RuCORD: Rule-based Composite Operation Recovering and Detection to Support Cooperative Edition of (Meta)Models

Amanuel Koshima and Vincent Englebert
PReCISE Research Center, University of Namur, Namur, Belgium

Keywords: Model Refactoring, Rule-based Refactoring, Composite Operation Detection.

Abstract: The cooperative edition of (meta)models may be enacted by the exchange of change operation journals between the participants. But these are often composed of atomic operations (create, delete, set, ...) that have no useful meaning for the users. Hence, detecting and recovering composite operations is a crucial step to help users understand the history of their (meta)models in terms of higher level operations. As a result, conflict detection, reconciliation, and merging of modeling artifacts will be improved. In addition, composite operations can also be used to generate model migration instructions that can automatically migrate instance models.

1 INTRODUCTION

Model Driven Engineering (MDE) approach raises the level of abstraction of software development from code-centric to model-centric (Bézivin, 2005). In (Kühne, 2006), a model is defined as *"[...] an abstraction of a (real or language-based) system allowing predictions or inferences to be made"*. Indeed, models can be used to perform analysis, validation, simulation, and source code generation (Bézivin, 2005).

MDE uses Domain Specific Modeling Languages (DSML) to comprehend requirements of software applications within a specific domain. G. Booch et al. wrote *"the full value of MDA is only achieved when the modeling concepts map directly to domain concepts rather than computer technology concepts"* (Booch et al., 2004). It helps to reduce accidental complexities of a development of software system by letting domain experts to focus on specific concepts which are related to their field of expertise instead of underlying platforms (software/hardware). This approach has already proved its usefulness in many application areas (Kelly and Tolvanen, 2008).

In MDE, concepts are defined at different abstraction levels such as models, meta-models and meta-meta-models. A meta-meta-model is a DSML oriented toward software development methods and allows the definition of modeling languages, for example, MOF (Object Management Group (OMG), 2002) and EMF/Ecore (Steinberg et al., 2009). Finally, meta-models are DSML instances that specify a set of rules to construct valid instance models (Gonzalez-Perez and Henderson-Sellers, 2008).

Modeling complex systems is far-fetched to achieve for a single user, it's very often the matter of groups of stakeholders with specific specializations. They need thus to cooperate together with respect to their respective viewpoints. But like any other software artifact, DSMLs evolve along with their domain, but also due to new requirements, error corrections, or iterations in the understanding of the domain (Bézivin, 2005; Meyers and Vangheluwe, 2011). It's thus important to facilitate the collaborative work of engineers at all the levels of abstraction, that is, to support concurrent edition of models but also of their DSML. This leads to specific and interesting challenges.

In our research, we are investigating a particular kind of collaborative work with the following hypotheses: 1) meta-models can be considered as specialized models, 2) users cooperate by exchanging (meta-)models as autonomous self-contained chunks of information, and 3) users wish to cooperate in a "loose" way (i.e. asynchronous). These use cases are encountered in many situations such as inter-enterprise cooperation, normalisation workgroups, outsourced contracts, etc. Supporting this activity entails several challenges such as model comparison, conflict detection, models reconciliation and merging, and model migration. They are discussed in (Koshima et al., 2013).

Model comparison is an important activity to identify differences (aka. delta) between evolving mod-

Koshima A. and Englebert V..
RuCORD: Rule-based Composite Operation Recovering and Detection to Support Cooperative Edition of (Meta)Models.
DOI: 10.5220/0005339305850591
In *Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development* (CMDD-2015), pages 585-591
ISBN: 978-989-758-083-3

eling artifacts. Deltas are used to perform conflict detection, reconciliation, and merging activities (Koshima and Englebert, 2014; Langer et al., 2013). Operation-based and state-based tactics are the two widely used approaches to find deltas between two versions of a (meta)model (Mens, 2002; Altmanninger et al., 2009).

State-based approaches derive deltas from two successive versions of (meta)models by comparing their states (Mens, 2002; Altmanninger et al., 2009) while *change-based approaches* focus only on modifications performed on the (meta)models (Mens, 2002; Altmanninger et al., 2009). These operations are then used to compute deltas between two successive versions.

Operations constituted in deltas improve users understanding about the evolution of modeling artifacts (Langer et al., 2013). These operations can be classified as atomic, refactoring, or composite change operations (Langer et al., 2013). *Atomic operations* (i.e., creation, addition, deletion, update) are low level operations and it is a cognitively challenging task for users to understand and re-construct high level changes (which might reflect the intention of modifications) from primitive changes. Hence, atomic operations do not scale (Langer et al., 2013). A *composite change operation* denotes any type of in-place model transformation that executes all its children operations within one transaction (Langer et al., 2013).

Refactoring operations are composite operations that modify internal structures of software artifacts (i.e., models) without changing their external behavior (Mens and Taentzer, 2007). This kind of operation helps to improve the understanding of modifications and intentions of a user who performed changes. The recovering of refactoring operations is an important activity in model management (Langer et al., 2013) and is even a crucial task to ensure collaborative modeling.

Meta-model adaptation might cause instance models inconsistent. Instance models might not anymore satisfy the set of rules and constraints specified by the meta-model. Therefore, models need to be co-evolved with their respective meta-models in order to preserve the conformance between models and meta-models (Herrmannsdoerfer, 2009). Indeed, complex operations, which adapt a meta-model, can be coupled with model adaptation instructions to migrate instance models (Herrmannsdoerfer, 2009). Detecting such complex operations helps to identify a set of model migration instructions (Vermolen et al., 2012) that transform instance models.

This article presents the principles of a framework to recover composite operations that denote refactoring tasks from histories recorded in a journal. This information is required to guide users when they have to reconcile concurrent (meta)model versions produced by cooperative editing tasks. Indeed the reconciliation process may oblige users to choose between concurrent and conflicting operations (e.g. two modifications on the same resource). Confronted with atomic operations, this decision may be impossible to take, since operations have no pertinent meaning from the user's viewpoint.

The paper is organized as follows: Section 2 presents the objective of the work and Section 3 describes the related work to detect and recover both refactoring and complex operations. Additionally, Section 4 gives a detailed description about the proposed framework, RuCORD. Finally, Section 5 presents the future research direction and conclusion.

2 OBJECTIVE

Figure 1 depicts a sample Petri net meta-model along with a journal of atomic operations. When cooperative frameworks use a change-based approach to merge and reconcile concurrent versions, users are confronted with this information that may prove very difficult to understand. Indeed, it is not related to users intents but to a technical API. Users mostly reason about modifications in terms of high level operations (i.e., refactoring or composite operations). For example, one refactoring operation like "refine a class into subtypes" may concern one hundred atomic operations that wouldn't have any meaning for the users. When they have to solve a conflict, they would prefer to accept or refuse the "refinement" operation and certainly not each atomic operation that constitutes it. There is thus a cognitive gap between the primitive operations and the mental setup of the modelers about changes.

The objective of this work is to recover and detect composite operations (including refactoring operations) from a journal composed of primitive operations. In addition, the detected complex operations can be used to specify model migration instructions that co-evolve instance models along with meta-model adaptation (Vermolen et al., 2012).

3 RELATED WORK

Much research work has already be done to detect refactoring and complex change operations in the context of object oriented programming. Demeyer

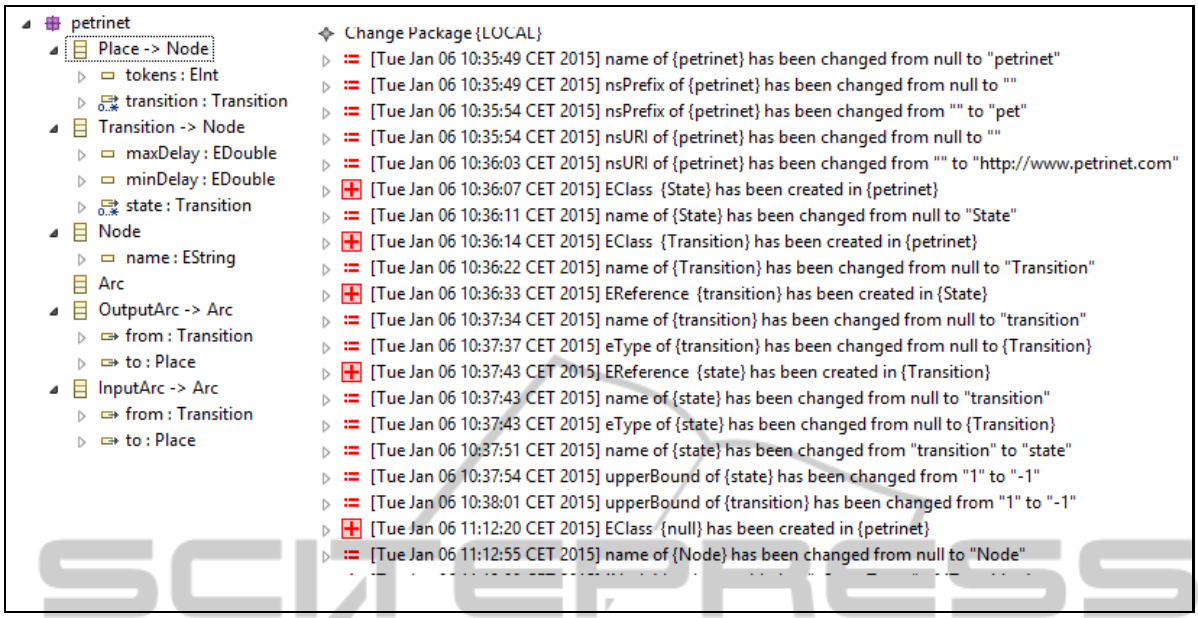

Figure 1: A Petri net meta-model and atomic meta-model adaptation operations.

et al. proposed a method to detect refactoring operations from successive versions based on change metrics (Demeyer et al., 2000). Besides, Dig et al. presented a RefactoringCrawler framework that uses a combination of syntactic analysis and semantic analysis to detect refactoring operations (Dig et al., 2006).

Composite operations can also be included as part of the development environment and they will be tracked whenever they are executed (Herrmannsdoerfer, 2009). However, this approach might produce a wrong list of composite operations, which contains some composite operation that are nullified by latest operations (Langer et al., 2013; Koshima et al., 2013). Besides, it cannot detect refactoring operations that are performed manually (Langer et al., 2013).

Operation-based collaborative modeling may canonize change operations to speed up the transfer of data and to facilitate the merging process. As a result, operations that are superseded by new ones can thus be cleaned from the history (i.e. the record of change operations). Hence, canonization of composite operations could produce a different type of composite operations that might not be defined by the modeling environment. Solving this problem requires to regroup primitive operations into other composite operations manually, that is a tedious and difficult task. Moreover, removing atomic changes from a composite operation might invalidate its constraints (Koshima et al., 2013). Hence, there should be a tool support to specify composite operations.

Prete et al. use refactoring template rules to recover refactoring operations between two program versions (Prete et al., 2010). In another work, Xing et al. (Xing and Stroulia, 2006) proposed an approach to detect refactoring operations based on UMLDiff. This is a domain specific algorithm (UML-aware) that compares the structural changes between two successive versions of class models and drives their differences (Xing and Stroulia, 2005). Queries are applied on deltas so as to detect different complex change operations. However, this approach is limited to a specific modeling language (Langer et al., 2013). Vermolen et al. also demonstrated a modeling language specific approach that reconstructs complex metamodel adaptation operations (Vermolen et al., 2012) . In (Langer et al., 2013), Philip et al. presented an approach that automatically detects composite operations between two successive models, which are defined in any Ecore (Steinberg et al., 2009) based modeling languages. However, their approach does not provide a facility to aggregate composite changes from another composite change(s).

Nevertheless, this raises interesting challenges. An automatic composite operation detection mechanism might find results that don't reflect users intentions. It would neither allow users interaction to identify the correct composite operations that reflect his/her intent. For instance, a user may wish to remove some atomic change operations from a composite operation while keeping the constraints of the composite operation. Indeed, an automatic process necessarily tries to compute the largest set of operations while recovering a composite operation. Nevertheless, some operations could be irrelevant because they

were executed with a different intent or in distinct stages. Besides, s/he might also add or remove rationale of changes attached to specific changes based on a new context of composite change operation. Hence, we argue that interactive composite change detection approach could improve the final result. However, the approach presented in (Prete et al., 2010; Xing and Stroulia, 2006; Langer et al., 2013; Vermolen et al., 2012) don't support user interactions to iteratively identify composite change operations.

4 RuCORD

RuCORD is an interactive rule-based composite operation detection and recovery framework. It is integrated within Eclipse and works with models expressed with EMF. Like the approach presented in (Langer et al., 2013) (see Section 3), RuCORD can work with any modeling language based on Ecore. RuCORD uses a rule-based engine to detect and to recover composite operations. Indeed, the definitions of composite operations are encoded as rules and they are stored in the rule-base, whereas deltas between two successive versions of a (meta)model are translated into facts in the fact-base. Afterwards, the Java Expert System Shell (Jess) rule engine (Hill, 2003) is used to identify the possible sets of composite operations. RuCORD also provides an interactive facility so that users can guide the detection and recovery process.

The aim of our work consists in recovering and detecting composite operations from a set of canonized operations, which are recorded during the (meta)model editing phase. Nevertheless, the work can also be used to find composite operations from *diffs* computed using state-based comparison. Figure 2 describes the proposed framework. Whenever a user adapts a (meta)model, edit operations are recorded in a history formally defined by the DiCoMEF meta-model (Koshima and Englebert, 2014). Later, these operations are canonized and translated into Jess facts by using a model-to-text transformation engine, Java Emitter Template JET (Foundation, 2014).

The definition of the composite operation patterns is written as a list of Jess rules. During the execution, when a rule's left hand side is satisfied, a new fact, which represents a composite operation, is asserted into the fact base. Composite operation facts can also be aggregated into larger facts. The result of this inference is a hierarchical representation of composite change operations.

The inference process may be non-deterministic.

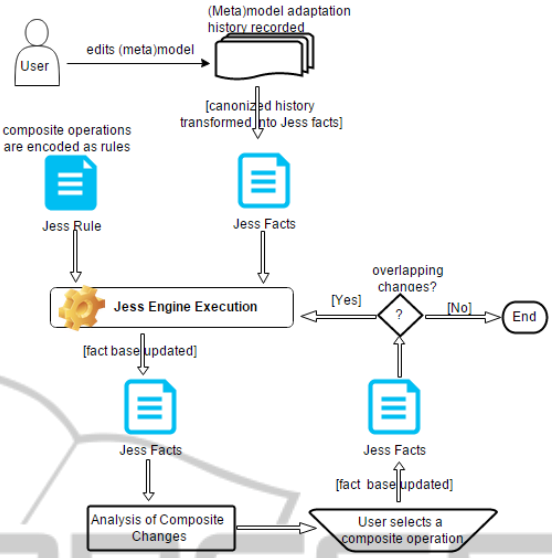

Figure 2: A rule-based composite operation detection and recovery steps.

Indeed, the analysis engine may identify overlapping composite operations. It then displays the result to a user who can then select the "best" composite operation, that is, the one which reflects his/her intentions. Besides, if rationale[1] of modifications are attached to change operations, the engine asks the user to verify if the rationale is still valid for the new composite operation. Indeed, a user can remove atomic operations from the proposed composite operation as long as constraints of the composite operations are satisfied.

The analysis engine examines dependencies between change operations and ordered composite operations. Of course, it uses the "*require*" relationship specified in (Koshima and Englebert, 2014) to identify the pre-condition of atomic and composite operations so as to order them. Hence, a user can sequentially execute an ordered list of composite operations on a base version of a (meta)model and studies their effect. Moreover, the analysis engine updates the fact-base based on the user's decisions and the Jess engine re-executes the rules until there is no more overlapping operations (composite operations that belong to different paths of a tree can't share the same operation(s)).

The appendix shows a code snippet of Jess facts that represents a set of operations. They create two classes (`State` and `Transition`) and one attribute (`name`). Besides, it specifies a creation rule and ag-

[1] A rationale is an extra information that provides some explanation about a decision, this information can be a text or any multimedia file for instance. This functionality is supported by the DiCoMEF framework.

gregation rules. For example, a composite operation, that creates an attribute, contains atomic operations such as create attribute, set name, set type, set lower bound, set upper bound, and other modifier operations related to the same attribute.

5 CONCLUSION AND FUTURE WORK

This article has presented a framework to recover refactoring operations from canonized operations or deltas. This framework is flexible enough to allow users to guide the result based on her/his preferences. Users can remove parts of composite operations detected by the analysis engine as long as the validity of the composite operations are preserved. The analysis rule can add/remove rationale of modifications to/from composite operations based on the user confirmation. It also analyzes dependencies among composite changes and orders them. Users can add their own composite operation patterns in defining their own Jess rules.

In the future work, we will study the analysis engine of RuCORD and integrate it with the DiCoMEF framework (Koshima and Englebert, 2014). Besides, we will encode different composite operations specified in (Herrmannsdoerfer, 2009) and evaluate the RuCORD framework.

REFERENCES

Altmanninger, K., Seidl, M., and Wimmer, M. (2009). A Survey on Model Versioning Approaches. Technical report, Johannes Kepler University Linz.

Bézivin, J. (2005). On the unification power of models. *Software and System Modeling*, 4(2):171–188.

Booch, G., Brown, A. W., Iyengar, S., Rumbaugh, J., and Selic, B. (2004). An MDA Manifesto.

Demeyer, S., Ducasse, S., and Nierstrasz, O. (2000). Finding refactorings via change metrics. *SIGPLAN Not.*, 35(10):166–177.

Dig, D., Comertoglu, C., Marinov, D., and Johnson, R. (2006). Automated detection of refactorings in evolving components. In *Proceedings of the 20th European Conference on Object-Oriented Programming*, ECOOP'06, pages 404–428, Berlin, Heidelberg. Springer-Verlag.

Foundation, T. E. (2014). JET (java emitter templates). http://www.eclipse.org/modeling/m2t/?project=jet.

Gonzalez-Perez, C. and Henderson-Sellers, B. (2008). *Metamodelling for Software Engineering*. John Wiley, New York.

Herrmannsdoerfer, M. (2009). Operation-based versioning of metamodels with COPE. In *Proceedings of the 2009 ICSE Workshop on Comparison and Versioning of Software Models*, CVSM '09, pages 49–54, Washington, DC, USA. IEEE Computer Society.

Hill, E. F. (2003). *Jess in Action: Java Rule-Based Systems*. Manning Publications Co., Greenwich, CT, USA.

Kelly, S. and Tolvanen, J.-P. (2008). *Domain-Specific Modeling. Enabling full code generation*. Wiley-IEEE Computer Society Pr.

Koshima, A. and Englebert, V. (2014). Collaborative editing of emf/ecore metamodels and models: Conflict detection, reconciliation, and merging in dicomef. In *Proc. of the MODELSWARD 2014*, Lisbon, Portugal.

Koshima, A., Englebert, V., and Thiran, P. (2013). A reconciliation framework to support cooperative work with dsm. In Reinhartz-Berger, I., Sturm, A., Clark, T., Cohen, S., and Bettin, J., editors, *Domain Engineering*, pages 239–259. Springer Berlin Heidelberg.

Kühne, T. (2006). Matters of (meta-)modeling. *Software and System Modeling*, 5(4):369–385.

Langer, P., Wimmer, M., Brosch, P., Herrmannsdörfer, M., Seidl, M., Wieland, K., and Kappel, G. (2013). A posteriori operation detection in evolving software models. *J. Syst. Softw.*, 86(2):551–566.

Mens, T. (2002). A state-of-the-art survey on software merging. *IEEE Trans. Softw. Eng.*, 28:449–462.

Mens, T. and Taentzer, G. (2007). Model-driven software refactoring. In Dig, D., editor, *1st Workshop on Refactoring Tools, WRT 2007, in conjunction with 21st European Conference on Object-Oriented Programming, July 30 - August 03, 2007, Berlin, Proceedings*, pages 25–27.

Meyers, B. and Vangheluwe, H. (2011). A framework for evolution of modelling languages. *Sci. Comput. Program.*, 76(12):1223–1246.

Object Management Group (OMG) (2002). Meta Object Facility(MOF) Specification. http://www.omg.org/spec/MOF/1.4/PDF.

Prete, K., Rachatasumrit, N., Sudan, N., and Kim, M. (2010). Template-based reconstruction of complex refactorings. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, ICSM '10, pages 1–10, Washington, DC, USA. IEEE Computer Society.

Steinberg, D., Budinsky, F., Paternostro, M., and Merks, E. (2009). *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd edition.

Vermolen, S. D., Wachsmuth, G., and Visser, E. (2012). Reconstructing complex metamodel evolution. In *Proceedings of the 4th International Conference on Software Language Engineering*, SLE'11, pages 201–221, Berlin, Heidelberg. Springer-Verlag.

Xing, Z. and Stroulia, E. (2005). UMLDiff: An algorithm for object-oriented design differencing. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, pages 54–65, New York, NY, USA. ACM.

Xing, Z. and Stroulia, E. (2006). Refactoring detection based on UMLDiff change-facts queries. In *Proceedings of the 13th Working Conference on Reverse Engineering*, WCRE '06, pages 263–274, Washington, DC, USA. IEEE Computer Society.

APPENDIX

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;; A template that represents
;;;;  the DiCoMEF history meta-model
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(deftemplate valueChange (slot name)
    (slot changeID)
    (slot element)
    (slot feature)
    (slot value)
    (slot oldValue))
(deftemplate nonDeleteChange (slot name)
    (slot changeID)
    (slot type)
    (slot element)
    (slot feature)
    (slot target)
    (slot source))
(deftemplate deleteChange (slot name)
    (slot changeID)
    (multislot element)
    (slot feature)
    (slot target))
(deftemplate createRefactoring
    (slot name)
    (slot constraint)
    (multislot changes))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;     Assertion of facts
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

 (deffacts primitiveChanges
    (nonDeleteChange (changeID "01")
        (type "EClass")
        (element "Pt-1")
        (name "Create")
        (feature "eEclassifier")
        (target "Pt-PK"))
    (valueChange (changeID "02")
        (element "Pt-1")
        (name "Set")
        (feature "name")
        (value "Place"))
    (nonDeleteChange (changeID "03")
        (type "EClass")
        (element "Pt-2")
        (name "Create")
        (feature "eEclassifier")
        (target "Pt-PK"))
    (valueChange (changeID "04")
        (element "Pt-2")
        (name "Set")
        (feature "name")
        (value "Transition"))
    (nonDeleteChange (changeID "05")
        (type "EAttribute")
        (element "Pt-3")
        (name "Create")
        (feature "eStructuralFeatures")
        (target "Pt-1"))
    (valueChange (changeID "06")
        (element "Pt-3")
        (name "Set")
        (feature "name")
        (value "name"))
    (valueChange (changeID "07")
        (element "Pt-3")
        (name "Set")
        (feature "eType")
        (value "EString"))
    (valueChange (changeID "08")
        (element "Pt-3")
        (name "Set")
        (feature "uperaBound")
        (value "1"))
    (valueChange (changeID "10")
        (element "Pt-3")
        (name "Set")
        (feature "lowerBound")
        (value "1")))

         . . .
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;  Jess Rules that encodes Composite
;;;;       operations Change pattern
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; a creation refactoring rule
(defrule create-modelElement
  ?c1 <-(nonDeleteChange(name "Create")(type ?t)
       (element ?id))
  ?c2 <-(valueChange (name "Set") (element ?id))
  =>
 (assertFacts(create$"createRefactoring" ?id ?t ?c1 ?c2)))

;;; a creation refactoring rule
(defrule create-modelElement
  ?c <-(nonDeleteChange(name "Create")(type ?t)
      (element ?id))
    =>
    (assertFacts(create$ "createRefactoring" ?id ?t ?c)))
       . . .

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;; Utility Jess Rules and Functions that
;;;;     aggregate facts
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;This function merges two lists
;;;;; -the first element is the length of the first list
;;;;; -the second element is the length of the second list
;;;;; -list1 and list2 added into $?args
(deffunction mergeChanges ($?args)
   (bind ?len1 (nth$ 1 ?args))
   (bind ?len2 (nth$ 2 ?args))
   (bind ?end1 (+ 2 ?len1))
   (bind ?end2 (+ ?end1 ?len2))
   (bind $?chg1 (subseq$ ?args 3 ?end1))
   (bind $?chg2 (subseq$ ?args (+ ?end1 1) ?end2))
   (bind $?temp (create$ ?chg1))
   (foreach ?c ?chg2
     (if (not (member$ ?c ?temp)) then
       (bind ?temp(insert$ ?temp(+ (length$ ?temp)1)?c))))
    (return ?temp))

;;;; this rule aggregates facts and
;;;; it also retracts children facts
(defrule merege-creates
    ?c1 <- (createRefactoring (name ?x)
             (constraint ?id) (changes $?cgs1))
    ?c2 <- (createRefactoring (name ?x)
             (constraint ?id) (changes $?cgs2))
    (test (not (eq ?cgs1 ?cgs2)))
    =>
    (bind $?intersection (intersection$ ?cgs1 ?cgs2))
    (bind ?len (length$ ?intersection))
    (if (and (> ?len 0) (= (length$ ?cgs1) ?len)) then
        (retract ?c1)
    else (if (and(> ?len 0)(=(length$ ?cgs2)?len))then
        (retract ?c2)
    else
        (bind ?p1 (length$ ?cgs1))
        (bind ?p2 (length$ ?cgs2))
        (bind ?merged(mergeChanges(create$ ?p1 ?p2 ?cgs1
          ?cgs2)))
        (assertFacts2(create$ "createRefactoring" ?id
          ?x ?merged))
        (retract ?c1)
        (retract ?c2))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;   Jess Facts      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

Fact-1  (nonDeleteChange (name "Create")
                (changeID "01")
                (type "EClass")
                (element "Pt-1")
                (feature "eEclassifier")
                (target "Pt-PK") (source nil))

Fact-2  (valueChange (name "Set")
                (changeID "02")
                (element "Pt-1")
                (feature "name")
                (value "Net")
                (oldValue nil))

Fact-3  (nonDeleteChange (name "Create")
                (changeID "03")
                (type "EClass")
```

```
                (element "Pt-2")
                (feature "eEclassifier")
                (target "Pt-PK")
                (source nil))

Fact-4  (valueChange (name "Set")
                (changeID "04")
                (element "Pt-2")
                (feature "name")
                (value "Transition")
                (oldValue nil))

Fact-5  (nonDeleteChange (name "Create")
                (changeID "05")
                (type "EClass")
                (element "Pt-3")
                (feature "eEclassifier")
                (target "Pt-PK")
                (source nil))

Fact-6  (nonDeleteChange (name "Create")
                (changeID "06")
                (type "EStructuralFeature")
                (element "Pt-4")
                (feature "eStructuralFeatures")
                (target "Pt-1")
                (source nil))

Fact-7  (valueChange (name "Set")
                (changeID "07")
                (element "Pt-4")
                (feature "name")
                (value "name")
                (oldValue nil))

Fact-8  (valueChange (name "Set")
                (changeID "08")
                (element "Pt-4")
                (feature "eType")
                (value "EString")
                (oldValue nil))

Fact-9  (valueChange (name "Set")
                (changeID "09")
                (element "Pt-4")
                (feature "uperaBound")
                (value "1")
                (oldValue nil))

Fact-10 (valueChange (name "Set")
                (changeID "10")
                (element "Pt-4")
                (feature "lowerBound")
                (value "1")
                (oldValue nil))

Fact-11(createRefactoring(name"CreateEStructuralFeature")
                (constraint "Pt-4")
                (changes <Fact-6>))

Fact-12(createRefactoring (name "CreateEClass")
                (constraint "Pt-3")
                (changes <Fact-5>))

Fact-13(createRefactoring (name "CreateEClass")
                (constraint "Pt-2")
                (changes <Fact-3> <Fact-4>))

Fact-14(createRefactoring (name "CreateEClass")
                (constraint "Pt-1")
                (changes <Fact-1> <Fact-2>))
```