParameter
Parameter
hasType
3:Type
1:Operation
3:Type
5:rType
4:Refactoring
type="Push Down Operation"

Figure 5.9: Push Down Operation - Remove Parameter

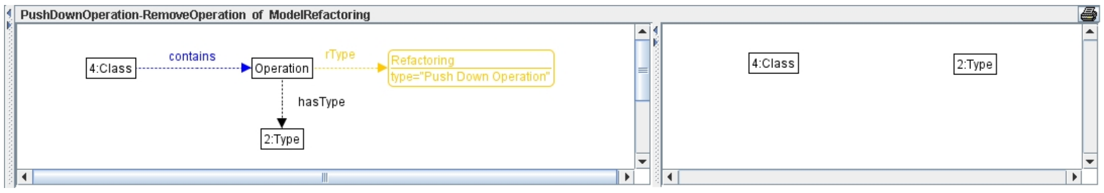

PushDownOperation-RemoveOperation of ModelRefactoring
4:Class
contains
Operation
rType
Refactoring
type="Push Down Operation"
hasType
2:Type
4:Class
2:Type

Figure 5.10: Push Down Operation - Remove Operation

When the operation has been copied to all the subclasses, it is possible to remove it from the superclass. The step named *Remove Parameter* removes a parameter from the original operation. This step must be repeated multiple times in order to remove all the associated parameters.

The last step named *Remove Operation* removes the operation from the superclass and, at the same time, the associated “Refactoring” node.

5.4.7 Consequences

This model refactoring might be extended with the functionality that allows to push down other class members, like properties. This would mean that the refactoring will be renamed from *Push Down Operation* to *Push Down Member*.

In the current state, the refactoring pushes the operation selected into all child classes. Another possible extension might be to let the user select the subclasses into which the operation will be pushed down. In this case, the list of subclasses into which the operation will be pushed down must be supplied to the refactoring as input parameter.

5.5 Pull Up Operation

5.5.1 Scope

The *Pull Up Operation* refactoring produces changes on a Class diagram only. Other diagrams are not affected by the transformations, but it is necessary to check that the operation involved in the refactoring is not used by other diagrams.

If the operation involved in the refactoring is used by other diagrams –for example by a State Machine diagram– application of some other complex transformations is necessary to ensure the models are consistent after the refactoring.

5.5.2 Description

This refactoring is the counterpart of *Push Down Operation*: it copies an operation from the subclasses to a super class, deleting the original. This prevents possible duplicated code if a super class has multiple subclasses.

5.5.3 Motivation and Applicability

The goal of the *Pull Up Operation* refactoring is to move one operation from one or more subclasses to its super class. The easiest case of using this refactoring occurs when the operations have the same body, implying there has been a copy and paste of the code.

The *Pull Up Operation* refactoring arises some limitations of the current graph representation of UML models. According to the standard catalogue for source code refactorings [3], necessary preconditions –like the accessing of properties– must be verified in order to apply successfully this refactoring. The graph representation does not allow to specify whether the operations access the properties, as this concept is not part of the UML Class diagram notation.

The signature of an operation is the combination of the operation name along with the number and types of the parameters and their order. Operations defined in the sibling classes could carry different signatures and, in this case, the operations can not be pulled up. On the contrary, the application of the refactoring will generate inconsistencies at source code level even if the resulting model is correct.

A simplification has been made in the current implementation of the refactoring, that is operations with the same name are considered identicals. The AGG tool does not provide a simple mechanisms to verify that the operations contain the same number of parameters. The operation selected must be compared to the corresponding defined in the sibling classes. If the operations have the same number of parameter, they must be marked as “equivalent”. After that step, it is possible to verify whether operations with the same name but not marked as “equivalent” exist in the sibling classes.

In order to verify that the types of the parameters and their order correspond for each operation, it is necessary to realise two different graph transformation rules, because there is a distinction between primitive types and user–defined types in the graph representation of UML models. The AGG tool does not provide any simple mechanisms to verify this precondition, and it is necessary to compare the parameters of each operations. Let N_c be the number of sibling classes involved in the refactoring and N_p the number of parameters of the selected operation, the computation cost C of the precondition will be $C = 2 \times N_c \times N_p$. Even if it is difficult to have a large number of parameters, this solution appear to be not scalable for large environment system.

If the operations involved in the refactoring are not referenced by any State Machine diagrams the task can be accomplished without problems. On the contrary, some considerations have to be made in order to preserve the consistency of the UML model. A subclass inherits operations from its superclass. When an operation is pulled up the subclass can continue making use of it and the operation is still part of the behaviour of the subclass. That way, it is possible for the State Machine diagrams to refer to the pulled up operation.

The current implementation of this refactoring imposes the precondition that the involved operations must not be referred by a State Machine diagram. This is due to the strict definition of consistency we have defined: “A transition can refer to operations contained only in the class represented by the State Machine diagram”.

5.5.4 Example

In the example shown in figure 5.2 the *snapshot* operation has been removed from the subclasses PhotoPlayer and MoviePlayer (b) and has been added to the Viewer class (a).

If the *snapshot* operation has the same behaviour in both subclasses, it is a good practice to apply a *Pull Up Operation* refactoring in order to move the operation to the super class and to avoid duplication of code.

5.5.5 Refactoring Implementation

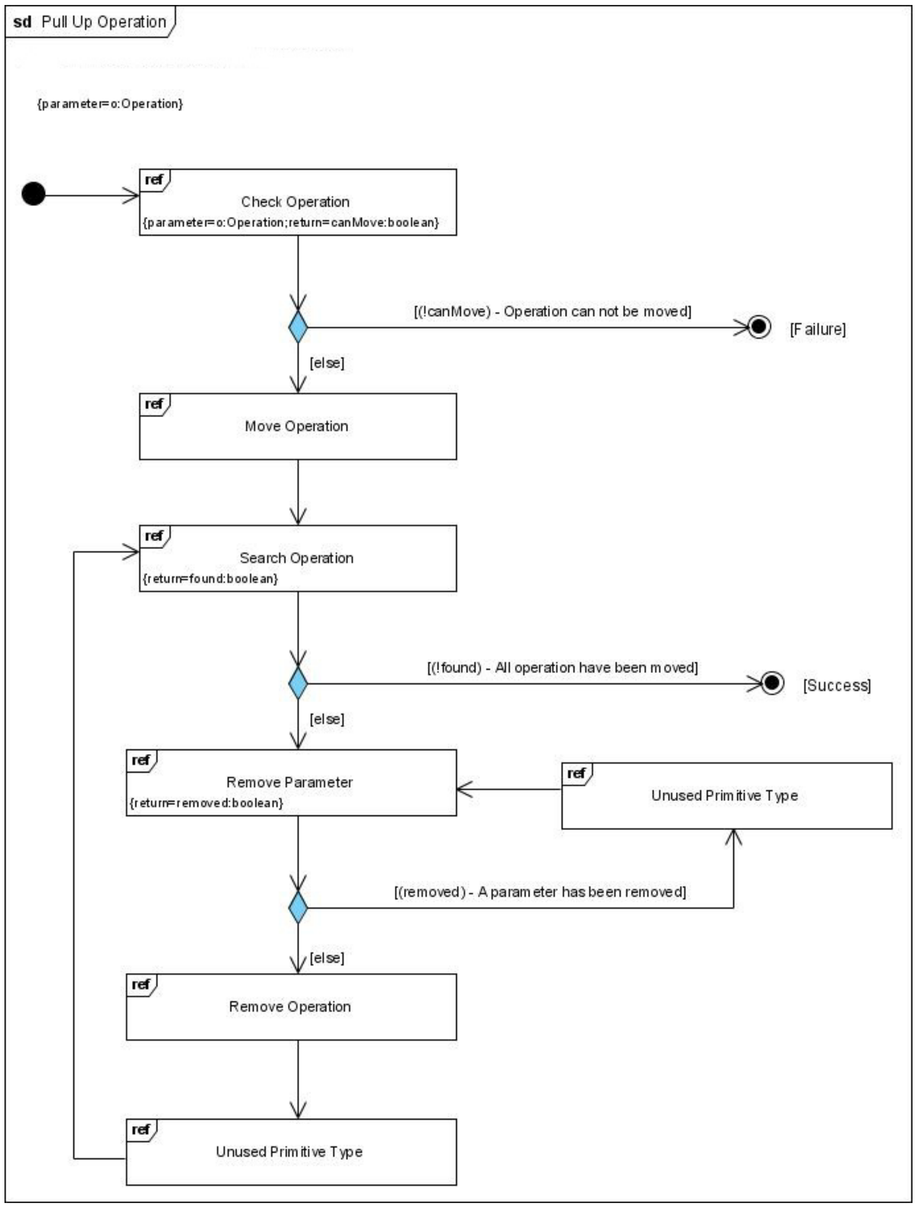

sd Pull Up Operation
{parameter=o:Operation}
ref Check Operation {parameter=o:Operation;return=canMove:boolean}
[(!canMove) - Operation can not be moved]
[Failure]
[else]
ref Move Operation
ref Search Operation {return=found:boolean}
[(!found) - All operation have been moved]
[Success]
[else]
ref Remove Parameter {return=removed:boolean}
ref Unused Primitive Type
[(removed) - A parameter has been removed]
[else]
ref Remove Operation
ref Unused Primitive Type

Figure 5.11: Pull Up Operation Refactoring

Figure 5.11 shows the *Pull Up Operation* refactoring as a sequence of primitive refactoring actions. In order to apply the refactoring, it is necessary to provide an input parameter *o* which identifies the operation that has to be pulled up.

The refactoring verifies whether it is possible to move the specified operation by checking all the necessary preconditions. It verifies that the operation is not referred by a State Machine diagram, that the operation is not already defined in the super class and that the superclass is not abstract. If at least one of these preconditions is not respected, the refactoring will be aborted.

After verification of all these preconditions, the refactoring will move the operation specified to the superclass. When the operation has been moved, it is possible to remove the operation from all the sibling classes, the parameters defined for them have to be removed as well. This action can not be performed by executing the step only once, instead the corresponding graph transformation rules must be repeated for each sibling class.

5.5.6 Graph Transformation Rules

The first step named *Check Operation* is shown in figure 5.12. It is composed of four NACs shown in figure 5.13, that verify whether the operation can be pulled up to the super class. The operation that has to be pulled up must be supplied as input parameter to this step. The *strName* variable defined in the rule is not an input parameter but is used by the NACs to search operations with the same name in the sibling classes and in the super class.

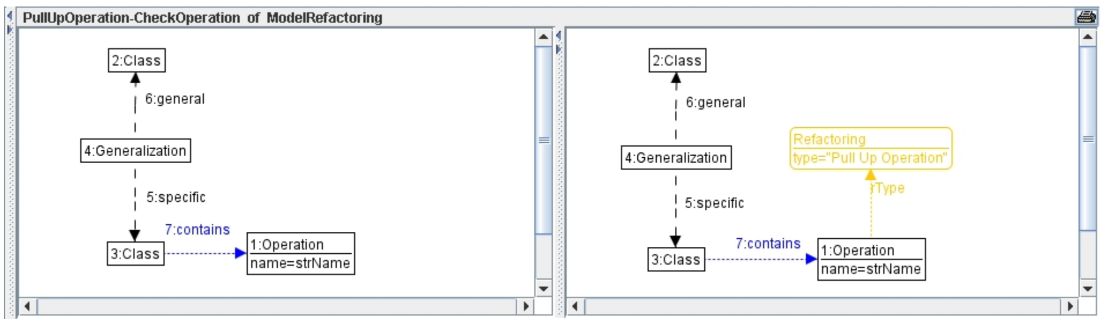
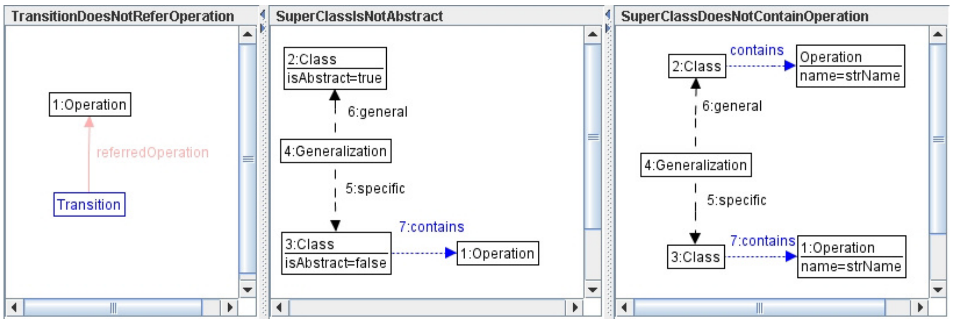


Figure 5.12: Pull Up Operation - Check Operation

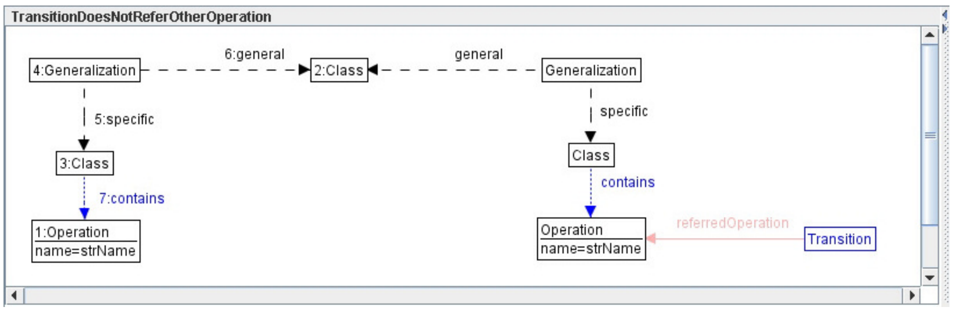
Input Parameters $o : Operation \Rightarrow 1$

The NAC *TransitionDoesNotReferOperation* verifies that the operation is not referenced by any transition in the State Machine diagram. The NAC *SuperClassDoesNotContainOperation* is used to verify that the operation is not already defined in the super class. The NAC *SuperClassIsNotAbstract* verifies that the refactoring is not trying to move a concrete operation to an abstract class. The NAC *TransitionDoesNotReferOtherOperation* verifies that the operation defined in the sibling classes is not referenced by any transition in the State Machine diagram.

If all preconditions are respected, the transformation rule marks the operation with a “Refactoring” node in order to recognize it during the execution of the subsequent steps.



(a)



(b)

Figure 5.13: Pull Up Operation - Check Operation NACs

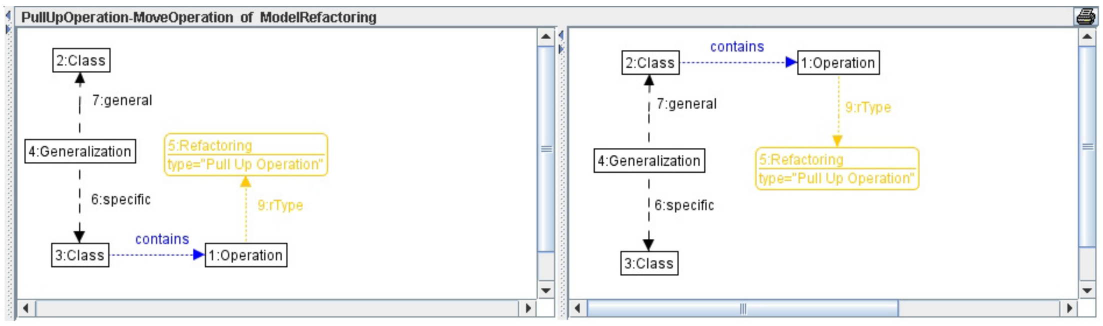


Figure 5.14: Pull Up Operation - Move Operation

The step named *Move Operation* moves the operation selected from the subclass to the super

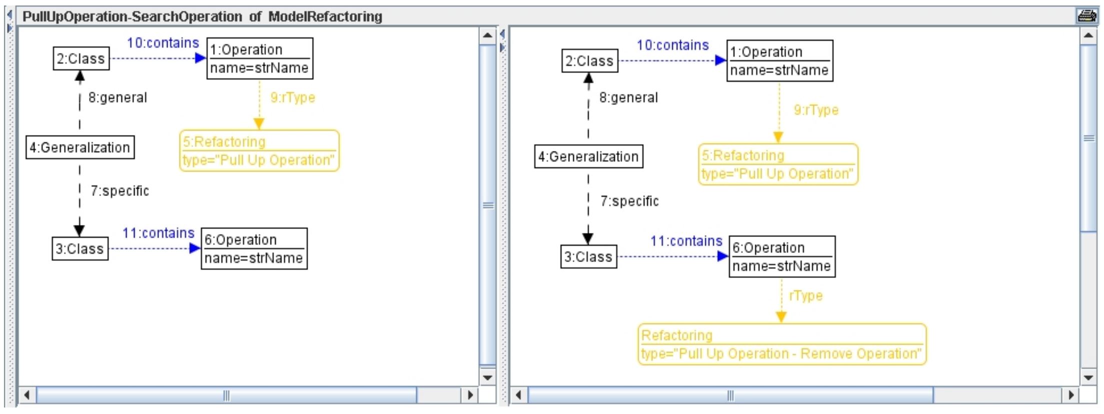


Figure 5.15: Pull Up Operation - Search Operation

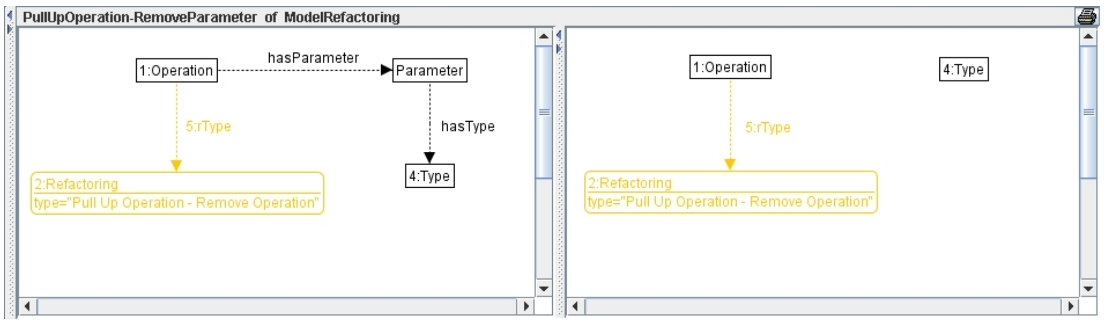

PullUpOperation-RemoveParameter of ModelRefactoring
1:Operation
hasParameter
Parameter
5:rType
hasType
4:Type
2:Refactoring
type="Pull Up Operation - Remove Operation"
1:Operation
4:Type
5:rType
2:Refactoring
type="Pull Up Operation - Remove Operation"

Figure 5.16: Pull Up Operation - Remove Parameter

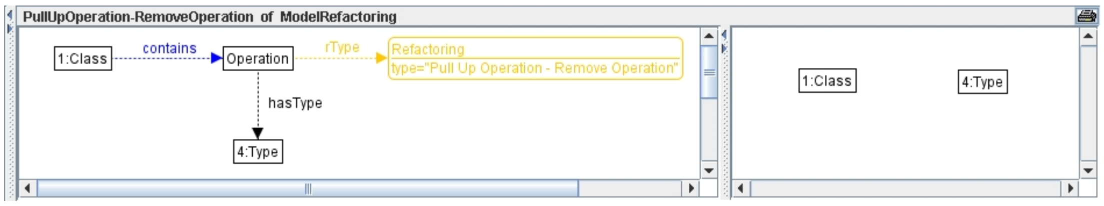

PullUpOperation-RemoveOperation of ModelRefactoring
1:Class
contains
Operation
rType
Refactoring
type="Pull Up Operation - Remove Operation"
hasType
4:Type
1:Class
4:Type

Figure 5.17: Pull Up Operation - Remove Operation

class. The step named *Search Operation* searches an operation with the same signature in the sibling classes; the *strName* variable is used to compare the names of the operations. When an operation is matched, the transformation rule marks it with a “Refactoring” node in order to recognize it during the execution of the subsequent steps. This step must be repeated multiple

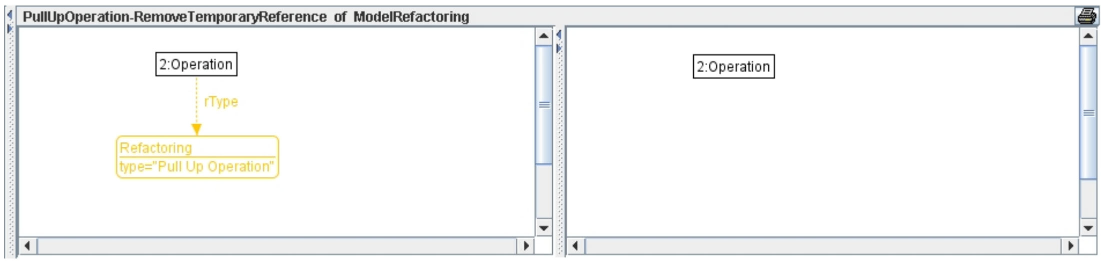


Figure 5.18: Pull Up Operation - Remove Temporary Reference

times in order to verify each sibling class.

If a sibling class contains an operation with the same signature, the operation has to be removed. The step named *Remove Parameter* removes a parameter from the operation matched. This step must be repeated multiple times in order to remove all the parameters associated. After that, removal of the matched operation is possible using the transformation rule *Remove Operation*.

The last step *Remove Temporary Reference* will be executed when the operation has been removed from all sibling classes and it removes the “Refactoring” node associated to the moved operation.

5.5.7 Consequences

This model refactoring might be extended with the functionality to pull up other class members like properties. This would mean that the refactoring will be renamed from *Pull Up Operation* to *Pull Up Member*. Moreover, the refactoring could be improved in order to verify whether the operation with the same name in the sibling classes has the same signature.

The graph representation of UML models should be extended to better verify all the necessary preconditions of this refactoring, as defined in the standard catalogue for source code refactorings [3].

5.6 Extract Class

5.6.1 Scope

The *Extract Class* refactoring produces changes on a UML Class diagrams only. Other diagrams are not affected by the transformations, but it is necessary to check that the classes and the operations involved in the refactoring are not used by other diagrams.

If the classes or the operations involved in the refactoring are used by other diagrams –for example by a State Machine diagram– application of some other complex transformations is necessary to ensure the models are consistent after the refactoring.

5.6.2 Description

The *Extract Class* refactoring splits one class into two separate classes. This is necessary if the functionality provided by one class would have a better logical structure when it is split into two. This will improve cohesion and, thereby, the structure of the system.

5.6.3 Motivation and Applicability

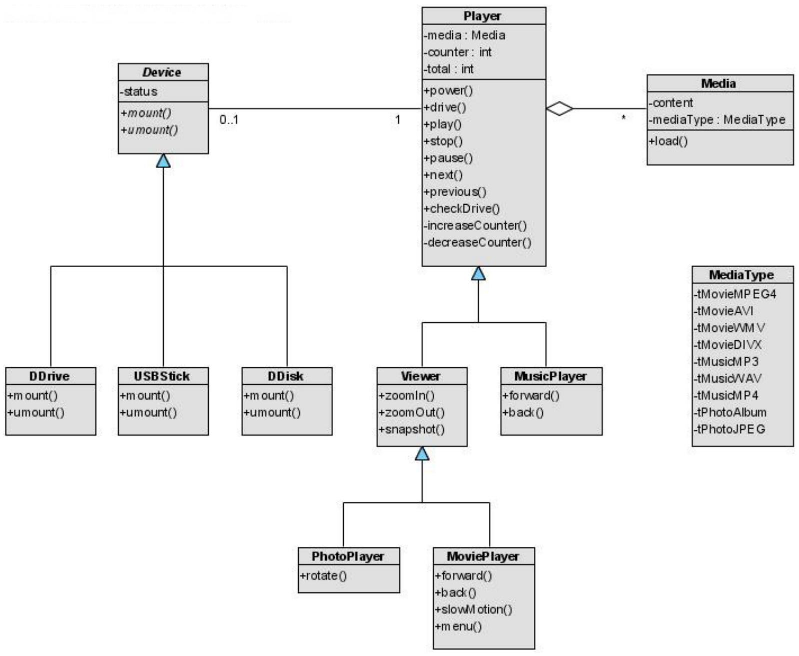
This kind of refactoring evolves and improves the structure of the UML models and –as suggested by the name– it extracts a class from an existing one exporting the operations and properties specified.

Sometimes, it is normal to start the design using a little set of classes and add the operations to that primitive classes. A more detailed analysis could show that some operations added to a class may be better encapsulated to an external class used by the existing one. In this case, it is necessary to create a new class and to export the operations to the newly created class. This actions is possible only if the operations are not referenced by any State Machine diagram.

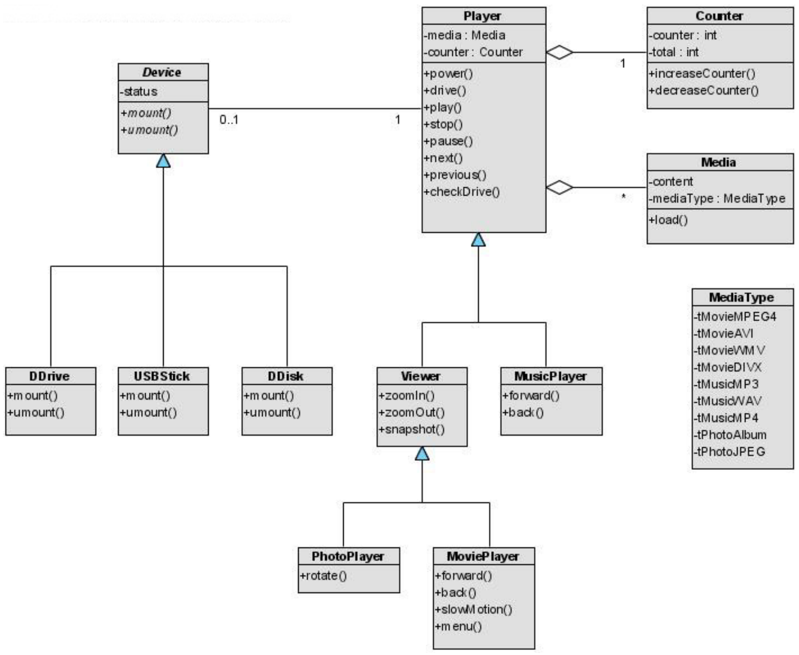
The refactoring may be also used to extract a subclass that will be inserted in the inheritance chain of the original class. In this case the class extracted and its operations represent a specialization of the original class.

5.6.4 Example

Figure 5.19 shows an example of the Extract Class refactoring applied to the Player application. In figure 5.19(a), the Player class contains two operations used to manage the internal counter



(a) Before Refactoring



(b) After Refactoring

Figure 5.19: UML Class Diagram - Extract Class

–*increaseCounter* and *decreaseCounter*– that contain the logic and the internal representation of the counter. The player does not need to know that informations and it just needs a way to access the counter.

A better design of the Player application could be done creating a class that encapsulates the counter's logic and that supplies the necessary operations to manage and access the counter. The Player class uses the operations exposed by the Counter class and does not know its internal representation. This kind of design is shown in figure 5.19(b).

The second class diagram could be obtained applying an Extract Class refactoring. A new class, named *Counter*, has been created and the interested operations, *increaseCounter* and *decreaseCounter*, have been exported to the newly created class. The variables *counter* and *total* previously defined in the Player class, respectively represent the current value of the counter and the maximum allowed value. They have been exported to the Counter class. Moreover, a variable of type Counter has been defined in the Player Class.

5.6.5 Refactoring Implementation

Figure 5.20 shows the Extract Class refactoring as a sequence of primitive refactoring actions. In order to apply the refactoring, it is necessary to provide input parameters that identify the class, the operations and the properties involved in the transformations. As reported in figure 5.20 they are:

- c is the existing source class that contains the properties and the operations.
- $strName$ is the name of the new class that has to be created by the refactoring.
- $setO[]$ is the set of operations that has to be exported to the new class.
- p is the property owned by the existing class for which type has to be changed.
- $setP[]$ is the set of properties that have to be exported to the new class.
- $blnSubclass$ specifies whether the new class has to be a subclass of the existing one.

The first step verifies that all the involved operations can be exported to the new class; if at least one of them raises a problem, the refactoring will be aborted. If all the operations specified can be exported, the refactoring continues verifying that the diagram does not contain another class with the name $strName$. If those requirements are respected, the new class can be created.

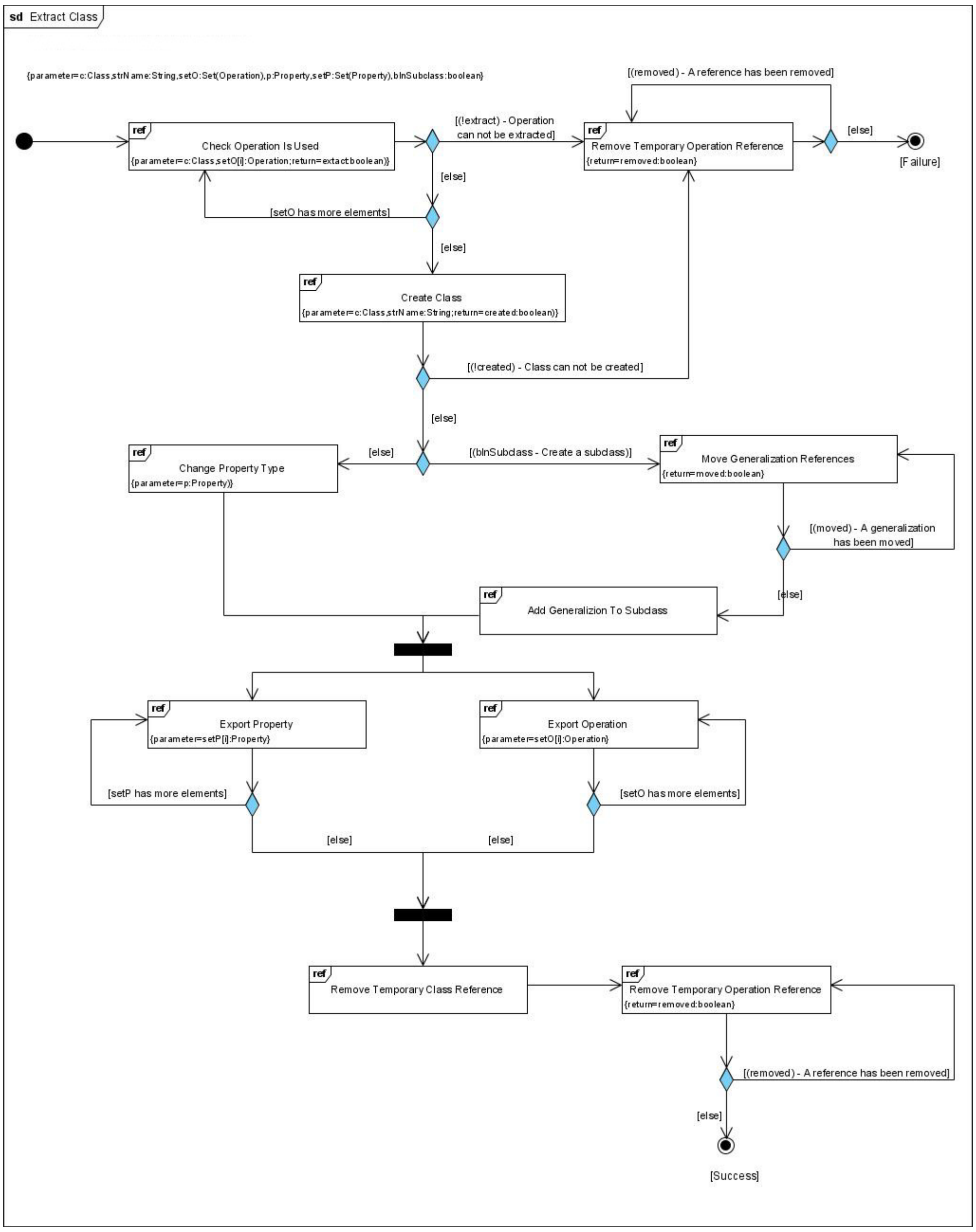

sd Extract Class
{parameter=c:Class,strName:String,setO:Set(Operation),p:Property,setP:Set(Property),blnSubclass:boolean}
ref Check Operation Is Used {parameter=c:Class,setO[i]:Operation;return=extact:boolean}}
[(!extract) - Operation can not be extracted]
ref Remove Temporary Operation Reference {return=removed:boolean}
[(removed) - A reference has been removed]
[else]
[Failure]
[else]
[setO has more elements]
[else]
ref Create Class {parameter=c:Class,strName:String;return=created:boolean)}
[(!created) - Class can not be created]
[else]
ref Change Property Type {parameter=p:Property)}
[else]
[(blnSubclass - Create a subclass)]
ref Move Generalization References {return=moved:boolean}
[(moved) - A generalization has been moved]
[else]
ref Add Generalizion To Subclass
ref Export Property {parameter=setP[i]:Property}
ref Export Operation {parameter=setO[i]:Operation}
[setP has more elements]
[setO has more elements]
[else]
[else]
ref Remove Temporary Class Reference
ref Remove Temporary Operation Reference {return=removed:boolean}
[(removed) - A reference has been removed]
[else]
[Success]

Figure 5.20: Extract Class Refactoring

The refactoring lets the user choose whether the new class has to be an external class or a subclass of the existing one. In the first case, it will change the property type that will be associated to the new class. In the second case, it will introduce the new class in the inheritance chain of the original class.

The refactoring continues exporting the operations and properties specified to the newly created class. The actions *Export Property* and *Export Operation* do not require any exact execution order and they may also be executed in parallel. After these steps, it is possible to remove all the "Refactoring" nodes added to the graph during the refactoring process.

5.6.6 Graph Transformation Rules

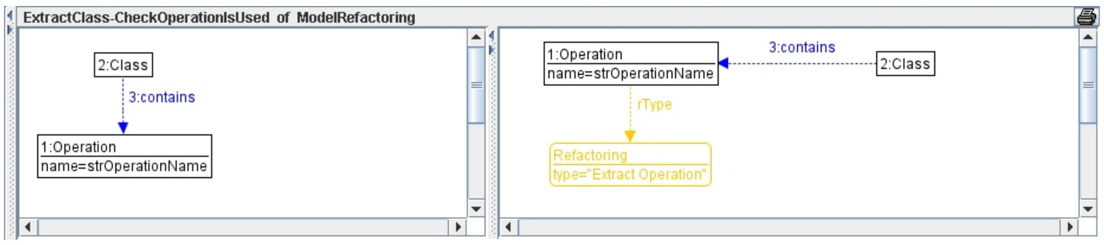


Figure 5.21: Extract Class - Check Operation Is Used

Input Parameters $setO[i] : Operation \Rightarrow 1, c : Class \Rightarrow 2$

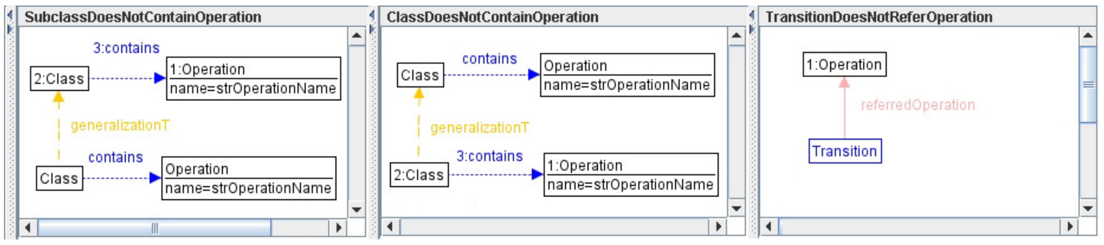


Figure 5.22: Extract Class - NACs

The first step, named *Check Operation Is Used*, is shown in figure 5.21. It is composed of three NACs shown in figure 5.22 that verify if the operation can be exported to another class. The source class and an operation must be supplied to this step as input parameters.

The *TransitionDoesNotReferOperation* NAC verifies that the operation is not referenced by a State Machine diagram, otherwise the refactoring will lead to an inconsistent diagram when the operation is moved to another class.

The *ClassDoesNotContainOperation* and *SubclassDoesNotContainOperation* NACs verify that the operation is not defined elsewhere in the hierarchy chain of the class. For example, if the operation is defined in other class of the source class hierarchy, and the operation is exported to a newly created class, the behaviour of the program may change due to the automatical inheritance mechanisms.

The *strOperationName* variable defined in the rule is not an input parameter, but it is used by the NACs to search operations with the same name in the class hierarchy.

If all those preconditions are respected, the refactoring step marks the operation with a "Refactoring" node that allows to recognize it during the execution of the subsequent steps. This step must be repeated for each operation that is going to be moved to the new class matching every time the corresponding nodes.

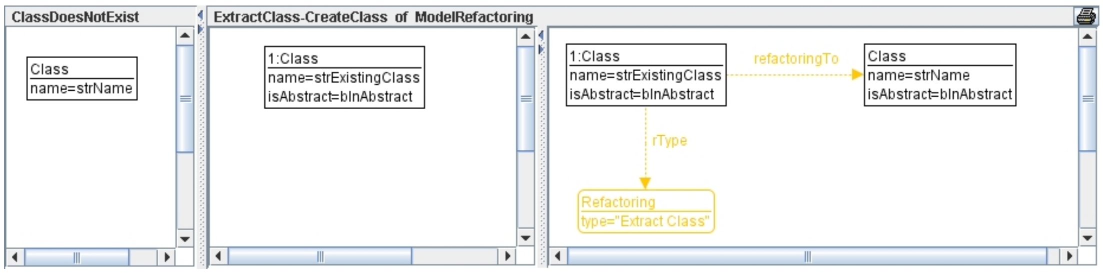


Figure 5.23: Extract Class - Create Class

Conditions $!(strName.equals(strExistingClass))$

Input Parameters $c : Class \Rightarrow 1, strName : String$

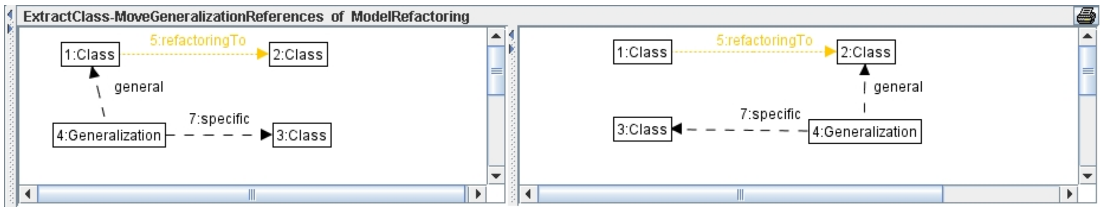


Figure 5.24: Extract Class - Move Generalization References

The step shown in figure 5.23 verifies whether the name specified for the new class is not defined in the diagram; this action is accomplished by the NAC named *Check Class Exists*. The name that has to be assigned to the new class and the source class must be provided as input

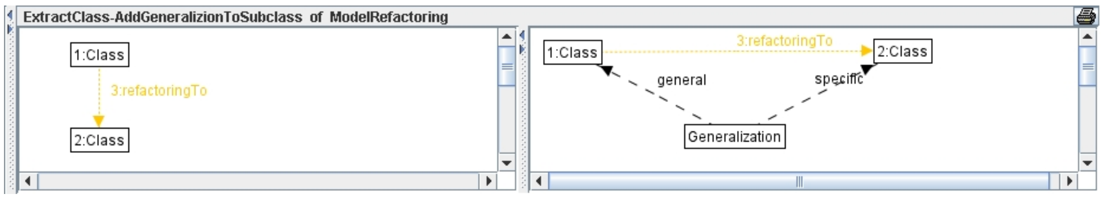


Figure 5.25: Extract Class - Add Generalization To Subclass

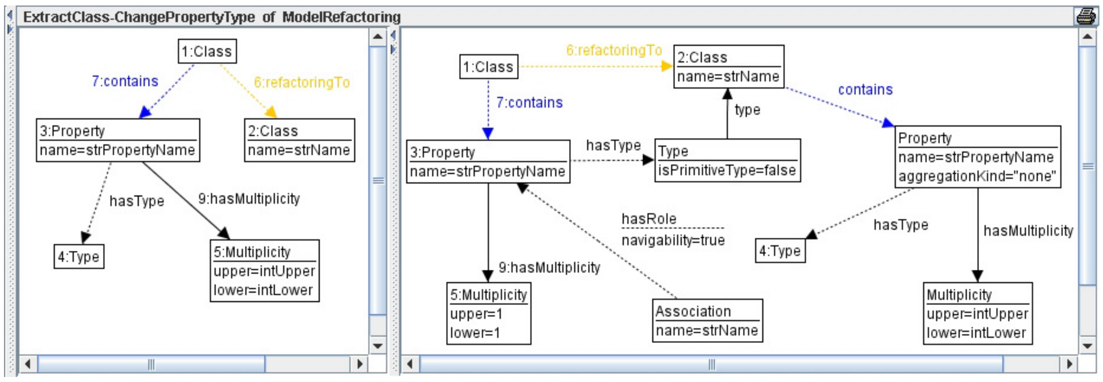


Figure 5.26: Extract Class - Change Property Type

Input Parameters $p : Property \Rightarrow 3$

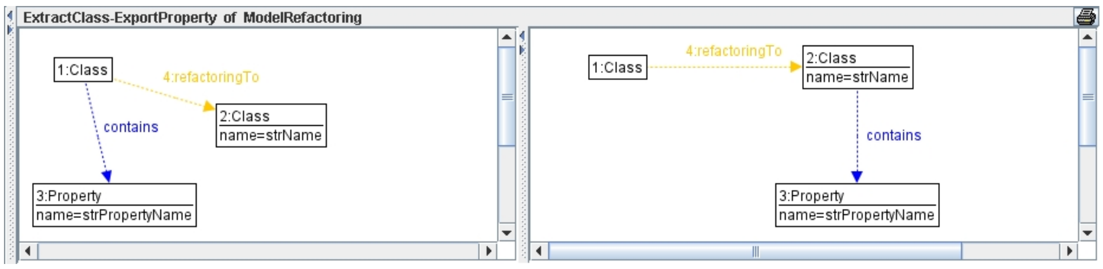


Figure 5.27: Extract Class - Export Property

Input Parameters $setP[i] : Property \Rightarrow 3$

parameters to this step.

Some variables are defined in the graph transformation rule. The *strName* variable represents the name specified for the new class, the *strExistingClass* variable represents the name of the source class and the *blnAbstract* variable is used to set the same abstraction level of the source

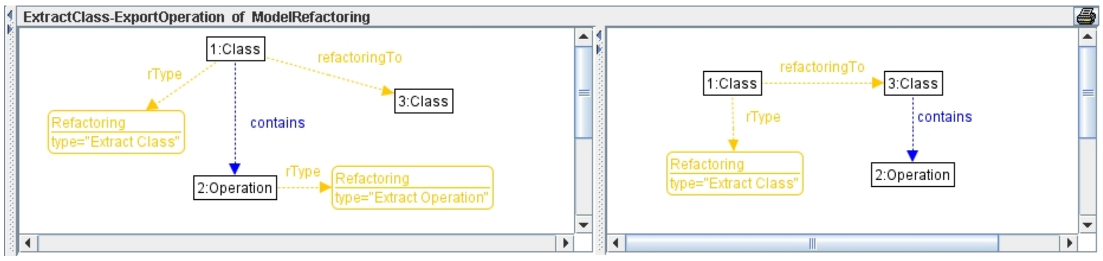


Figure 5.28: Extract Class - Export Operation

Input Parameters $setO[i] : Operation \Rightarrow 2$

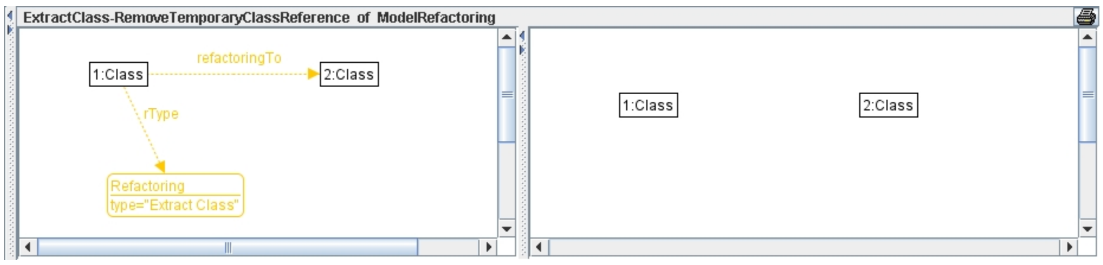


Figure 5.29: Extract Class - Remove Temporary Class Reference

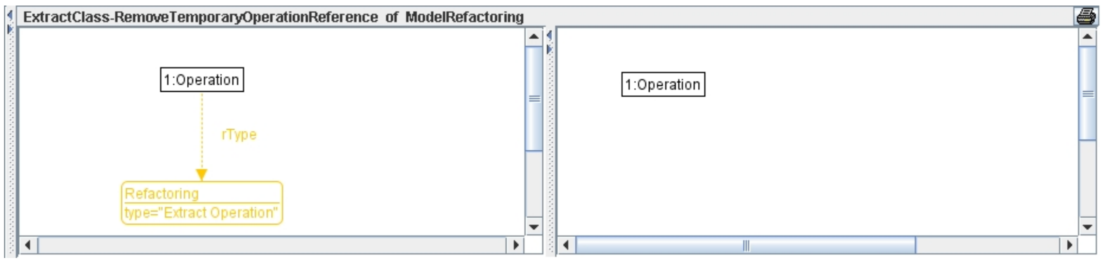

ExtractClass-RemoveTemporaryOperationReference of ModelRefactoring
1:Operation
rType
Refactoring
type="Extract Operation"
1:Operation

Figure 5.30: Extract Class - Remove Temporary Operation Reference

class to the new one. The condition $!(strName.equals(strExistingClass))$ has been defined in order to verify that the specified name does not correspond to the name of the source class.

If these tests are successful, the graph transformation rule creates the new class and marks the source class with a “Refactoring” node that allows its recognition during the execution of the subsequent steps.

Figure 5.26 shows the step that takes in charge the property provided as input parameter. The graph transformation rule modifies the property specified and assignes to it the newly created class as type. Moreover, it creates a new property in the new class with the same characteristics of the one supplied as input parameter. All the variables defined in the graph transformation rule are necessary in order to copy the attributes of the existing property to the new one. This step is executed only in the case it is necessary to create an external class.

The step named *Move Generalization References* shown in figure 5.24 changes the hierarchy chain of the class. The subclasses of the source class will become subclasses of the newly created class. The step named *Add Generalization To Subclass* shown in figure 5.25 adds a generalization relationship between the newly created class and the source class. The new class will become a subclass of the existing one. The steps in figures 5.24 and 5.25 are executed only in the case that the new class must be a subclass of the existing one.

The step shown in figure 5.27 exports a property supplied as input parameter to the newly created class. If the property specified does not belong to the source class, no changes will be made to the diagram. However, a different implementation could choose to abort the refactoring. This step must be executed for each property defined in the set of properties supplied as input parameter to the refactoring, matching every time the corresponding nodes.

The step shown in figure 5.28 exports an operation belonging to the source class to the newly created class. This step must be executed for each operation defined in the set of operations supplied as input parameter, matching every time the corresponding nodes.

Figure 5.29 shows the *Remove Temporary Class Reference* step that removes the "Refactoring" node added to the source class during the process.

Figure 5.30 shows the *Remove Temporary Operation Reference* step that removes the "Refactoring" nodes added to the operations exported during the process. If the refactoring is aborted before its completion, execution of this step is necessary in order to restore the original state of the diagram.

5.6.7 Consequences

This refactoring evolves and improves the structure of the UML models, but it can only be applied to simple models where the operations are not referenced by State Machine diagrams and the class inheritance is not used. In chapter 8, one further solution will be analysed in order to apply this refactoring and preserve the consistency among the different kinds of UML diagrams.

5.7 Generate Subclass

5.7.1 Scope

The *Generate Subclass* refactoring produces changes on a UML Class diagrams only. Other diagrams are not affected by the transformations, but it is necessary to check that the involved class has not an associated State Machine diagram.

5.7.2 Description

This kind of refactoring differs from the *Extract Subclass* refactoring. Indeed, it generates a new subclass using the source class as a template. If the existing class is a concrete class, this will be transformed in an abstract class. All operations defined in the superclass will be copied to the new subclass.

5.7.3 Motivation and Applicability

During the design phase, it may occur that a class needs one or more specializations that make use of the same set of operations. It is possible to add the classes needed as subclasses of the original one. This way the existing "Association" relationships among classes defined in the UML Class diagrams will be preserved.

The current implementation of the refactoring imposes that the existing class is transformed to an abstract class. However, it is possible to let the user choose where this change is necessary.

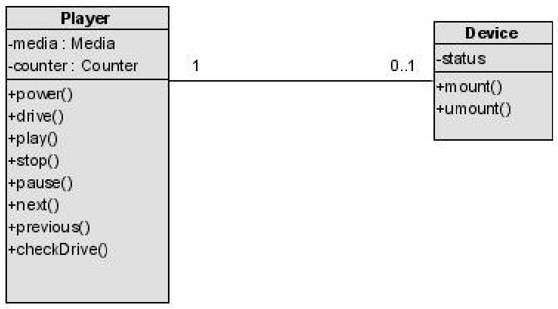
In the UML Sequence diagrams a *message send* represents a call to an instance of remote operations. After application of the refactoring, the calls specified to the operations of the super class are still valid. This is due to the *polymorphism* of the Object–Oriented programming languages.

Polymorphism means the ability of a single variable of a given type to be used to reference objects of different types, and automatically call the operation that is specific to the type of object which the variable references.

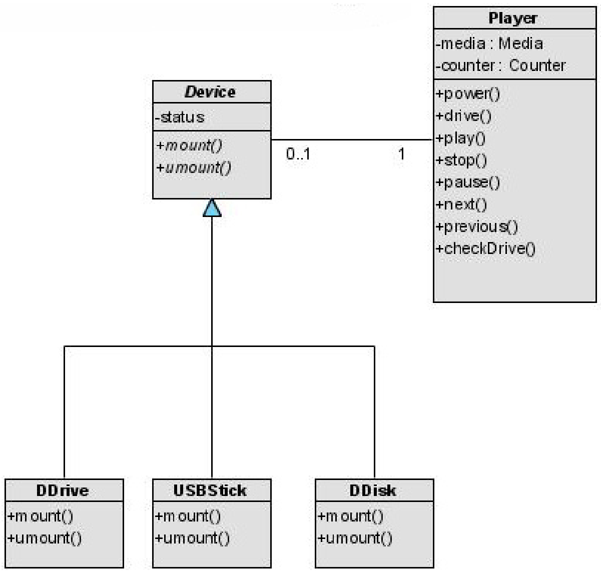
The classes that call operations of the existing class will be still correct after the refactoring. An instance of one of the subclasses created must be supplied to them instead of an instance of the superclass.

If a State Machine diagram exists for the class, the refactoring can not be applied. This is due to the fact that an abstract class can not have any associated State Machine diagram. A different approach is possible: the State Machine diagram can be removed from the existing class and associated to the newly created subclass.

5.7.4 Example



(a)



(b)

Figure 5.31: UML Class Diagram - Generate Subclass

Figure 5.31 shows an example of this kind of refactoring. The *Device* class is used by the *Player* class and represents the physical device that contains the media. In figure 5.31(a) the *Device* class is a concrete class.

If multiple kinds of Device are needed, a suitable solution is the specialisation of the *Device* class. Figure 5.31(b) shows the Class diagram after the application of the *Generate Subclass* refactoring. It has been applied three times in order to generate the subclasses *DDisk*, *DDrive* and *USBStick*. The *Device* class has become an abstract class. All subclasses implement the operations defined in the superclass.

The operations of the *Player* class that call operations of the *Device* class do not need to be modified. An instance of the new subclasses will be supplied to the *Player* class instead of an instance of the *Device* class.

5.7.5 Refactoring Implementation

Figure 5.32 shows the *Generate Subclass* refactoring as a sequence of primitive refactoring actions. In order to apply the refactoring, it is necessary to provide the input parameter c that identifies the source class and the input parameter $strName$ that will be the name of the subclass generated.

The refactoring verifies whether it is possible to complete the action checking all the necessary preconditions. It verifies that the class does not have any associated State Machine diagram and that no classes exist with the specified name. If at least one of these preconditions is not respected, the refactoring will be aborted.

If these requirements are respected, the new class can be created. Subsequently, the refactoring copies the operations of the existing class to the subclass. All parameters of each operation must be copied, too. This action can not be performed executing the step only once, instead the corresponding graph transformation rule must be repeated for each operation.

The operations of the existing class are transformed to abstract operations and the class itself is transformed to an abstract class.

5.7.6 Graph Transformation Rules

The first step, named *Create Subclass*, is shown in figure 5.33. It is composed of two NACs shown in figure 5.34. The source class must be supplied as input parameter to this step. The NAC *ClassDoesNotExist* verifies that the name specified as input parameter does not correspond to the name of an existing class. The NAC *StateMachineDoesNotExist* verifies that a State Machine diagram is not defined for the source class. The *strExistingName* variable defined in the rule is not an input parameter, it corresponds to the name of the existing class.

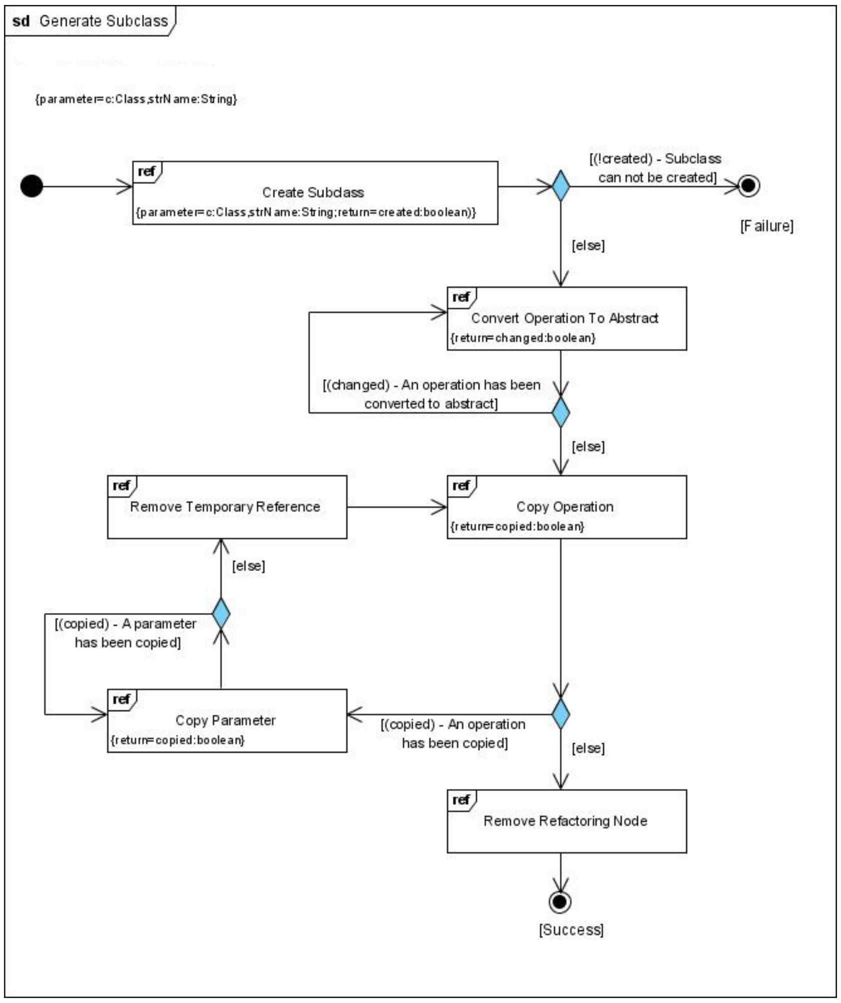

sd Generate Subclass
{parameter=c:Class,strName:String}
ref
Create Subclass
{parameter=c:Class,strName:String;return=created:boolean)}
[(!created) - Subclass can not be created]
[Failure]
[else]
ref
Convert Operation To Abstract
{return=changed:boolean}
[(changed) - An operation has been converted to abstract]
[else]
ref
Remove Temporary Reference
ref
Copy Operation
{return=copied:boolean}
[else]
[(copied) - A parameter has been copied]
ref
Copy Parameter
{return=copied:boolean}
[(copied) - An operation has been copied]
[else]
ref
Remove Refactoring Node
[Success]

Figure 5.32: Generate Subclass Refactoring

If all preconditions are respected, the refactoring step creates the new subclass and marks the source class with a “Refactoring” node that allows its recognition during the execution of the subsequent steps. The source class is modified in order to be an abstract class.

The step *Convert Operation To Abstract* transforms an operation of the source class in an abstract operation. The graph transformation rule must be repeated for each operation of the source class.

The step named *Copy Operation* copies an operation to the subclass; a NAC is defined for this rule in order to avoid that the operation is copied multiple times. This step together with

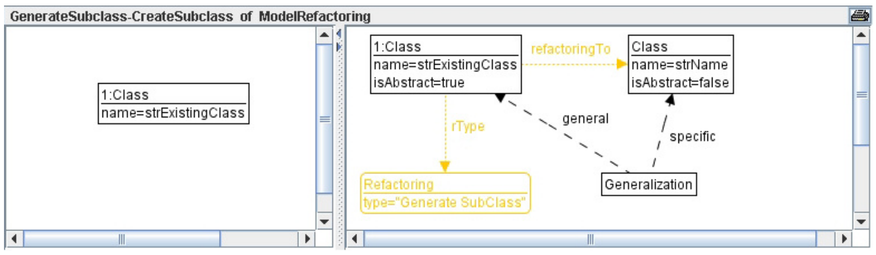


Figure 5.33: Generate Subclass - Create Subclass

Conditions $!(strName.equals(strExistingClass))$

Input Parameters $c : Class \Rightarrow 1$

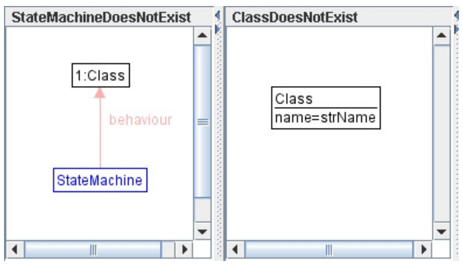


Figure 5.34: Generate Subclass - Create Subclass NACs

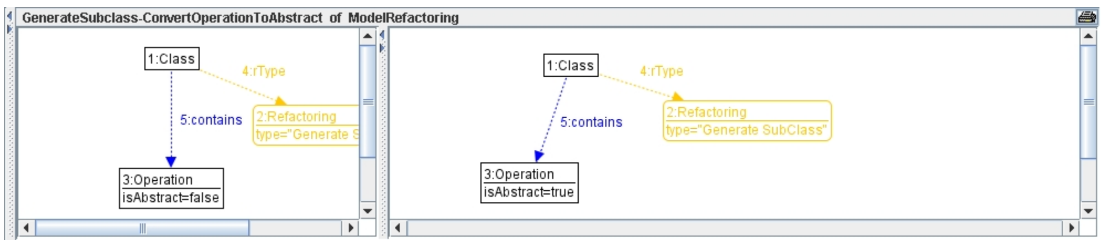


Figure 5.35: Generate Subclass - Convert Operation To Abstract

the one that copies the parameters must be repeated multiple times in order to copy all the operations.

The transformation rule *Copy Operation* contains some variables that are not defined as input parameters. The *strOperationName* and *intParam* variables are used to assign respectively the same name and the same number of parameters to the operation copied.

The step named *Copy Parameter* is shown in figure 5.37. It copies a parameter from the

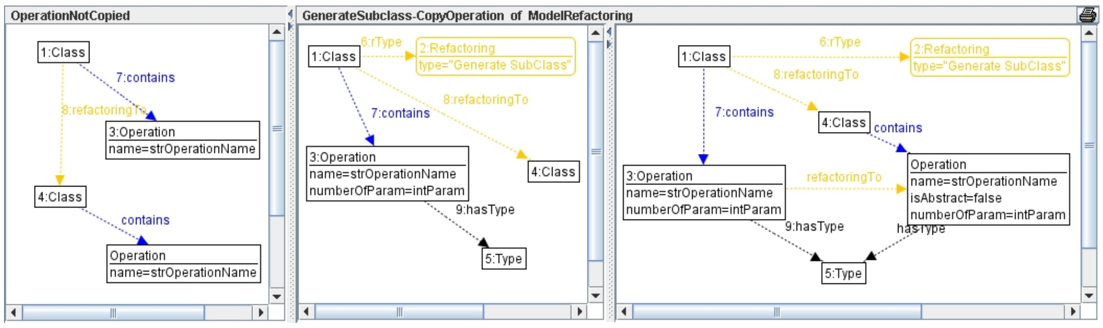


Figure 5.36: Generate Subclass - Copy Operation

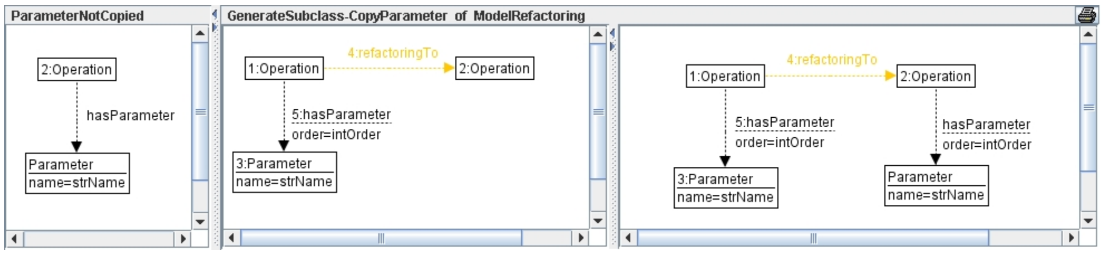


Figure 5.37: Generate Subclass - Copy Parameter

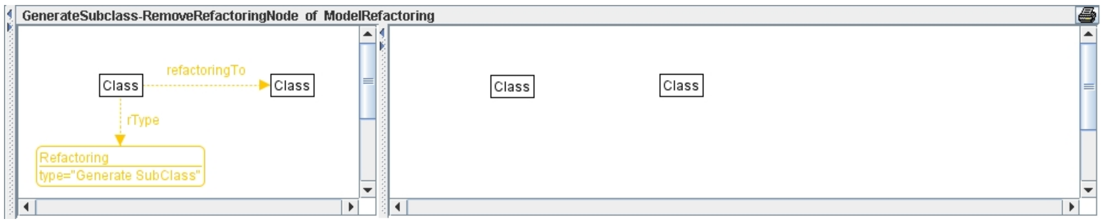


Figure 5.38: Generate Subclass - Remove Refactoring Node

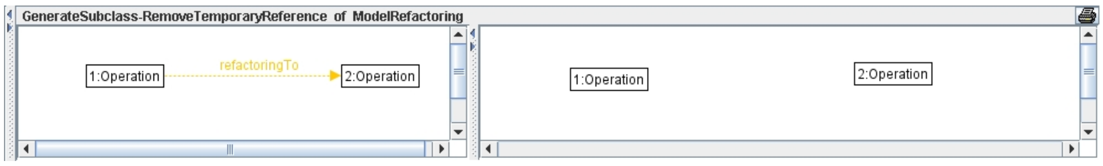

GenerateSubclass-RemoveTemporaryReference of ModelRefactoring
1:Operation
refactoringTo
2:Operation
1:Operation
2:Operation

Figure 5.39: Generate Subclass - Remove Temporary Reference

operation contained in the source class to the one created in the previous step. A NAC is defined for this rule in order to avoid that it copies multiple times the same parameter. The graph transformation rule must be repeated multiple times in order to copy each parameter of the original operation.

The transformation rule *Copy Parameter* contains some variables that are not defined as input parameters. The *strName* and *intOrder* variables are used to assign respectively the same name and the same position to the parameter copied.

The step named *Remove Temporary Reference* removes the “Refactoring” edge associated to the operation involved in the previous steps. At this point, it is possible to copy another operation from the source class to the subclass. The last step named *Remove Refactoring Node* removes the “Refactoring” node associated to the source class.

5.7.7 Consequences

This refactoring improves the structure of the UML model and allows to easily add subclasses to an existing class.

The *Generate Subclass* refactoring may be combined with a *Push Down Property* refactoring in order to move properties defined in the superclass to the subclasses.

5.8 Introduce Initial Pseudostate

5.8.1 Scope

This kind of refactoring produces changes on a State Machine Diagram only. Other diagrams are not affected by the transformation.

5.8.2 Description

The *Introduce Initial Pseudostate* refactoring is used to improve the structure of a State Machine diagram. As suggested by the name, it adds an initial pseudostate to a composite state, or region.

5.8.3 Motivation and Applicability

When all the incoming transitions of a region have the same target state, the diagram can be simplified by adding an initial pseudostate to the region and by setting the region itself as target of the incoming transitions.

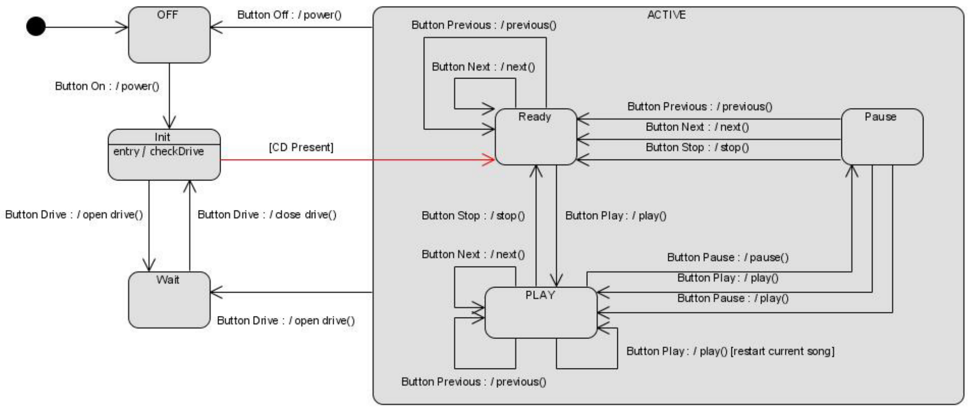
This kind of refactoring does not produce any changes over other kinds of diagrams because it does not modify the structure of the UML model. No check over the Class diagrams is needed because the operations associated to the transitions involved in the refactoring will not be modified.

5.8.4 Example

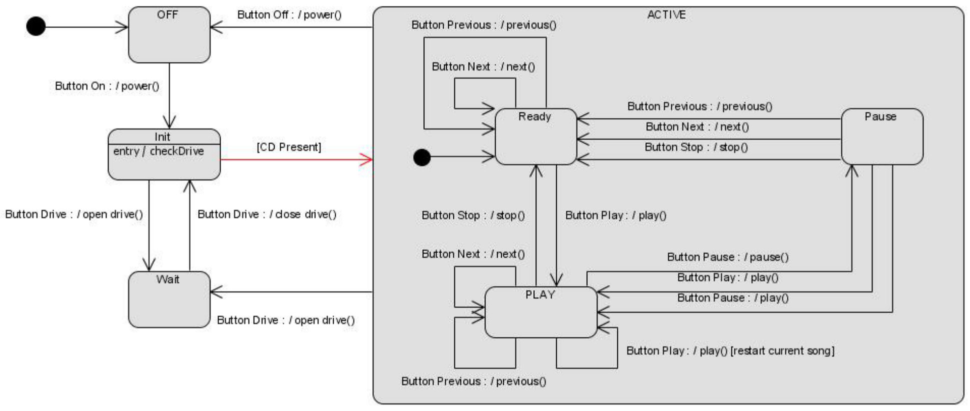
Figure 5.40 shows a simple example of this kind of refactoring. An initial pseudostate has been added to the *ACTIVE* composite state and the target of the transition –that initially refers to the *Ready* state– has been changed to become the region itself. An automatic transition has been defined between the initial pseudostate and the *Ready* state.

The *Ready* state will become the default initial state of the *ACTIVE* region; a transition whose target is the *ACTIVE* state will lead the State Machine to the *Ready* state.

The graph representation of UML Models used for this dissertation does not consider the actions attached to states, such as *do*, *entry*, and *exit* actions. The former 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



(a) Before Refactoring



(b) After Refactoring

Figure 5.40: UML State Machine Diagram - Introduce Initial Pseudostate

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.

Referring to the above example, some considerations can be made. If the *ACTIVE* composite state contains an *entry* action, the refactoring become slightly more complicated. In figure 5.40(a) the *entry* action is executed when the transition leaving the *Init* state crosses the border of the *ACTIVE* composite state.

In figure 5.40(b) the transition does not cross the border of the *ACTIVE* composite state.

In order to preserve the behaviour of the system, moval of the *entry* action to the transition is necessary.

5.8.5 Refactoring Implementation

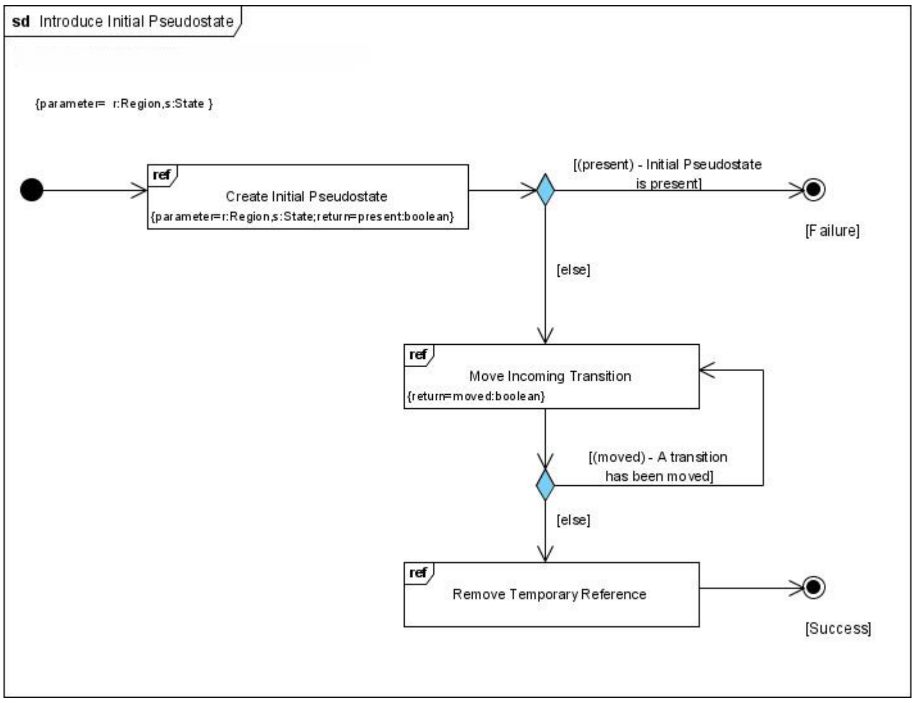

sd Introduce Initial Pseudostate
{parameter= r:Region,s:State }
ref
Create Initial Pseudostate
{parameter=r:Region,s:State;return=present:boolean}
[(present) - Initial Pseudostate is present]
[Failure]
[else]
ref
Move Incoming Transition
{return=moved:boolean}
[(moved) - A transition has been moved]
[else]
ref
Remove Temporary Reference
[Success]

Figure 5.41: Introduce Initial Pseudostate Refactoring

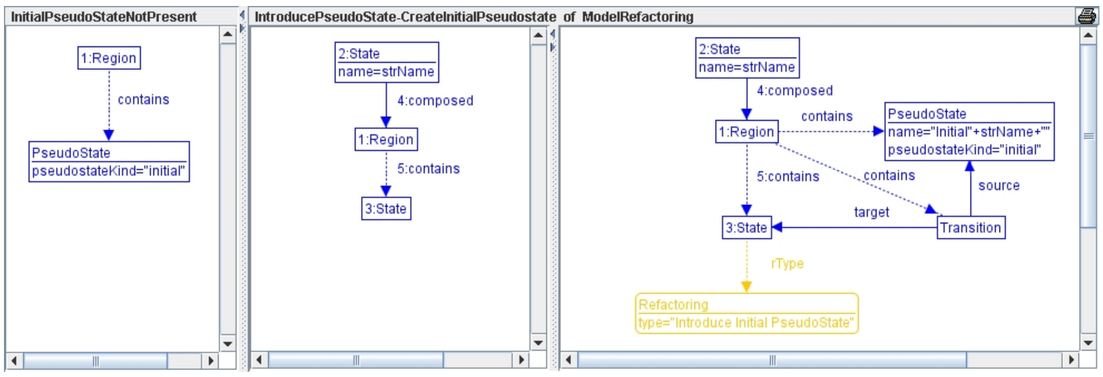
The UML 2.0 Superstructure Specification [4] (p. 591) defines an initial pseudostate in the following way: “An initial pseudostate represents a default vertex that is the source for a single transition to the default state of a composite state. There can be at most one initial vertex in a region. The initial transition may have an action.”.

Before applying the refactoring, it is necessary to verify that the composite state does not contain an initial pseudostate; this check will be implemented as a precondition.

Figure 5.41 shows the refactoring as a sequence of primitive refactoring actions. In order to apply the refactoring, it is necessary to provide two input parameters. The parameter r specifies which composite state, or region, will be modified by the refactoring. The parameter s specifies the default state of the region.

If the precondition is respected, the refactoring proceeds creating the initial pseudostate inside the composite state. Subsequently, the refactoring changes the target of all transitions for which the default state is the target. The target state of those transitions will become the composite state that contains the region r. This action can not be performed executing the step only once, instead the corresponding graph transformation rules must be repeated for each transition.

5.8.6 Graph Transformation Rules



InitialPseudoStateNotPresent
1:Region
contains
PseudoState
pseudostateKind="initial"
IntroducePseudoState-CreateInitialPseudostate of ModelRefactoring
2:State
name=strName
4:composed
1:Region
5:contains
3:State
2:State
name=strName
4:composed
contains
PseudoState
name="Initial"+strName+""
pseudostateKind="initial"
1:Region
5:contains
contains
source
target
3:State
Transition
rType
Refactoring
type="Introduce Initial PseudoState"

Figure 5.42: Introduce Initial Pseudostate - Create Initial Pseudostate
Input Parameters $r : Region \Rightarrow 1, s : State \Rightarrow 3$

The first step, named *Create Initial Pseudostate* is shown in figure 5.42. It is composed of a NAC to ensure that the region does not contain any initial pseudostate. The state s provided as input parameter will become the default state of the region. The *strName* variable defined in the rule is not an input parameter but is used in order to set the name of the initial pseudostate. The name of the initial pseudostate is composed using the name of the composite state.

The state s provided as input parameter must be part of a composite state, otherwise application of this kind of refactoring will not be possible. If the precondition is respected, the transformation rule marks the default state with a “Refactoring” node in order to recognize it during the execution of the subsequent steps.

The second step named *Move Incoming Transition* is shown in figure 5.43. It takes into account the transitions for which the default state is defined as target. It modifies the target state of a transition to point to the composite state. For this transformation rule a NAC has been defined to ensure that only transitions that are defined outside the region will be modified. The graph transformation rule must be repeated multiple times in order to modify all transitions.

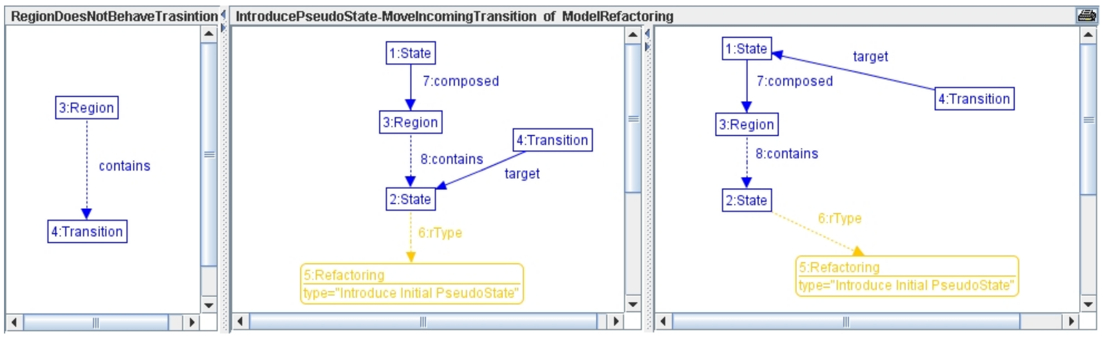


Figure 5.43: Introduce Initial Pseudostate - Move Incoming Transition

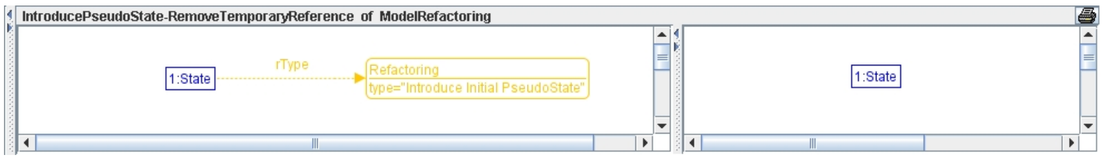

IntroducePseudoState-RemoveTemporaryReference of ModelRefactoring
1:State
rType
Refactoring
type="Introduce Initial PseudoState"
1:State

Figure 5.44: Introduce Initial Pseudostate - Remove Temporary Reference

The last step named *Remove Temporary Reference* is shown in figure 5.43. It removes the "Refactoring" node that has been associated to the default state during the execution of the first step.

5.8.7 Consequences

The *Introduce Initial Pseudostate* refactoring is used to improve the structure of a State Machine diagram. It is often combined with the other refactorings like *Introduce Region*.

Improval of the refactoring is possible extending the graph representation of UML Models, in order to support the actions attached to states, such as *do*, *entry*, and *exit* actions.

5.9 Introduce Region

5.9.1 Scope

This kind of refactoring produces changes on a State Machine Diagram only. Other diagrams are not affected by the transformation.

5.9.2 Description

The *Introduce Region* refactoring is used to improve the structure of a State Machine diagram. As suggested by the name, it introduces a region creating a composite state. The refactoring adds to the region a set of states selected by the user.

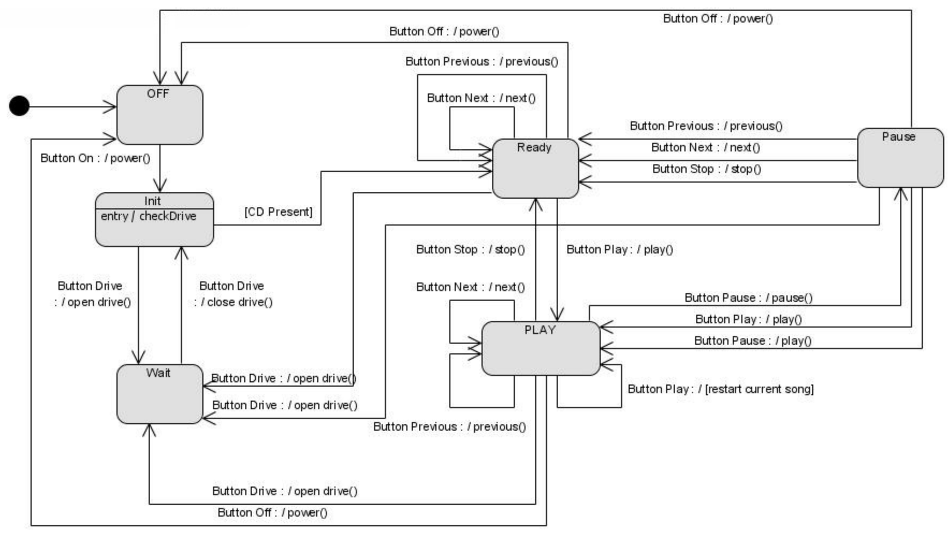
This refactoring is usually combined with the *Introduce Pseudostate* refactoring and the *Fold States Transitions* refactoring in order to simplify the structure of the UML model. The *Fold States Transitions* refactoring is the counterpart of *Flatten States Transitions* refactoring discussed in chapter 5.11.

5.9.3 Motivation and Applicability

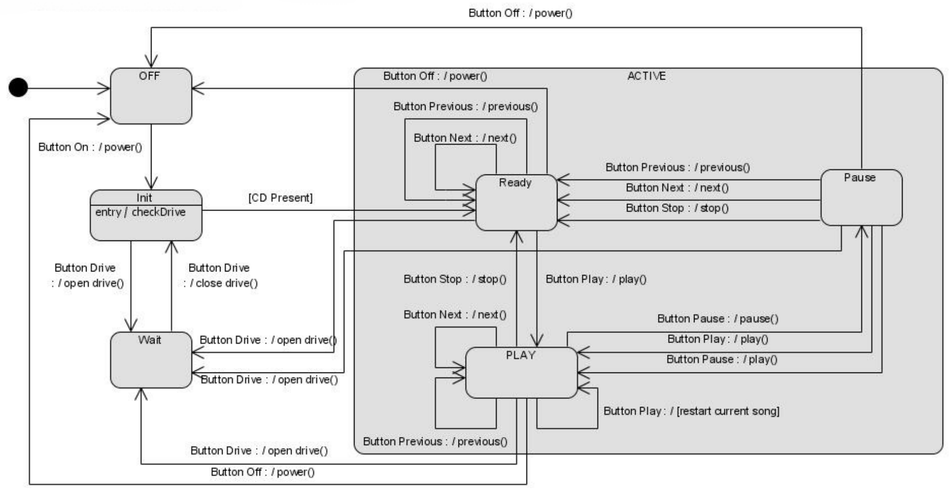
Regions are useful in order to define nested states and transitions. They group states and transitions in a logical and organized way. This kind of refactoring is particularly useful for the maintenance of State Machine diagrams. When an application is extended with new functionalities, the State Machine diagrams may be modified adding new states. A subsequent analysis could reveal that the states would be better grouped together by means of regions.

The UML 2.0 Superstructure Specification [4] (p. 531) defines a composite state in the following way: “A composite state either contains one region or is decomposed into two or more orthogonal regions. Each region has a set of mutually exclusive disjoint subvertices and a set of transitions.”

Current implementation of this refactoring adds a new composite state that will contain the region. The possibility to create a concurrent region inside an existing composite state should be taken into consideration as a possible improvement.



(a) Before Refactoring



(b) After Refactoring

Figure 5.45: UML State Machine Diagram - Introduce Region

5.9.4 Example

Figure 5.45 shows an example of this kind of refactoring. The "ACTIVE" composite state containing a region has been added to the State Machine diagram. The states "Ready", "Pause" and "Play" have been moved into the region.

Applying an *Introduce Pseudostate* refactoring and the *Fold States Outgoing Transitions* refactoring, it is possible to obtain the State Machine diagram shown in figure 5.40(b).

5.9.5 Refactoring Implementation

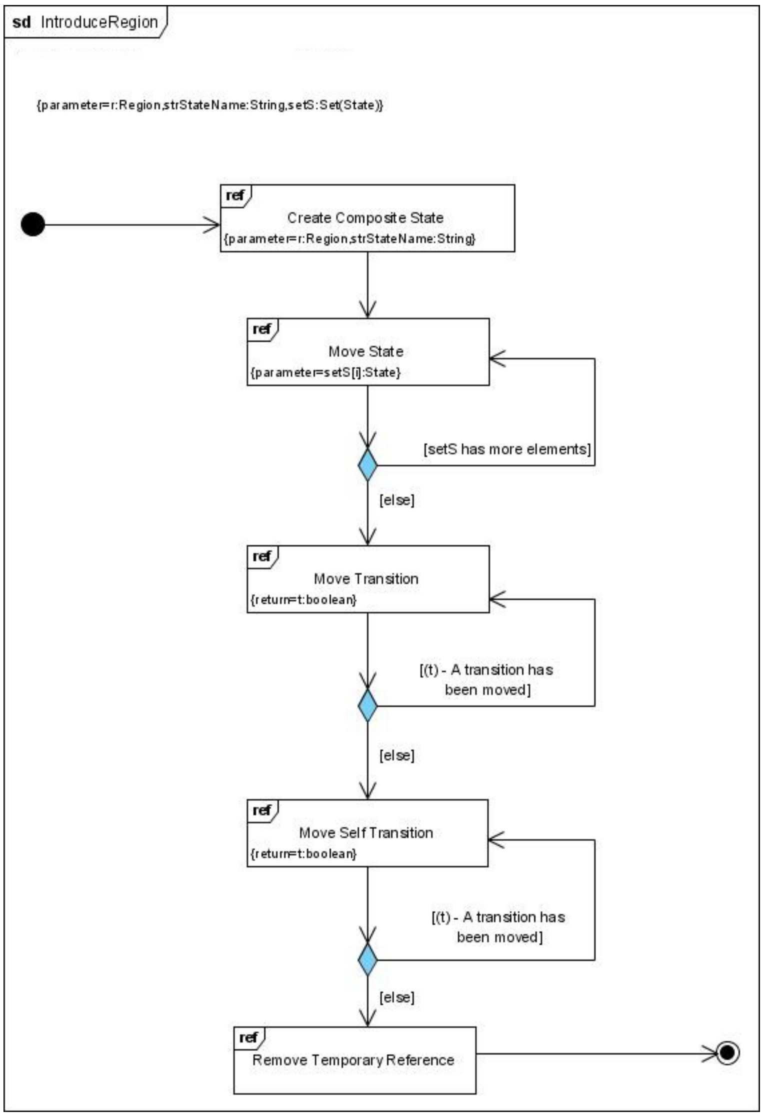

sd IntroduceRegion
{parameter=r:Region,strStateName:String,setS:Set(State)}
ref
Create Composite State
{parameter=r:Region,strStateName:String}
ref
Move State
{parameter=setS[i]:State}
[setS has more elements]
[else]
ref
Move Transition
{return=t:boolean}
[(t) - A transition has been moved]
[else]
ref
Move Self Transition
{return=t:boolean}
[(t) - A transition has been moved]
[else]
ref
Remove Temporary Reference

Figure 5.46: Introduce Region Refactoring

Figure 5.46 shows the refactoring as a sequence of primitive refactoring actions. In order to apply the refactoring it is necessary to provide three input parameters. The parameter r specifies which region will contain the new composite state. The parameter $strStateName$ specifies the name of the new composite state. The parameter $setS$ is the set of states that will be moved inside the new region.

The current implementation of the refactoring will not move states defined outside of the region r. An extension of the refactoring is possible in order to permit the selection of states defined outside of the region. However, application of a *Move State* refactoring is preferable in order to move the state from a region to one other.

The refactoring creates the composite state and assigns the specified name to it. Subsequently, the refactoring moves the selected states from the region r to the new one. If one or more states supplied as input parameter to this step are not contained in the region r they will not be considered. This action can not be performed executing the step only once but the corresponding graph transformation rules must be repeated for each state.

As mentioned in section 4.4, for the purpose of this dissertation a transition between two states is owned by the "closest" region that contains both states. The refactoring will move to the new region the transitions between two states that belong to the new region itself. The graph transformation rule must be repeated multiple times in order to move all the interested transitions.

5.9.6 Graph Transformation Rules

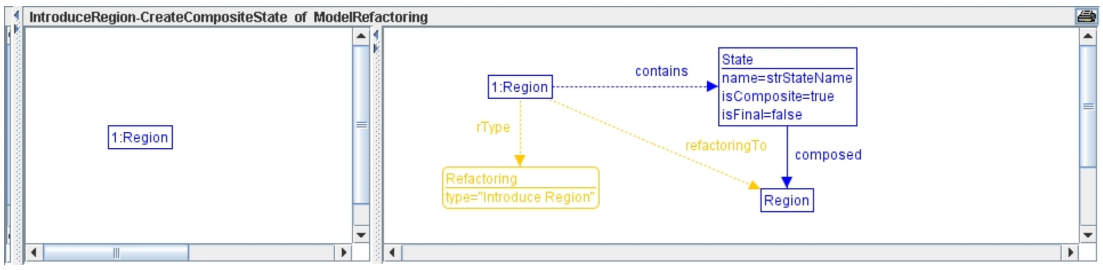


Figure 5.47: Introduce Region - Create Composite State
Input Parameters $r : Region \Rightarrow 1, strStateName : String$

The first step named *Create Composite State* is shown in figure 5.47. The region that will contain the new composite state and the name of the new composite state must be provided as input parameters to the graph transformation rule.

The transformation rule creates the new region inside a composite state. It marks the source

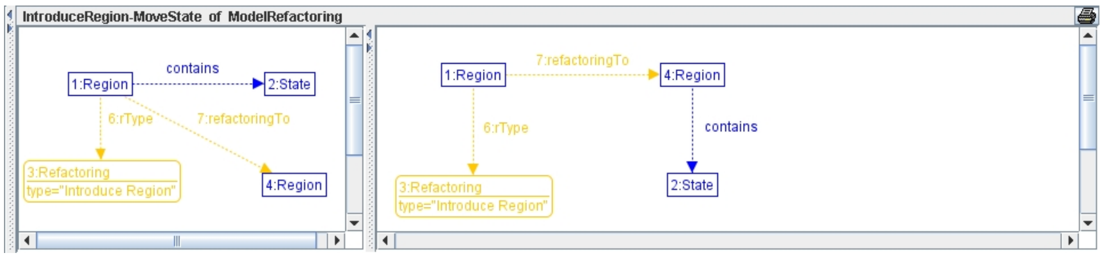


Figure 5.48: Introduce Region - Move State

Input Parameters $setS[i] : State \Rightarrow 2$

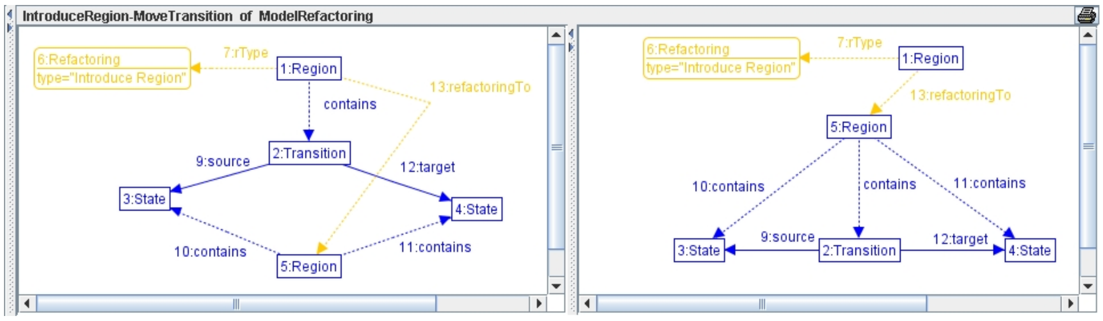


Figure 5.49: Introduce Region - Move Transition

region with a “Refactoring” node in order to recognize it during the execution of the subsequent steps. It also adds a “Refactoring” edge between the two regions, that means the refactoring will move the states from the source region to the new one.

The step named *Move State* is shown in figure 5.48. It moves a state supplied as input parameter from the source region to the one that has been created in the previous step. If the state does not belong to the source region the refactoring will produce no changes. The graph transformation rule must be repeated for each state selected by the user.

The step named *Move Transition* is shown in figure 5.49. It moves to the new region a transition between two states that belong to the source region. The graph transformation rule must be repeated multiple times in order to move all the interested transitions.

The step named *Move Self Transition* is shown in figure 5.50. It moves to the new region a self transition defined for a state that belongs to the source region. The graph transformation rule must be repeated multiple times in order to move all the interested transitions.

The step named *Remove Temporary Reference* removes the “Refactoring” node associated

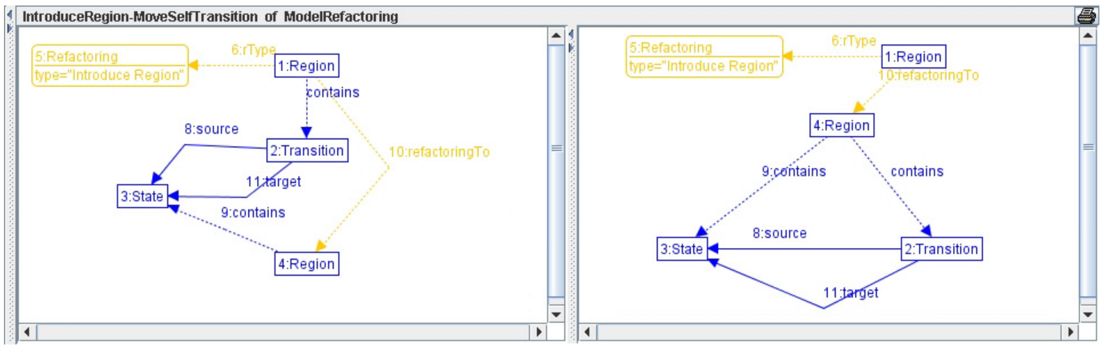


Figure 5.50: Introduce Region - Move Self Transition

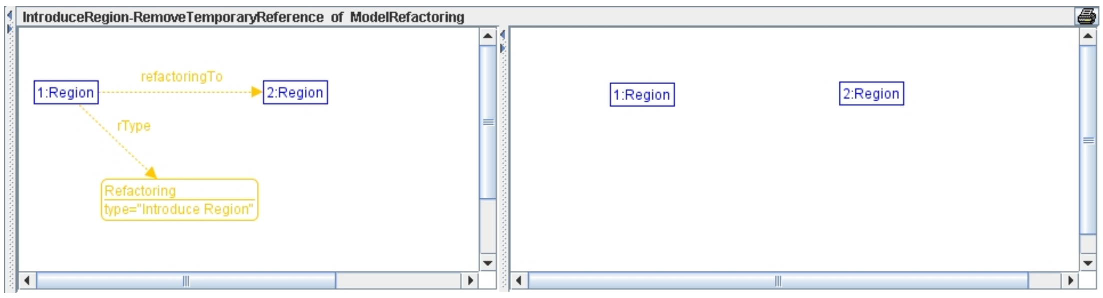


Figure 5.51: Introduce Region - Remove Temporary Reference

to the source region during the execution of the first step.

5.9.7 Consequences

The *Introduce Region* refactoring improves the structure of a State Machine diagram. It is useful in order to organise states and transitions in a logical way.

Some improvements may be done to this refactoring in order to support the creation of concurrent regions.

5.10 Remove Region

5.10.1 Scope

This kind of refactoring produces changes on a State Machine Diagram only. Other diagrams are not affected by the transformation.

5.10.2 Description

The *Remove Region* refactoring is the counterpart of *Introduce Region* refactoring previously discussed. As suggested by the name, it removes a region from a State Machine diagram. It moves states and transitions contained in the region to the parent region. The refactoring also removes the composite state that contains the region.

This refactoring is usually combined with the *Remove Pseudostate* refactoring and the *Flatten State Transitions* refactoring in order to reorganize the structure of the UML model.

5.10.3 Motivation and Applicability

The *Remove Region* refactoring is particularly useful for maintenance of State Machine diagrams. When an application evolves, the functionalities of the system may change and the State Machine diagrams may be modified to reflect these changes. A subsequent analysis could reveal that the states contained in a region are not related anymore and they would be better grouped in a different way.

The upper region, directly connected to the State Machine, can not be removed as it represents the whole State Machine diagram.

The UML 2.0 Superstructure Specification [4] (p. 531) defines a composite state in the following way: "A composite state either contains one region or is decomposed into two or more orthogonal regions. Each region has a set of mutually exclusive disjoint subvertices and a set of transitions."

Current implementation of this refactoring does not allow removal of concurrent regions. The possibility to remove a concurrent region defined inside a composite state should be taken into account as a possible improvement.

The graph representation of UML Models used for this dissertation does not consider the actions attached to states, such as *do*, *entry*, and *exit* actions. If the composite state that is going to be removed contains *entry* or *exit* actions, the refactoring becomes slightly more complicated.

If *entry* and *exit* actions are simply removed, the behaviour of the system will change. In order to preserve the behaviour, it is necessary to move the *entry* action to the incoming transitions and the *exit* action to the outgoing transitions.

5.10.4 Example

Figure 5.45(b)(a) shows an example of this kind of refactoring. The "ACTIVE" composite state containing a region has been removed from the State Machine diagram. The states "Ready", "Pause" and "Play" are part of the upper region, directly connected to the State Machine.

5.10.5 Refactoring Implementation

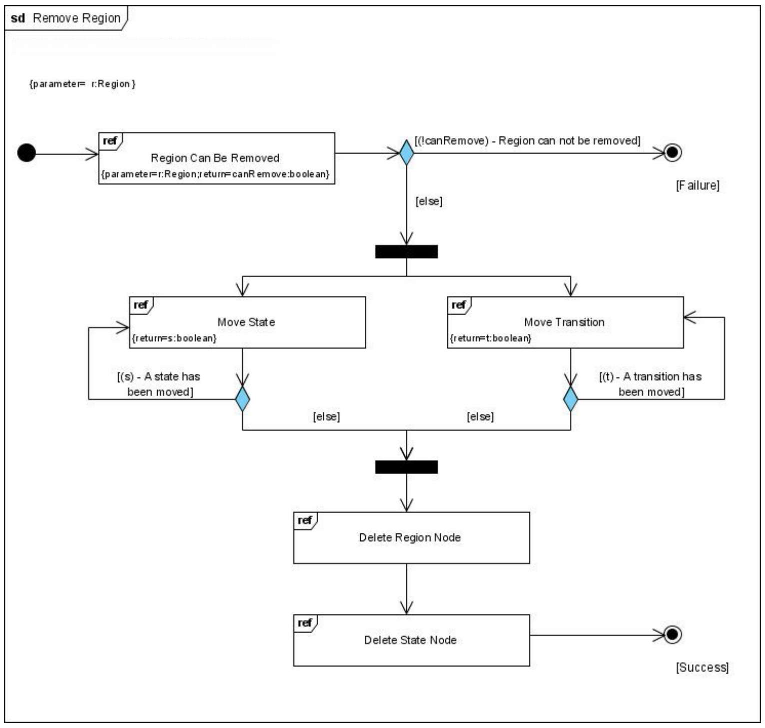


Figure 5.52: Remove Region Refactoring

Figure 5.52 shows the refactoring as a sequence of primitive refactoring actions. In order to apply the refactoring, it is necessary to provide an input parameters r that specifies which is the region that will be removed.

The refactoring checks that the region r is not a concurrent region. Subsequently, it verifies that ingoing and outgoing transitions do not exist for the specified region. The last precondition checked by the refactoring avoids that a region containing pseudostates could be removed. If at least one of these preconditions is not respected the refactoring will be aborted.

After verification of all these preconditions, the refactoring proceeds moving all states and transitions from the region r to the parent region. The actions *Move State* and *Move Transition* do not require any exact execution order and they may also be executed in parallel. This action can not be performed executing the steps only once, but the corresponding graph transformation rules must be repeated for each state and transition.

When all states and transitions has been moved, the refactoring continues removing the region r. The last step removes the composite state that was the container of the region r.

5.10.6 Graph Transformation Rules

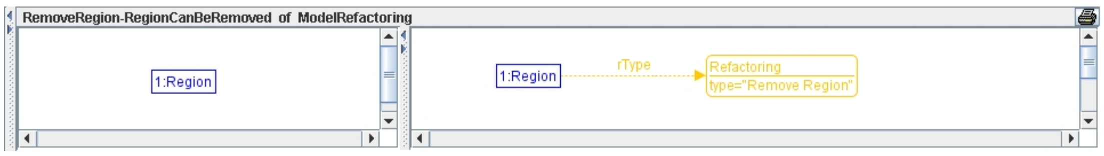


Figure 5.53: Remove Region - Region Can Be Removed

Input Parameters $r : Region \Rightarrow 1$

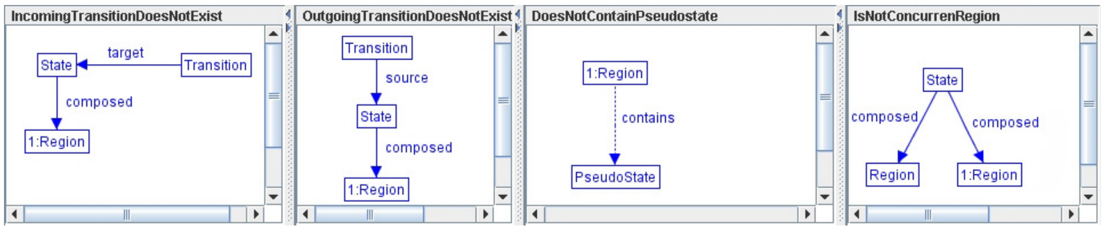


Figure 5.54: Remove Region - Region Can Be Removed NACs

The first step named *Region Can Be Removed* is shown in figure 5.53. It is composed of four NACs shown in figure 5.54, that verify all necessary preconditions. The region that have to be removed must be provided as input parameter to the graph transformation rule.

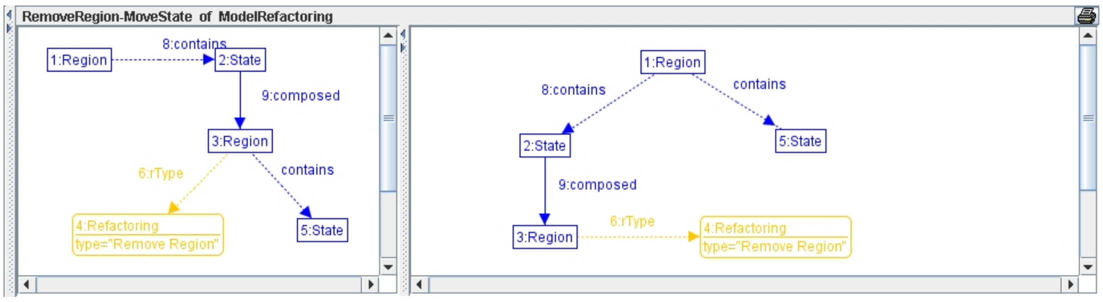


Figure 5.55: Remove Region - Move State

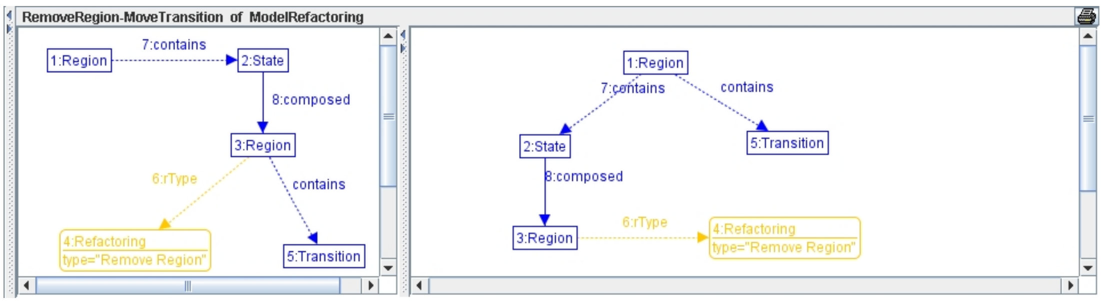


Figure 5.56: Remove Region - Move Transition

The NAC *IncomingTransitionDoesNotExist* verifies that the selected region is not the target of any incoming transition. The NAC *OutgoingTransitionDoesNotExist* verifies that the region is not the source of any outgoing transition. The NAC *DoesNotContainPseudostate* is used to verify that the selected region does not contain any pseudostates. The NAC *IsNotConcurrenRegion* verifies that the region is not defined as concurrent region.

If the preconditions are respected, the transformation rule marks the region with a "Refactoring" node in order to recognize it during the execution of the subsequent steps.

The step named *Move State* is shown in figure 5.55. It moves a state from the selected region to the parent region. The graph transformation rule must be repeated multiple times in order to move all states contained by the region.

The step named *Move Transition* is shown in figure 5.56. It moves a transition from the selected region to the parent region. The graph transformation rule must be repeated multiple times in order to move all transitions contained by the region.

The actions *Move State* and *Move Transition* do not require any exact execution order and

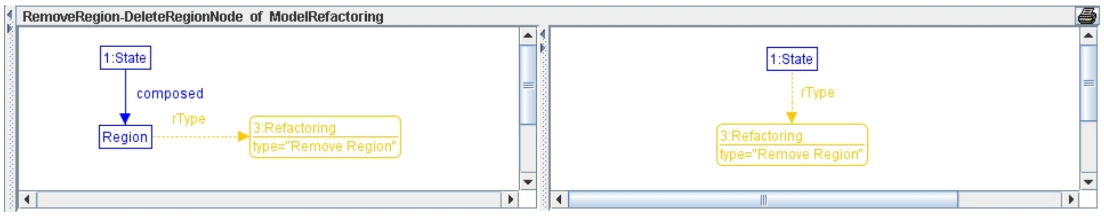


Figure 5.57: Remove Region - Delete Region Node

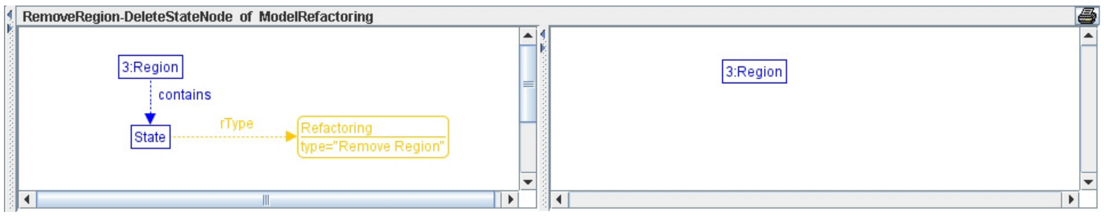


Figure 5.58: Remove Region - Delete State Node

they may also be executed in parallel.

The step named *Delete Region Node* removes the selected region from the graph. The “Refactoring” node is moved to the composite state in order to recognize it in the next step.

The step named *Delete State Node* is shown in figure 5.58. It removes the composite state that previously contained the region.

5.10.7 Consequences

The *Remove Region* refactoring is useful to reorganize states and transitions during the design phase.

Some improvements may be done to this refactoring in order to support the deletion of concurrent regions. Moreover, it is possible to extend the graph representation of UML Models in order to support the actions attached to states, such as *do*, *entry*, and *exit* actions.

5.11 Flatten States Transitions

5.11.1 Scope

This kind of refactoring produces changes on a State Machine Diagram only. Other diagrams are not affected by the transformation.

5.11.2 Description

The *Flatten States Transitions* refactoring is used to improve the structure of a State Machine diagram. This kind of refactoring is often part of more complex refactorings. It is usually combined with the *Remove Region* refactoring previously formalised. It is composed of two different transformations that are identified with the name *Flatten States Outgoing Transitions* and *Flatten States Incoming Transitions*.

The *Flatten States Outgoing Transitions* transformation replaces an outgoing transition, starting from a composite state, with the corresponding transitions associated to each sub–states.

The *Flatten States Incoming Transitions* transformation replaces an incoming transition, leading to the composite state, with the corresponding transitions associated to each sub–states.

The two transformations present the same behaviour in order to modify outgoing and incoming transitions. This chapter formalise the *Flatten States Outgoing Transitions* transformation, and in section 5.11.8, the differences of the *Flatten States Incoming Transitions* transformation will be discussed.

In this chapter, the *Flatten States Outgoing Transitions* transformation is also referred as *Flatten States Outgoing Transitions* refactoring.

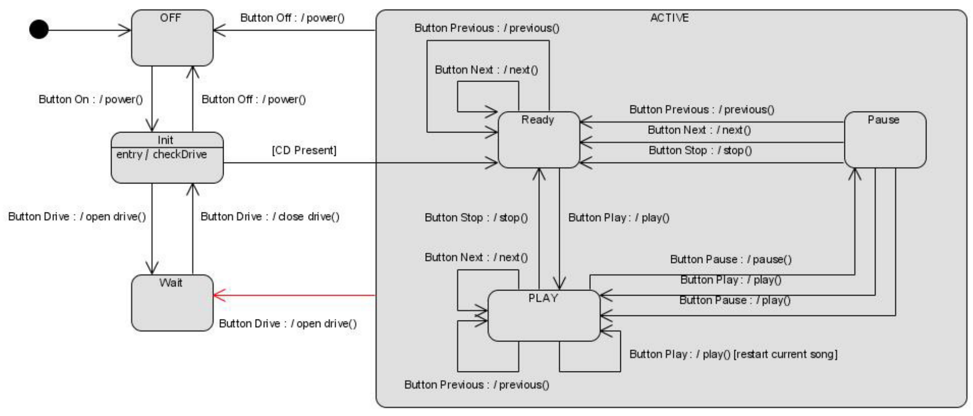
5.11.3 Motivation and Applicability

When an outgoing transition starting from a composite state is triggered by its event, it is executed independently of the sub–state the State Machine resides in. This means that the outgoing transition could be replaced by transitions associated to each sub–state of the composite state; these transitions would have the same target state of the original one.

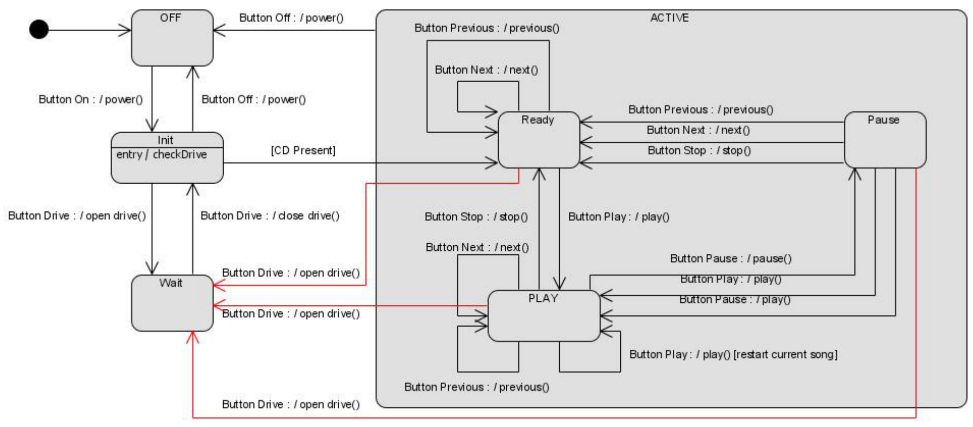
An outgoing transition that is automatically triggered exiting the composite state could be defined in the State Machine diagram. This particular transition can not be replaced by transitions leaving each sub–states. Therefore, the refactoring will be applied only to the transitions for which an event is defined.

The *Flatten States Outgoing Transitions* refactoring modifies a State Machine diagram only. It does not produce any changes over other kinds of diagrams, because it does not modify the structure of the system and all transitions involved in the refactoring preserve the same characteristics.

5.11.4 Example



(a) Before Refactoring



(b) After Refactoring

Figure 5.59: UML State Machine Diagram - Flatten States Outgoing Transitions

Figure 5.59 shows an example of this kind of refactoring. Before applying the refactoring there is a transition whose source and target are respectively the *ACTIVE* region and the *Wait* state. That transition is triggered by the event *Drive Button* of the *Player*. The refactoring adds a similar transition to each sub–state of the *ACTIVE* region and removes the original one. The target state of these new transitions is the same as the original one.

A transition could be characterized also by a guard and an associated action. If this information is defined in the UML model for the original transition it must be copied to the new transitions.

The same could be done for the outgoing transition whose target is the *OFF* state, triggered by the *Off Button* of the *Player*. When the Off Button of the *Player* is pressed, the sub–state the *Player* currently is in is not relevant, the *Player* being always turned off.

The graph representation of UML Models used for this dissertation does not consider any actions attached to states, such as *do*, *entry*, and *exit* actions.

Referring to the above example, some considerations can be made. If the *ACTIVE* composite state contains an *exit* action, the refactoring becomes slightly more complicated. In figure 5.59(a) the transition between the composite state and the *Wait* state does not cross the border of the *ACTIVE* composite state. In figure 5.59(b) the *exit* action of the composite state is executed when the transitions between the composite state and the *Wait* state are triggered. This will cause the behaviour of the system to change. In order to preserve the behaviour of the system, it is necessary to move the *exit* action to the transitions for which it has to be executed and remove the *exit* action from the composite state.

5.11.5 Refactoring Implementation

Figure 5.60 shows the refactoring as a sequence of primitive refactoring actions, the action named *Transform Transitions* is a complex action and will be detailed later.

In order to apply the refactoring, it is necessary to provide an input parameter r that represents the region that is going to be modified. The refactoring verifies that the region is not part of a composite state containing concurrent regions. After verification of the necessary preconditions, it will automatically verify whether at least one outgoing transition is defined for the region supplied as input parameter.

The *Select Target State* step searches in the graph a state that is defined as target for an outgoing transition. With the subsequent steps the transitions associated to each sub–state of the region will be created. During this process, some “Refactoring” nodes are used to identify the

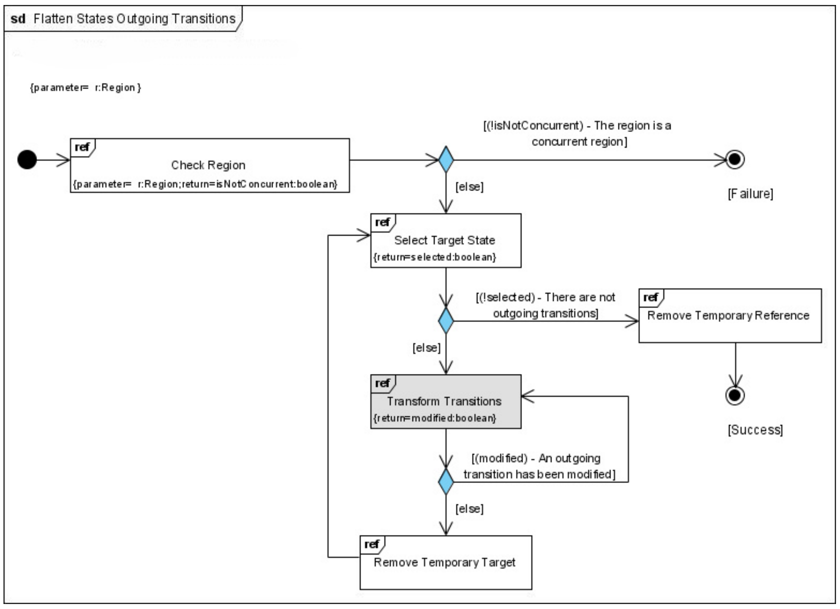

sd Flatten States Outgoing Transitions
{parameter= r:Region}
ref
Check Region
{parameter= r:Region;return=isNotConcurrent:boolean}
[(!isNotConcurrent) - The region is a concurrent region]
[Failure]
[else]
ref
Select Target State
{return=selected:boolean}
[(!selected) - There are not outgoing transitions]
ref
Remove Temporary Reference
[Success]
[else]
ref
Transform Transitions
{return=modified:boolean}
[(modified) - An outgoing transition has been modified]
[else]
ref
Remove Temporary Target

Figure 5.60: Flatten States Outgoing Transitions Refactoring

involved transitions. When the transition has been copied to each sub–state of the region, the original one will be removed. If the diagram still contains outgoing transitions for the specified region a new state will be taken into account and the same actions will be repeated.

When the refactoring has moved all the transitions, it will remove the "Refactoring" nodes that have been added in order to recognize the interested region during the transformations. This action is reported in the diagram as *Remove Temporary Reference*.

The *Transform Transitions* action is a complex action and its implementation using the AGG tool has underlined some limitations of the tool chosen. When an outgoing transition has been detected, it is necessary to copy it to all the sub–states of the region, and the AGG tool does not provide a mechanism to copy a node multiple times. This is especially the case if the number of copies varies for different application of the graph transformation rule.

It is therefore necessary to repeat multiple times the copy of the current transition and provide the logic for verifying that the transition has been copied to all sub–states.

Figure 5.61 shows the implementation of the *Transform Transitions* action. In section 5.11.7 some possible alternatives will be discussed.

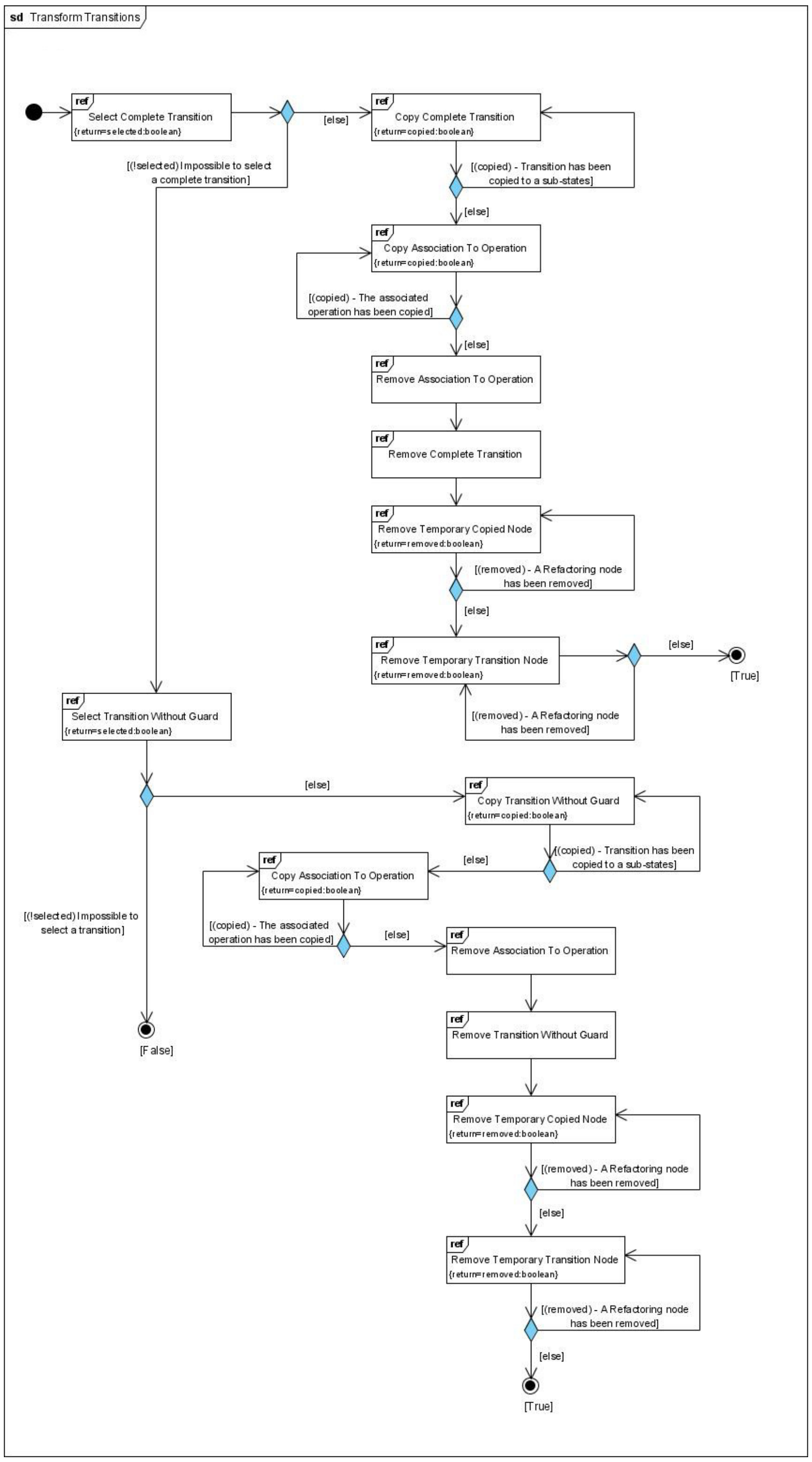

sd Transform Transitions
ref Select Complete Transition {return=selected:boolean}
[else]
ref Copy Complete Transition {return=copied:boolean}
[(!selected) Impossible to select a complete transition]
[(copied) - Transition has been copied to a sub-states]
[else]
ref Copy Association To Operation {return=copied:boolean}
[(copied) - The associated operation has been copied]
[else]
ref Remove Association To Operation
ref Remove Complete Transition
ref Remove Temporary Copied Node {return=removed:boolean}
[(removed) - A Refactoring node has been removed]
[else]
ref Remove Temporary Transition Node {return=removed:boolean}
[else]
[True]
[(removed) - A Refactoring node has been removed]
ref Select Transition Without Guard {return=selected:boolean}
[else]
ref Copy Transition Without Guard {return=copied:boolean}
[(copied) - Transition has been copied to a sub-states]
[else]
ref Copy Association To Operation {return=copied:boolean}
[(!selected) Impossible to select a transition]
[(copied) - The associated operation has been copied]
[else]
ref Remove Association To Operation
[False]
ref Remove Transition Without Guard
ref Remove Temporary Copied Node {return=removed:boolean}
[(removed) - A Refactoring node has been removed]
[else]
ref Remove Temporary Transition Node {return=removed:boolean}
[(removed) - A Refactoring node has been removed]
[else]
[True]

Figure 5.61: Transform Transitions

5.11.6 Graph Transformation Rules

The first step named *Check Region* is shown in figure 5.62. The NAC *IsNotConcurrentRegion* of this rule verifies that the region supplied as input parameter is not part of any composite state that contains concurrent regions. If the precondition is respected, a “Refactoring” node is added in order to identify the region during the execution of the subsequent steps. An input parameter r that specifies which region will be modified by the refactoring has to be provided to the graph transformation rule.

A composite state may have multiple outgoing transitions leading to different states. The step shown in figure 5.63 searches an outgoing transition and marks the target state with a “Refactoring” node in order to identify it. If any state could be selected by this transformation step, the region does not have any outgoing transition and the refactoring must be terminated as described in the previous section.

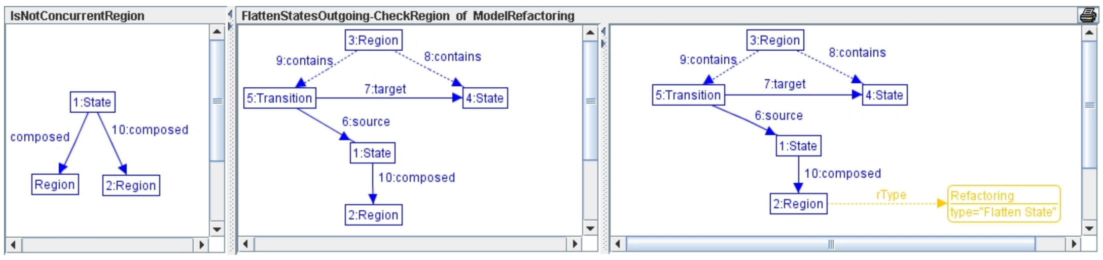


Figure 5.62: Flatten States Outgoing Transitions - Check Region

Input Parameters $r : Region \Rightarrow 2$

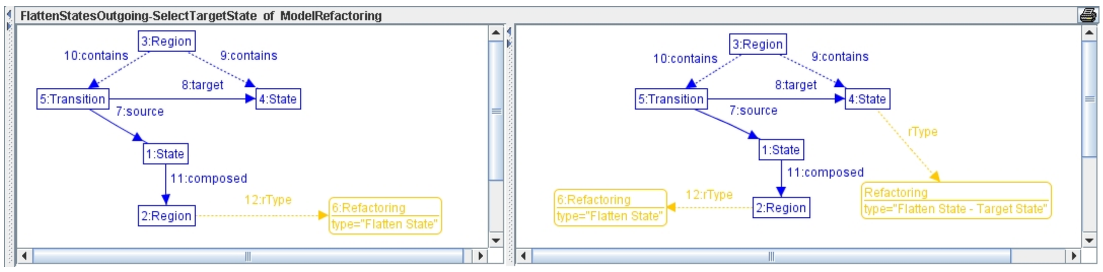


Figure 5.63: Flatten States Outgoing Transitions - Select Target State

The UML 2.0 Superstructure Specification [4] allows multiple transitions to share the same source and target states. In that case, it is possible to have more outgoing transitions that share the

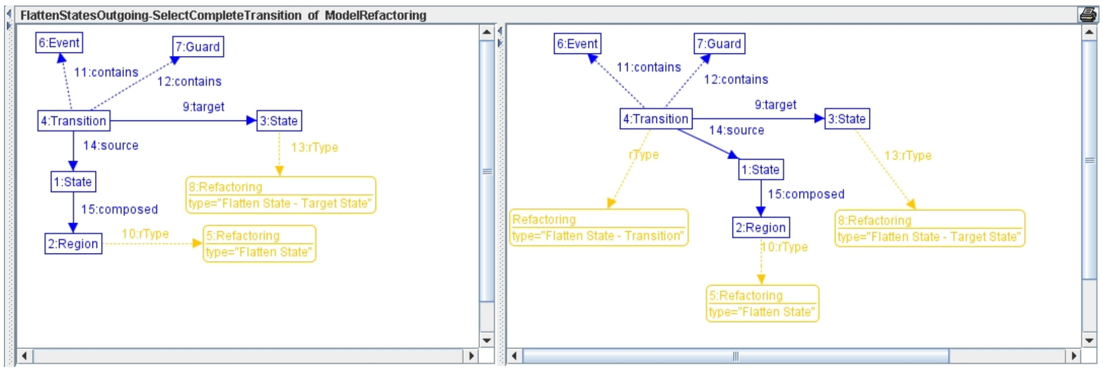


Figure 5.64: Flatten States Outgoing Transitions - Select Complete Transition

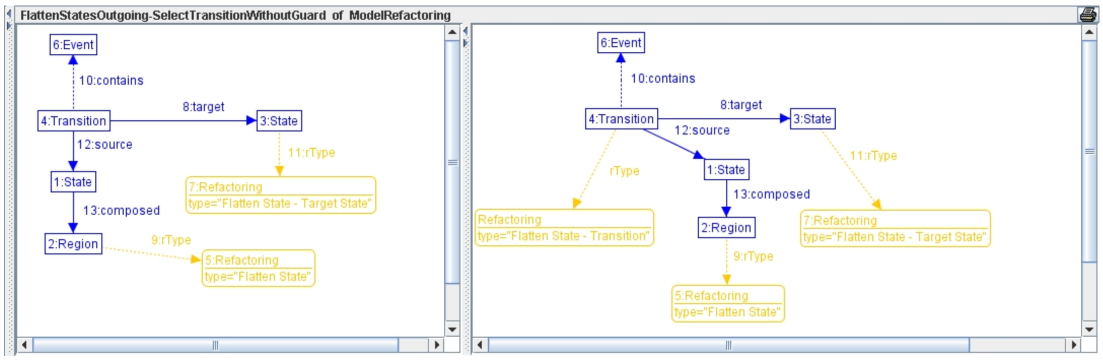


Figure 5.65: Flatten States Outgoing Transitions - Select Transition Without Guard

same target state and the different transitions need to be managed separately. The transformation rules shown in figures 5.64 and 5.65 select a transition and mark it with a “Refactoring” node, the former searching the complete transitions and the latter matching transitions that do not contain a guard.

The following steps, that copy the transition to each sub–state of the region, must be repeated multiple times and all added “Refactoring” nodes are necessary to clearly identify the transition that has to be copied.

Figures 5.66 and 5.67 represent the steps that copy the transition and its associated nodes to a sub–state of the selected region. The new transition is created in the region that contains the original one and is marked with a “Refactoring” node that allows to recognize it during the execution of the next step. Two variables are defined for that rules and they are necessary

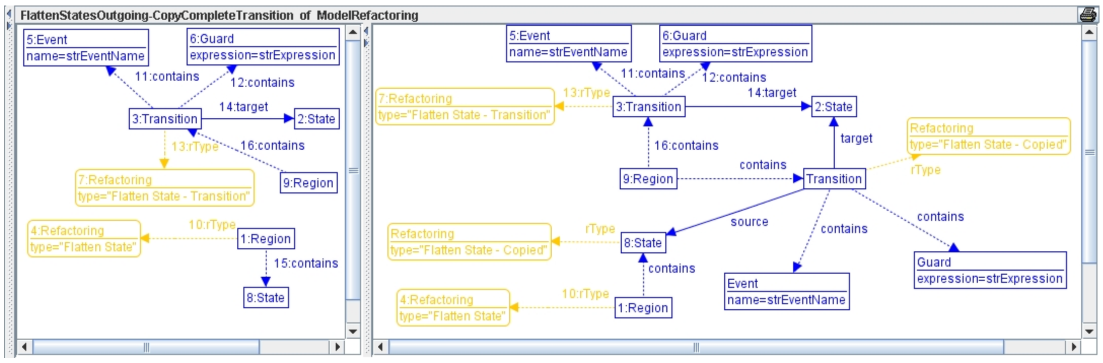

FlattenStatesOutgoing-CopyCompleteTransition of ModelRefactoring
5:Event name=strEventName
6:Guard expression=strExpression
11:contains
12:contains
14:target
3:Transition
2:State
13:rType
16:contains
7:Refactoring type="Flatten State - Transition"
9:Region
4:Refactoring type="Flatten State"
10:rType
1:Region
15:contains
8:State
Refactoring type="Flatten State - Copied"
Transition
source
target
contains
rType
Guard expression=strExpression
Event name=strEventName

Figure 5.66: Flatten States Outgoing Transitions - Copy Complete Transition

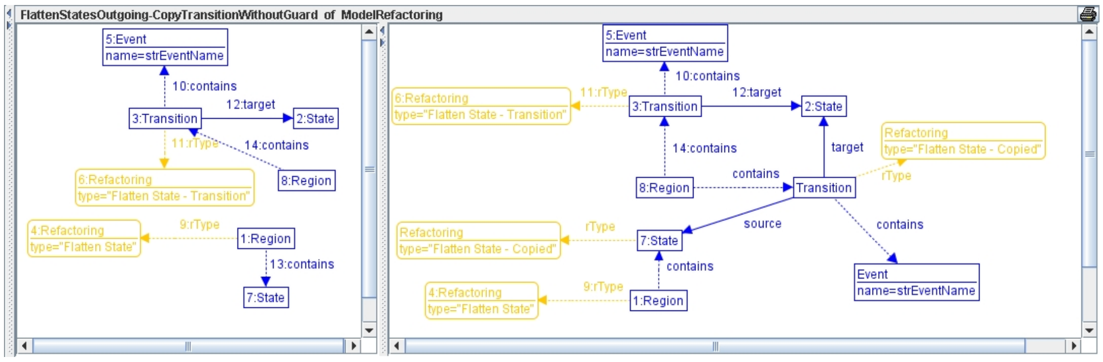

FlattenStatesOutgoing-CopyTransitionWithoutGuard of ModelRefactoring
5:Event name=strEventName
10:contains
12:target
3:Transition
2:State
11:rType
14:contains
6:Refactoring type="Flatten State - Transition"
8:Region
4:Refactoring type="Flatten State"
9:rType
1:Region
13:contains
7:State
Refactoring type="Flatten State - Copied"
Transition
source
target
contains
rType
Event name=strEventName

Figure 5.67: Flatten States Outgoing Transitions - Copy Transition Without Guard

to recreate the transition with the same informations. The *strEventName* variable is used to copy the event that triggers the transition, and the *strExpression* variable memorises the boolean expression of the guard optionally associated to the transition. A “Refactoring” node is added to the source sub–state of the region to indicate that the transition has already been copied into it.

When a transition is triggered by its event, an operation may be executed and an association between the transition and the operation is defined in this case. Figure 5.68 describes the rule named *Copy Association To Operation* that copies the association to the new created transition if it is defined for the original one. The association must be copied to each transition that has been generated in the previous step and, for that purpose, the properties of the “Refactoring” node associated to the transitions are modified to clearly recognize the nodes that have been processed.

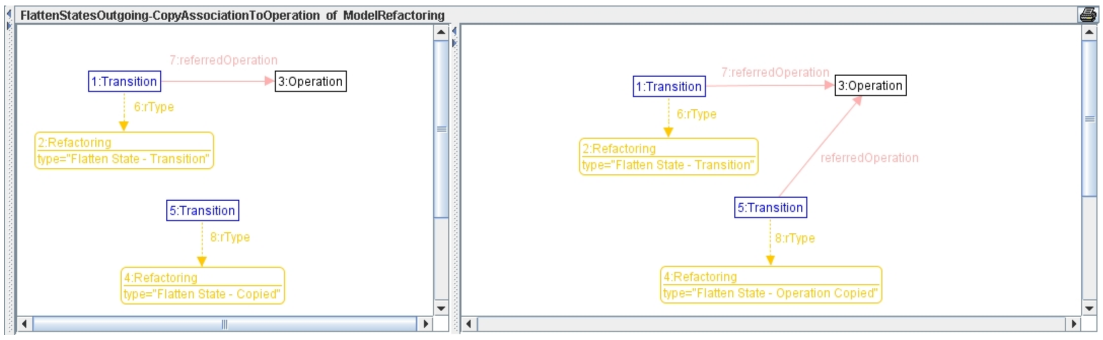


Figure 5.68: Flatten States Outgoing Transitions - Copy Association To Operation

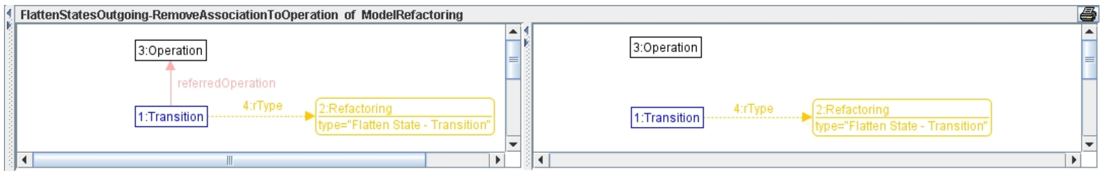

FlattenStatesOutgoing-RemoveAssociationToOperation of ModelRefactoring
3:Operation
referredOperation
1:Transition
4:rType
2:Refactoring
type="Flatten State - Transition"
3:Operation
1:Transition
4:rType
2:Refactoring
type="Flatten State - Transition"

Figure 5.69: Flatten States Outgoing Transitions - Remove Association To Operation

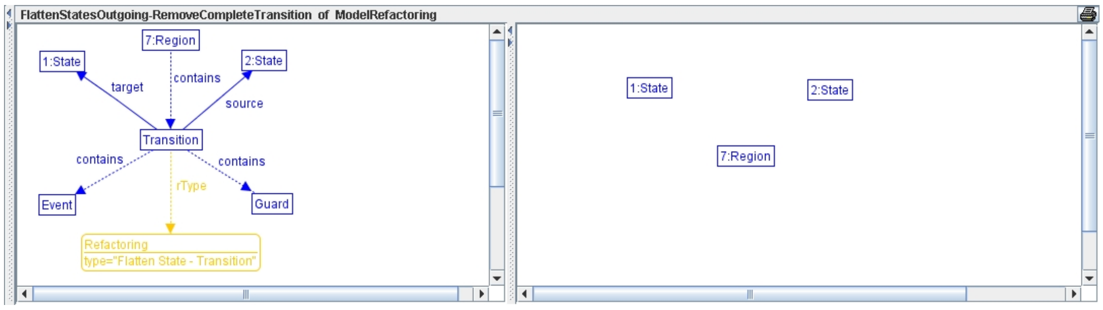


Figure 5.70: Flatten States Outgoing Transitions - Remove Complete Transition

The step named *Remove Association To Operation* is shown in figure 5.69. It is used to remove the optional association between the original transition and the operation that is executed when the transition is triggered by its event. If the original transition is not associated with an operation, the execution of this step will produce no changes over the diagram.

The steps represented in figures 5.70 and 5.71 have the responsibility to remove the original transition and all its associated nodes. The former removes transitions characterized by a guard

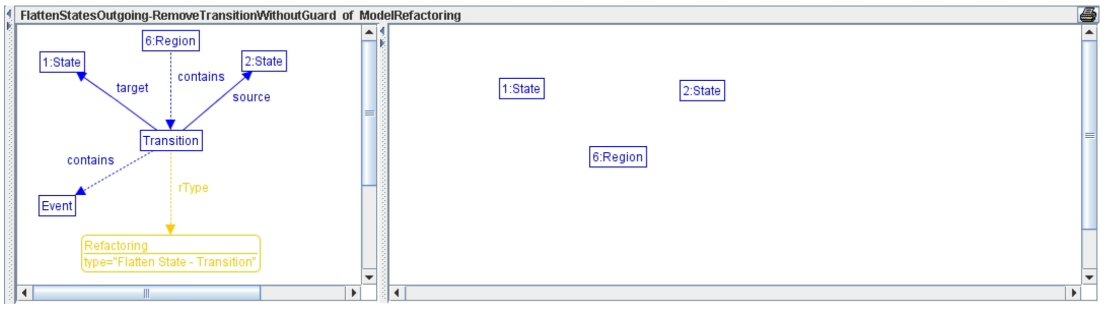

FlattenStatesOutgoing-RemoveTransitionWithoutGuard of ModelRefactoring
6:Region
1:State
2:State
target
contains
source
Transition
contains
rType
Event
Refactoring
type="Flatten State - Transition"
1:State
2:State
6:Region

Figure 5.71: Flatten States Outgoing Transitions - Remove Transition Without Guard

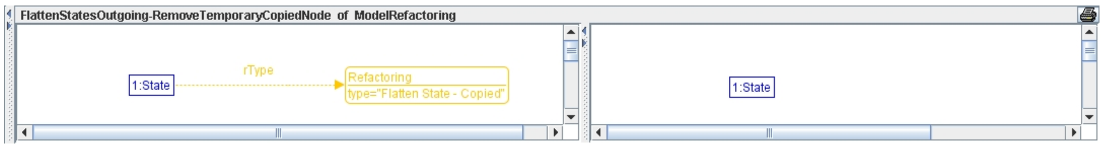

FlattenStatesOutgoing-RemoveTemporaryCopiedNode of ModelRefactoring
1:State
rType
Refactoring
type="Flatten State - Copied"
1:State

Figure 5.72: Flatten States Outgoing Transitions - Remove Temporary Copied Node

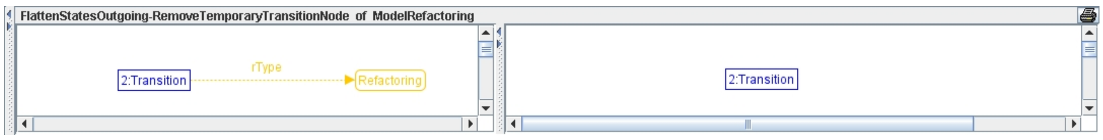

FlattenStatesOutgoing-RemoveTemporaryTransitionNode of ModelRefactoring
2:Transition
rType
Refactoring
2:Transition

Figure 5.73: Flatten States Outgoing Transitions - Remove Temporary Transition Node

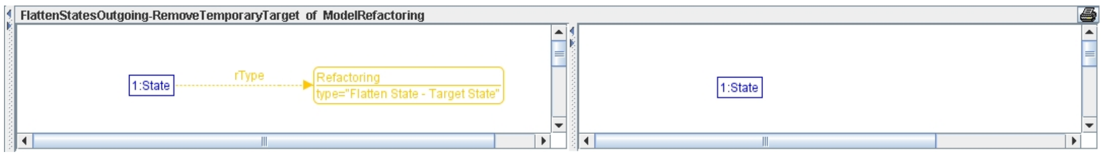

FlattenStatesOutgoing-RemoveTemporaryTarget of ModelRefactoring
1:State
rType
Refactoring
type="Flatten State - Target State"
1:State

Figure 5.74: Flatten States Outgoing Transitions - Remove Temporary Target

and the latter transitions without a guard.

The rules *Remove Temporary Copied Node* and *Remove Temporary Transition Node* are shown respectively in figure 5.72 and 5.73. They remove the “Refactoring” node added to the sub–states during the copy of the transition and used by the previous steps. This steps must be repeated multiple times in order to remove all the “Refactoring” nodes.

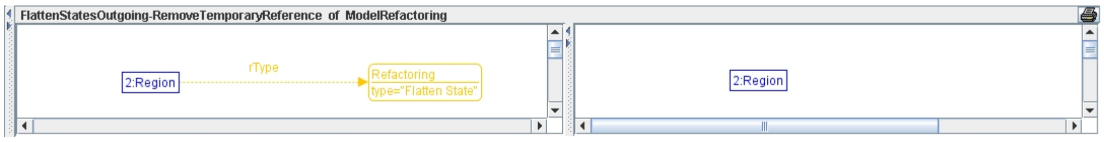

FlattenStatesOutgoing-RemoveTemporaryReference of ModelRefactoring
2:Region
rType
Refactoring
type="Flatten State"
2:Region

Figure 5.75: Flatten States Outgoing Transitions - Remove Temporary Reference

The step *Remove Temporary Target* shown in figure 5.74 removes the “Refactoring” node that have been added to the selected target state during the execution of the *Select Target State* step. At this point it is possible to look for another outgoing transition with a different target state. The final step *Remove Temporary Reference*, shown in figure 5.75, removes the “Refactoring” node that has been associated to the region provided as input parameter.

5.11.7 Alternative implementations

As introduced in previous sections the *Transform Transitions* action is a complex action and its implementation using the AGG tool has underlined some limitations of the tool chosen. When an outgoing transition has been detected, it is necessary to copy it to all the sub–states of the region and the AGG tool does not provide any mechanism to easily copy a node multiple times. This is especially the case if the number of copies varies for different application of the graph transformation rule. It is therefore necessary to repeat the copy of the current transition multiple times and provide the logic for verifying that the transition has been copied to all sub–states.

The current graph transformation notation is not powerful enough and does not suffice to easily define this kind of refactoring. Many studies have been done to search a formal solution to this problem, and some proposals have been presented to close this gap. In particular *D. Janssens* and *N. Van Eetvelde* have proposed and formally defined in their article [29] a way for cloning and expanding graph nodes and graph patterns. These operations make graph transformations more expressive and facilitate definition of refactorings using graph transformations.

Moreover, a transition could be characterized by a guard and an associated action. The guard is a boolean condition that is evaluated when a transition initiates. The associated action is an operation that will be performed when the transition is triggered. The AGG tool does not provide any mechanism to specify optional nodes that have to be managed by the transformation. The possibility to define optional graph patterns in the graph transformation rules is another necessary enhancement and makes graph transformations more powerful.

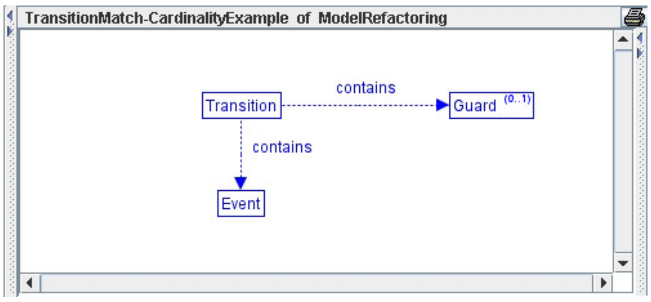


Figure 5.76: Cardinality Example

A. Agrawal, *G. Karsay* and *F. Shi* [30] propose to define a cardinality property for the graph nodes and graph edges in the graph transformation rules. This technique allows the expression of a large number of graph patterns in a compact way and the specification of optional components in a pattern by having the cardinality of optional components as (0..1).

Figure 5.76 shows a possible example of the use of cardinalities. In the right–upper corner of the “Guard” node, a cardinality has been specified. The value (0..1) means that the “Guard” node could be present 0 or 1 times. That way, the graph pattern represented in the figure will match transitions that contain a guard and transitions that do not contain a guard.

Using the AGG tool, there are two possible ways for implementing the *Transform Transitions* action. The first option, described in figure 5.77, consists of creation of a rule for each possible graph pattern. For example, referring to figure 5.76 it is necessary to create two different rules in order to match transitions that contain a guard and transitions that do not contain a guard. Obviously, these rules are very similar and they differ only for the optional nodes.

In the solution shown in figure 5.77, a first set of rules is defined for a complete transition that contains an event, a guard, and an associated operation. This set of rules consists of the necessary operations to select a complete transition, copy it to each sub–state, and remove it. After copy of all the complete transitions, it is necessary to copy the transition that contains only the guard or only the associated operation, and two similar sets have been defined to match and manage these patterns. Finally, it is necessary to define another set of rules for the transitions that do not contain the guard nor the associated operation.

Each set of rules is composed of three rules. One rule is used to select the transition, one is used to copy the transition to each sub–state, and one more removes the original transition. Let N_{opt} be the number of optional nodes associated to the transition, the total number of necessary

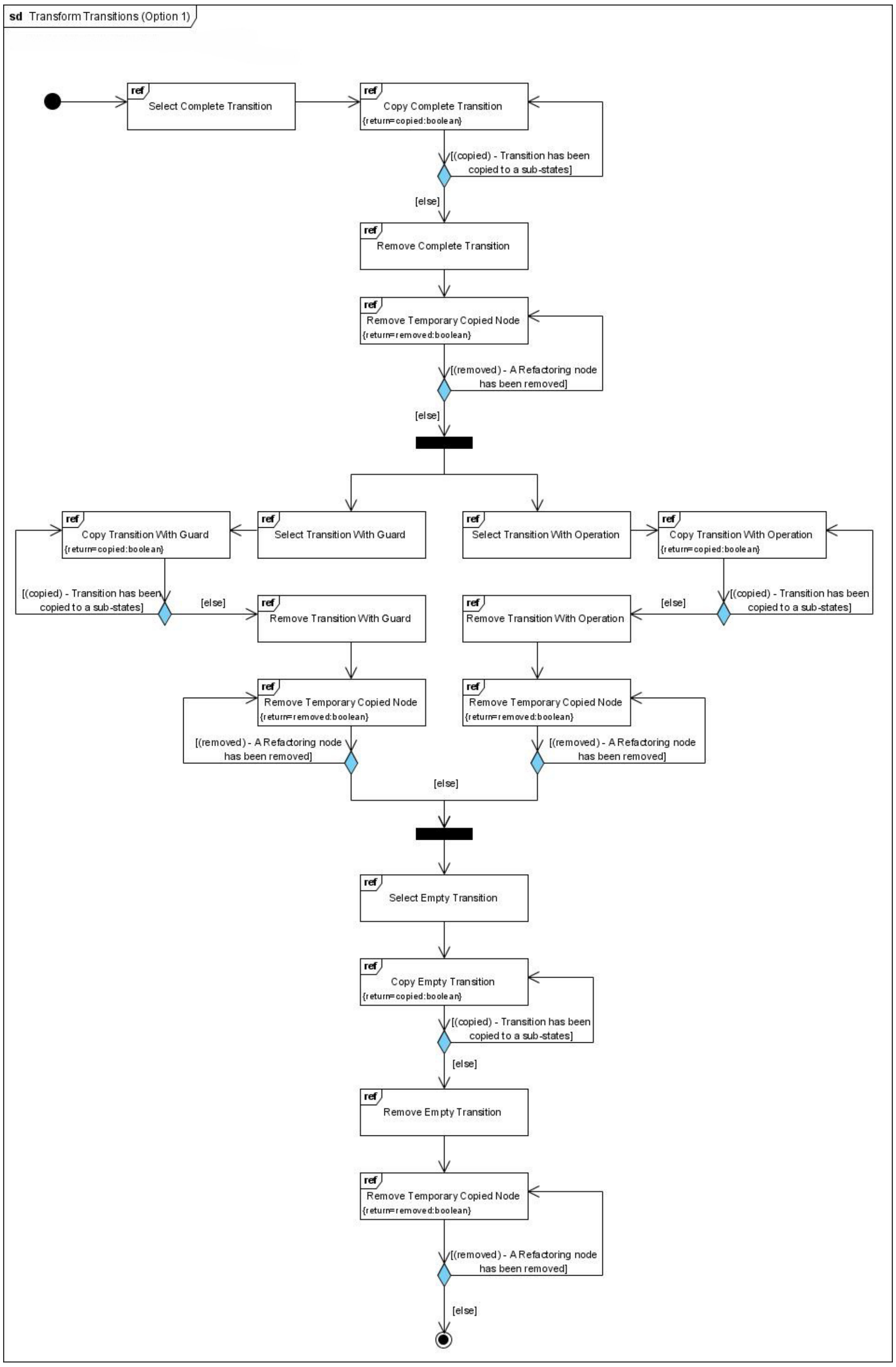

sd Transform Transitions (Option 1)
ref Select Complete Transition
ref Copy Complete Transition {return=copied:boolean}
[(copied) - Transition has been copied to a sub-states]
[else]
ref Remove Complete Transition
ref Remove Temporary Copied Node {return=removed:boolean}
[(removed) - A Refactoring node has been removed]
[else]
ref Copy Transition With Guard {return=copied:boolean}
ref Select Transition With Guard
ref Select Transition With Operation
ref Copy Transition With Operation {return=copied:boolean}
[(copied) - Transition has been copied to a sub-states]
[else]
ref Remove Transition With Guard
ref Remove Transition With Operation
[else]
[(copied) - Transition has been copied to a sub-states]
ref Remove Temporary Copied Node {return=removed:boolean}
ref Remove Temporary Copied Node {return=removed:boolean}
[(removed) - A Refactoring node has been removed]
[(removed) - A Refactoring node has been removed]
[else]
ref Select Empty Transition
ref Copy Empty Transition {return=copied:boolean}
[(copied) - Transition has been copied to a sub-states]
[else]
ref Remove Empty Transition
ref Remove Temporary Copied Node {return=removed:boolean}
[(removed) - A Refactoring node has been removed]
[else]

Figure 5.77: Transform Transitions Option 1

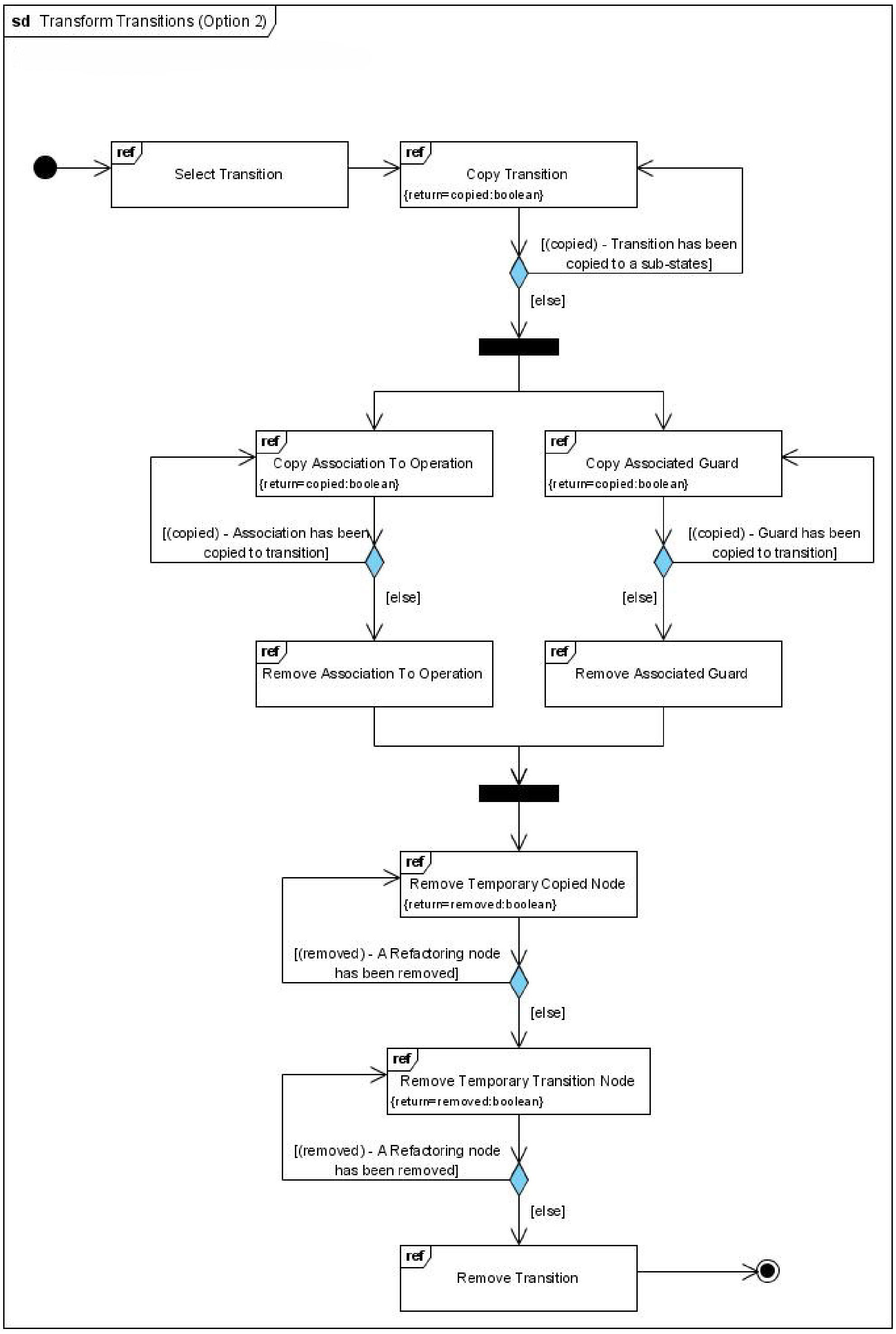

sd Transform Transitions (Option 2)
ref Select Transition
ref Copy Transition {return=copied:boolean}
[(copied) - Transition has been copied to a sub-states]
[else]
ref Copy Association To Operation {return=copied:boolean}
[(copied) - Association has been copied to transition]
[else]
ref Copy Associated Guard {return=copied:boolean}
[(copied) - Guard has been copied to transition]
[else]
ref Remove Association To Operation
ref Remove Associated Guard
ref Remove Temporary Copied Node {return=removed:boolean}
[(removed) - A Refactoring node has been removed]
[else]
ref Remove Temporary Transition Node {return=removed:boolean}
[(removed) - A Refactoring node has been removed]
[else]
ref Remove Transition

Figure 5.78: Transform Transitions Option 2

graph transformation rules will be $N_{rules} = 3 \times 2^{N_{opt}}$. If other optional nodes are added, the number of necessary graph transformation rules will increase exponentially.

Let N_s be the number of sub–states of the region and N_t the number of transitions that have to be modified, the computation cost of the transformation will be $C = N_t \times (2 + N_s)$.

The second option described in figure 5.78 uses a different approach to implement this transformation. A transition along with its event is selected and is copied to each sub–state of the region. If the original transition contains a guard or an associated operation, they will be copied to each newly created transition. For each optional node, there is a correspondent graph transformation rule that matches it and copies it. One more rule is necessary in order to delete the original transition.

The total number of necessary graph transformation rules of this solution will be $N_{rules} = 3 + N_{opt}$. If other optional nodes are added, the number of necessary graph transformation rules will increase linearly.

The computation cost of the transformation will be $C = N_t \times (2 + N_s + N_s \times N_{opt})$. The computation cost will increase proportionally to the number of optional elements. The graph transformation rules that remove the "Refactoring" node have been excluded from the computations because they are not relevant.

Both solutions are not really satisfiable, and an improvement of the graph transformation rules is necessary to achieve good results. The first approach will generate an higher number of transformation rules and it would not be easily maintainable and scalable. The second approach requires a smaller number of transformation rules but presents an higher computation cost.

For the current implementation, a combination of the two options has been used; this way it is possible to give an example of the graph transformation rules of both solutions. This solution is specific for the current graph representation and tries to limit at the same time the number of graph transformation rules and the computation cost of the transformation.

The final solution described in figure 5.61 is divided in two similar parts: the former manages the complete transitions, and the latter manages the transitions without the guard. The operation associated to the transitions is copied as described in the second option.

5.11.8 Flatten States Incoming Transitions

The *Flatten States Incoming Transitions* transformation is similar to the *Flatten States Outgoing Transitions*. It replaces an incoming transition, with the corresponding transitions associated to each sub–states, leading to the composite state.

Like for the *Flatten States Outgoing Transitions* transformation, if an *entry* action is defined in the composite state the refactoring becomes slightly more complicated. In order to preserve the behaviour of the system, it is necessary to move the *entry* action to the transitions for which it has to be executed and remove the *entry* action from the composite state.

The necessary transformation steps are the same as for the *Flatten States Outgoing Transitions* and they are functionally equivalents. In this case, the transformation steps take into account incoming transitions instead of outgoing transitions. The graph transformation rule shown in figure 5.79 has been implemented as example. It correspond to the step *Select Target State* shown in figure 5.63. The two graph transformation rules differs only for the order of the "source" and "target" edges associated to the "Transition" node. This consideration is correct for all the graph transformation rules defined in the *Flatten States Outgoing Transitions* transformation.

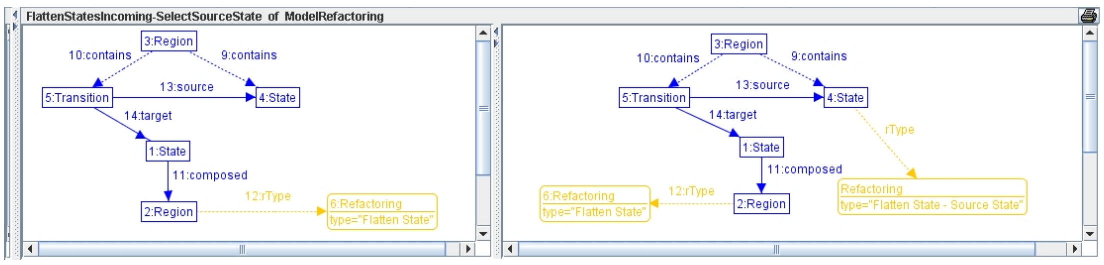


Figure 5.79: Flatten States Incoming Transitions - Select Source State

The AGG tool does not provide any mechanism to parametrise the edge defined in the graph transformation rule. The possibility to parametrise the edge defined in the graph transformation rules could be another possible enhancement and would make graph transformations more powerful.

Such a feature would permit to use the same set of graph transformation rules in order to apply the *Flatten States Outgoing Transitions* and *Flatten States Incoming Transitions* transformations.

The AGG tool offers another solution to this problem. It is possible to change the graph representation of UML models in order to have a single set of graph transformation rule. Figure 5.80 shows a simplified version of the Type Graph where "source" and "target" edges has been replaced by nodes.

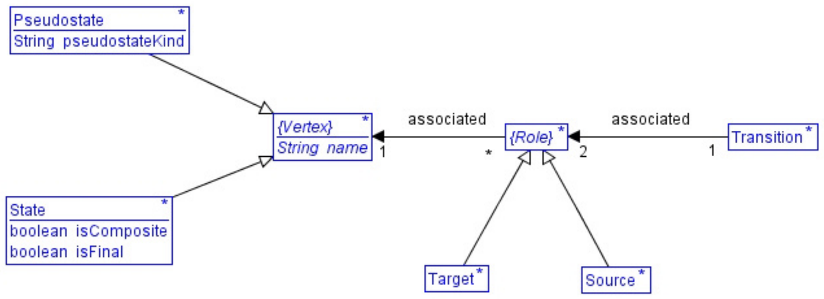


Figure 5.80: Type Graph alternative

The “Transition” node is associated with the “Vertex” node. The role of the association is specified using the “Source” and “Target” nodes. The “Role” node is defined as *abstract* node, i.e. it can not be instantiated in the graph. However, the “Role” node can be used to define the graph transformation rule.

Figure 5.81 shows the graph transformation rule that corresponds to the rules *Select Target State* and *Select Source State* previously discussed. This graph transformation rule allows to match at the same time both incoming and outgoing transitions.

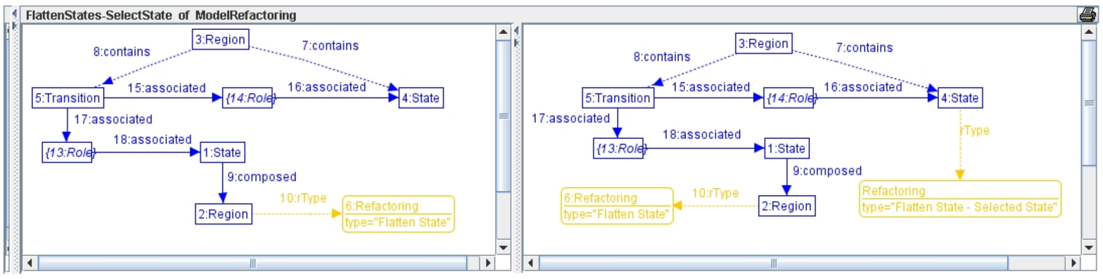


Figure 5.81: Flatten States Transitions - Select State

Using this graph representation of UML Models, it is necessary to add a check in order to ensure that a “Transition” node is associated with exactly one “Source” node and exactly one “Target” node. It is not possible to express such a condition in the Type Graph.

5.11.9 Consequences

This kind of refactoring produces a more intuitive model but does not really improve the structure if it is used by itself. It is often part of more complex refactorings, or in general it is necessary in order to apply some complex changes to the models. It is usually combined with the *Remove Region* refactoring.

Some improvements may be done to this refactoring, in order to support the actions attached to states, such as *do*, *entry*, and *exit* actions. Moreover, it is possible to extend the graph representation of UML Models in order to have a single set of graph transformation rule.

Chapter 6

Model Refactoring Tool

In this chapter we will illustrate the possibility of developing model refactoring tools using graph transformations. For this purpose, we have developed a prototype application that serves as a feasibility study. Using the prototype application, we have also performed a validation of model refactorings previously formalised and we have verified their correctness.

6.1 Implementation

The AGG graph transformation engine is delivered together with an API (Application Programming Interface) that allows to integrate the internal graph transformation engine into other environments.

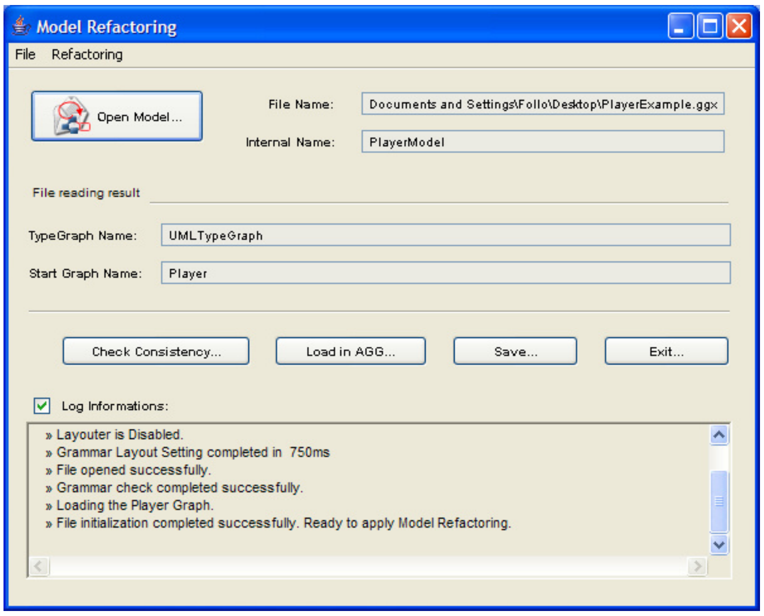
We have chosen to developed a prototype application using the AGG API. That way, it has been possible to make use of the graph transformation rules defined in chapter 5. The prototype application serves as a feasibility study to illustrate that development of model refactoring tools is possible. Moreover, it serves to verify the correctness of the model refactorings previously formalised. This prototype application will supply guidelines for future work and put into evidence some possible enhancements for AGG in order to better support model refactorings.

The representation of the control flow has been a crucial point for the implementation of the prototype application. The control flow describes the order in which the individual graph transformation rules of each model refactoring have to be executed.

At the moment, the AGG tool does not provide any satisfactory solution for organizing and combining rules, and the supplied mechanisms were not sufficient for describing model refactorings.

The prototype application avoids the underlied problem by using a custom control structure that represents the control flow of model refactorings. In section 6.2.1 we will describe how this control structure has been implemented.

Figure 6.1 shows the graphical user interface of the model refactoring application that we developed in JAVA by making use of the AGG API.



File Name: Documents and Settings\Follo\Desktop\PlayerExample.ggx
Internal Name: PlayerModel
File reading result
TypeGraph Name: UMLTypeGraph
Start Graph Name: Player
☑ Log Informations:

```
» Layouter is Disabled.
» Grammar Layout Setting completed in  750ms
» File opened successfully.
» Grammar check completed successfully.
» Loading the Player Graph.
» File initialization completed successfully. Ready to apply Model Refactoring.
```

Figure 6.1: Model Refactoring Application

The application internally loads a file containing the model refactoring specifications and the necessary graph transformation rules. It allows then to open files in GGX format containing the UML models to be refactored. The UML model must respect the syntax, or type graph, defined in section 4.4. Using the “Refactoring” context menu, the user can apply the different model refactorings. When necessary, the user will be prompted to enter the input parameter values and if necessary to supply a match if the model refactoring can be applied to different parts of the UML model.

The prototype application also allows the user to save the resulting UML model and visualise it using the AGG graphical user interface.

6.2 Control Structures

As anticipated in section 6.1, there exists a need for a high-level control structure that can drive the application of elementary graph transformation rules and allows to manage the complexity of model refactorings.

The syntax of *UML Interaction Overview diagrams* has been used in chapter 5 to formally depict the control flow and to specify in which order the rules must be applied (see figure 5.41 for an example of Interaction Overview diagrams). The *interaction occurrence frames* that compose the diagram indicate an activity or operation to be invoked. For the purpose of defining model refactoring, they have been associated to graph transformation rules. Some custom notations have been added to enrich the diagram with all the necessary informations. In particular, the input and output parameters for each atomic step have been reported.

In order to apply a model refactoring, it is necessary to provide the corresponding control flow to the model refactoring application. We have focused our efforts on searching a suitable control structure that allows representation of the control flow of model refactorings and drives the application of graph transformation rules.

The AGG tool does not provide any satisfactory solution for organizing and combining rules, and the supplied mechanisms were not sufficient for describing model refactorings. In section 6.2.1 we will explain the custom control structure that we have implemented in order to represent the control flow of model refactorings and drive the application of graph transformation rules. Section 6.2.2 outlines the control structures available in the AGG tool.

6.2.1 Graph–based control structure

The major concept of this dissertation is the representation of UML models using graph–based structures. The same way, the control flows based on the *UML Interaction Overview diagram* syntax have been represented using graphs, and have been used to drive the application of graph transformation rules. The custom control structure adds the notion of *"controlled" graph transformation*, which was not previously available in AGG.

When the prototype application has to apply a model refactoring, it loads the corresponding graph representing the control flow. It searches the starting point and prompts the user to insert the necessary parameter values. The application walks through the graph to determine which graph transformation rules have to be applied. It continues exploring the graph until it reaches a *final point* reporting the result to the user.

Figure 6.2 reports the Type Graph implemented with AGG that we have defined in order to represent the control flow.

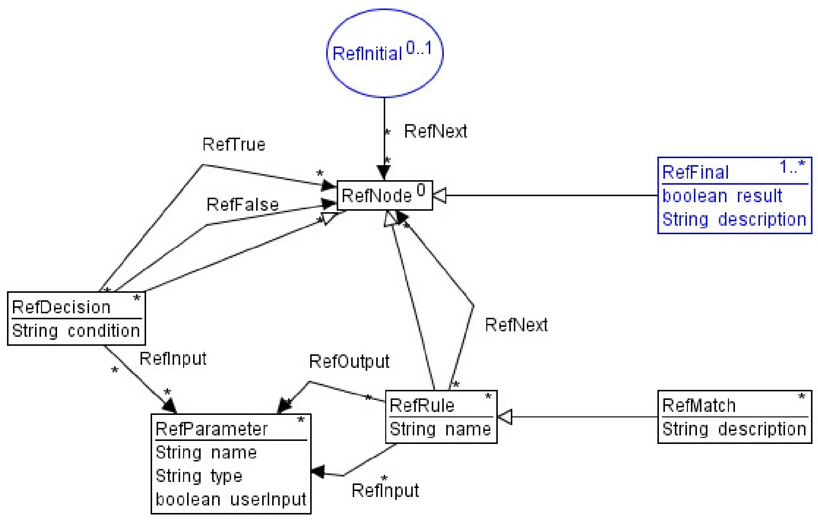


Figure 6.2: Interaction Overview Diagram Type Graph

The *RefInitial* entity corresponds to the starting point of the flow of control. The *RefFinal* entity is used to terminate the execution. Several nodes of this kind may exist at the same time in order to specify a successful conclusion of the refactoring or to signal errors during its execution.

The *RefDecision* entity corresponds to the *decision point* of the *UML Interaction Overview diagram*, like the *RefRule* entity corresponds to the *interaction occurrence frames*. The *RefNode* entity has been added in order to simplify the Type Graph, but it can not be instantiated in the graph. The *RefMatch* entity does not correspond to any entity of the *UML Interaction Overview diagram*, but is used to select a list of matches which will be subsequently used to apply a graph transformation rule.

The *RefParameter* entity is used to define the input and output parameters for each refactoring step. Only *RefRule* and *RefMatch* are allowed to set output parameters. Table 6.1 illustrates the meaning of the output parameters based on their types.

The *RefParameter* entities can be defined as input parameters for the refactoring steps. In this case, values defined by the user are also permitted. The attribute *userInput* of the *RefParameter*

	RefRule	**RefMatch**
Matches	–	*List of selected matches*
Int	–	*Number of matches*
Boolean	*Result of the execution*	–
String	–	–

Table 6.1: Meaning of the output parameters.

entity is used to detect which parameters require user interaction.

Decision points contain boolean expressions which must be evaluated by the application, in order to determine the subsequent step of the model refactoring. For that purpose, we have added an interpreter to the prototype application.

Figure 6.3 shows the control flow we have implemented for the *Introduce Initial Pseudostate* refactoring. It corresponds to the one reported in figure 5.41.

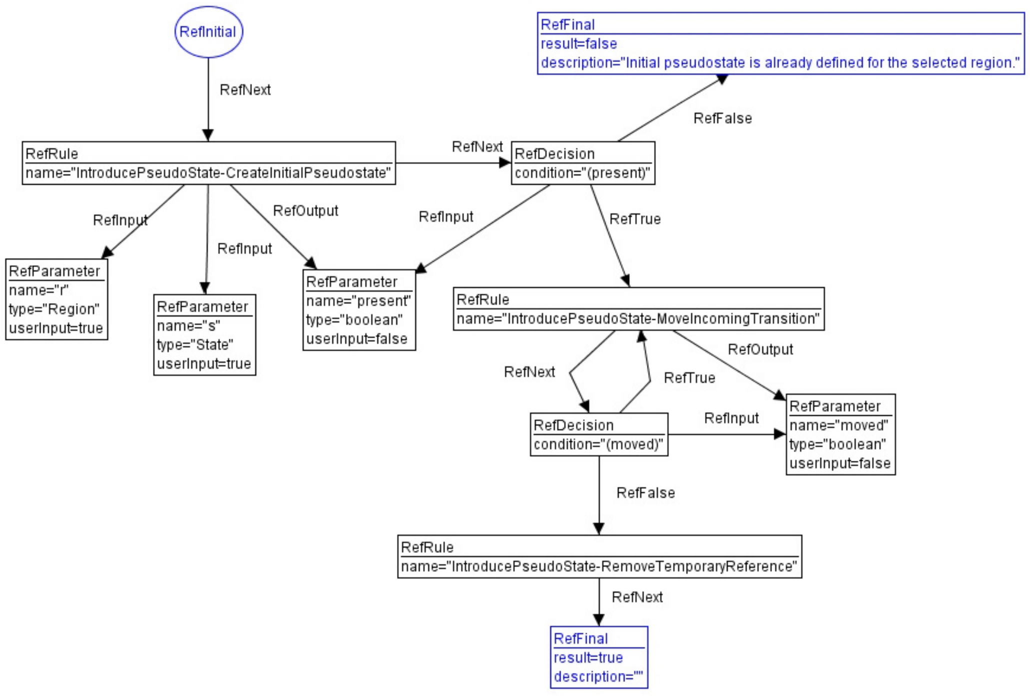


Figure 6.3: Control Flow – Introduce Initial Pseudostate

The *RefInitial* node is the starting point of the flow of control. The user has to specify two input parameters that are provided to the subsequent rule named *IntroducePseudoState-CreateInitialPseudostate*. The parameter r specifies which is the composite state or region that will be modified by the refactoring. The parameter s specifies the default state of the region.

The result of the execution of the rule is saved to the parameter named $present$. It is provided as input parameter to the subsequent *RefDecision* node, identified using the *RefNext* edge. The *RefDecision* node evaluates the value of the parameter $present$. If the value is "false", the subsequent step is a *RefFinal* node that signal the error: the refactoring can not be applied because an Initial Psuedostate is already present in the selected region.

If the value of the parameter $present$ is "true", the rule named *IntroducePseudoState-MoveIncomingTransition* will be executed. It will be repeated for each transition that lead to the default state. When all transitions have been moved, the rule named *IntroducePseudoState-RemoveTemporaryReference* will be executed. It removes all "Refactoring" nodes added to the graph during the execution of the first step. The target of the *RefNext* edge associated to the node is a *RefFinal* node. It specifies a successful conclusion of the refactoring.

6.2.2 AGG Rule Sequence

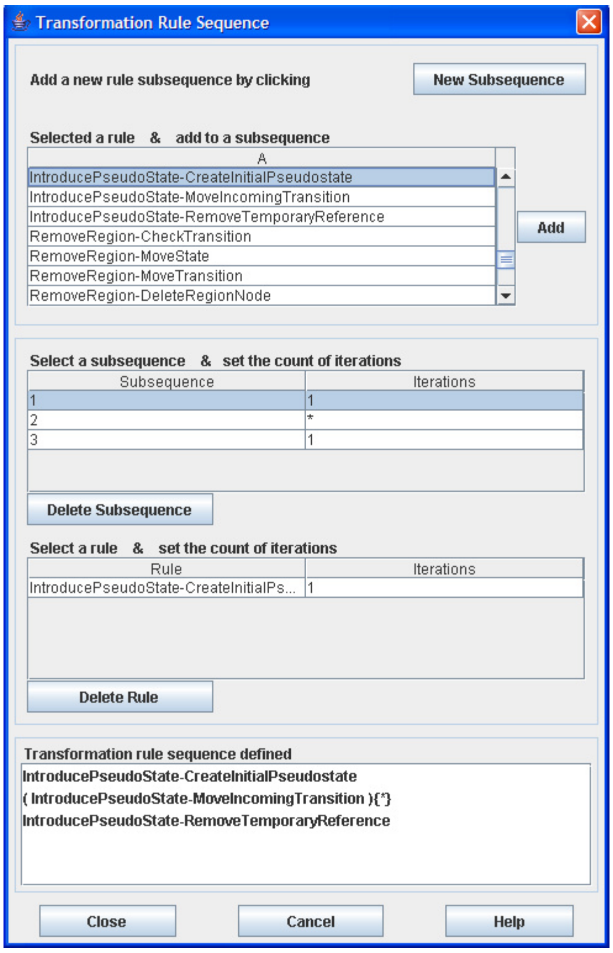
The AGG graph transformation tool [24] allows the use of *rule layers* in order to coordinate the application of rules. Each rule is assigned to a certain layer. Rules of the same layer are applied in a non–deterministic manner. Starting with layer 0, the rules of one layer are applied as long as possible. Thereafter, the next layer is executed. After execution of the highest layer, the transformation is finished.

Recently, a newer version of the AGG tool has been released [31]. The graph transformation engine has been improved and two new control structures for graph transformation have been added.

Control of the execution order is now possible by setting *rule priorities*. However, this option can not be combined with the use of layers, and they appear to be very similar. Both those mechanisms do not suffice to express model refactoring.

The real improvement of the newer version of AGG is the possibility to define a *transformation rule sequence* that specifies in which order the rules must be applied. A transformation rule sequence is a control structure which defines an ordered set of rule subsequences and rule iterations.

The dialog in figure 6.4 is part of the AGG tool and allows to define a rule sequence. In particular, it shows the *transformation rule sequence* that corresponds to the *Introduce Initial Pseudostate* refactoring described in section 5.8.5.


Transformation Rule Sequence
Add a new rule subsequence by clicking
New Subsequence
Selected a rule & add to a subsequence
Add
Select a subsequence & set the count of iterations
Delete Subsequence
Select a rule & set the count of iterations
Delete Rule
Transformation rule sequence defined
Close
Cancel
Help

A
IntroducePseudoState-CreateInitialPseudostate
IntroducePseudoState-MoveIncomingTransition
IntroducePseudoState-RemoveTemporaryReference
RemoveRegion-CheckTransition
RemoveRegion-MoveState
RemoveRegion-MoveTransition
RemoveRegion-DeleteRegionNode

Subsequence	Iterations
1	1
2	*
3	1

Rule	Iterations
IntroducePseudoState-CreateInitialPs...	1

```
IntroducePseudoState-CreateInitialPseudostate
( IntroducePseudoState-MoveIncomingTransition ){*}
IntroducePseudoState-RemoveTemporaryReference
```

Figure 6.4: AGG rule sequence example

Looking at the textual view of the *transformation rule sequence* reported in the bottom of the dialog, we can see three rule subsequences that are being performed. Creation of a new empty subsequence is possible using button *“New Subsequence”*. By selecting one or more rules in the first table and clicking on the button *“Add”*, they can be put into the currently selected subsequence. The first subsequence with the rule *IntroducePseudoState-CreateInitialPseudostate* should be applied only once. The *Iterations* field of the second and third tables can be used to define how long a rule subsequence or a single rule should be applied. It is also possible to put a star (*), which means “as long as possible”. The second subsequence *IntroducePseudoState-MoveIncomingTransition* should be applied as long as possible, and the third subsequence with the rule *IntroducePseudoState-RemoveTemporaryReference* should be applied only once.

The behaviour of this *transformation rule sequence* corresponds to the one described in figure 5.41, meaning that for one set of input values the resulting set of output values is the same. However, a relevant difference can be noticed: when the *IntroducePseudoState-CreateInitialPseudostate* rule of the *transformation rule sequence* fails, the subsequent rules will be executed producing no changes. In fact, it would be preferable to stop the refactoring.

This difference is caused by some limitations of the *transformation rule sequence*. It does not allow to have a complete control of the rules execution process. It is not possible to stop the application of rules and it is not possible to apply a different rule, nor set of rules, depending on the result of a previous one.

Moreover, the *transformation rule sequence* allows the definition of just one level of subsequences, and it is not possible to group the rules in a more complex way. The refactoring *Flatten States Outgoing Transition* described in section 5.11.5 requires at least one more level of subsequence in order to be correctly represented.

The limitation of having only one *transformation rule sequence* definitely discourages its use in order to represent model refactorings. It would be preferable to have a *transformation rule sequence* corresponding to each model refactoring.

6.3 Validation

In this section the results of some detailed tests that have been performed with the prototype application are reported. The Player model previously illustrated has been used as running example. It was not possible to test the implemented application using industrial UML models, because the conversion of UML models to graphs has not been automated yet and it would go beyond the scope of this dissertation. Other researches are currently exploring the possibility to automatically convert UML model to graphs.

The correctness of model refactorings has been validated through a large number of tests. The resulting graphs have been analysed in order to ensure that they conform to the Type Graph defined in chapter 4.4. The AGG tool allows easy verification of this requirement. The conversion of graphs to UML models has not been automated yet. We have manually verified that the resulting graphs conform to the UML metamodel. Each graph has been analysed in order to ensure that it corresponded to a consistent UML Model.

When possible, we have subsequently applied the inverse model refactoring. We have compared the resulting graph to the original one in order to ensure they are identical.

6.3.1 Pull Up Operation and Push Down Operation

The example shown in this section corresponds to the one illustrated in figure 5.2.

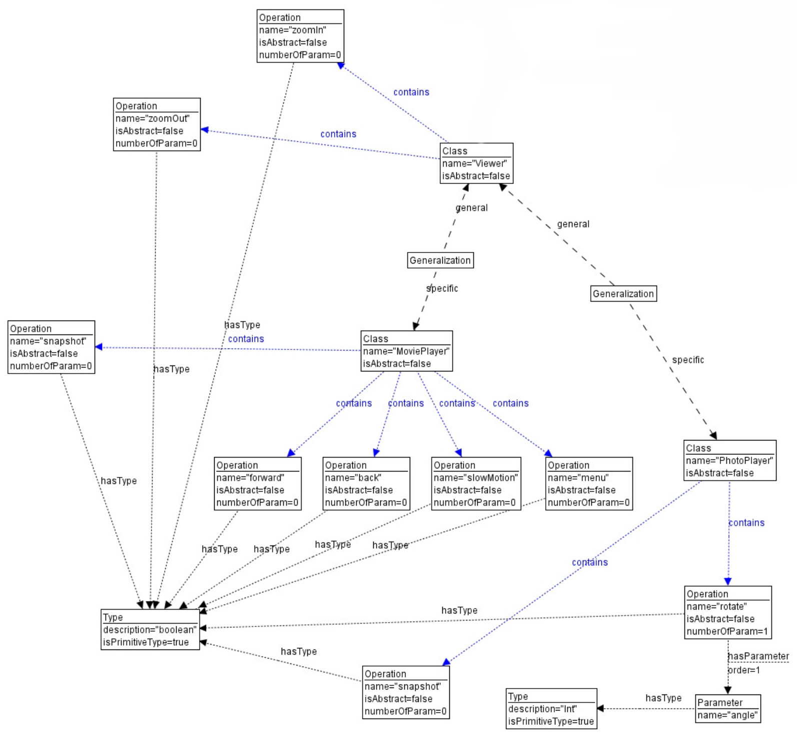


Figure 6.5: Operation Pushed Down

Figure 6.5 shows the graph produced by the prototype application after application of the *Push Down Operation* refactoring to the graph reported in figure 6.6. The snapshot operation has been pushed down to the classes MoviePlayer and PhotoPlayer.

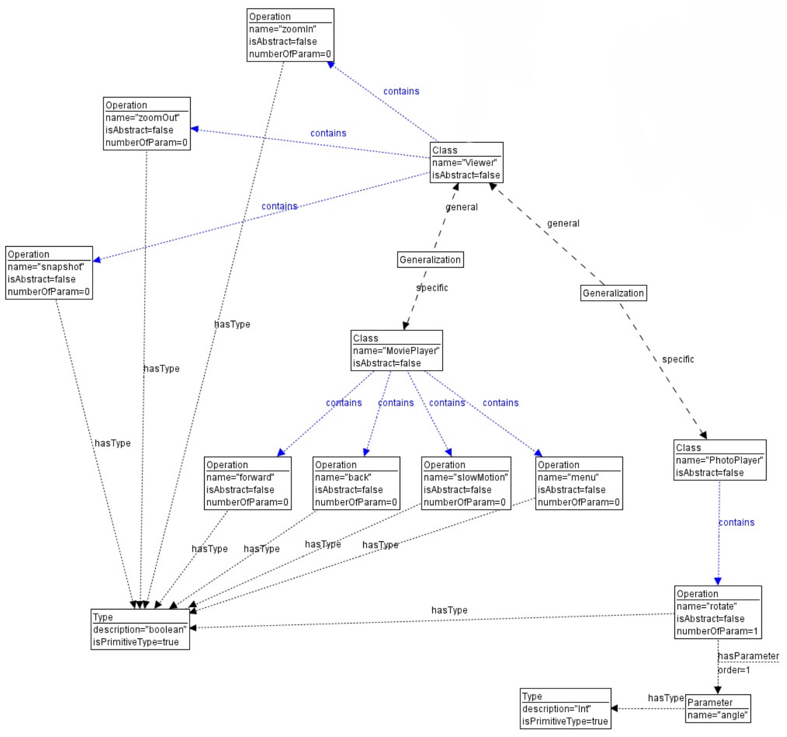


Figure 6.6: Operation Pulled Up

Figure 6.6 show the graph produced by the prototype application after application of the *Pull Up Operation* refactoring to the graph reported in figure 6.5. The snapshot operation has been pulled up to the class Viewer.

The *Pull Up Operation* and *Push Down Operation* refactorings have been applied multiple times in order to verify that they produce each time the same results.

6.3.2 Introduce Initial Pseudostate

The example shown in this section corresponds to the one illustrated in figure 5.40. Figure 6.7 shows the starting graph representing the State Machine diagram. The *ACTIVE* composite state does not contain an initial pseudostate.

Figure 6.8 shows the graph produced by the prototype application after the execution of the *Introduce Initial Pseudostate* refactoring. An initial pseudostate has been added to the *ACTIVE*

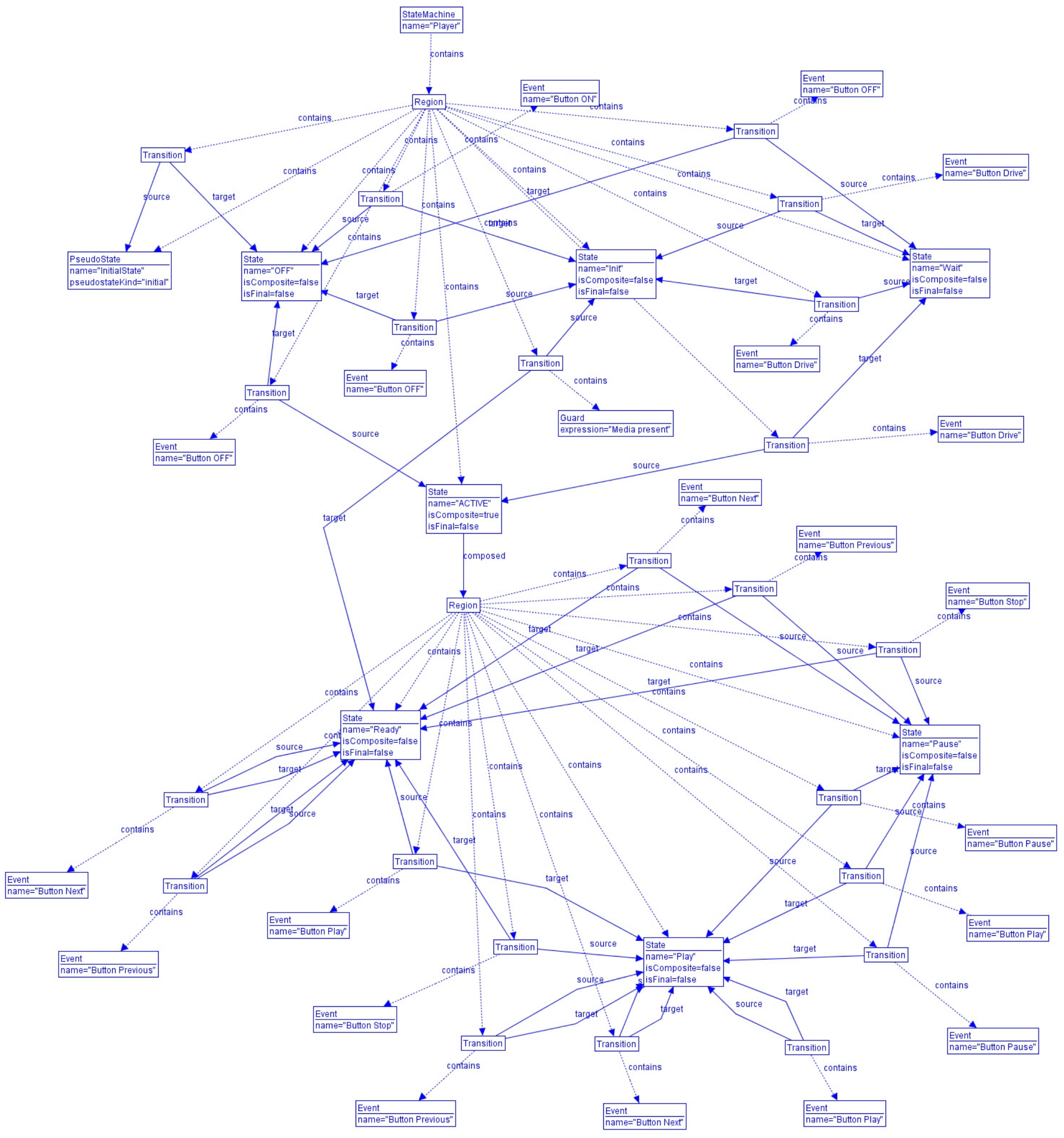

StateMachine name="Player"
Region
Event name="Button ON"
Event name="Button OFF"
Transition
Event name="Button Drive"
PseudoState name="InitialState" pseudostateKind="initial"
State name="OFF" isComposite=false isFinal=false
State name="Init" isComposite=false isFinal=false
State name="Wait" isComposite=false isFinal=false
Event name="Button OFF"
Event name="Button Drive"
Guard expression="Media present"
Event name="Button OFF"
Event name="Button Drive"
State name="ACTIVE" isComposite=true isFinal=false
Event name="Button Next"
Event name="Button Previous"
Event name="Button Stop"
Region
State name="Ready" isComposite=false isFinal=false
State name="Pause" isComposite=false isFinal=false
Event name="Button Pause"
Event name="Button Next"
Event name="Button Play"
Event name="Button Previous"
Event name="Button Play"
State name="Play" isComposite=false isFinal=false
Event name="Button Stop"
Event name="Button Pause"
Event name="Button Previous"
Event name="Button Next"
Event name="Button Play"
contains
source
target
composed

Figure 6.7: Initial Pseudostate not present

composite state. An automatic transition has been defined between the initial pseudostate and the *Ready* state.

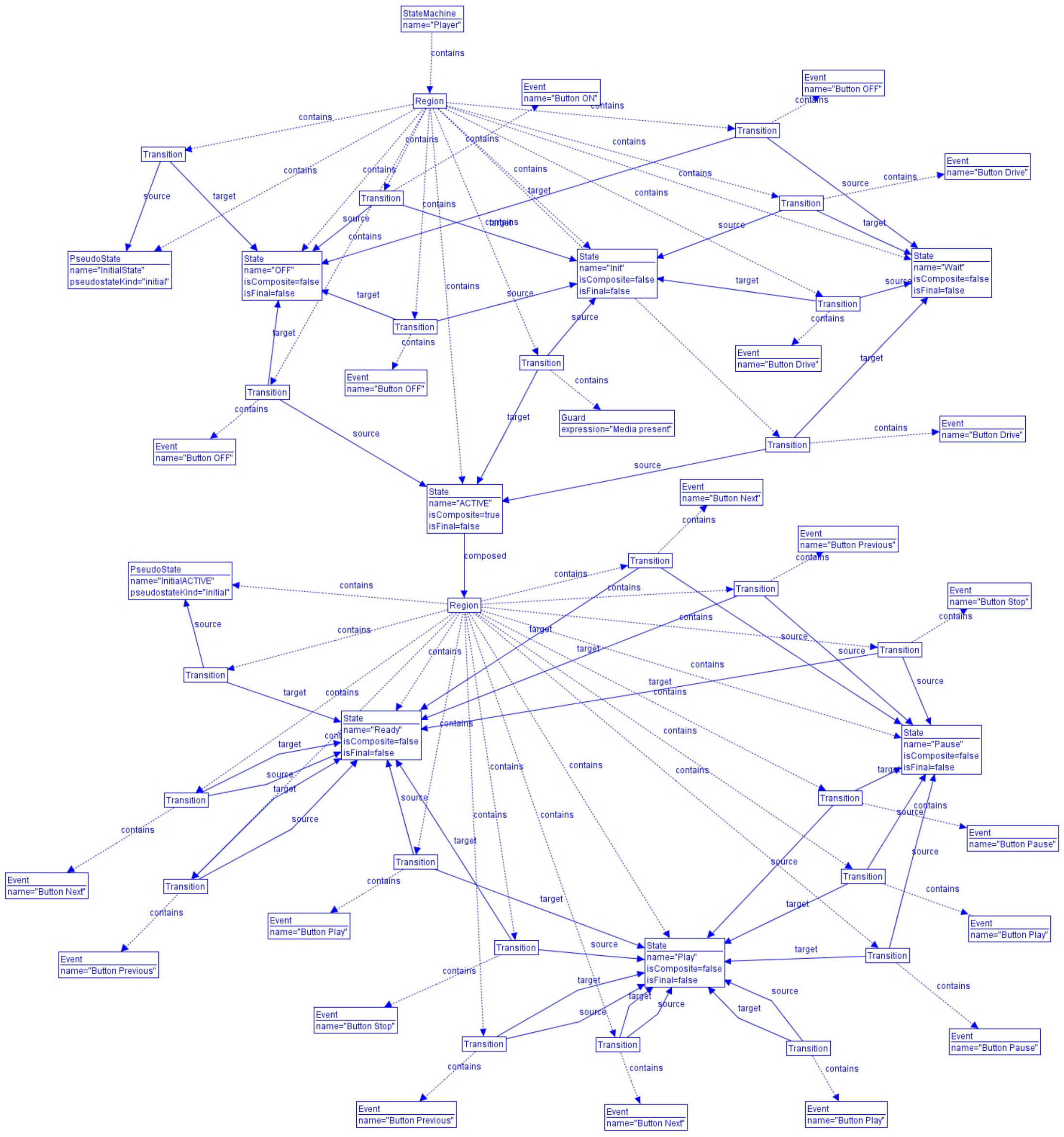

StateMachine name="Player"
contains
Region
Event name="Button ON"
Event name="Button OFF"
Event name="Button Drive"
Transition
PseudoState name="InitialState" pseudostateKind="initial"
State name="OFF" isComposite=false isFinal=false
State name="Init" isComposite=false isFinal=false
State name="Wait" isComposite=false isFinal=false
Guard expression="Media present"
State name="ACTIVE" isComposite=true isFinal=false
composed
PseudoState name="InitialACTIVE" pseudostateKind="initial"
Event name="Button Next"
Event name="Button Previous"
Event name="Button Stop"
Event name="Button Play"
Event name="Button Pause"
State name="Ready" isComposite=false isFinal=false
State name="Pause" isComposite=false isFinal=false
State name="Play" isComposite=false isFinal=false
source
target

Figure 6.8: Initial Pseudostate present

The *Ready* state has become the default initial state of the *ACTIVE* region. A transition whose target is the *ACTIVE* state will lead the State Machine to the *Ready* state.

6.3.3 Flatten States Transitions

The example shown in this section corresponds to the one illustrated in figure 5.59. Figure 6.7 shows the starting graph representing the State Machine diagram.

Figure 6.9 shows the graph produced by the prototype application after execution of the *Flatten States Outgoing Transition* refactoring. Before applying the refactoring, there are transitions whose source is the *ACTIVE* region. The refactoring adds similar transitions to each sub–state of the *ACTIVE* region and removes the originals.

6.3.4 Remove Region

The example shown in this section corresponds to the one illustrated in figure 5.45(b)(a). Figure 6.9 shows the starting graph representing the State Machine diagram.

Figure 6.10 shows the graph produced by the prototype application after the execution of the *Remove Region* refactoring. The “ACTIVE” composite state containing a region has been removed from the State Machine diagram. After the refactoring, the states “Ready”, “Pause”, and “Play” are contained in the upper region, directly connected to the State Machine.

6.3.5 Introduce Region

The example shown in this section corresponds to the one illustrated in figure 5.45(a)(b). Figure 6.10 shows the starting graph representing the State Machine diagram.

Figure 6.11 shows the graph produced by the prototype application after the execution of the *Introduce Region* refactoring. The “ACTIVE” composite state containing a region has been added to the State Machine diagram. The states “Ready”, “Pause” and “Play” have been moved into the region. Transitions between states contained in the new region has been moved to the region itself.

The refactoring has added the region that was removed in the previous example. The resulting graph shown in figure 6.11 is identical to the one illustrated in figure 6.9.

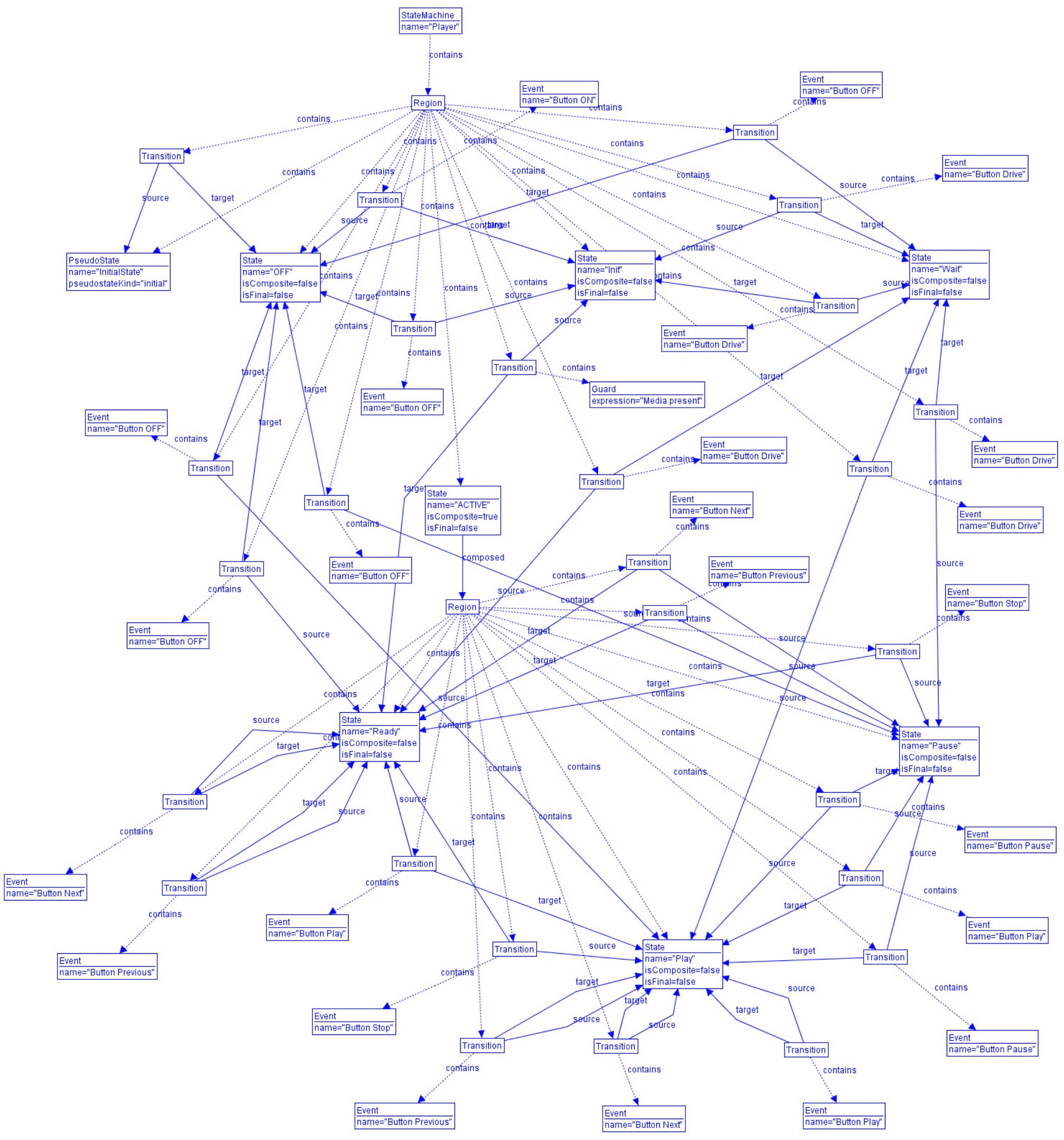


Figure 6.9: Flatten States Transitions

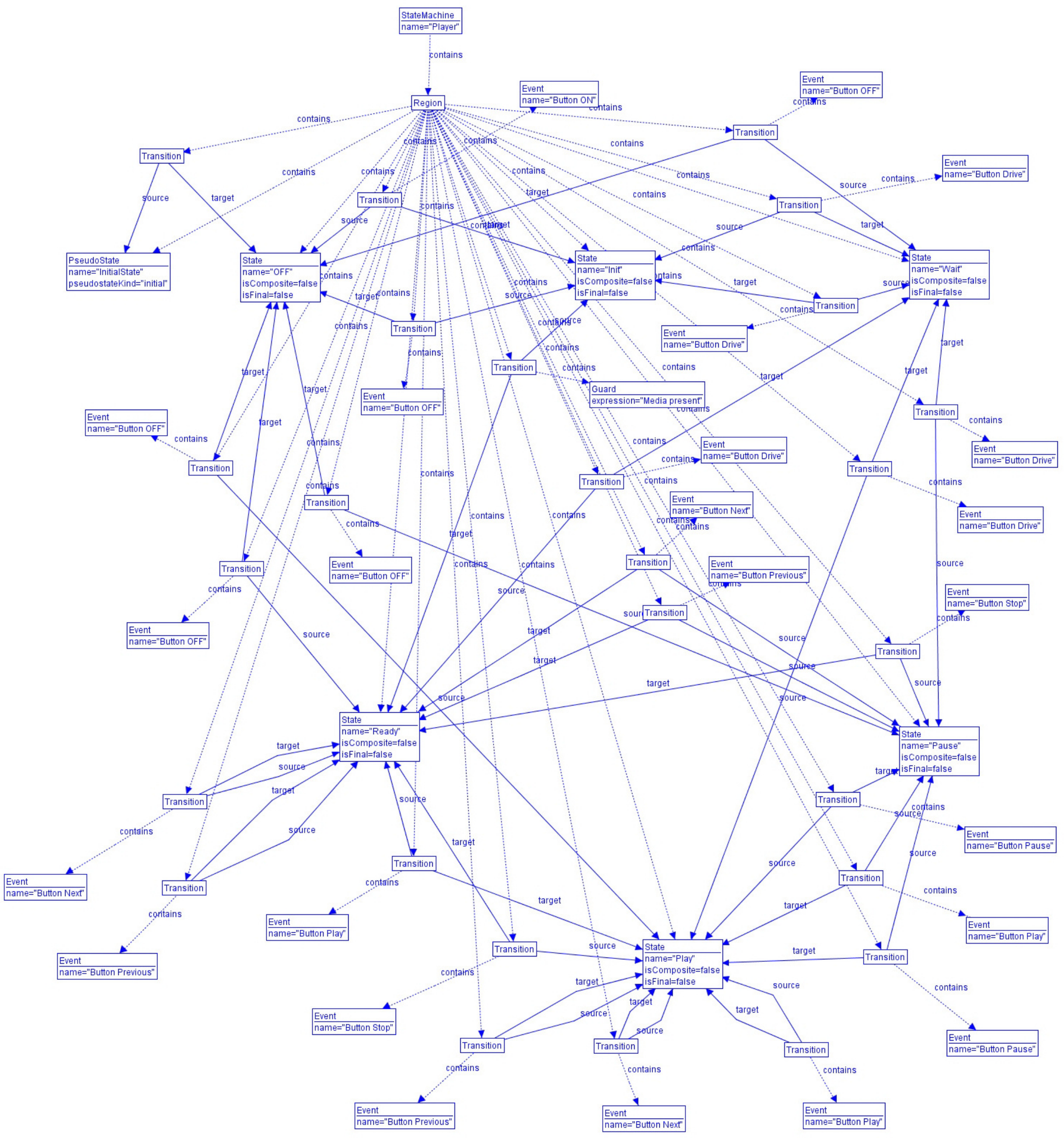


Figure 6.10: Remove Region

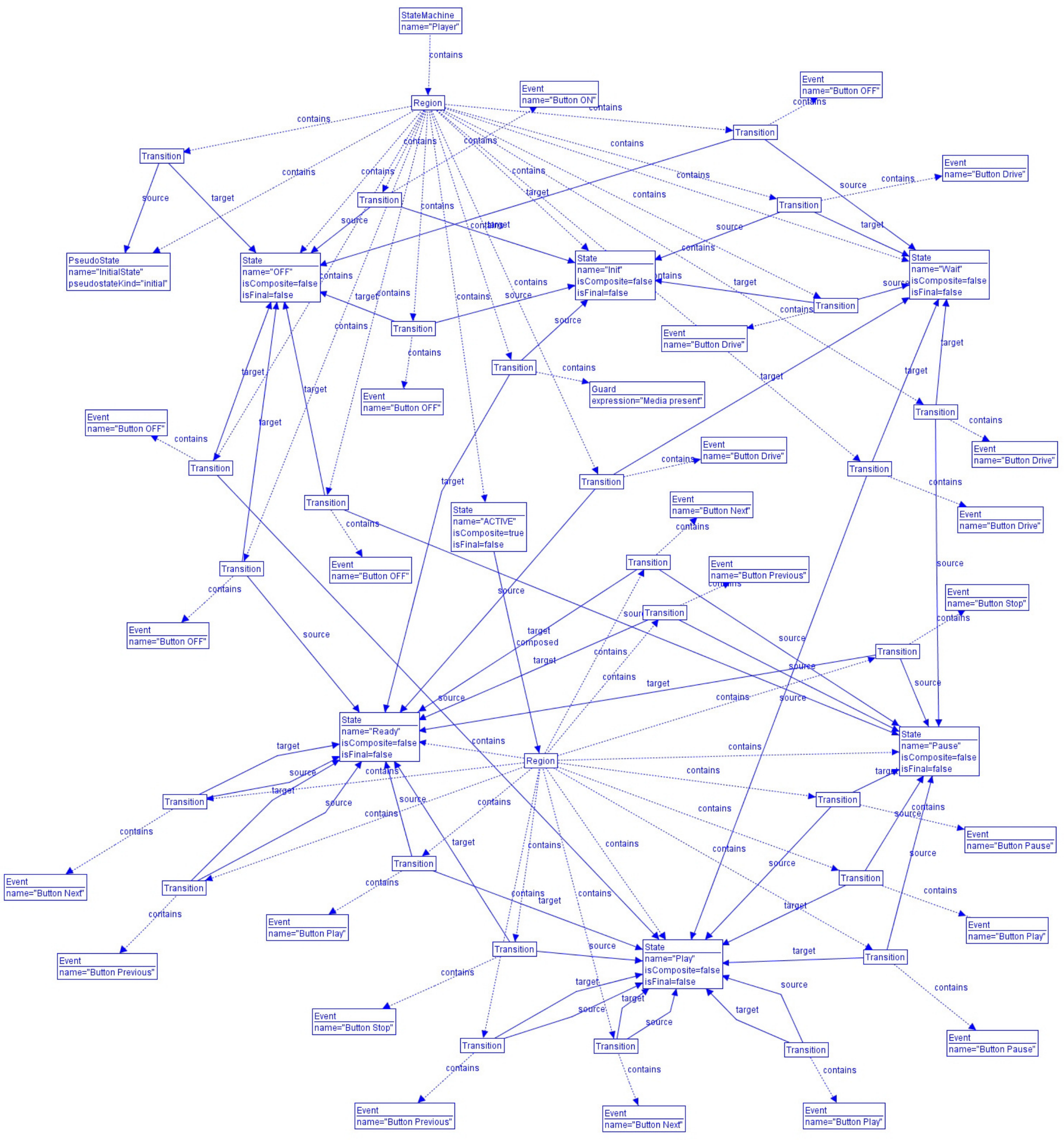


Figure 6.11: Introduce Region

6.4 Limitations and known issues

The testing phase has given us the possibility of finding bugs in the developed application. Moreover, this phase has been very useful in order to discover errors or not well–defined concepts in the refactoring specification and the Type Graph specification.

Some changes have been apported to the Type Graph in order to better support model refactorings. The graph representation of UML Models has been completed with missing elements. The custom control structure used to drive the application of the graph transformation rules has been updated in order to better specify the control flow. For example, in the initial version of the Type Graph the *RefFinal* node did not contain attributes *result* and *description* used to specify a successful conclusion of the refactoring or to signal errors during its execution.

During the testing phase, errors have been found in the implementation of model refactorings. We have corrected the graph transformation rules that did not work as expected and we have added new rules in order to manage conditions that had not been considered. For example, the *Move Self Transition* step illustrated in figure 5.50 has been added after some tests of the *Introduce Region* refactoring. In fact, an analysis of the resulting graphs has revealed that self transitions were not moved by transformation rules.

The development of the prototype application and the testing phase have put in evidence some problems and limitations of the AGG tool, too. Moreover, in some cases, the documentation of the AGG API is not sufficiently detailed and the development of the application has been sometimes more complex than we initially thought.

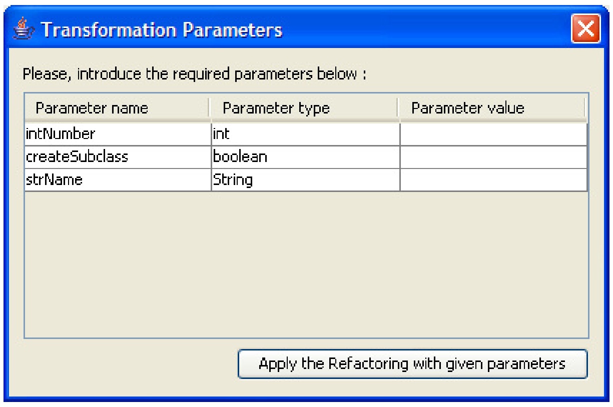
In section 6.4.1 we will give an overview of the limitations on the prototype and AGG. In section 6.4.2 we will summarise the known issues of the developed application.

6.4.1 Limitations

The prototype application presents some limitations and it slightly differs from the guidelines defined in chapter 5. For each model refactoring we have formally defined the input parameters that have to be provided to each atomic step. Two different types of input parameters are allowed in our approach, i.e. *Java types* and *Node types*.

The *Java types* consist of the *Java primitive types* (i.e., String, int, boolean and Float) and *Data types* which may come from user–defined Java classes. The parameters of those types are defined as variables in graph transformation rules. They can be used for evaluating JAVA expressions during the application of rules or for assigning a value to the attributes.

The prototype application considers only the *String*, *int*, and *boolean* JAVA primitive types. This is due to the fact that they are the only types used by the current graph representation of UML models. Figure 6.12 shows the dialog of the application used to fill in the input parameter values.


Transformation Parameters
Please, introduce the required parameters below :
Apply the Refactoring with given parameters

Parameter name	Parameter type	Parameter value
intNumber	int	
createSubclass	boolean	
strName	String	

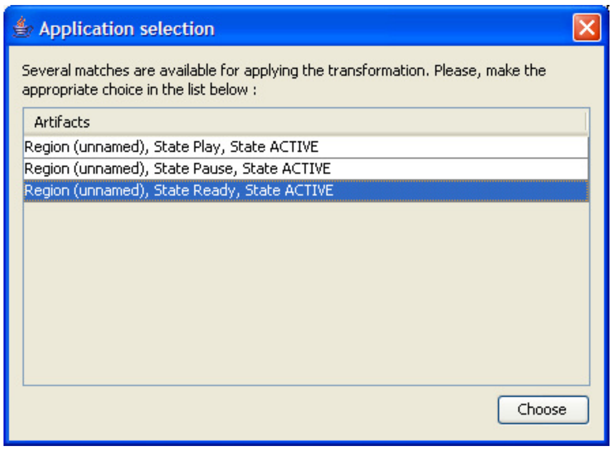
Figure 6.12: Application Dialog – Primitive parameters

Node types correspond to all node types defined in the Type Graph. The nodes supplied as input parameter specify a partial match between the graph and the left–hand side of the graph transformation rule. Using the graphical user interface of AGG it is possible to match nodes of the left–hand side with nodes of the graph before the application of the graph transformation rule.

AGG does not allow to use *Node types* in order to specify the type of other entities. For example, only *JAVA types* are allowed for attributes in the graph. This is a limitation that is not present in other graph transformation tools.

The prototype application adopts a simple approach to deal with input parameters. The user is asked to fill in only the value of parameters whose type is a *JAVA primitive type*. Subsequently, the transformation engine searches the possible matches for the graph transformation rule that has to be execute. If the pattern of the rule applies to different parts of the UML model, the user will be prompted to choose the interested match.

Figure 6.13 shows the dialog of the application used to choose a match when the rule can be applied to different parts of the UML model. In particular, it shows the possible matches for the rule *Create Initial Pseudostate* of the *Introduce Initial Pseudostate* refactoring. In the graph


Application selection

Several matches are available for applying the transformation. Please, make the appropriate choice in the list below :

Artifacts
Region (unnamed), State Play, State ACTIVE
Region (unnamed), State Pause, State ACTIVE
Region (unnamed), State Ready, State ACTIVE

Figure 6.13: Application Dialog – Match selection

of the running example only the composite state named "ACTIVE" is present. The composite state contains three sub–states: each of them can be chosen as default state of the region. If the pattern of the graph transformation rule is more complex, it become slightly difficult to understand which parts of the UML Model correspond to the matches proposed by the dialog.

The same way, when the model refactoring requires a list of nodes, the user will be asked to select the items he is looking for within the list of complete matches found for the current rule. For example, the user is prompted to select the set of operations that have to be exported during the execution of the *Extract Class* refactoring.

This behaviour of the prototype application is due to the fact that the AGG tool does not manage *Node types* in the same way as *JAVA types*.

6.4.2 Known issues of the prototype

The model refactoring application lacks some features that have not been implemented yet, because they do not affect the normal functioning of the program nor the correctness of the model refactoring tests.

- The application allows the user to visualize the graph using the AGG graphical user interface. Closing the GUI will cause the program termination. It is therefore preferable to save the UML model and open it manually using the AGG tool.

- The user is requested twice to specify the value for input parameters defined in the control flow. The user is first requested to specify the input parameter value that will be provided to the transformation rule. When the transformation rule is going to be applied, the user is requested to confirm the parameter value.

- In case of a failure occuring during the execution of the model refactoring, the previous version of the model should be restored. When a model is restored the layout of the graph is lost and the graph is unreadable. Recently, the version 1.6.0 of the AGG tool has been released [31]. It better supports the "Undo" and "Redo" of editing operations and transformation steps.
- Once the refactoring has been started, it is not possible to stop its execution. It will be preferable to add the possibility to terminate it.

Chapter 7

MOFLON

In this chapter we will compare AGG with other graph transformation tools in order to explore whether the latter could provide better support for model refactoring. This comparison allows to give more insight in how AGG support could be improved.

The *MOFLON meta modeling framework* appears to be a suitable tool for the specification of the refactoring of UML models. This chapter outlines the capabilities of the MOFLON tool with the help of an example. A simplified representation of UML State Machine diagrams will be presented. We have reimplemented the *Introduce Initial Pseudostate* and *Flatten States Outgoing Transitions* refactorings in order to assess the possibility of using the MOFLON tool for the specification of model refactorings.

7.1 The MOFLON tool

Recently, the *MOFLON meta modeling framework* has been released [32]. It combines the MOF 2.0 meta modeling language [33], Triple Graph Grammars [34] and JMI code generation [35]. The MOFLON tool allows to perform a wide range of tasks in the fields of model analysis, model transformation, and model integration for standard modeling languages like UML or domain–specific modeling languages. It is a diagram editor and code generator plug–in for the Fujaba tool suite. [36]

Fujaba [37] is a graph based tool which uses the Unified Modeling Language UML for design and realization of software projects. Fujaba uses UML class diagrams for the specification of graph schema. It uses a combination of Activity diagrams and Collaboration diagrams, so–called *Story Diagrams* for the specification of operational behaviour. The semantics of Story Diagrams is based on programmed graph rewriting rules [38]. In contrast to other graph based tools, Fujaba does not rely on proprietary runtime environments. Instead, Fujaba generates standard Java source code that can be easily integrated with other Java program parts and that runs in a common Java runtime environment. [39]

The Meta–Object Facility (MOF) is an extensible modeling language for defining, manipulating, and integrating metadata and data in a platform independent manner. The MOFLON tool implements several concepts of MOF 2.0 [33]:

- **Packages:** Packages, Package import, Package merge and Element import.
- **Types:** Classes, Datatypes, Enumerations and Generalization.
- **Associations:** Associations, Unidirectional and Mutual references, Association refinement and Association ends.
- **Annotations:** Constraints, Comments and Tags

In this chapter we are going to illustrate the functioning of the MOFLON tool comparing it with the AGG tool. The descriptions are illustrated and motivated using some examples.

7.2 Metamodel

The MOFLON tool is capable of creating MOF 2.0 compliant metamodels. In MOF 2.0 [33], a metamodel can be divided into several packages. A MOFLON project must always contain an outermost package that will contain all the elements of the project.

Figure 7.1 shows the packages we have defined for the specification of UML State Machine diagrams. The outermost package named *UML* contains three packages. The “Primitive” package contains the primitive data types. The “StateMachineDiagram” package contains the representation of UML State Machine diagrams. The “Transformation” package contains the model refactoring implementations.

Primitive data types are used to describe the properties of the instances of each class defined in the metamodel. It is necessary to specify the primitive data types in each metamodel, this

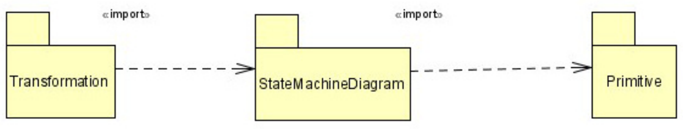


Figure 7.1: Metamodel Packages

allows to choose arbitrary names for the primitive types. The primitive data types can be mapped to predefined Java classes that represent primitives type at compile time. Figure 7.2 shows the primitive data types defined for the specification of UML State Machine diagrams.

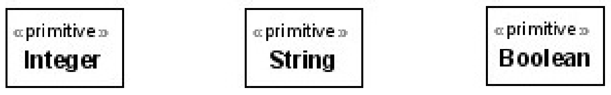


Figure 7.2: Primitive types

The MOFLON tool allows the mapping of primitive types defined in the metamodel to primitive types from the UML infrastructure library [5], which is reused in MOF. These primitive types are mapped to JAVA classes and primitives at compile time, so the primitive types in the metamodel adopt the semantics of the corresponding JAVA elements. The MOFLON tool allows the use of *String*, *Integer*, *Boolean*, *UnlimitedNatural*, *Float*, and *Double*. Moreover, the MOFLON tool allows the definition of Enumerations and custom data types that can be used in property declarations. That way, the MOFLON tool is more expressive and does not impose any limitations on the set of possible data types.

In the MOFLON tool the nodes and edges of graphs are respectively represented by classes and associations. The MOFLON tool permits to add classes and associations inside a package. It is possible to set the visibility of classes in order to specify whether a class have to be visible only to other classes within its package or elsewhere. Classes allow to specify properties and operations inside them. The possibility of specifying operations inside a class adds expressiveness to the MOFLON tool. In section 7.4 we will show that use of classes as data types is allowed, too. The MOFLON tool also permits to set a default value for the properties of a class.

An association represents a relationship between classes and allows to specify the roles played by the classes. In the MOFLON tool it is possible to specify plain associations and compositions. Associations are more expressive than edges because it is possible to specify their *Visibility* and *Navigability*. The *Navigability* simply means whether –given an instance of a class on one side of an association– you can access an instance on the other side. In contrast to edges, associations do not permit to define any custom properties inside them.

UML 2.0 [4, 5] is formally defined using the OMG Meta–Object Facility (MOF). As the MOFLON tool implements several concepts of MOF 2.0, it allows to easily represent the UML Models. Figure 7.3 shows the metamodel that corresponds to UML State Machine diagrams. The implemented metamodel is very similar to the UML 2.0 specification [4, 5] (p. 509).

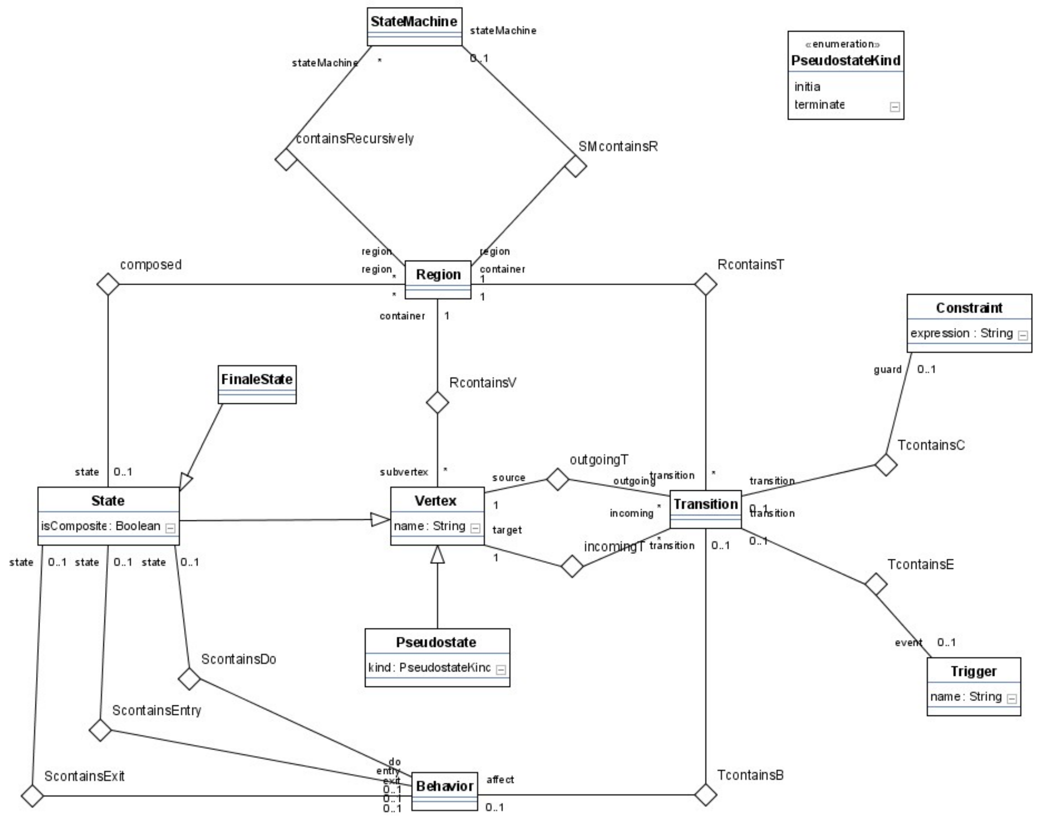
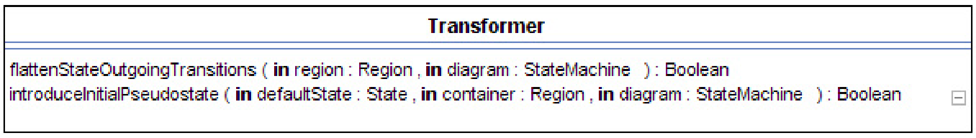


Figure 7.3: State Machine diagram

The MOFLON tool does not define the concept of *aggregation* and this kind of relationship must be represented using a plain association. The Enumeration type is useful to represent the

PseudostateKind element and the property *kind* of the *Pseudostate* element. The AGG tool does not permit to define enumerations and a different representation had been chosen for those elements (see figure 4.4).

The “Transformation” package shown in figure 7.4 contains the *Transformer* class. That class contains the model refactoring implementation. Each model refactoring is specified as an operation of the class. The parameters required for the refactoring are defined as input parameters of the operations.



```
Transformer
flattenStateOutgoingTransitions ( in region : Region , in diagram : StateMachine ) : Boolean
introduceInitialPseudostate ( in defaultState : State , in container : Region , in diagram : StateMachine ) : Boolean
```

Figure 7.4: Transformation Package

In the MOFLON tool, operations of a class can be specified as Story Diagrams [40] in a graphical editor rather than being written as piece of source code. Story Diagrams are composed of activities and transitions, which define the order in which the activities are executed. Story Diagrams support two kinds of activities: statement activities and *Story Patterns*. In sections 7.3 and 7.4 we will explain the Story Diagram and the graph transformation notation used by the MOFOLN tool.

7.3 Primitive transformations

In MOFLON primitive graph, transformations are represented with *Story Patterns*. A Story Pattern is a graph rewrite rule showing left and right–hand side within one picture. The combination of left and right–hand side into a single picture results in a more concise and readable notation.

Figure 7.5 shows an example of Story Pattern. The depicted structure defines the left–hand side of the rule, it is used to search a match of the represented pattern in the graph. The cross-out of classes and associations means that those elements must not be present in the graph. The cross-out has the same function of Negative Application Conditions in AGG.

The Story Pattern is composed of variables. Unbound variables are shown as boxes containing name and type, e.g. $initial : Pseudostate$. Bound variables are shown as boxes containing

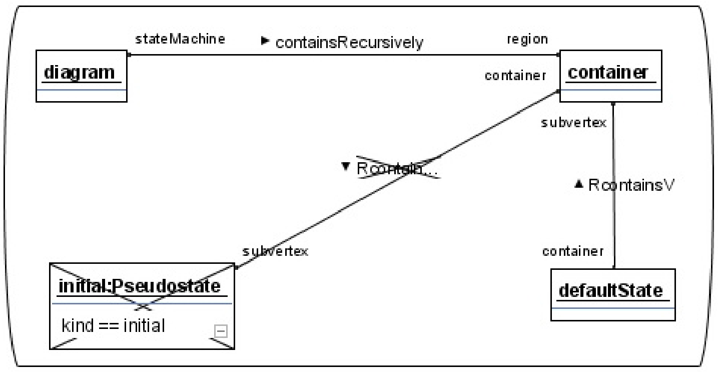


Figure 7.5: Story Pattern example

only their name, e.g. $diagram$. Subsequent Story Patterns may use variables bound in previous Story Patterns (or statements) of the same Story Diagram.

Figure 7.6 shows an example of Story Pattern where the left and right–hand sides are combined together. Creation of elements is shown in green colour and by attaching a «create» marker. Deletion of elements is shown in red colour and by attaching a «destroy» marker. Thus, the left–hand side of the depicted rule consists of the normal elements together with the (red) cancelled elements. The right–hand side consists of the normal and (green) created elements.

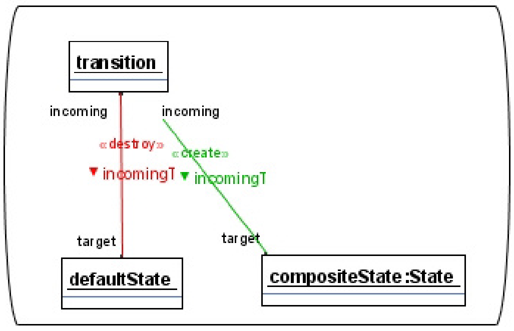


Figure 7.6: Story Pattern example

The rule shown in figure 7.6 destroys the old “incomingT” association between “transition” and “defaultState” and creates a new “incomingT” association between “transition” and “compositeState”.

It is possible to specify the creation of a class using the same mechanism. Moreover, values for properties of the class can be specified.

7.4 Composite transformations

The behaviour of class operations can be specified using Story Diagrams [40]. Story diagrams are a combination of UML Activity and Collaboration diagrams. MOFLON uses UML Collaboration diagrams as a notation for graph rewrite rules that have been explained in section 7.3. Story patterns are embedded into an UML activity diagram, specifying the control flow. [41]

Story Diagrams may have formal parameters for passing attribute values and object references. Story Diagrams adapt UML Activity diagrams to represent control flow, graphically. Thus, the basic structure of a Story Diagram consists of a number of so-called activities shown by big rectangles with rounded left and right sides. Story diagrams support two kinds of activities: statement activities and Story Patterns. A statement activity consists of a chunk of UML pseudo code that can be used for I/O-operations, mathematical computations, and operation invocations. Activities are connected by transitions, that specify the execution sequence. Execution starts at the unique start activity represented by a filled circle. Execution proceeds following the outgoing transition(s). Multiple outgoing transitions are guarded by mutual exclusive boolean expressions shown in square brackets. Diamond shaped activities express branching. When the stop activity is reached, operation execution terminates.

All data types and classes defined in the metamodel can be used for specifying types of formal parameters. That feature is very useful in order to implement model refactorings. It easily permits to provide to the operation the nodes of the graph that have to be used for applying transformations. AGG permits the only use of Java types to specify input parameters, and it is necessary to manually match nodes of the left–hand side with nodes of the graph before the application of the graph transformation rule.

In order to give an overview of the capabilities of the MOFLON tool, we have reimplemented the *Introduce Initial Pseudostate* and *Flatten States Outgoing Transitions* refactorings. However, we deliberately made some simplifications because the implementation of a complete application for model refactoring using the MOFLON tool was out of the scope of this dissertation.

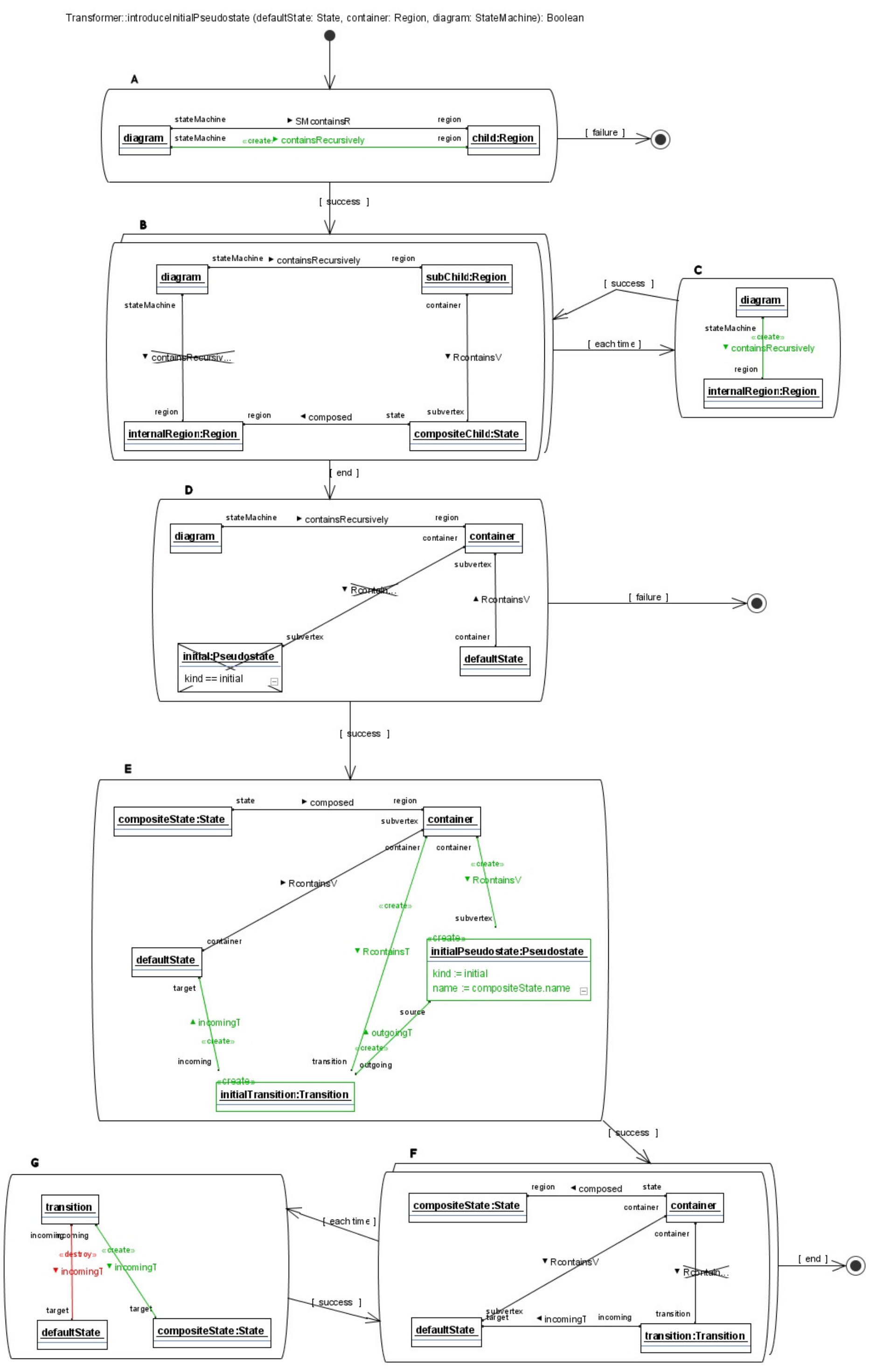


Figure 7.7: Introduce Initial Pseudostate refactoring

7.4.1 Introduce Initial Pseudostate

Figure 7.7 shows the Story Diagram that implements the *Introduce Initial Pseudostate* refactoring introduced in section 5.8. The refactoring is represented as a sequence of primitive transformation steps. In order to apply the refactoring, it is necessary to provide three input parameters. The parameter $diagram$ represents the State Machine diagram that is going to be modified. The parameter $container$ represents the region subject of the refactoring. The parameter $defaultState$ specifies the default state of the region $container$.

The activities A, B, and C in figure 7.7 are necessary in order to verify that the region specified as input parameter belongs to the diagram. Each instance of the State Machine class corresponds to a diagram. The concept of "diagram" was not present in the graph representation that we have implemented using the AGG tool. Those steps are necessary to ensure that the region is part of the selected diagram.

The activity A adds an association "containsRecursively" between the diagram and its outermost region. The activity B contains a graph transformation rule that matches all the sub–region for which the association "containsRecursively" with the diagram have not been defined yet. The cross-out of the "containsRecursively" association between the diagram and the "internalRegion" specifies that the association must not be present. The activity B is defined as a *"For Each"* activity, it means that the activity is repeated multiple times. Each time when a match is found the activity C is executed. The activity C adds a "containsRecursively" association between the diagram and the "internalRegion" found by the activity B.

The activity D verifies the necessary preconditions of this refactoring. The region must not contain an initial pseudostate and the default state provided as input parameter must belong to the region. The cross-out of the "initial" node means that the region must not contain a pseudostate whose type is "initial". If the preconditions are not respected the refactoring is aborted.

The activity E adds the initial pseudostate to the region and defines the automatic transition between the initial pseudostate and the default state of the region. The activity F is a *"For Each"* activity. It contains a graph transformation rule that matches all transitions whose target is the default state. The cross-out of the "RcontainsT" association between the "region" and the "transition" means that the transition must be external at the region provided as input parameter. Transitions between internal states of the region are not modified. Each time when a transition is found the activity G is executed. The target of the transition is modified and becomes the composite state. The refactoring ends when the activity F has matched all transitions.

The implementation of the refactoring using the MOFLON tool does not require to add any additional elements to the metamodel. It is not necessary to use "Refactoring" elements like in the AGG tool to recognize the nodes subjects of the refactoring. Variables are used in order to indentify elements. Subsequent activities may use variables defined in previous activities. The *"For Each"* activity allows to easily repeat activities, in this case to create a node multiple times.

7.4.2 Flatten States Outgoing Transitions

Figure 7.8 shows the Story Diagram that implements the *Flatten States Outgoing Transitions* refactoring introduced in chaper 5.11. The refactoring is represented as a sequence of primitive transformation steps. In order to apply the refactoring, it is necessary to provide two input parameters. The parameter $diagram$ represents the State Machine diagram that is going to be modified. The parameter $region$ represents the region subject of the refactoring.

The activities A, B, and C in figure 7.8 are necessary in order to verify that the region specified as input parameter belongs to the diagram. Their behaviour has been explained in section 7.4.1.

The activity D is a *"For Each"* activity. It contains a graph transformation rule that matches all transitions whose source is the composite state. The "event" element associated to transition must be present as defined in the formalisation of the refactoring. The MOFLON tool allows to specify optional element in the graph transformation rule. The "guard" element could not be present and it defined as optional. The refactoring ends when the activity D has matched all transitions. Each time when a transition is found, the activity E is executed. It is a *"For Each"* activity and matches all sub–states of the region. For each sub–states matched by the graph transformation rule the activity F is executed. The activity F copies the original transition setting the matched sub–state as source. As seen in the previous section the operation is repeated automatically multiple times; it is not necessary to mark the transition with a "Refactoring" elements in order to recognize it.

When the activity E has matched all the sub-states of the region the activity G is executed. It deletes the original transition and then returns to the activity D. The red elements in the graph transformation rule identify the elements that are going to be deleted.

It is important to notice that the graph transformation rule of the activity F is incorrect. The "flattenGuard" node is always created even when it is not present in the original transition. This problem can easily be avoided using branch conditions. That mechanism must be applied for

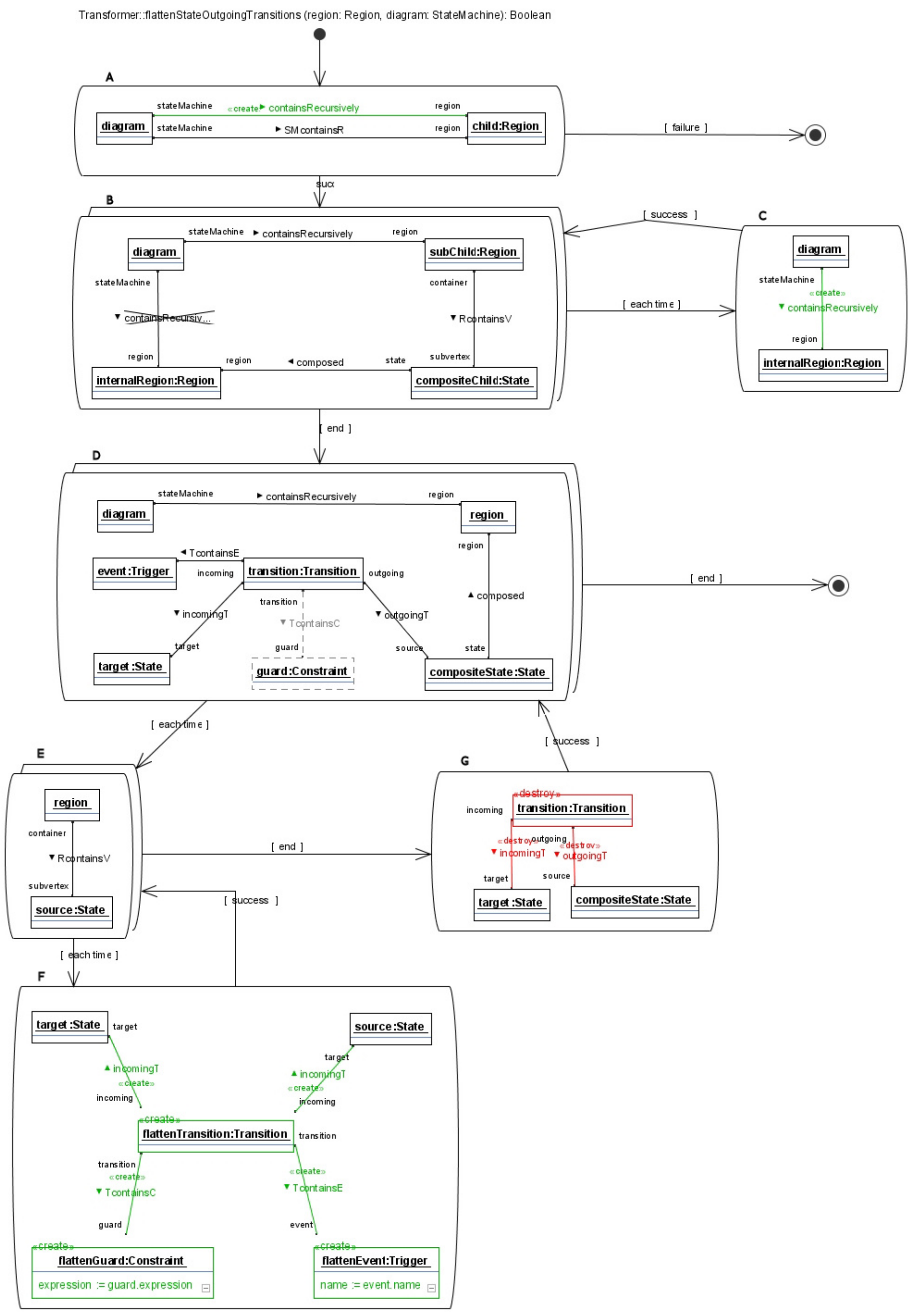


Figure 7.8: Flatten States Outgoing Transitions refactoring

each optional element. We have intently presented the incorrect version in order to show that the MOFLON tool does not allow to specify *Conditional creation* of elements. It is not possible to specify by means only of one Story Pattern that the "flattenGuard" node must be created only if the "guard" node is defined for the original transition.

The possibility to define *Conditional creation* of elements in the graph transformation rules could be an interesting enhancement and would make graph transformations more powerful.

With the MOFLON tool, it is possible to generate JAVA code for the implemented specifications. The generated code complies to the JAVA Metadata Interface (JMI), a standard for metadata management published by Sun MycroSystem [35]. JMI defines both a tailored and a reflective interface for the metamodels.

The generated JAVA code for the *Flatten States Outgoing Transitions* refactoring is incorrect. In figure 7.8 is possible to see that the expression attribute of the "flattenGuard" node assumes the value specified in the "guard" node. This assignment is not valid because the "guard" node may be undefined.

The incorrect version of the *Flatten States Outgoing Transitions* refactoring has been presented because it gives the possibility to avoid the lack of the *Conditional creation* concept. Modification of the generated JAVA code makes possible to specify that the "flattenGuard" node must be created only if the "guard" node is associated to the original transition. This approach can be used for all optional elements.

A comparison of the resulting implementation with the solutions proposed in chapter 5.11.7 let notice some important improvements. Let N_{opt} be the number of optional nodes associated to the transition, the total number of necessary graph transformation rules will be $N_{rules} = K$ where K is a constant value. If other optional nodes are added, the number of necessary graph transformation rules will not change.

Let N_s be the number of sub–states of the region and N_t the number of transitions that have to be modified, the computation cost of the transformation will be $C = N_t \times (2 + 2N_s)$. In the solution implemented with the MOFLON tool both the computation cost and the number of necessary graph transformation rules do not depend on the number of optional nodes N_{opt}.

7.5 Conclusion

As the MOFLON tool implements several concepts of MOF 2.0, it allows to easily represent the UML Models. Classes and associations are more expressive than nodes and edges and offer a better support for representing the elements of the UML Metamodel. In contrast to edges, associations do not permit the definition of custom properties inside them. In the Type Graph implemented with AGG we have used an edge property to specify the order of parameters contained by an operation. In MOFLON this concept can be represented using *Ordered Associations*. In other situations a different representation must be chosen.

The MOFLON tool allows to define Enumerations and custom data types that can be used in property declarations. The Classes can also be used as types for the formal parameters of the operations. That way, the MOFLON tool is more expressive and does not impose limitations on the set of possible data types.

Story Patterns offer a more concise and readable notation. Moreover, the MOFLON tool allows to specify optional element inside Story Patterns. The AGG tool prefers to keep the transformation model rather simple by supporting the standard transformation concepts with negative application conditions. Story Diagrams are useful in order to describe the control flow of model refactorings. They allow to easily combine and organise the primitive transformation specifying their execution order. The control structures of AGG were not sufficient for describing model refactorings and we have implemented a custom control flow mechanism.

The *MOFLON meta modeling framework* appears to be a suitable tool for the specification of the refactoring of UML models. Its expressive power offers a better support and avoids many issues encountered in the AGG tool during the implementation of the model refactoring.

Furthermore, the MOFLON tool supports the import of XMI files exported from Rational Rose enabling users to import legacy metamodels into MOFLON. XML Metadata Interchange (XMI) [42] is an OMG's standard for defining, interchanging, manipulating, and integrating XML data and objects.

Chapter 8

UML Model Consistency

8.1 Model Consistency

Different aspects of a software system are covered by different UML diagrams. There is an inherent need to preserve consistency between these UML diagrams, in particular during the application of model refactorings.

Currently, consistency of UML models is only partially ensured by the language specification. While it is an accepted fact that consistency maintenance is an issue in UML–based system development, appropriate definitions of consistency are still an open research topic. B. Hnatkowska, Z. Huzar and J. Magott have considered the problem of consistency among components of an UML system model in [43]. They proposed OCL (Object Constraint Language) to formalize the consistency conditions that must hold among model components.

W. Liu, S. Easterbrook and J. Mylopoulos in [44] proposed a rule–based approach for the detection of inconsistencies in UML Models. They have defined a language for production system and rules specific to software designs modeled in UML. Using that approach, they are able to: detect inconsistencies, notify the users, recommend resolutions, and automatically fix the inconsistency during the design process. [44]

J. Küster and G. Engels in [45, 46, 47] have defined an approach to model consistency management that has led to the development of a general methodology for dealing with consistency in UML-based development processes. Their methodology requires the identification of consistency problems and then the development of partial mappings of UML models in a formal semantic domain. In this formal semantic domain, formal consistency conditions can be defined.

The rule–based approach to consistency management relying on the theory of graph transformation can be characterized by the following steps [45, 46, 47]:

- Identification of a consistency problem and informal description of required consistency.
- Formalization of the consistency concept in form of a mapping in a formal semantic domain and the definition of formal consistency conditions.
- Development of an operational consistency concept that can be applied in practical development.
- Definition of consistency checks for the consistency concept.

In section 8.2 we will explain some of the consistency checks implemented in AGG using a similar approach.

In chapter 5 consistency problems arising between static and dynamic diagrams have been analysed. More precisely, we have discussed problems arising between UML Class diagrams and their associated UML State Machine diagrams. For each model refactoring, we have analysed the effects on the consistency of the overall model and provided localized consistency checks for those parts of the model that are going to be changed. The formalisation of each refactoring contains strict preconditions that have been defined in order to preserve the consistency amongst the different kinds of UML diagrams.

The approach to model consistency chosen for the purpose of this dissertation is discussed in next section. Some of the consistency constraints and consistency checks implemented are presented, in order to demonstrate the correctness of this approach.

In section 8.3 one more solution will be analysed in order to apply the refactorings and preserve the consistency between different kinds of UML diagrams.

8.2 Consistency constraints

The Type Graph shown in figure 4.4 represents a subset of the concepts defined by the UML metamodel [4, 5]. The inconsistencies that could affect UML diagrams and the inconsistencies that could arise in the chosen representation between UML Class diagrams and UML State Machine diagrams have been identified and classified. However, a complete representation of UML Models may be affected by further consistency problems. Table 8.1 shows the possible inconsistencies identified.

Name	Description
Abstract State Machine	The State Machine diagram is associated to an abstract class.
Incorrect Operation Reference	The operation associated to a transition does not belong to the class represented by the State Machine diagram.
Incorrect Composite State	A composite state is composed of a region that contains the state itself.
Dangling Type Reference	Type has not been defined for operations, parameters or properties.
Generalization Error	A class is a generalization of an interface or an interface is a generalization of a class.
Incorrect Interface Implementation	A class does not implement all operations of an interface.
Unused Interface	An interface defined in the UML Class diagram is not implemented by any class.
Abstract Operation	An abstract operation is defined inside a concrete class.

Table 8.1: List of consistency problems

The *Abstract State Machine* and *Incorrect Operation Reference* consistency problems represent inconsistencies between UML Class diagrams and UML State Machine diagrams. The *Incorrect Composite State* inconsistency affects UML State Machine diagrams. The other inconsistencies affect UML Class diagrams. In this section we will present and discuss in details the *Abstract State Machine* and *Incorrect Operation Reference* consistency problems.

For each possible inconsistency, the corresponding consistency check has been implemented using a graph transformation rule. A consistency check performs the necessary verifications to ensure the model to be consistent. Figure 8.1 shows the *Abstract State Machine* consistency check. If a class is an abstract class and has an associated State Machine diagram, there is an inconsistency in the UML model. If the inconsistency is detected, the graph transformation rule adds a “Conflict” node in the graph in order to point out this consistency problem.

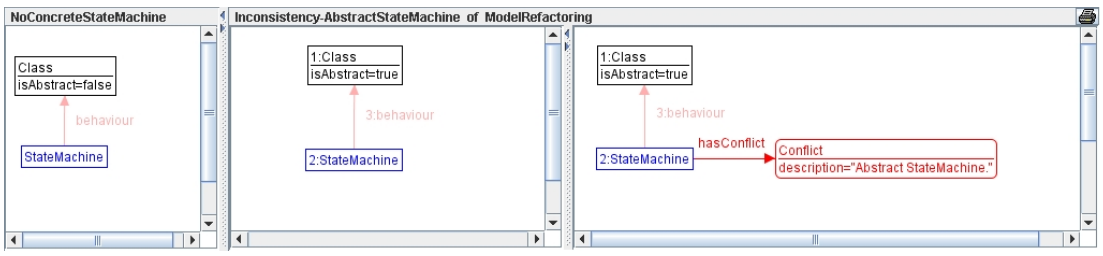


Figure 8.1: Abstract State Machine

Figure 8.2 shows the *Incorrect Operation Reference* consistency check. The State Machine diagram represents the behaviour of a class and contains transitions that refer to operations. If an operation referred by transitions is not contained in the class associated to the State Machine diagram, there is an inconsistency in the UML model. If the inconsistency is detected, the graph transformation rule adds a “Conflict” node in the graph in order to point out this consistency problem. The relationship “containsT” among the “StateMachine” node and the “Transition” nodes is used to easily identify all transitions contained in a State Machine diagram.

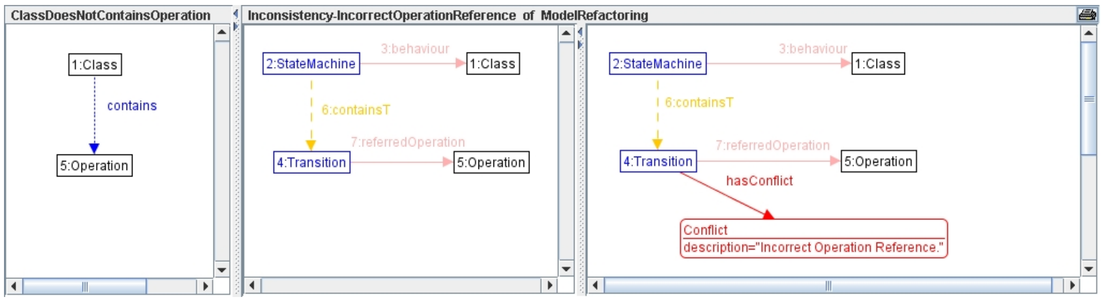

ClassDoesNotContainsOperation
1:Class
contains
5:Operation
Inconsistency-IncorrectOperationReference of ModelRefactoring
2:StateMachine
3:behaviour
1:Class
6:containsT
7:referredOperation
4:Transition
5:Operation
hasConflict
Conflict
description="Incorrect Operation Reference."

Figure 8.2: Incorrect Operation Reference

The prototype application illustrated in chapter 6 allows the user to check the consistency of the UML model. That function applies a set of graph transformation rules that implement the consistency checks and verifies that the consistency of the UML model has been preserved after the application of the model refactoring.

However, the implementation of a complete system of detection and resolution for model inconsistencies would go beyond the scope of this dissertation. The approach to model consistency chosen for the purpose of this dissertation aims to preserve the consistency. A transformation that could lead to an inconsistent model should not be applied. To achieve this result, the consistency checks have been transformed in consistency constraints.

The formalisation of each refactoring contains consistency constraints, or preconditions, that have to be respected in order to preserve the consistency among the different kinds of UML diagrams. If the model does not respect the consistency constraint, the model refactoring can not be applied.

For example, a model refactoring that transforms a class to an abstract class could generate an *Abstract State Machine* inconsistency. It should be applied only if the class subject of the model refactoring does not have an associated State Machine diagram.

Figure 8.3 shows the NAC *State Machine Does Not Exist* defined for the *Generate Subclass* refactoring. That precondition verifies that a State Machine diagram is not defined for the class.

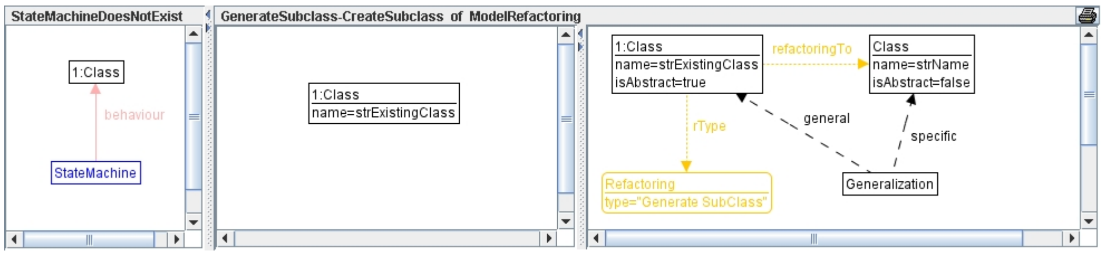


Figure 8.3: Create Subclass transformation

A model refactoring that moves an operation from a class to another one could generate an *Incorrect Operation Reference* inconsistency. It should be applied only if the operation is not referenced by any transitions in the State Machine diagram.

Figure 8.4 shows the NAC *Transition Does Not Refer Operation* defined for the *Extract Class* refactoring. That precondition verifies that in the State Machine diagram the operation is not referenced by any transition.

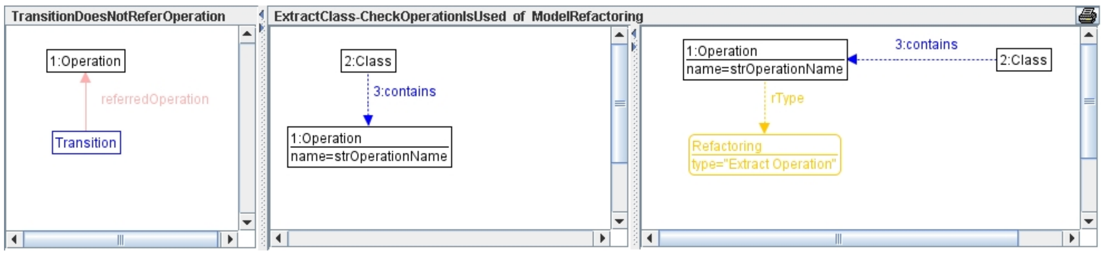


Figure 8.4: Check Operation Is Used transformation

This approach avoids the application of model refactorings that could lead to an inconsistent model. The user is requested to manually change the UML Models before the application of the refactoring. In the next section other possible approaches to this problem are presented.

8.3 Model Synchronization

I. Ivkovic and K. Kontogiannis in [48] proposed a methodology to keep synchronized models at different levels of abstraction. Their conceptual view of software models is as graphs, and model transformations are viewed in terms of basic graph transformations such as node insertions and deletions. Based on that view, a set of transformations applied to one model is traced and propagated to the other by choosing from a set of possible transformations. [48]

A similar approach could be used to maintain consistency between different kinds of UML diagrams. Once the dependencies among different kinds of UML diagrams have been identified, it is possible to trace and propagate a set of transformations applied to one UML diagram to another diagram, by choosing from a set of possible transformations.

The dependencies among different kinds of UML diagrams correspond to the same concepts previously used to define consistency constraints. If a model refactoring applied to an UML diagram could generate a model inconsistency, then the application of a transformation on the other UML diagrams is necessary in order to preserve the synchronization between them.

A suitable solution that applies to all possible cases can rarely be found. A satisfactory approach would consist of identifying a set of possible solutions for each consistency problem and requesting the user to specify which one best applies to the situation.

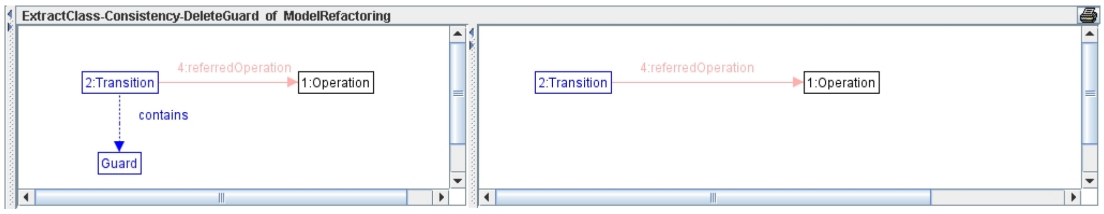
An explanation of this approach is illustrated referring to the example of model inconsistency shown in chapter 3.3.2. The *Extract Class* refactoring extracts a class from an existing one exporting a set of operations and properties. After the application of the refactoring the operations are not contained by the class anymore and the State Machine diagrams could be incorrect. It is possible to find three solutions to preserve the synchronization between the different kinds of UML diagrams involved in the model refactoring. In next sections each solution will be presented and discussed.

8.3.1 Delete transition

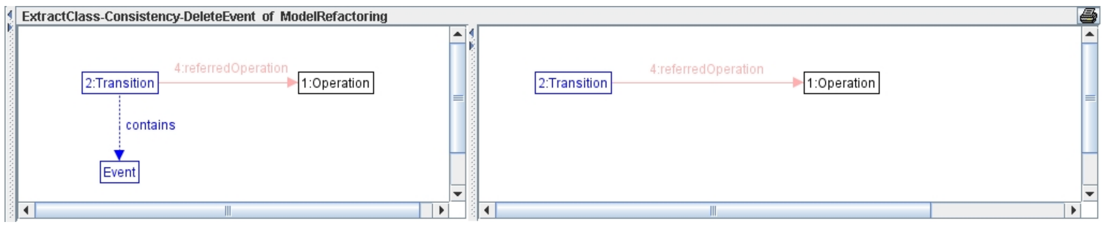
It is possible to delete transitions that refer to the operation involved from the State Machine diagram. This task can be accomplished through the graph transformation rules shown in figure 8.5.

The step (a) deletes the guard associated to the transition and the step (b) deletes the event associated to the transition. It is necessary to have separate graph transformation rules because the elements could be missing in the graph. The step (c) deletes the transition associated to the operation. These steps must be repeated multiple times in order to delete all transitions that refer the operation.

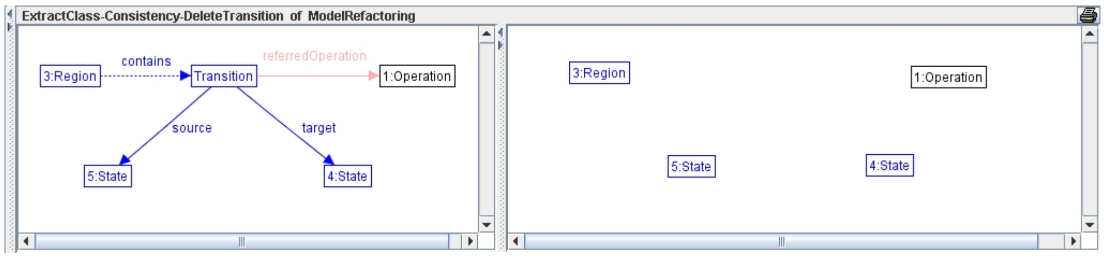
Even if this approach preserves the consistency between different kinds of UML diagrams, it will modify the behaviour of the system. Moreover, without verification of some necessary preconditions this transformation could generate problems in the State Machine diagram, leading to an inconsistent model.



(a) Delete Guard



(b) Delete Event



(c) Delete Transition

Figure 8.5: Delete transition from State Machine diagram

Input Parameters $o : Operation \Rightarrow 1$

If the transition that is going to be deleted is the only outgoing transition of a state, after the refactoring the state can never be left. If the transition that is going to be deleted is the only incoming transition of a state, after the refactoring the state can never be reached.

This solution appears to be rarely applicable. Moreover, it should be used only if the user explicitly intends to change the behaviour of the system.

8.3.2 Move the association to operations

In order to avoid the problems of the *Delete transition* solution, only removal of the action associated to the transitions is possible. That way the transformation does not generate unreachable or dead states in the State Machine diagram. In order to completely preserve the behaviour of the system, creation of a State Machine diagram associated to the new class is necessary. The execution of the exported operations is specified in the new State Machine diagram.

Referring to the example in chapter 3.3.2, it is possible to analyse the necessary transformations. The operations *increaseCounter()* and *decreaseCounter()* are associated to the transitions triggered respectively by the “Button Next” and “Button Previous” events. These operations are exported to the *Counter* class during the application of the refactoring and their associations with the transitions in the State Machine diagram have to be deleted. When the modified transitions are triggered by the events “Button Next” and “Button Previous” they continue leading the State Machine to the *Ready* state.

In order to preserve the behaviour of the system it is necessary to specify that the *increaseCounter()* and *decreaseCounter()* operations of the *Counter* class must be executed respectively when the “Button Next” and “Button Previous” events occur. This task can be accomplished by creating a State Machine diagram for the *Counter* class where the transitions associated to the *increaseCounter()* and *decreaseCounter()* operations will be specified. Figure 8.6 shows a possible State Machine diagram for the extracted class *Counter*.

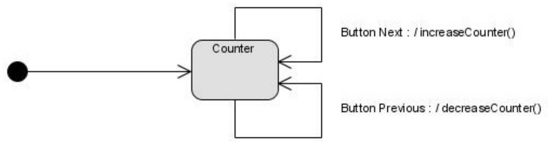

Counter
Button Next : / increaseCounter()
Button Previous : / decreaseCounter()

Figure 8.6: Counter - State Machine diagram

These steps could be implemented using graph transformation rules. The transformation shown in figure 8.7 creates a new State Machine diagram associated to the extracted class. The variable *strName* is used to assign a name to the state contained in the State Machine diagram. The transformation shown in figure 8.8 creates a self transition in the State Machine diagram

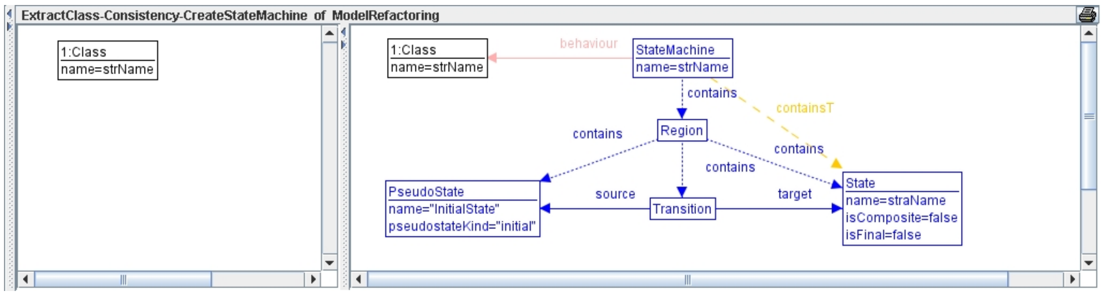


Figure 8.7: Create State Machine

Input Parameters $newClass : Class \Rightarrow 1$

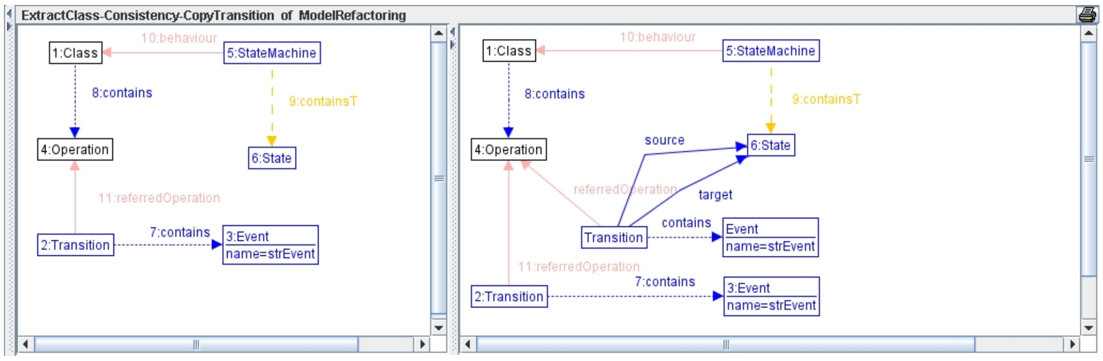


Figure 8.8: Copy Transition

Input Parameters $exportedOperation : Operation \Rightarrow 4$

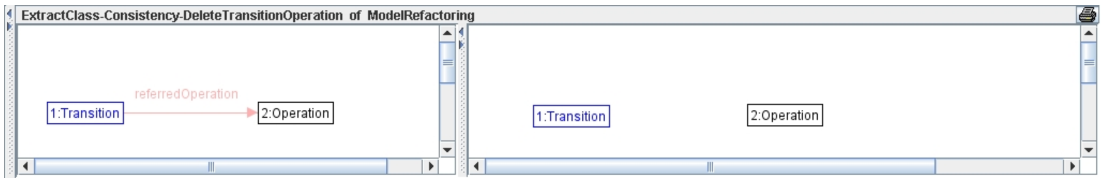

ExtractClass-Consistency-DeleteTransitionOperation of ModelRefactoring
referredOperation
1:Transition
2:Operation

Figure 8.9: Delete Transition Operation

Input Parameters $exportedOperation : Operation \Rightarrow 2$

copying the characteristics of the original transitions. The transformation shown in figure 8.9 removes the association between the operation and the original transition. The last two steps must be executed for each operation that has to be exported to the new class.

A simplification in the implementation of the graph transformation rules has been made.

The transformations consider that transitions always have an associated event and they do not consider the guard optionally associated. The guard could easily be copied to the transition in the same way analysed during the formalisation of model refactorings. If the original transition does not have an associated event, the transformation will add a corresponding self transition in the State Machine diagram associated to the exported class. A transition without an event is an automatic transition and will cause the system to continuously call the operation referred by the transition. If the transition has an associated guard, a different consideration can be made. However, in order to avoid the system to continuously call the operation associated to the transition it is necessary that the evaluation of the guard is changed after the execution of the operation. The graph transformation rules can not determine when the execution of the operation changes the evaluation of the guard and it is not possible to decide whether the transition can be safely copied or not.

If at least one of the transitions associated to the exported operations does not have an associated event, this solution can not be applied.

Referring to the example in chapter 3.3.2 a further consideration can be made. If the *next()* operation does not internally call the *increaseCounter()* operation, those transformations will change the behaviour of the system. After the application of those transformations the *increaseCounter()* operation is executed every time when the "Button Next" event occurs. If the exported operations are not executed, nor called by other operations, every time when the event occurs, this solution can not be applied.

If the State Machine diagram contains two transitions that refer two different exported operations but are triggered by the same event, an other problem arises. In the State Machine diagram associated to the exported class, it is not possible to add two self transitions that are triggered by the same event.

Referring to the example in chapter 3.3.2, it is possible to apply an *Extract Class* refactoring in order to export the *opendrive()* and *closedrive()* operations to a new class named *Device*. Both operations are executed when the "Button Drive" event occur.

Figure 8.10(b) shows a possible State Machine diagram for the extracted class *Device*. Using the transformations previously described it is possible to export the *opendrive()* operation and obtain the State Machine diagram shown in figure 8.10(a). The transformation in figure 8.11 allows to export the *closedrive()* operation obtaining the State Machine diagram shown in figure 8.10(b). That transformation converts the existing state to a composite state and creates

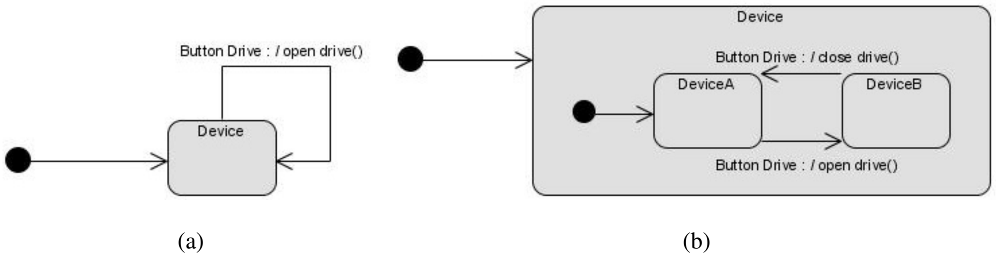

Button Drive : / open drive()
Device
(a)
Device
Button Drive : / close drive()
DeviceA
DeviceB
Button Drive : / open drive()
(b)

Figure 8.10: Split State example

two sub–states inside it. The transitions cause the state of the system to change between one sub–state to the other when the associated events occur.

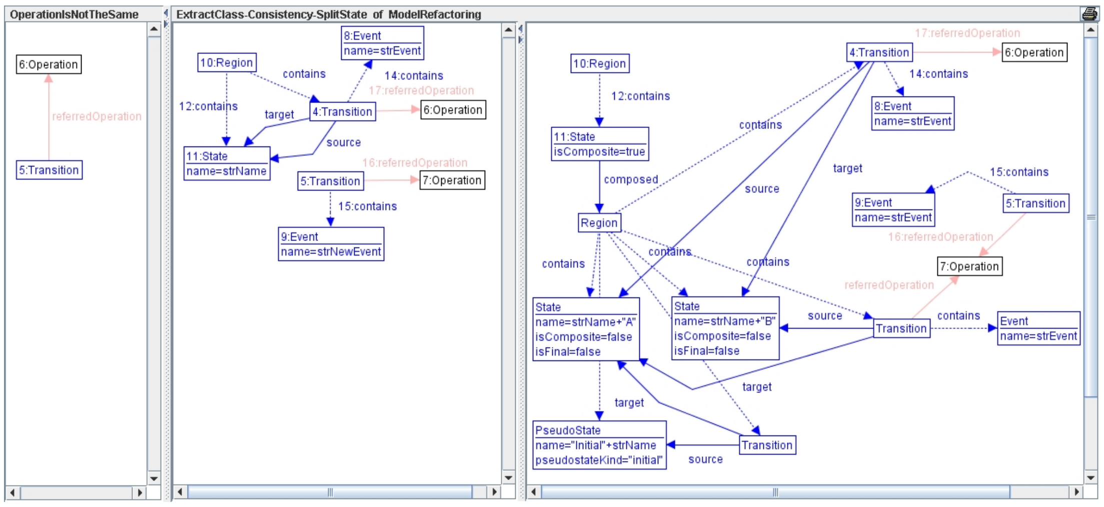


Figure 8.11: Split State

Input Parameters $exportedOperation : Operation \Rightarrow 7$

However, this solution still presents some limitations. If the operations are exported in the opposite order, the State Machine diagram is not correct. Specification is necessary of the exact order in which operations have to be exported. Moreover, if a third operation has to be exported and it is executed when the same event occurs, it is not possible to apply this transformation.

This solution appears to be applicable only to simple models. All the problems identified must be avoided by creating all the necessary preconditions.

8.3.3 Wrap operation

This solution is specific for refactorings that move an operation from a class to another one (i.e., Extract Class, Move Operation).

In order to preserve the consistency between the Class Diagram and the State Machine diagram, it is possible to specify –for each operation that has to be moved– an operation that will wrap the call to the original operation. It is possible to choose the wrap operation among existing operations of the class or add a new operation.

Figure 8.12 shows the transformation rule that accomplishes this task. It simply changes the referred operation associated to the transition. The step must be repeated multiple times in order to modify all transitions that refer to the operation.

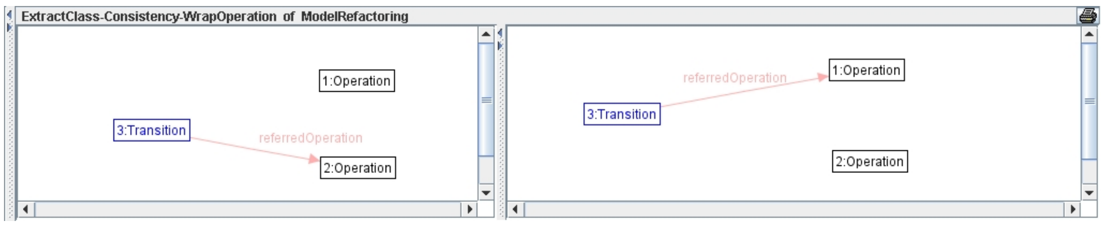


Figure 8.12: Wrap Operation

Input Parameters $wrapperO : Operation \Rightarrow 1, originalO : Operation \Rightarrow 2$

Referring to the example in chapter 3.3.2, it is possible to apply the *Extract Class* and to preserve the consistency between the Class Diagram and the State Machine diagram. The State Machine diagram shown in figure 3.5(a) can be obtained by specifying that the next() operation wraps the call to the increaseCounter() operation and the previous() operation wraps the call to the decreaseCounter() operation.

This solution preserves the behaviour of the system and appears to be always applicable. If the UML Model contains a Sequence diagram, the calls to the wrapped operations must be added to the diagram.

Chapter 9

Conclusions

In this dissertation we have shown how the formalism of graph transformation can be used as an underlying foundation for the specification of model refactoring. We have presented an initial catalogue of model refactorings using a simplified version of the UML metamodel [4, 5]. Adapting refactorings to the model level has been sometimes more complex than we initially thought, especially when we wanted transformations to have an impact on different UML diagrams.

The UML metamodel specification [4, 5] presents some poorly–defined concepts, and decisions have been taken to ensure a correct definition of model refactorings. However, in some cases those decisions could generate incompatibilities with tools that use a different approach.

We have discussed eight primitive model refactorings and we have shown that it is possible to formalise their execution using graph transformation rules with a superimposed control flow mechanism that we developed especially for this purpose. Graph grammars appear to be a suitable technique for model transformation, as explored by many other works [21, 22, 49, 50]. However, the formalisation of model refactorings has pointed out some limitations of the graph transformation notation and current–day graph transformation tools.

We have taken into account two graph transformation tools –the AGG tool [24] and the MOFLON tool [32]– in order to compare their functionalities. The MOFLON tool allows a better representation of UML models, due to the similarities between the UML metamodel and the MOFLON concepts. The MOFLON tool supports a number of advanced transformation concepts and additional forms for structuring rule sets. The AGG tool prefers to keep the transformation model rather simple by supporting the standard transformation concepts with negative application conditions in addition. As advantage, AGG graph transformations can be verified based on the theory of algebraic graph transformation. The MOFLON tool offers a better support for the implementation of model refactorings. The application of graph transformation rules can be easily specified by means of Story Diagrams.

An important potential advantage of graph transformation is that rules may yield a concise visual representation of complex transformations. Unfortunately, current graph transformation notation does not suffice to easily define model refactorings so their expressive power must be increased. Two mechanisms have been proposed so far: one for cloning, and one for expanding nodes by graphs. [29]

We have developed a prototype application in order to verify the usability of graph transformations for the purpose of model refactorings. It also serves as a feasibility study to illustrate that development of model refactoring tools is possible. The correctness of model refactorings presented in this dissertation has been verified using the prototype application.

The prototype application has been developed using the AGG API delivered together with the tool. AGG does not provide any satisfactory control structure for organizing and combining rules, and the supplied mechanisms were not sufficient to describe model refactorings. In order to reach our goal, we have represented the execution order of rules by means of graphs that are used to drive the control flow of model refactorings. That way, we have added the notion of *"controlled" graph transformation*, which was not previously available in AGG.

We have also explored some possible approaches in order to preserve the consistency among different kinds of UML diagrams during the application of model refactorings. A suitable solution that applies to all possible cases can be rarely found. A satisfactory approach would consist of identifying a set of possible solutions for each consistency problem and requesting the user to specify which one best applies to the situation.

In this dissertation we have shown the feasibility of model refactoring using the formalism of graph transformation. Our initial catalogue of model refactorings could be expanded, and directly supported in standard UML tools. The prototype application illustrates that development of such a model refactoring tool is possible. We have also analysed and discussed the necessary improvements of the graph transformation notion and graph transformation tools in order to increase their expressive power and allow a better support for the specification of model refactorings.

Bibliography

[1] S. Sendall and W. Kozaczynski, "Model transformation: The heart and soul of model-driven software development," *IEE Software*, pp. 42–45, 2003.

[2] W. F. Opdyke, *Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[3] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[4] Object Management Group, "Unified Modeling Language: Superstructure version 2.0." formal/2005-07-04, August 2005.

[5] Object Management Group, "Unified Modeling Language: Infrastructure version 2.0." formal/2005-07-05, August 2005.

[6] A. Finkelstein, G. Spanoudakis, and D. Till, "Rule-based detection of inconsistency in UML models," in *Joint proceedings of second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints 96) on SIGSOFT 96 workshops*, pp. 172–174, ACM Press, 1996.

[7] G. Spanoudakis and A. Zisman, *Handbook of Software Engineering and Knowledge Engineering*, ch. Inconsistency management in software engineering: Survey and open research issues, pp. 329–380. World scientific, 2001.

[8] R. Van Der Straeten, *Inconsistency Management in Model–Driven Engineering: An Approach using Description Logics*. PhD thesis, Department of Computer Science, Vrije Universiteit Brussel, Belgium, September 2005.

[9] Wikipedia, "Unified Modeling Language," November 12 2006.

[10] T. Mens and P. Van Gorp, “A taxonomy of model transformation,” in *Proc. Int’l Workshop on Graph and Model Transformation (GraMoT 2005)*, September 2005.

[11] J.-M. Favre, “Towards a basic theory to model model driven engineering,” in *Proc. 3rd Workshop in Software Model Engineering (Satellite workshop at the 7th International Conference on the UML)*, 2004.

[12] T. Mens and T. Tourwé, “A survey of software refactoring,” *IEEE Transactions on Software Engineering*, vol. 30, pp. 126–162, February 2004.

[13] D. B. Roberts, *Practical Analysis for Refactoring*. PhD thesis, University of Illinois at Urbana-Champaign, 1999.

[14] J. Zhang, Y. Lin, and J. Gray, “Generic and domain-specific model refactoring using a model transformation engine,” in *Model-Driven Software Development*, pp. 199–217, Springer Berlin Heidelberg, 2005.

[15] T. Mens, S. Demeyer, and D. Janssens, “Formalising behaviour preserving program transformations,” in *Proc. Int’l Conf. Graph Transformation* (A. Corradini, H. Ehrig, H.-J. Kreowski, and G. Rozenberg, eds.), vol. 2505 of *Lecture Notes in Computer Science*, pp. 286–301, Springer-Verlag, 2002.

[16] M. Nagl, ed., *A tutorial and bibliographical survey on graph-grammars*, Graph-Grammars and their Application to Computer Science and Biology, Springer-Verlag, 1979.

[17] H. Ehrig, M. Nagl, G. Rozenberg, and A. Rosenfeld, eds., *Graph-Grammars and Their Application to Computer Science*, vol. 291 of *Lecture Notes in Computer Science*, Springer-Verlag, 1987.

[18] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer, *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006.

[19] A. Corradini, U. Montanari, and F. Rossi, “Graph processes,” *Fundamenta Informaticae*, vol. 26, no. 3 and 4, pp. 241–265, 1996.

[20] J. Niere and A. Zündorf, “Using Fujaba for the development of production control systems,” in *Proc. Int. Workshop Agtive 99* (M. Nagl, A. Schürr, and M. Münch, eds.), vol. 1779 of *Lecture Notes in Computer Science*, pp. 181–191, Springer-Verlag, 2000.

[21] T. Mens, “On the use of graph transformations for model refactoring,” in *Generative and transformational techniques in software engineering* (J. V. Ralf Lämmel, Joao Saraiva, ed.), pp. 67–98, Departamento di Informatica, Universidade do Minho, 2005.

[22] T. Mens, P. Van Gorp, D. Varró, and G. Karsai, “Applying a model transformation taxonomy to graph transformation technology,” in *Proc. Int’l Workshop on Graph and Model Transformation (GraMoT 2005)*, September 2005.

[23] G. Taentzer, “Agg: A tool environment for algebraic graph transformation,” in *Applications of Graph Transformations with Industrial Relevance*, vol. 1779 of *Lecture Notes in Computer Science*, pp. 481–488, Springer-Verlag, 1999.

[24] TU Berlin, “The Attributed Graph Grammar System, version 1.5.0,” 2006.

[25] H. Ehrig and Michael Löwe, “Parallel and distributed derivations in the single-pushout approach,” *Theoretical Computer Science*, vol. 109, pp. 123–143, 1993.

[26] G. Taentzer, I. Fischer, M. Koch, and V. Volle, “Visual design of distributed systems by graph transformation,” in *Concurrency, Parallelism, and Distribution.*, vol. 3 of *Handbook of Graph Grammars and Computing by Graph Transformation*, World Scientific, 1999.

[27] Object Management Group, “Unified Modeling Language specification version 1.5.” formal/2003-03-01, March 2003.

[28] G. Sunyé, D. Pollet, Y. LeTraon, and J.-M. Jézéquel, “Refactoring UML models,” in *Proc. UML 2001*, vol. 2185 of *Lecture Notes in Computer Science*, pp. 134–138, Springer-Verlag, 2001.

[29] B. Hoffmann, D. Janssens, and N. Van Eetvelde, “Cloning and expanding graph transformation rules for refactoring,” *Proc. Int’l Workshop on Graph and Model Transformation (GraMoT 2005)*, vol. 152, pp. 53–67, 2006.

[30] A. Agrawal, G. Karsai, and F. Shi, "A UML–based graph transformation approach for implementig domain-specific model transformations," *International Journal on Software and Systems Modeling*, 2003.

[31] TU Berlin, "The Attributed Graph Grammar System, version 1.6.0," April 1 2007.

[32] Real-Time Systems Lab, Darmstadt University of Technology., "Moflon, version 1.0.0," December 15 2006.

[33] Object Management Group, "Meta object facility (MOF) 2.0 specification," April 2003.

[34] Andy Schürr, "Specification of graph translators with triple graph grammars," Tech. Rep. AIB 94-12, RWTH Aachen, 1994.

[35] Sun Microsystems, "The Java Metadata Interface (JMI) Specification." JSR-000040, June 2002.

[36] U. Nickel, J. Niere, and A. Zündorf, "The fujaba environment," in *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, (New York, NY, USA), pp. 742–745, ACM Press, 2000.

[37] Universitat–Gesamthochschule Paderborn, "Fujaba," 1998.

[38] A. Zündorf, "Rigorous object oriented software development," 2001. Habilitation Thesis.

[39] L. Geiger and A. Zündorf, "Graph based debugging with fujaba," *IEE Software*, vol. 72, no. 2, 2002.

[40] T. Fischer, J. Niere, L. Torunski, and A. Zündorf, "Story diagrams: A new graph rewrite language based on the unified modeling language and java.," in *TAGT*, pp. 296–309, 1998.

[41] I. Diethelm, L. Geiger, and A. Zündorf, "Systematic story driven modeling, a case study," in *Workshop on Scenarios and state machines: models, algorithms, and tools*, May 2004.

[42] Object Management Group, "XML Metadata Interchange (XMI), v2.1." formal/05-09-01, January 2005.

[43] B. Hnatkowska, Z. Huzar, and J. Magott, "Consistency checking in UML models."

[44] W. Liu, S. Easterbrook, and J. Mylopoulos, “Rule-based detection of inconsistency in UML models,” 2002.

[45] G. Engels, R. Heckel, J. M. Kuster, and L. Groenewegen, “Consistency-preserving model evolution through transformations.,” in *UML*, pp. 212–226, 2002.

[46] G. Engels, R. Heckel, and J. M. Kuster, “The consistency workbench: A tool for consistency management in UML-based development.,” in *UML*, pp. 356–359, 2003.

[47] J. M. Kuster and G. Engels, “Consistency management within model-based object-oriented development of components.,” in *FMCO*, pp. 157–176, 2003.

[48] I. Ivkovic and K. Kontogiannis, “Model synchronization as a problem of maximizing model dependencies,” in *OOPSLA ’04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, (New York, NY, USA), pp. 222–223, ACM Press, 2004.

[49] G. Csertan, G. Huszerl, I. Majzik, Z. Pap, A. Pataricza, and D. Varro, “Viatra – visual automated transformations for formal verification and validation of UML models,” *ase*, vol. 00, p. 267, 2002.

[50] S. Sendall, “Combining generative and graph transformation techniques for model transformation: An effective alliance?,” in *2nd OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture*, Lecture Notes in Computer Science, 2003.