Automated Model Transformations Based on STRIPS Planning

Oldřich Nouza[1], Vojtěch Merunka[2], and Miroslav Virius[3]

[1] Czech Technical University in Prague, Faculty of Nuclear Sciences and Physical Engineering, Department of Software Engineering in Economy,
nouza@fjfi.cvut.cz
[2] Czech University of Life Sciences Prague, Faculty of Economics and Management, Department of Information Engineering,
merunka@pef.czu.cz
[3] Czech Technical University in Prague, Faculty of Nuclear Sciences and Physical Engineering, Department of Software Engineering in Economy,
virius@fjfi.cvut.cz

Abstract. This paper deals with application of STRIPS planning in automated model transformations. Object oriented model is viewed as a state space containing possible models as states and elementary transformations as state transitions. A source model is represented by an initial state, a target model by a goal state. Automation of model transformation consists in finding a plan to reach a goal state in this state space.

Key words: automated model transformations, modeling, object oriented approach, refactoring, SBAT, STRIPS planning

1 Introduction

Transformations play a key role in software engineering. Although there exist satisfiable solutions of automated transformations of models to text, the same cannot be said about transformations of models to models. This area is still in phase of research and exploring of possibilities [1].

According to the approaches of present implemented in several CASE tools, every model transformation requires application of the corresponding transformation rule with suitable parameters [3, 4, 6, 13] Unfortunately, as mentioned in [4], this approach has one big disadvantage, which is a low reusability. For every transformation that has no rule defined, it is necessary to either apply a composition of other rules, or define a new transformation rule. In this paper, we introduce transformation engine named SBAT (STRIPS Based Transformation Engine) that does not require such steps.

2 Model Transformations

There are several ways how to define model transformation. In this paper, we introduce the definition presented in [5] and cited by [8]:

J. Barjis, M.M. Narasipuram, G. Rabadi, J. Ralyté, and P. Plebani (Eds.):
CAiSE 2010 Workshop EOMAS'10, Hammamet, Tunisia, pp. 1-13, 2010.

> *A* transformation *is the automatic generation of a target model from a source model, according to a transformation definition. A* transformation definition *is a set of transformation rules that together describe how a model in the source language can be transformed into a model in the target language. A* transformation rule *is a description of how one or more constructs in the source language can be transformed into one or more constructs in the target language.*

2.1 Classification of Transformations

Publication [8] describes two criteria of classification of model transformations. The first one is a difference of abstraction level of source and target models:

Horizontal transformation – A transformation where source and target models have the same level of abstraction. A typical example is refactoring.
Vertical transformation – A transformation where source and target models have a different level of abstraction. A typical example is refinement.

The second classification criteria is a difference of modeling languages in which the source and targets models are expressed:

Endogenous transformation – A transformation where source and target models are expressed in the same language. Typical examples are refactoring and normalization.
Exogenous transformation – A transformation where source and target models are expressed in different languages. Typical examples are code generation and reverse engineering.

In this paper, we focus more closely on refactoring.

3 Refactoring

There exist several ways how to define refactoring. The definition presented in [7] says that refactoring is an improvement of software system without changing its behavior. In other words, for the same input, the refactored software must return the same output as the original software. Detail information on refactoring is available in [2].

3.1 Complex Refactorings and Primitive Refactorings

The idea of complex refactoring as composition of finite primitive (atomic) refactorings was first published in [10], where formal definition of C++ code refactoring was discussed, and later in [12], where it was demonstrated on refactoring of UML models. We have used the same idea to construct the SBAT transformation engine.

4 STRIPS Planning

Technical report [9] defines STRIPS as following:

> *STRIPS (STanford Research Institute Problem Solver) belongs to the class of problem solvers that search a space of "world models" to find one in which a given goal is achieved. For any world model, we assume there exists a set of applicable operators each of which transforms the world model to some other world model. The task of the problem solver is to find some composition of operators that transforms a given initial world model into one that satisfies some particular goal condition.*

The STRIPS language is based on the calculus of first-order predicate logic. Formally, the STRIPS problem can be expressed by the following definitions:

Definition 1. STRIPS problem *is an ordered triple* (I, O, G)*, where* I *is an initial state,* O *is a set of operators, and* G *is a goal state condition.*

Definition 2. Operator $o(\bar{x})$ *is defined as an ordered triple*(P, A, D)*, where*$P = (\bar{x})$ *is an* application condition*,* $A = [A_1(\bar{x}), \ldots, A_l(\bar{x})]$ *is a set of formulas which become true after operator application,* $D = [D_1(\bar{x}), \ldots D_m(\bar{x})]$ *is a set of formulas which become false after operator application,* $\bar{x} = (x_1, \ldots x_n)$ *are free variables contained in formulas* $P, A_1, \ldots, A_l, D_1, \ldots D_m$*,* $n \in \mathbb{N}^+, l, m \in \mathbb{N}^0$*. Elements of* A *are called* add-effects*, elements of* D delete-effects*.*

Definition 3. *Let* $o(x_1, \ldots, x_n) = (P, A, D) \in O$ *be operator.* A transition function $o' : (x_1 \times x_2 \times \ldots \times x_n \times S) \to S$*, where* S *is a state set, is defined in the following way:*

$$o'(x_1, \ldots, x_n, s) \stackrel{def}{=} \frac{(s \cup A) - D \qquad \varnothing}{s \vDash P} \tag{1}$$

A goal is to find such list of operators applications, which cause transition for initial state to the state satisfying the goal state condition. More formally:

Definition 4. *State* s_m *is a* solution *of problem* (I, O, G)*, if there exists a list of operators applications* $o_1(\overline{h_1}), \ldots, o_m(\overline{h_m})$*, where* $\overline{h_i}$ *are vectors of constants,* $m \in \mathbb{N}$*, and:*

1. $s_0 = I$
2. $(\forall i \in \hat{m})\, s_i = o'_{i-1}(\overline{h_{i-1}}, s_{i-1})$
3. $s_m \vDash G$

If $I \vDash G$*, then the solution, which is* I *in this case, is called trivial solution.*

5 SBAT Transformation Engine

5.1 Requirements

Resulting from the facts on present state model transformation discussed in the introduction of this paper, we have decided to design a new transformation engine fulfilling the following requirements:

1. The engine will support model transformations such as refactoring.
2. The input of transformation process will be the source model and conditions of a target model.
3. The output of transformation process will be a target model, or information that the source model cannot be transformed to any model fulfilling the input conditions.
4. The source model and the target model will be consistent in light of behavior of modeled system.
5. The transformation process will be fully automatized, without need to specify transformation rules on input.
6. The engine will be universal and sufficiently reusable for wide scale of object models.

5.2 Formal Definition of Object Model

In this paper, formal definition of object model is based on simplified metamodel of UML 2.0 class diagrams, in detail described in [11]. Because of intended generality, we do not focus on implementation details, such as method parameters and bodies, access mode of class members, etc.

5.2.1 Model as State Space For our purposes we use the primitive refactoring composition mentioned earlier. For this reason, we define model as a state space, where state set represents model in all possible states and state transitions represent primitive refactorings.

Definition 5. *Let C be a class universe, A attribute universe, and F method universe. Let$Object, Client \in C$ be classes, where $\forall\,(x \in C - [Object])\,(Object \prec c)$, which means that Object is parent of all other classes and Client represent a client class, whose object sends messages to objects of classes in model. Letϵ be "empty value".* Model state *s is an ordered 5-tuple $(C_s, in_s, super_s, type_s, send_s)$, where*

- *$C_s \subset C - [Object, Client]$ is a finite set of* classes of model in state *s,*
- *$in_s \subseteq ((A \cup F) \times C_s)$ is a binary relation named* "is in class" *defined as*

$$in_s \stackrel{def}{=} \{(x, y)\,|x \in (Attr\,(y) \cup Meth\,(y)) \wedge y \in C_s\}\,, \tag{2}$$

where $Attr\,(y)$ and $Meth\,(y)$ are sets of attributes and methods respectively of class y,

– $super_s \subseteq ((C_s \cup [Object]) \times C_s)$ *is a binary relation* "is superclass" *defined as*

$$super_s \stackrel{def}{=} \{(x,y) \,|x = super\,(y) \wedge x, y \in C_s\}\,, \tag{3}$$

where $super\,(y)$ *means "a superclass of* y*",*

– $type_s \subseteq (A \times C_s) \cup (F \times (C_s))$ *is a binary relation named* "is of type" *defined as*

$$type_s \stackrel{def}{=} \{(x,y) \,|x = type\,(y) \wedge y \in C_s\}\,, \tag{4}$$

where $type\,(y)$ *means "a type of* y*",*

– $send_s \subseteq (F \times (C_s \cup [Client]) \times (A \cup F) \times C_s)$ *is a 4-ary relation named* "sends message" *defined as*

$$\begin{aligned} send_s \stackrel{def}{=} \{(x,y,u,v) \,| &(x,y) \in in_s \\ &\wedge (\exists w)\,((u,w) \in in_s \wedge (u \prec w \vee u = w)) \\ &\wedge \langle x, \Lambda\rangle \in Meth\,(y)\}\,, \end{aligned} \tag{5}$$

where lambda-expression Λ *contains message sending* $o \lhd u$*, where* o *is an instance of class* w*.*

The following conditions must be fulfilled:

– *in the class hierarchy, each attribute appears at most once, so*

$$(\forall x \in A)\,(\forall y, z \in C)\,(in_s\,(x,y) \wedge in_s\,(x,z) \rightarrow \neg\,(y \prec z) \wedge \neg\,(z \prec y))\,, \tag{6}$$

– *each attribute is of some type, so*

$$(\forall x \in A)\,(\exists y \in C_s)\,(type_s\,(x,y))\,, \tag{7}$$

– *each attribute is of at most one type and each method has at most one return value type, so*

$$(\forall x \in (A \cup F))\,(\forall y, z \in C_s)\,(type_s\,(x,y) \wedge type_s\,(x,z) \rightarrow y = z)\,. \tag{8}$$

Definition 6. Model *is a state space* (M, Φ)*, where* M *is a finite set of model states and* $\Phi = \bigcup_{i=1}^{n} \left[\varphi_i : M \times E^{k_i} \rightarrow M\right], (\forall i \in \hat{n})\,(k_i \in \mathbb{N})$ *is a set of transformation rules.*

5.3 Model Transformation Problem

Each model transformation requires answers to the following questions [8]:

1. What needs to be transformed?
2. What will be the result of the transformation?

To find answers, we have to formulate the transformation problem and set the principle of its solution. This can be done by several ways, we have decided to apply the STRIPS planning.

5.4 STRIPS Planning application

Let's assume any finite subset B of object universal, containing elements of all possible model states. To formulate STRIPS problem (I, O, G) for model transformation, we must describe the model states and transformation rules using first-order predicate logic calculus. For this purpose, we define predicates shown in table 1.

Table 1. Predicates for STRIPS problem formulation in SBAT engine

Predicate	Declaration	Definition
class c is in model	$inModel\,(c)$	$c \in C_s$
class c is not in model	$outOfModel\,(c)$	$c \in (B - C_s)$
attribute a is in class c	$attrInClass\,(a, c)$	$(a, c) \in in_s$
attribute a is not in class c	$attrOutOfClass\,(a, c)$	$\neg\,((a, c) \in in_s) \wedge a \in (B \cap A)$
method μ is in class c	$methInClass\,(\mu, c)$	$(\mu, c) \in in_s$
method μ is not in class c	$methOutOfClass\,(\mu, c)$	$\neg\,((\mu, c) \in in_s) \wedge \mu \in (B \cap F)$
attribute a is of type t	$hasType\,(a, t)$	$(a, t) \in type_s$
method μ has return value type t	$hasRetType\,(\mu, t)$	$(\mu, t) \in type_s$
class c is superclass of d	$superClass\,(c, d)$	$(c, d) \in super_s$
class c is a parent of d	$parent\,(c, d)$	$c \prec d$
method $\mu$of class c sends messageη to objects of class d	$sending\,(\mu, c, \eta, d)$	$(\mu, c, \eta, d) \in send_s$

5.4.1 Initial State Formulation Let $m_I = (C_I, in_I, super_I, type_I, send_I)$ be a model in initial state. We construct initial state I of STRIPS problem by the following steps:

1. Put $I := \varnothing$.
2. Add formulas about existence of classes in model:
   a) $(\forall c \in C_I)$ put $(I := I \cup [inModel\,(c)])$ and
   b) $(\forall c \in (B - C_I))$ put $(I := I \cup [outOfModel\,(c)])$.
3. Add formulas about class attributes:
   a) $(\forall\,(a, c) \in (in_I \cap (B \cap A, C_I)))$ put
   $(I := I \cup attrInClass\,(a, c))$ and

b) $(\forall (a, c) \in ((B \cap A, B \cap C) - in_I))$ put
$(I := I \cup attrOutOfClass\,(a, c))$.

4. Add formulas about class methods:
   a) $(\forall (\mu, c) \in (in_I \cap (B \cap F, C_I)))$ put
   $(I := I \cup methInClass\,(\mu, c))$ and
   b) $(\forall (\mu, c) \in ((B \cap F, B \cap C) - in_I))$ put
   $(I := I \cup methOutOfClass\,(\mu, c))$.
5. Add formulas about inheritance:
   $(\forall (c, d) \in super_I)$ put $(I := I \cup superClass\,(c, d))$.
6. Add formulas about attribute types and return value types of methods:
   a) $(\forall (a, c) \in (type_I \cap (B \cap A, C_I)))$ put
   $(I := I \cup hasType\,(a, c))$ and
   b) $(\forall (\mu, c) \in (type_I \cap (B \cap F, C_I)))$ put $(I := I \cup hasType\,(\mu, c))$.
7. Add formulas about message sending:
   $(\forall (a, c, b, d) \in send_I)$ put $(I := I \cup sending\,(a, b, c, d))$.

5.4.2 Formulation of Goal State Condition A goal state condition G is defined by the formula that is true in any goal state.

5.4.3 Formulation of Operator Set The set of operators O is defined identically for each particular problem, because it represents a set of primitive refactorings. The complete definition of the operator set is described in table 3 in appendix.

5.5 Example

5.5.1 Problem Let's suppose a simplified class model of file system (see fig. 1).

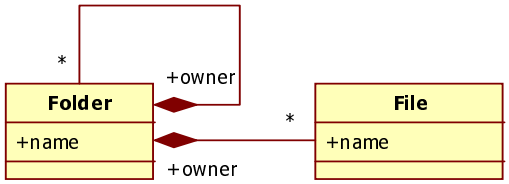

Fig. 1. File system class model

The goal is to transform this model into state which satisfies the composite design pattern.

5.5.2 Solution The initial model state $m_s = (C_s, in_s, super_s, type_s, send_s)$, where:

- $C_s = [Folder, File, String]$
- $in_s = [(name, Folder), (owner, Folder), (name, File), (owner, File)]$
- $super_s = [(Object, Folder), (Object, File), (Object, String)]$
- $type_s = [(name, String), (owner, Folder)]$
- $send_s = \{(Client, main, x, y) \,|(x, y) \in in_s\}$

The corresponding initial state of STRIPS problem is the following:

$$\begin{aligned} I = [&inModel(Folder), inModel(File), inModel(String), \\ &outOfModel(Element), \\ &attrInClass(name, Folder), attrInClass(owner, Folder), \\ &attrInClass(name, File), attrInClass(owner, File), \\ &attrOutOfClass(name, Elememt), attrOutOfClass(owner, Element), \\ &superClass(Object, File), superClass(Object, Folder), \\ &hasType(owner, Folder), hasType(name, String), \\ &sending(Client, main, Folder, name), \\ &sending(Client, main, Folder, owner), \\ &sending(Client, main, File, name), \\ &sending(Client, main, File, owner)] \end{aligned} \tag{9}$$

The goal state condition is the following:

$$\begin{aligned} G = &inModel(Element) \wedge attrInClass(name, Element) \wedge \\ &attrInClass(owner, Element) \wedge superClass(Element, File) \wedge \\ &superClass(Element, Fodler) \end{aligned} \tag{10}$$

The STRIPS planner reaches the goal state by application of satisfiable operators (see table 2).

Table 2. List of operators application to reach the goal state

Operator application	Description
$addClass\,(Element)$	Add class *Element* to the model.
$changeSup\,(File, Object, Element)$	Change superclass *Object* of class *File* to *Element*.
$changeSup\,(Folder, Object, Element)$	Change superclass *Object* of class *Fodler* to *Element*.
$attrUp\,(name, Element, File, Folder)$	Move attribute *name* to class *Element* from its subclasses *File* and *Folder*.
$attrUp\,(owner, Element, File, Folder)$	Move attribute *owner* to class *Element* from its subclasses *File* and *Folder*.

The goal state is as follows:

$$\begin{aligned} Goal = [&inModel\,(Folder)\,, inModel\,(File)\,,\\ &inModel\,(String)\,, inModel\,(Element)\,,\\ &attrInClass\,(name, Element)\,, attrInClass\,(owner, Element)\,,\\ &attrOutOfClass\,(name, Folder)\,,\\ &attrOutOfClass\,(owner, Folder)\,,\\ &attrOutOfClass\,(name, File)\,, attrOutOfClass\,(owner, File)\,,\\ &superClass\,(Element, File)\,, superClass\,(Element, Folder)\,,\\ &hasType\,(owner, Folder)\,, hasType\,(name, String)\,,\\ &sending\,(Client, main, Folder, name)\,,\\ &sending\,(Client, main, Folder, owner)\,,\\ &sending\,(Client, main, File, name)\,,\\ &sending\,(Client, main, File, owner)] \end{aligned} \tag{11}$$

A class model in UML notation corresponding to the goal state is shown in fig. 2.

6 Conclusion and Future Work

In this paper we have introduced SBAT transformation engine based on STRIPS planning system. This engine automates refactoring of the given source model to a target model fulfilling the input condition.

The main asset of SBAT engine for practice is a contribution to an improvement of automation of object model transformations, which consequently would implicate saved human resources for software projects. Then, these resources could be allocated for example on software debugging or testing tasks, rather than on model transformation ones. Another asset should be a theoretical background for research activities in the area of model transformations.

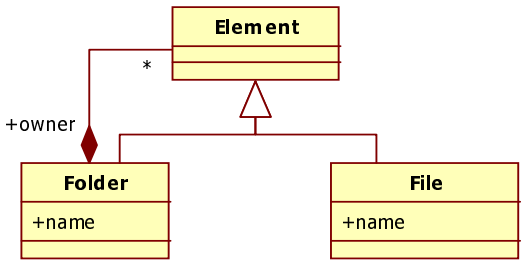

Fig. 2. File system as Composite design pattern

Several consecutive research topics appeared during the SBAT development, e.g. the model state space optimization by reduction of states count or implementation of SBAT in some CASE tool.

Acknowledgment

The authors would like to acknowledge the support of the research grant project SGS10/094 of the Czech Ministry of Education, Youth and Sports.

References

1. Czarnecki K. and Helsen S.: Feature-based survey of model transformation approaches. In: IBM Systems Journal 2006, vol. 45, no. 3, pp. 621-645. (2006)
2. Fowler M.: Refactoring. Addison-Wesley. ISBN 0-201-48567-2. (1999)
3. Gray J., Lin Y., and Zhang J.: Automating Change Evolution in Model-Driven Engineering. In Computer 2006, vol. 31, no. 2, pp. 51. (2006)
4. Jézequel J-M.: Model Transformation Techniques. Available online at `http://modelware.inria.fr/static_pages/slides/ModelTransfo.pdf`. (2005)
5. Kleppe A. G., Warmer J., and Bast W.: MDA Explained: The Model Driven Architecture: Practice and Promise. Boston (MA, USA): Addison-Wesley Longman Publishing Co., Inc., 170 p. ISBN:032119442X. (2003)
6. Lin Y amd Gray J: A model transformation approach to automatic model construction and evolution. In Proceedings of the 20th IEEE/ACM international Conference on Automated Software Engineering, pp. 448-451. (2005)
7. Markovic S.: Composition of UML Described Refactoring Rules. In OCL and Model Driven Engineering, UML 2004 Conference Workshop, pp. 45-59. (2004)
8. Mens T., Czarnecki K, and Gorp P. V.: A Taxonomy of Model Transformations. In Language Engineering for Model-Driven Software Development, ser. Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Dagstuhl (Germany). (2005)
9. Nilson N. J. and Fikes R. E. STRIPS: A new approach to the application of theroem proving to problem solving. Standford Research Institute, Menlo Park (California) 34 p. (1970).

10. Opdyke W. Refactoring Object-Oriented Frameworks. University of Illinois at Urbana-Champaign, Champaign, (IL, USA), PhD. thesis, 197 p. (1992)
11. Object Management Group (OMG): OMG Unified Modeling Language (OMG UML), Infrastructure: Version 2.2. 226 p. Available online at `www.omg.org`. (2009)
12. Sunyé G., Pollet D., Le Traon Y., Jézéquel J-M.: Refactoring UML Models. In Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools. Springer-Verlag, London (UK), pp. 134-148. ISBN 3-540-42667-1. (2001)
13. Zhang J., Lin Y., and Gray, J.: Generic and Domain-Specific Model Refactoring using a Model Transformation Engine. In Volume II of Research and Practice in Software Engineering, pp. 199-218. (2005)

Appendix: Primitive Refactorings as STRIPS Operators

Table 3: Primitive refactorings as STRIPS operators

Primitive refactoring	**STRIPS Operator**	
Add class c	Declaration	$addClass\,(c)$
	Condition	$outOfModel\,(c)$
	Add-effects	$inModel\,(c)$ $(\forall x \in B \cap C)\, attrOutOfClass\,(x, c)$ $(\forall x \in B \cap F)\, methOutOfClass\,(x, c)$
	Delete-effects	$outOfModel\,(c)$
Add attribute a of type t into class c	Declaration	$addAttr\,(a, t, c)$
	Condition	$(\forall x)\,(attrOutOfClass\,(x, d)$ $\vee \neg superClass\,(x, c) \vee \neg superClass\,(c, x)$ $\vee x \neq c) \wedge$ $(\forall x)\,(\neg hasType\,(a, x) \vee (x = t))$
	Add-effects	$attrInClass\,(a, c)$
	Delete-effects	$attrOutOfClass\,(a, c)$
Add method μ to class c	Declaration	$addMeth\,(\mu, c)$
	Condition	$methOutOfClass\,(\mu, c) \wedge (\forall x, y, z)$ $(\neg sending\,(x, y, \mu, z) \vee \neg parent\,(z, c)$ $\vee (z \neq c))$
	Add-effects	$methInClass\,(\mu, c)$
	Delete-effects	$methOutOfClass\,(\mu, c)$
Remove class c	Declaration	$removeClass\,(c)$
	Condition	$inModel\,(c) \wedge (\forall x)\,(\neg parent\,(c, x)) \wedge$ $(\forall x)\,\neg attrInClass\,(x, c) \wedge$ $\neg methInClass\,(x, c)$
	Add-effects	$outOfModel\,(c)$
	Delete-effects	$inModel\,(c)$

Primitive refactoring	**STRIPS Operator**	
Remove attribute a from class c	Declaration	$removeAttr\,(a, c)$
	Condition	$attrInClass\,(a, c) \wedge (\forall x, y, z)$ $(\neg sending\,(x, y, \mu, z) \vee \neg parent\,(c, z)$ $\vee (z \neq c))$
	Add-effects	$attrOutOfClass\,(a, c)$
	Delete-effects	$attrInClass\,(a, c)$
Remove method μ from class c	Declaration	$removeMeth\,(\mu, c)$
	Condition	$methInClass\,(\mu, c) \wedge (\forall x, y, z)$ $(\neg sending\,(x, y, \mu, z) \vee \neg parent\,(c, z)$ $\vee (z \neq c))$ $\wedge (\forall x, y)\,(\neg sending\,(x, y, \mu, c) \vee (y = c))$
	Add-effects	$methOutOfClass\,(\mu, c)$
	Delete-effects	$methInClass\,(\mu, c)$
Change type t of attribute a to u	Declaration	$changeAtrrType\,(a, t, u)$
	Condition	$hasType\,(a, t) \wedge super\,(u, t)$
	Add-effects	$hasType\,(a, u)$
	Delete-effects	$hasType\,(a, t)$
Change type t of values returned by method μ to u	Declaration	$changeMethType\,(\mu, t, u)$
	Condition	$hasRetType\,(\mu, t) \wedge super\,(u, t)$
	Add-effects	$hasRetType\,(\mu, u)$
	Delete-effects	$hasRetType\,(\mu, t)$
Change superclass b of class a to c	Declaration	$changeSup(a, b, c)$
	Condition	$inModel\,(c) \wedge$ $superClass\,(b, a) \wedge (\neg parent\,(a, c)) \wedge$ $\forall\,(x, u, v, w)\,((\neg superClass\,(x, c)) \wedge$ $(\neg superClass\,(x, b)) \vee$ $(\neg sending\,(v, w, u, x))$ $\wedge (\neg sending\,(u, x, v, w)))$
	Add-effects	$superClass\,(c, a)$
	Delete-effects	$superClass\,(b, a)$
Move attribute a from class c to its all subclasses $b_1, \ldots, b_n$	Declaration	$attrDown\,(a, c, (b_1, \ldots, b_n))$
	Condition	$attrInClass\,(a, c) \wedge (\forall x)$ $(x = b_1 \vee \ldots \vee x = b_n$ $\vee \neg superClass\,(c, x))$ $\wedge (\forall x, y)\,\neg sending\,(x, y, a, c)$
	Add-effects	$attrOutOfClass\,(a, c)$ $(\forall i \in n)\,attrInClass\,(a, b_i)$
	Delete-effects	$attrInClass\,(a, c)$ $(\forall i \in n)\,attrOutOfClass\,(a, b_i)$
Move attribute a to class c from all its subclasses $b_1, \ldots, b_n$	Declaration	$attrUp\,(a, c, (b_1, \ldots, b_n))$
	Condition	$(attrInClass\,(a, x) \vee \neg superClass\,(c, x))$ $\wedge (\forall x)\,((x \neq b_1 \vee \ldots \vee x \neq b_n)$ $\vee \neg superClass\,(c, x))$
	Add-effects	$attrInClass\,(a, c)$ $(\forall i \in n)\,attrOutOfClass\,(a, b_i)$
	Delete-effects	$attrOutOfClass\,(a, c)$ $(\forall i \in n)\,attrInClass\,(a, b_i)$

Primitive refactoring	**STRIPS Operator**	
Copy methodμ from class c to its subclass b	Declaration	$methDown\,(\mu, c, b)$
	Condition	$methInClass\,(\mu, c) \wedge$ $methOutOfClass\,(\mu, b)$ $\wedge\,(superClass\,(c, b))$
	Add-effects	$methInClass\,(\mu, b)$
	Delete-effects	$methOutOfClass\,(\mu, b)$
Copy method μ to class c from its subclass b	Declaration	$methUp\,(\mu, c, b)$
	Condition	$methOutOfClass\,(\mu, c) \wedge$ $methInClass\,(\mu, b)$ $\wedge\,(superClass\,(c, b))$
	Add-effects	$methInClass\,(\mu, c)$
	Delete-effects	$methOutOfClass\,(\mu, c)$
Add message η sent to objects of class d from method μ of class c	Declaration	$addSend\,(\mu, c, \eta, d)$
	Condition	$(\neg sending\,(\mu, c, \eta, d) \wedge methInClass\,(\mu, c)$ $\vee\,(\exists x)\,((parent\,(x, d) \vee (x = d)) \wedge$ $(methInClass\,(\eta, x))$ $\vee\,(attrInClass\,(\eta, x))))$ $\wedge\,(\forall x, y)\,(\neg sending\,(x, y, \mu, c)$ $\wedge\,(c \neq Client))$
	Add-effects	$sending\,(\mu, c, \eta, d)$
	Delete-effects	$\varnothing$
Remove message η sent to objects of class d from method μ of class c	Declaration	$removeSend\,(\mu, c, \eta, d)$
	Condition	$sending\,(\mu, c, \eta, d) \wedge (c \neq Client)$ $\wedge\,(\forall x, y)\,(\neg sending\,(x, y, \mu, c))$
	Add-effects	$\varnothing$
	Delete-effects	$sending\,(\mu, c, \eta, d)$
Split message sending $d \lhd \eta$ to $(d \lhd l) \lhd \eta$, where l is of type e.	Declaration	$splitSend\,(\mu, c, \eta, d, l, e)$
	Condition	$sending\,(\mu, c, \eta, d) \wedge attrInClass\,(l, d)$ $\wedge hasType\,(l, e)$
	Add-effects	$sending\,(\mu, c, l, d)$ $sending\,(\mu, c, \eta, e)$
	Delete-effects	$sending\,(\mu, c, \eta, d)$
Merge message sending $(d \lhd l) \lhd \eta$ into $d \lhd \eta$, where l is of type e.	Declaration	$mergeSend\,(\mu, c, \eta, d, l, e)$
	Condition	$sending\,(\mu, c, \eta, e) \wedge sending\,(\mu, c, l, d)$ $\wedge attrInClass\,(l, d) \wedge hasType\,(l, e)$
	Add-effects	$sending\,(\mu, c, \eta, d)$
	Delete-effects	$sending\,(\mu, c, l, d)$ $sending\,(\mu, c, \eta, e)$