Model Refactorings through Rule-Based Inconsistency Resolution

Ragnhild Van Der Straeten
Vrije Universiteit Brussel
System and Software Engineering Lab
Pleinlaan 2, 1050 Brussels, Belgium
rvdstrae@vub.ac.be

Maja D'Hondt
Université des Sciences et Technologies de Lille
Laboratoire d'Informatique Fondamentale
59655 Villeneuve d'Ascq Cédex, France
Maja.D-Hondt@lifl.fr

ABSTRACT

The goal of *model-driven engineering* is to raise the level of abstraction by shifting the focus to models. As a result, complex software development activities move to the modelling level as well. One such activity is *model refactoring*, a technique for restructuring the models in order to improve some quality attributes of the models. As a first contribution of this paper, we argue and show that refactoring a model is enabled by *inconsistency* detection and resolution. Inconsistencies in or between models occur since models typically describe a software system from different viewpoints and on different levels of abstraction. A second contribution of this paper is *rule-based* inconsistency resolution, which enables reuse of different inconsistency resolutions across model refactorings and manages the flow of inconsistency resolution steps automatically.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Object-oriented design methods*; D.1 [**Programing Techniques**]: Miscellaneous; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic; I.2.4 [**Artificial Intelligence**]: Knowledge Representation Formalisms and Methods

Keywords

Model Refactoring, Inconsistency Management, Description Logics, Rule-Based Systems

1. INTRODUCTION

Model-driven engineering is an approach to software engineering where the primary focus is on models, as opposed to source code. Model transformation is considered to be the heart and soul of model-driven engineering [18].

The inherent complexity of (design) models increases as more software engineering activities move to the modelling level. According to Lehman's second law of software evolution: "As a program is evolved its complexity increases unless work is done to maintain or reduce it" [11]. To counter this growing complexity, we need *model refactorings* [1, 3, 21]. Model refactorings are model transformations that restructure the models in order to improve some quality attributes of the models but preserve the behaviour. They are the design level equivalent of source code refactorings [7]. In this paper, we will focus on the *application of model refactorings* [13].

The activity of applying a model refactoring consists of changing a particular configuration of model elements. This usually brings the model in an inconsistent state, which needs to be resolved subsequently. A first insight and contribution of this paper is that applying most model refactorings boils down to detecting and resolving a series of inconsistencies. *Inconsistency management* is well-studied and well-known domain within software engineering. In [24], the first author *et al.* present a classification of different inconsistencies in the context of UML and a formalism for detecting inconsistencies.

A second insight and contribution of this paper is that reuse of inconsistency resolutions in and across model refactorings is necessary and can be achieved by *rule-based inconsistency resolution*. In order to validate this and the above claim, we analyse a representative set of model refactorings and show that (1) executing most model refactorings consists indeed of resolving a number of inconsistencies and (2) concrete resolutions of inconsistencies are reused in and across model refactorings.

This paper is structured as follows. Section 2 introduces model refactorings, inconsistencies and the relationship between both concepts. Section 3 argues the necessity of a rule-based inconsistency resolution approach and introduces our particular approach and tool support. In Section 4 we provide a validation and discussion of our approach. Section 5 discusses additional issues and future work. Section 6 presents related work and Section 7 concludes this paper.

2. MODEL REFACTORINGS AND INCONSISTENCIES

First, we introduce model refactorings through source code refactorings and their relation to inconsistencies and secondly, we introduce one refactoring in depth.

2.1 Source Code versus Model Refactoring

Some model refactorings are inspired by source code refac-

SAC'06 April 23-27, 2006, Dijon, France

torings [7], others are specifically defined from the model point of view (e.g., state machine refactorings) [3]. In contrast to model refactorings, source code refactorings have been thoroughly studied in literature [7, 16].

Source code refactorings are defined on certain source code elements which are chosen by the developer. Each source code refactoring description in [7] contains a *mechanics* section. That section gives a step-by-step description on how to carry out the refactoring. Each step consists of a certain activity, for example, *copy the body of a certain method*. In most cases, this step is followed by a description of corrections necessary to make the code compile. These corrections not only affect the newly created elements but also existing elements. For each application of a refactoring, these corrections can be different. Consequently, although a refactoring is defined on only those elements indicated by the user, it can affect different elements.

The activity of how to carry out a *model refactoring* can also be described step by step. Similar to the execution of source code refactorings, a certain step can affect several elements in the model and introduce *inconsistencies*. The inconsistencies resulting from the execution of a certain refactoring step need to be resolved and (similar to corrections in source code refactorings) inconsistency resolutions can affect not only the newly created elements but also existing ones.

Different techniques exist for helping the user to resolve inconsistencies (for a detailed overview, see [20]). In our work, we focus on *resolution actions* that modify models in order to resolve inconsistencies. We use the term *inconsistency resolution* to indicate a set of resolution actions that resolve a certain inconsistency. We do not consider other techniques such as enforcing of consistency of models at any time or preserving consistency as much as possible. Such techniques are considered to be unrealistic in a real-world project on which several developers are working at the same time [14].

In this paper, we show that **most model refactorings are executed through the resolution of inconsistencies**.

In the remainder of this section, we will give an in-depth description of the execution of *Move Operation*. We have chosen this particular refactoring because, although it is conceptually a small refactoring, it illustrates very well how inconsistency resolutions can be used for the support of the execution of most model refactorings.

2.2 Executing *Move Operation*

In Figure 1, the decision diagram and the different activities of the *Move Operation* refactoring are shown. Activity diagrams as defined in the UML are used to specify the decision diagram of a model refactoring. The actions representing a refactoring step in the diagram are marked by a *S*. We will now describe step by step the execution of this model refactoring similar to the description of source code refactorings by Fowler [7].

Declare the operation in the target class
A possible inconsistency that can occur after this step is that the operation's name and signature already exist in the target class. However as already mentioned in the introduction, our inconsistencies are classified in the context of UML. In this classification we do not consider well-formedness rules. As a result, no inconsistencies are checked after this step.

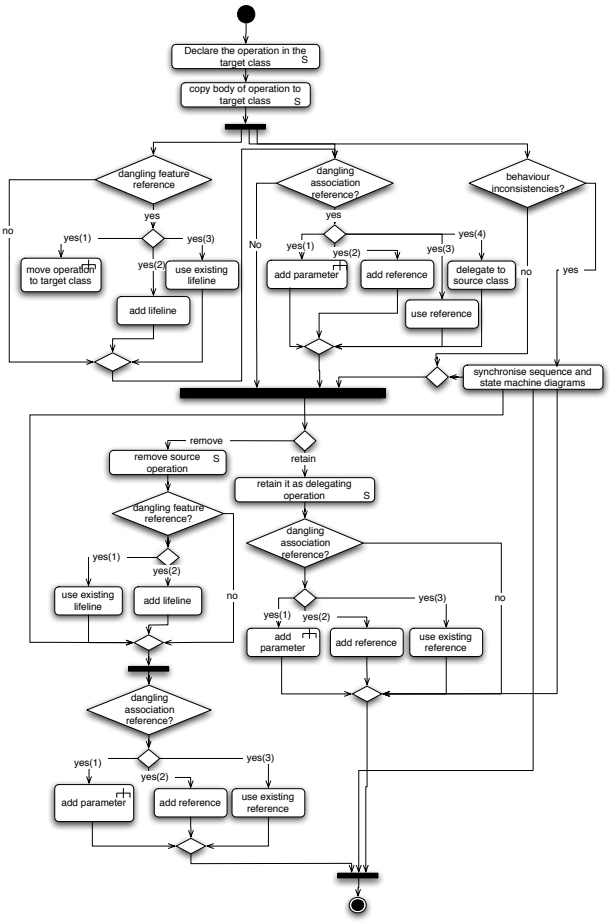

Figure 1: Decision activities for *Move Operation*.

Copy the body of this operation to its new target
In this step, answers to the question *how to reference the source object from the target class* and the question *how to reference the referenced objects in the body of the operation from the target class* need to be given. Support for answering these questions can be delivered by an inconsistency resolution approach at the model level. Possible answers to the first question are: (1) move the operation to the target class as well, in this case the model refactoring *Move Operation* is executed first for the operation in question; (2) create or use a reference from the target class to the source, (3) pass the source object as a parameter to the operation, in this case the model refactoring *Add Parameter* (similar to the *Add Parameter* source code refactoring [7]) is executed first. Different solutions are possible for when objects are referenced in the body of the operation that are not known to the target class: (1) an explicit reference can be added to the class in question; (2) a parameter can be added to the operation; (3) as the objects are known to the source class, operation calls can be sent to the source class, but this involves the addition of different operations in the source class, which are delegating operations. Figure 1 shows the different inconsistencies and possible resolutions. For an explanation of the different inconsistencies, see [22].

Behavioural inconsistencies [22] can also occur in this step,

e.g., when the sequence of messages received by an object of a certain class in a particular scenario does not conform with the specification of the behaviour for this particular class. These inconsistencies can be resolved in this step, or after the last step in the execution of this refactoring.

Remove the operation in the source class or retain it as a delegating operation

If the operation is *removed from the source class*, an inconsistency can occur in the scenarios where the operation is sent to an object of the source class while this operation is not known anymore by the source class or in the specification of the behaviour of the source class. In this case, the receiver of the operation must be changed to the correct object that is an instance of the target class or such an object must be created. This resolution can cause another inconsistency if the target class cannot be reached from the object invoking the operation or if the relationship on which the message is sent is not known between the objects. This inconsistency can be resolved by either creating a reference or using an existing one, or adding a parameter to the operation in question.

In case the operation is *turned into a delegating method*, the question *how to reference the correct target object from the source object* needs to be answered. Solving this inconsistency boils down to an inconsistency resolution. Possible resolutions are: (1) reference the correct target object through the creation of a new reference, or (2) use an existing reference or add a parameter to the operation. If there are still inheritance inconsistencies or incompatible behaviour, these inconsistencies can be resolved at the end of this step.

3. INCONSISTENCY DETECTION AND RESOLUTION

3.1 Inconsistency Detection

In [24], the first author *et al.* present a classification of inconsistencies. To be able to define and detect these classified inconsistencies in a precise yet executable way, we need a formal specification and a formal reasoning engine relying on this specification. The formalism used and introduced for this purpose in [24] is Description Logics (DLs) [2]. DLs are a family of formalisms to represent knowledge. They are less expressive than first-order logic but have more specific reasoning abilities. DLs represent knowledge by defining *concepts*, *roles* and *individuals*. Individuals represent instances of the defined concepts and can be related to each other by the defined roles.

The UML is currently the most widely used object-oriented modelling language. The visual representation of the UML consists of a set of different diagram types. The different diagram types describe different aspects of a software system. In our work, *class diagrams*, *sequence diagrams* and *protocol state machine* diagrams of the UML version 2.0 [15] are considered. The discussion in this paper however, can be applied to any object-oriented modelling language.

The UML metamodel is translated into DL concepts and roles. There is a one-to-one mapping between UML metaclasses and concepts and between UML meta-associations and roles. User-defined models are interpreted as instances of the UML metamodel and are asserted in a factbase as instances of a concept representing a certain metaclass and related to each other by roles, representing meta-associations. The state-of-the-art DL system RACER [9] is used in our approach for the representation and reasoning over factbases representing UML models, because this system has the most extensive reasoning abilities, an extensive query language and rule system. The syntax for asserting an individual in this DL system, is `(instance` In C`)` declaring In as being an individual of the concept C. For example, `(instance session class)` specifies that the individual `session` is an instance of the concept `class`. The syntax for asserting that the individuals $In1$ and $In2$ are related by the role R is `(related` $In1$ $In2$ R`)`. For example, the assertion `(related session getcustomerspecifics ownedoperation)` relates the individual `session` with `getcustomerspecifics` through the role `ownedoperation`.

Inconsistencies are detected by querying the assertions that represent the model. This is expressed in the query language of RACER, *nRQL*. It is possible to group queries together in one "super" query. In [24], the first author *et al.* present queries for detecting the classified inconsistencies.

3.2 Rule-Based Inconsistency Resolution

Section 2 has illustrated that executing a model refactoring boils down to resolving a number of possibly recurring inconsistencies. The decisions that have to be taken in order to execute the *Move Operation* model refactoring are depicted in the activity diagram in Figure 1. However, even for a model refactoring as moderately complex as this one, this diagram is non-trivial. Indeed, every possible situation in the model to be refactored leads to a potentially different flow of inconsistency resolutions. Moreover, the same inconsistency resolution can occur multiple times in different combinations with other inconsistency resolutions. Although the previous section only illustrated one model refactoring, we observe that the same inconsistencies occur in multiple model refactorings. As such, a first important criterion when resolving inconsistencies as part of model refactorings is **reuse of inconsistency resolutions in and across model refactorings**.

Each detected inconsistency can be resolved in several different ways. The selection of a particular resolution for an inconsistency can depend on the particular state of the model. For the most part, however, selecting a resolution is a matter of preference of the person who is executing the model refactoring. Therefore, we require choice points in the flow of inconsistency resolutions where the refactorer is able to communicate his or her preference for a particular resolution. This choice affects the subsequent flow of inconsistency resolutions. As such, a second criterion we aim to address is **support for user-guided selection of inconsistency resolutions**.

In this paper we argue that a rule-based approach to resolving inconsistencies meets the two criteria specified above. In the following we introduce our approach to rule-based inconsistency resolution specifically. In Section 4 we validate and discuss this claim.

3.2.1 Inconsistency Resolution Rules

A generic inconsistency resolution rule has the form: `IF inconsistency X occurs in model M THEN change model M so that X is resolved`. There are typically multiple resolutions for a particular inconsistency and each one is represented by one rule. Hence, all rules pertaining to a certain

inconsistency `X` have the same expression `inconsistency X occurs in model M` in their conditions. The occurence of an inconsistency in a model is detected by querying the data representing the model, i.e., the model elements. A certain state of the model attests to the presence of a particular inconsistency.

A rule's conclusion states how to resolve the detected inconsistency. It consists of a sequence of statements, where each statement is responsible for either adding data to the model or removing data from the model. As such, the model elements are rearranged so that the inconsistency is resolved. However, in order for a certain inconsistency resolution to be applicable, some model elements typically need to be present or in a particular configuration. Therefore, this is also checked in the condition of the rule, after checking the occurrence of the inconsistency.

In our approach, a set of inconsistency resolution rules corresponding to an inconsistency is by no means unique or complete. The rules embody certain strategies for resolving inconsistencies, other or more strategies can typically be devised. Therefore, there is no formal verification of the completeness and soundness of the rules foreseen.

The rule below represents a possible resolution of an inconsistency occurring in the *Move Operation* model refactoring. In particular, this rule ensures the creation of a new association from the target class to the source class and an instance of this association being used in the sequence diagram. The rule is expressed in the rule-based system of RACER.

```
1. (firerule
2.   (and (check-DAR ?class ?op ?m)
3.        (?m ?con connectorr)
4.        (?m ?mend sendEvent)
5.        (?mend ?lifeline coveredsub)
6.        (?lifeline ?connectableel represents)
7.        (?connectableel ?class2 base)
8.        (?u ?assocname user-option-addAssoc))
9.   ((related (new-ind assoc ?assocname) ?assocname name)
10.    (related (new-ind assoc ?assocname)
11.             (new-ind end ?class) memberend)
12.    (related (new-ind assoc ?assocname)
13.             (new-ind end ?class2) memberend)
14.    (related ?class (new-ind end ?class) ownedattribute)
15.    (related ?class2 (new-ind end ?class2) ownedattribute)
16.    (related (new-ind connector ?assocname)
17.             (new-ind assoc ?assocname) associationtype)
18.    (related ?m (new-ind connector ?assocname) connectorr)
19.    (forget-role-assertion ?m ?con connectorr)))
```

The entire conjunction is the rule's condition (lines 2-8), whereas the rest (lines 9-19) is the rule's conclusion. The first element of the condition (line 2) is a *nRQL* query for detecting the inconsistency (see Section 3.1). Rule variables are preceded by a question mark. Noteworthy in this rule is the last element of the condition (line 8), used for asking the user for a name for the association that is going to be created in the rule's conclusion, expressed by `(new-ind assoc ?assocname)`. Furthermore, new association ends are created and related to the new association and to the correct classes (lines 10-15). Also a new connector, representing a link in a sequence diagram on which a message is sent, is created and related to the newly created association (lines 16-18). The old connector is removed from the original message (line 19). The rule is executed for each set of bindings of the variables, in other words, for each detected instance of the inconsistency.

3.2.2 Rule Engine

In our approach we employ a forward-chaining rule engine since the engine has to be activated when model elements change. As a result of firing a rule that resolves an inconsistency, the model is changed anew, which again ensures that the rule engine looks for rules that are applicable in the new situation. One of the criteria we established earlier in this section is that selecting one of the applicable inconsistency resolutions is up to the developer. Therefore, after the rule engine has determined the applicable rules, the developer is required to select the preferred resolution of an inconsistency. The corresponding rule is subsequently actually fired. Note that resolving an inconsistency might have as a side effect that another previously detected inconsistency is resolved. After each inconsistency resolution, the applicable inconsistency resolutions are updated by the rule engine.

3.3 Inconsistency Management Tool

We set up a proof-of-concept inconsistency management tool, called *RACOoN*[1] that is plugged into Poseidon [8], which is a state-of-the-art UML CASE tool. RACER is used as the underlying reasoning system. *RACOoN* allows for its configuration and for importing the DL translation of the UML metamodel into RACER. Users can choose and execute the detection of particular inconsistencies on UML models. User-defined models are translated into DL assertions and loaded into RACER. Different rules are implemented and loaded in *RACOoN*. Using these rules, refactorings can be executed.

Different model refactorings are implemented in the environment. Figure 2 shows the refactoring manager pane of our tool incorporated in Poseidon. The user can select a refactoring, e.g., the *Move Operation*. After executing the first two steps of this refactoring, a rule engine is called and different rule sets are fired. If there are inconsistencies in the model(s) under consideration, different resolutions are presented to the user.

The user can select a certain resolution as shown in Figure 2. If user input is necessary for the execution of this resolution, such as the name of a model element that is to be created, the necessary information is requested from the user. After the execution of this resolution, the cycle begins anew and the rule engine again looks for applicable rules. If none is found, the tool moves on to the next step in the refactoring.

4. VALIDATION AND DISCUSSION

4.1 Model Refactorings and Inconsistencies

In this section, we motivate our first claim (Section 2) by means of a set of analysed refactorings. We considered one or two representative refactorings of each category defined by Fowler [7] (see Table 1). Several reasons for considering this particular set of refactorings can be stated: (1) the refactorings defined by Fowler are well-known and some of these refactorings have already been defined at design level [21]; (2) the corresponding model refactorings are model transformations that transform not only the specification of the static structure as suggested by the UML drawings used in Fowler. These refactorings affect also the specification of

[1] *RACOoN* stands for *Resolution Actions for inCONsistencies*

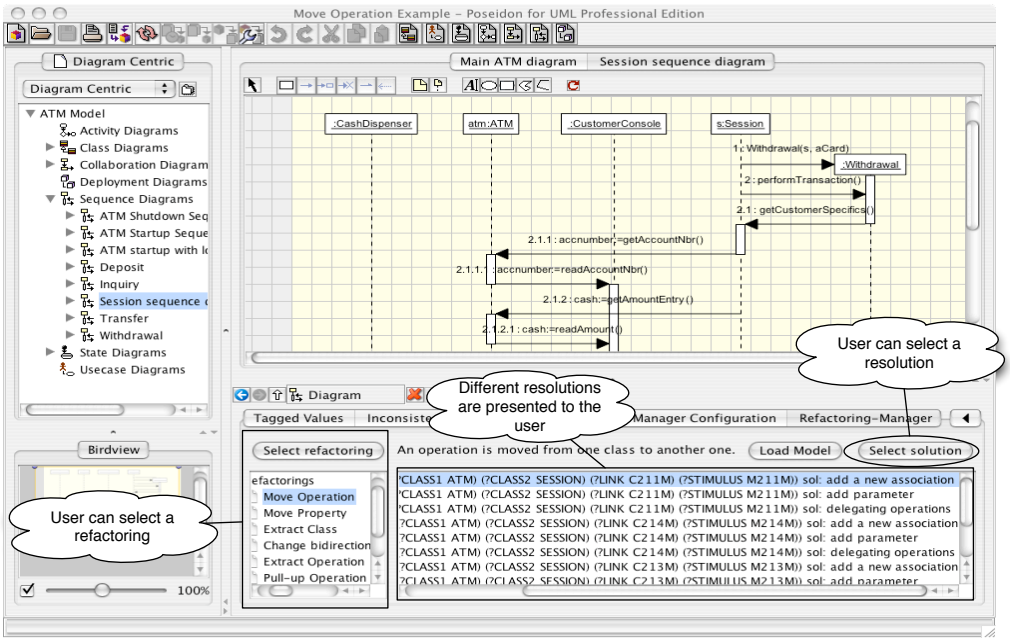

Figure 2: Screenshot of *RACOoN* presenting different resolutions to the user.

the behaviour of the system under study; (3) because the source code refactorings defined in Fowler are well-known, it is obvious to consider their corresponding model refactorings in view of exploring the link between these refactorings and (generated) source code in the larger context of MDE.

Table 1: Analysis of relation between model refactorings and inconsistencies.

	inheritance	incompatibel declaration	incompatible behaviour	dangling type reference	inherited association reference	instance declaration missing	disconnected model
Add Parameter	×		×	×		×	
Change Bidirectional Association to Unidirectional		×	×			×	
Extract Class	×	×	×		×	×	
Move Property		×			×	×	
Move Operation	×	×	×		×	×	
Pull up Operation					×	×	
Push down Operation			×		×	×	
Extract Operation	×		×				
Replace Conditional with Polymorphism	×		×		×	×	

Table 1 summarises which inconsistencies can be detected and resolved to support a certain model refactoring. The rows contain the different refactorings we (re)designed at the model level so far. A *X* in a cell of the table indicates that the resolutions of the corresponding inconsistency can be used to support the execution of the corresponding refactoring. This table indicates that executing the studied model refactorings consists indeed of resolving inconsistencies and that the same inconsistencies occur in different model refactorings.

Some refactorings however, can be executed without this process of inconsistency detection and resolution. Consider a refactoring where a class is inserted in a hierarchy. In a first step, the new class is created. This does not introduce any inconsistencies. A second step is to add a generalisation relationship between this new class and the superclass. This also does not introduce any inconsistencies. A third step is to add a generalisation relationship between the subclass and the new class. Finally, the generalisation relationship between the superclass and subclass can be removed without causing any inconsistencies.

4.2 Rule-Based Inconsistency Resolution

We review the criteria we identified in Section 3:

Reuse of inconsistency resolutions in and across model refactorings. As introduced earlier in Section 3, an inconsistency resolution encompasses the detection of a particular inconsistency and one possible way of resolving it. Rule-based systems provide an explicit *rule* construct, which we use for representing inconsistency resolutions. This means that the detection and resolution parts of an inconsistency resolution are encapsulated. A rule engine treats rules as stand-alone modules.

Rule-based systems dramatically boost reuse of rules, since they only have to be defined once and the rule engine fires the ones appropriate for a certain situation. As such, the

rule engine constructs *implicit* flows of rules or, in our case, inconsistency resolutions.

Table 2: Analysis of reuse of inconsistency resolutions in and across model refactorings.

		Add Parameter	Change Bidirectional Association to Unidirectional	Extract Class	Move Property	Move Operation	Pull up Operation	Push down Operation	Extract Operation	Replace Conditional with Polymorphism
incompatible declaration	S1		1							
	S2		1							
dangling type reference	S3	1								
inherited association reference	S4				1	2	1	1		2
	S5				1	2	1	1		2
instance declaration missing	S6		1						1	1
	S7			1		2				2
	S8					2				2
	S9					2		1		2
	S10				1	2	1	1		2
	S11				1	2	1			2
	S12				1	2				2
	S13				1	2				2
	S14			1	1					

S7→Move Operation; S13→Add Parameter; S14→Move Property;

We now show that reuse of inconsistency resolutions such as described above is crucial in the context of model refactorings, and can be addressed by a rule-based approach. We present an analysis of the relation between several model refactorings and inconsistency resolutions. The results are shown in Table 2. This table presents the same 9 model refactorings as before (colums), but this time in relation to the concrete inconsistency resolutions (rows) that we have defined for supporting the execution of these model refactorings. We discovered 14 different resolutions for 4 inconsistency categories. The cells indicate if an inconsistency resolution has been employed to execute a model refactoring. In some cases a particular resolution can occur twice in the same model refactoring, as is also illustrated in the *Move Operation* model refactoring presented in Section 2. It is clear from the table that there is significant reuse of resolutions in and across model refactorings. Resolutions S1 and S2 are reused once. Resolution S6 is reused 3 times. The number of times an inconsistency resolution is reused is not merely the sum of the numbers of a certain row. The sum of these numbers is a lower bound. S7, S13 and S14 cause the execution of the model refactoring, *Move Operation*, resp., *Add Parameter* and, resp., *Move Property*. As a consequence, S3, for example, is reused 6 times.

Note that the inconsistency resolutions belong to structural inconsistency categories [24]. In this paper we have omitted the behavioural inconsistency resolutions that occur in the analysed model refactorings since it gives similar results.

Support for user-guided selection of inconsistency resolutions. A very important characteristic of a rule-based system is that the *definition* of a rule is separated from the *method* employed by the rule engine for selecting the rules to be fired from the applicable ones. In our context, this method is user-guided selection of a particular inconsistency resolution. As such, our rule-based approach to inconsistency resolution provides a mix between automation and user input: the rule engine automatically finds all applicable rules in each situation and as such automatically constructs an implicit flow of inconsistency resolutions, whereas the developer is able to select his or her preferred resolution out of the applicable ones. The developer also has the freedom to leave certain inconsistencies unresolved.

5. ISSUES AND FUTURE WORK

A first issue is that the resolutions implemented in our approach can be too *fine-grained*. Depending on the application and system, it must be possible to group different resolutions or to define new resolutions. This can be realised by extending our tool support with a rule editor allowing the addition, removal and grouping of resolutions and allowing to enable or disable certain resolutions for certain refactorings.

A next issue is a more optimal ordering of the different rules. Different rules can be fired at the same time, user interaction is used in our approach to decide which inconsistency must be resolved first. However, heuristics and algorithms can be developed to order the different rules. As a result of the resolution of a certain inconsistency, it can be that other inconsistencies are resolved as well. This can happen in our approach as a side-effect of choosing a particular resolution. To be able to discover the resolutions of inconsistencies that will resolve other inconsistencies, the dependencies between the inconsistencies need to be determined. This determination is future work.

Even with a more optimal ordering of the rules, it is still possible that a lot of inconsistencies are detected and for each inconsistency, a lot of resolutions are possible. The question is how to manage all these inconsistency occurrences and their resolutions. Several possibilities can be investigated. One possibility is to use learning techniques where the selection of particular resolutions by the user in a particular context is learned by the resolution approach. Another possibility is to use predefined resolution alternatives defined by the user. However, we believe none of these approaches enables the automatic resolution of inconsistencies and we even believe that full automatic inconsistency resolution is not desirable. For example, if a new model element needs to be added, the user has to specify its name. A computer-generated name can be used, but in that case, the user still has to edit the model if he/she does not agree on the name.

A final issue is the correctness of a refactoring. A refactoring is *correct* if it terminates, the model is syntactically correct and some behavioural properties of the model are preserved [17]. Due to the fact that resolutions can introduce new inconsistencies, a rule can be fired more than once. This can introduce cycles leading to an infinite chain of resolutions. In our tool support, the firing of the same rule on the same data is allowed only once preventing such cycles.

Formal techniques can be used to prove termination of the rules. This is outside the scope of this paper. The syntactical correctness of the model is guaranteed because of the usage of DLs. As already explained, the UML metamodel is translated into concepts and roles, called definitions, and the user-defined models are translated into assertions which can be checked for conformance with the definitions using a standard DL reasoning task. A last issue of correctness is the preservation of some behavioural properties of the model by the model refactoring. Not a lot of work has been presented in this context. In [5], transformation rules are defined in the context of model evolution. However, these rules are not to be interpreted as rules in a rule-based system. These rules have another purpose than our inconstency resolution rules. Some behavioural properties of source-code refactorings have been defined in [12] and graph transformations have been defined to support them. These properties are quite general and one of them is refined into several possible behavioural properties and redefined on model level in the work of the first author *et al.* [23]. In the work presented in this paper, only particular resolution rules can be triggered after a specific inconsistency in a certain step in the refactoring. As a result the behaviour of the application is preserved. Note that we did not proof this formally.

Until now, we only carried out experiments on small examples. It would be useful to validate our approach on some large-scale industrial cases. This validation would provide us with some empirical data such as the most used resolutions, the most used model refactorings, the domain-dependent or company-dependent resolutions. This information is very useful for improving the usability of our tool support.

6. RELATED WORK

Inconsistency management is well-known in software engineering. However, resolving inconsistencies is a difficult research problem. Different techniques have been developed coping with this problem. Synoptic is a technique developed by Easterbrook [4] in which stakeholders are expected to define and select resolution actions. In [19] a *reconciliation method* is developed which uses distance metrics to indicate the type and extent of inconsistencies. Based on these distances, actions are generated and proposed to the users that can partially resolve the different types of model inconsistencies. van Lamsweerde *et al.* [25] developed the technique called KAOS that uses divergence resolution patterns but only specific kinds of such divergences can be handled. In these approaches, either the stakeholder gets a lot of responsibility by defining the possible resolution actions, or the set of actions is restricted to a specific domain such as requirements. These approaches also do not take into account that resolution actions also can introduce other inconsistencies.

In [6], inconsistency handling between *Viewpoints*, locally managed software models, is presented. Viewpoints and inter-Viewpoint rules are all translated to first-order predicate logic, and inconsistencies are identified using the *Closed World Assumption*. A meta-language based on first-order temporal logic, which uses a set of meta-level axioms, is employed for defining inconsistency handling rules. The main difference with our approach is that the rules give the user a likely explanation for the occurence of an inconsistency, such as *typographical error* or *conflict between specification*. Kozlenkov *et al.* [10] use abductive reasoning for establishing a user-defined goal consisting of sequences of conditions required for the goal to be achieved. Our approach uses deduction rather than abduction for the detection of inconsistencies. For inconsistency resolution Kozlenkov *et al.* use Prolog rules that are only used as a querying mechanism on assertions as in our approach.

Research on model refactoring is emerging. A set of basic UML refactorings is provided in Sunyé *et al.* [21] to improve the software design in a stepwise fashion. That work redefines a few source code refactorings on UML diagrams. Very strict preconditions are specified that need to be satisfied before the refactoring can be executed. As a result, no inconsistencies occur during the execution of these refactorings. The disadvantage of this approach is that model refactorings can only be executed on particular designs, obeying very strict conditions. A model violating these preconditions, can be changed so that it enforces these conditions. Our inconsistency resolution approach can be used in the process of changing the model.

Model refactorings are defined in [17] as a sequence of transformation rules. Reusability of refactoring steps across different refactorings is not considered, whereas in our work inconsistency resolutions are decidedly reused, not only conceptually but also their actual definitions as rules. Moreover, the transformation rules are executed in the order they are defined and there is no rule engine that chains rules, i.e. (re)activating rules when data is changed or created.

To the best of our knowledge, tool support for model refactorings is only discussed in [3], [1] and [26]. Boger *et al.* [3] show how model refactorings can be integrated in the Poseidon UML refactoring browser. However, this plug-in is not available anymore for Poseidon. Astels [1] uses a UML tool to perform source-code refactorings more easily, and also to aid in code smell detection. However, possible inconsistencies or problems are left for detection by the source-code compiler. A *Move Method* e.g., is done by just dragging the method in the corresponding UML class diagram to the target class. The work on generic and domain-specific model refactoring using a model transformation engine [26] has some similarities with our work. First of all, they also provide a set of predefined model refactoring rules and allow new rules to be defined since they also have a language for describing refactorings. Another similarity is that model refactoring rules consist of a precondition and a strategy. The former tests if certain model elements fulfil certain conditions and the latter are made up of operations that add or remove basic model elements. However, there are also a number of important differences between the approaches. The most important difference is that our model refactorings automatically detect inconsistencies in a model and hence also the particular model elements that cause the inconsistency. Each set of model elements that are thus detected are treated by the model refactoring rules in order to resolve the inconsistency. In the aforementioned approach, however, the required model elements are selected manually and passed as parameters to the model refactoring strategy. As such, these model refactorings are actually more like functions that are called with actual parameters, than like rules that are triggered when certain elements match with the (pre)condition. An important side effect is that there is no automatic construction of a flow of the model refactoring rules, as is the case in our approach.

7. CONCLUSION

In this paper model refactorings based on source code refactorings are introduced. We elaborate on the *Move Operation* model refactoring and show that in order to execute these refactorings, a chain of inconsistency detection and resolution steps is actually performed. We argue that manually determining inconsistency resolution scenarios that correspond to all possible situations is a daunting and unmanageable task. The identified problems are exactly those that are addressed by rule-based systems.

A rule-based inconsistency resolution approach enables *reuse of inconsistency resolutions in and across model refactorings* and *support for user-guided selection of inconsistency resolutions*. This is exemplified by our particular approach using the DL formalism and by demonstrating our proof-of-concept inconsistency management tool support including model refactorings.

8. REFERENCES

[1] D. Astels. Refactoring with UML. In *Proc. Int'l Conf. eXtreme Programming and Flexible Processes in Software Engineering*, pages 67–70, Alghero, Sardinia, Italy, 2002.

[2] F. Baader, D. McGuinness, D. Nardi, and P. Patel-Schneider. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, 2003.

[3] M. Boger, T. Sturm, and P. Fragemann. Refactoring browser for UML. In *Proc. Int'l Conf. eXtreme Programming and Flexible Processes in Software Engineering*, pages 77–81, Alghero, Sardinia, Italy, 2002.

[4] S. Easterbrook. Handling conflict between domain descriptions with computer-supported negotiation. *Knowledge Acquisition*, 3(3):255–289, 1991.

[5] G. Engels, R. Heckel, J. M. Küster, and L. Groenewegen. Consistency-preserving model evolution through transformations. In J.-M. Jézéquel, H. Hußmann, and S. Cook, editors, *Proc. Int'l Conf. UML 2002 - The Unified Modeling Language.*, number 2460 in LNCS, pages 212–227, Dresden, Germany, October 2002. springer.

[6] A. Finkelstein, D. M. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh. Inconsistency handling in multi-perspective specifications. In *European Software Engineering Conference*, pages 84–99. springer, 1993.

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

[8] Gentleware. Poseidon, http://www.gentleware.com/products/poseidonpe.php3, March 18 2004.

[9] V. Haarslev and R. Möller. RACER system description. In *Int'l Joint Conf. Automated Reasoning (IJCAR 2001)*, 2001.

[10] A. Kozlenkov and A. Zisman. Discovering, recording, and handling inconsistencies in software specifications. *International Journal of Computer and Information Science*, 5(2), June 2004.

[11] M. M. Lehman, J. F. Ramil, P. Wernick, D. E. Perry, and W. M. Turski. Metrics and laws of software evolution - the nineties view. In *Proc. Int'l Symposium Software Metrics*, pages 20–32. IEEE Computer Society Press, 1997.

[12] T. Mens, S. Demeyer, and D. Janssens. Formalising behaviour preserving program transformations. In *Proceedings of the First International Conference on Graph Transformation*, pages 286–301. springer, 2002.

[13] T. Mens and T. Tourwé. A survey of software refactoring. *Trans. Software Engineering*, 30(2):126–139, February 2004.

[14] B. Nuseibeh, S. Easterbrook, and A. Russo. Leveraging inconsistency in software development. *IEEE Computer*, 33(4):24–29, 2000.

[15] Object Management Group. Unified Modeling Language 2.0 Superstructure Draft Adopted Specification. ptc/03-08-02, February 2005.

[16] W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, Urbana-Champaign, IL, USA, 1992.

[17] I. Porres. Model refactorings as rule-based update transformations. In P. Stevens, J. Whittle, and G. Booch, editors, *Proc. Int'l Conf. UML 2003*, volume 2863 of *LNCS*, pages 159–. springer, 2003.

[18] S. Sendall and W. Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE Software*, 20(5):42–45, September/October 2003.

[19] G. Spanoudakis and A. Finkelstein. Reconciling requirements: a method for managing interference, inconsistency and conflict. *Ann. Softw. Eng.*, 3:433–457, 1997.

[20] G. Spanoudakis and A. Zisman. *Inconsistency Management in software engineering: Survey and open research issues*, volume 1, pages 329–380. World Scientific Pub. Co., 2001.

[21] G. Sunyé, D. Pollet, Y. LeTraon, and J.-M. Jézéquel. Refactoring UML models. In *Proc. UML 2001*, volume 2185 of *LNCS*, pages 134–138. springer, 2001.

[22] R. Van Der Straeten. *Inconsistency Management in Model-driven Engineering. An Approach using Description Logics*. PhD thesis, Vrije Universiteit Brussel, Brussels, Belgium, September 2005.

[23] R. Van Der Straeten, V. Jonckers, and T. Mens. Supporting model refactorings through behaviour inheritance consistencies. In T. Baar, A. Strohmeier, A. Moreira, and S. Mellor, editors, *UML 2004 - The Unified Modeling Language*, volume 3273 of *LNCS*, pages 305–319. springer, 2004.

[24] R. Van Der Straeten, T. Mens, J. Simmonds, and V. Jonckers. Using description logics to maintain consistency between UML models. In P. Stevens, J. Whittle, and G. Booch, editors, *UML 2003 - The Unified Modeling Language*, volume 2863 of *LNCS*, pages 326–340. springer, 2003.

[25] A. van Lamsweerde, E. Letier, and R. Darimont. Managing conflicts in goal-driven requirements engineering. *IEEE Trans. Softw. Eng.*, 24(11):908–926, 1998.

[26] J. Zhang, Y. Lin, and J. Gray. Generic and domain-specific model refactoring using a model transformation engine. In S. Beydeda, M. Book, and V. Gruhn, editors, *Model-driven Software Development*, chapter 9, pages 199–218. Springer, 2005.