From Well-Formedness to Meaning Preservation: Model Refactoring for Almost Free

Friedrich Steimann

Lehrgebiet Programmiersysteme
Fernuniversität in Hagen
D-58084 Hagen
steimann@acm.org

Abstract: Modelling languages such as the UML specify well-formedness as constraints on models. For the refactoring of a model to be correct, it must take these constraints into account and check that they are still satisfied after the refactoring has been performed — if not, execution of the refactoring must be refused. By replacing *constraint checking* with *constraint solving*, we show how the role of constraints can be lifted from permitting or denying a tentative refactoring to computing additional model changes required for the refactoring to be executable. Thus, to the degree that the semantics of a modelling language is specified using constraints, refactorings based on these constraints are guaranteed to be meaning preserving. To be able to exploit constraints available in the form of a language's well-formedness rules for refactoring, we present a mapping from these rules to the constraint rules required by constraint-based refactoring. Where there are no gaps between well-formedness and (static) semantics of a modelling language, these mappings enable structural refactorings of models at no extra cost; where there are, we identify ways of detecting and filling the gaps.

1 Introduction

Refactoring is the discipline of modifying a software artefact so as to improve one or more of its non-functional properties (such as readability, changeability, etc.) whilst maintaining its meaning. While originally conceived as program restructuring [7], refactoring is today applied to all kinds of software artefacts, including models [3, 6, 8, 12, 13, 14, 15, 23].

Constraint-based refactoring is a refactoring technique that relies on constraints for the specification of invariants that a refactoring must regard [18, 19, 24, 25]. In constraint-based refactoring, a refactoring problem is translated to a constraint satisfaction problem

(CSP) whose solutions represent legal refactorings of the artefact to be refactored. For this, the semantic rules of the language of the artefact must be transcribed to so-called *constraint rules* which, when applied to the artefact to be refactored, generate the constraints expressing all relevant invariants. With the exception of Ref. [20], the precursor to this article, constraint-based refactoring has so far exclusively been applied in programming, a field in which it has however proven highly successful (see, e.g., [18, 19, 24, 25]).

In modelling, constraints are commonly used to express invariants such as multiplicities etc. On the metalevel, at which modelling languages are specified, these invariants play the role of well-formedness rules, expressing parts of the static semantics of a language. Like invariants, well-formedness rules are evaluated by a constraint checker: if the constraints are satisfied, the model is well-formed and ok; if not, it is malformed and hence rejected.

In this article, we show how the well-formedness rules used in specifying the static semantics of a modelling language can be used for constraint-based refactoring of models expressed in that language, giving us model refactorings for almost free. Where well-formedness rules leave static semantics underspecified, we show how they can be systematically enhanced to fill the gap. The transformation of well-formedness rules to the constraint rules required for constraint-based refactoring faces several technical difficulties, including the adaptation to specific refactorings and the handling of the indirection established by path expressions, for which we present solutions. The viability of our approach is demonstrated by applying it to a number well-formedness rules taken from the UML standard.

The remainder of this article is organized as follows. In Section 2, we present an instructive example explaining the idea of constraint-based model refactoring based on two simple well-formedness rules expressed in first-order predicate logic. In Section 3, we recapitulate as much of the theory of constraint-based refactoring as is necessary to make this article self contained. Section 4 specifies how well-formedness rules are transformed to the constraint rules underlying constraint-based refactoring, taking the special problems of integrating the refactoring specification and dealing with path expressions into account. While Section 5 applies this procedure to a number of real examples from the UML standard expressed in OCL, the general limitations of our approach are discussed in Section 6. We conclude with a brief comparison of related work with ours.

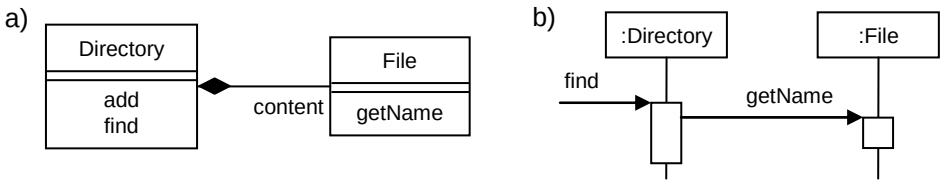

Figure 1. UML model of a flat file system: a) a class diagram and b) a sequence diagram.

2 Constraint-Based Model Refactoring by Example

Consider the simple model of a flat file system shown in Figure 1. Files are found in a directory by sending the latter a message named “find”, which in turn leads to a message “getName” being sent to a file. From an operating system perspective, this model is certainly overly simplistic; however, in the context of model refactoring, it will serve our purposes well.

2.1 Two Simple Well-Formedness Rules

In human-readable representations, model elements such as classes, attributes, or operations are usually identified by a (locally) unique name. For the operations of a classifier, for instance, this uniqueness constraint is expressed by the well-formedness rule

$$\forall\ o_1, o_2 \in \mathit{classifier.operations}, o_1 \neq o_2 : o_1.\mathit{name} \neq o_2.\mathit{name} \qquad (1)$$

which is based on the metamodel shown in Figure 2 and in which *classifier* is the classifier to be checked (the *context* in OCL terms). This well-formedness rule rejects all models in which any single classifier declares two or more operations of the same name. As can be easily verified, for the two classifiers of our example (Figure 1 a), Directory and File, this is not the case, so that it is well-formed with respect to well-formedness rule (1).

Another, slightly more involved well-formedness rule requires that only those elements be referenced (or used) that have been declared (or defined) elsewhere (a typing constraint). More specifically, and assuming that referencing is name-based, we have that for a sequence diagram such as that of Figure 1 b) to be well-formed, the names of the mes-

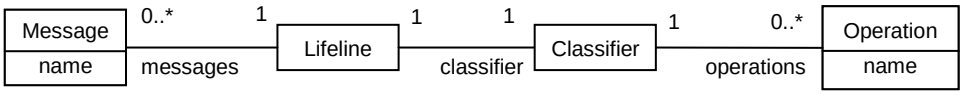

Figure 2: Common metamodel for the diagrams of Figure 1.

sages must correspond to names of operations defined by the classifiers associated with the lifelines of the objects to which the messages are sent.[1] This is expressed as

$$\forall l \in \mathit{lifelines}\ \forall m \in l.\mathit{messages}\ \exists o \in l.\mathit{classifier}.\mathit{operations} : m.\mathit{name} = o.\mathit{name} \quad (2)$$

which is based on the metamodel of Figure 2 and in which *lifelines*, the set of lifelines of the sequence diagram to be checked, is the context. The reader can easily verify that, with respect to well-formedness rule (2), the model of Figure 1 b) is also well-formed.

Admittedly, the typing rule expressed in (2) is somewhat simplistic in that it does not consider inheritance or subtyping; however, we do not delve into the technicalities necessary for this here, because they complicate matters unduly (and have been addressed in great detail elsewhere; see, e.g., [11, 24]).

2.2 Refactoring as Constraint Solving

Now suppose that the message labelled “getName” in the sequence diagram of Figure 1 b) is to be renamed, say to “x”. Re-evaluating well-formedness rule (2) immediately tells us that this renaming is not acceptable, since the rule now evaluates to *false*: for the message renamed to “x” and the classifier File, there exists no operation defined by File that has the same name “x”. Quite obviously, the problem can be fixed by renaming the operation labelled “getName” to “x” also. The question that remains, however, is, how can the renaming of the operation be automated?

As it turns out, the necessary secondary change, the renaming of the referenced operation, can be computed, simply by replacing constraint *checking* used for probing well-formedness (by way of evaluating the constraint expression to *true* or *false*) with con-

[1] For the sake of simplicity, we assume here that all operations of a classifier are defined in the same class diagram. In practice, different operations may be introduced in different class diagrams, showing different views on the model; yet, for a sequence diagram to be checkable for well-formedness, it cannot introduce the required operations itself.

straint *solving*, which can change the values of the constrained properties[2] of the model so that the constraints are satisfied (and the model is well-formed). For this, well-formedness rules such as (1) and (2) must be used to create from a model such as that of Figure 1 a constraint satisfaction problem (CSP), i.e., a set of constraint variables and constraints suitable for submission to a constraint solver. This is done by unrolling the quantifiers, instantiating the quantified variables with the elements of the model the rule is applied to. For the model of Figure 1 and well-formedness rule (1), it leads to the constraint set

$$\{o_{find}.name \neq o_{add}.name\} \quad (3)$$

in which o_{find} is an object literal denoting the operation originally named "find", $o_{find}.name$ denotes the *name* property of o_{find}, and so forth. This constraint prevents that the operation o_{find} or o_{add} (the only two operations of classifier Directory; note that classifier File has only a single operation $m_{getName}$, for which no constraint can be generated using (1)) is renamed so that their names equal (and the model becomes ill-formed). For well-formedness rule (2), the constraint set

$$\{m_{find}.name = o_{add}.name \vee m_{find}.name = o_{find}.name,\ m_{getName}.name = o_{getName}.name\} \quad (4)$$

is generated, making sure that for every message of the model an operation with the same name exists in the classifier to whose lifeline the message is sent. Based on these constraints, when the *name* property of $m_{getName}$ is set to "x", solving the CSP will force a change of the *name* property of $o_{getName}$ to "x" also, thereby making the model well-formed again. Since the model has also kept its original meaning (the message $m_{getName}$ binds to the same operation as before), it is a refactoring.

2.3 Removing Semantic Underspecification by Eliminating Existential Quantification

The previous refactoring is meaning preserving since it makes sure that the message $m_{getName}$ and the only available operation it could bind to, $o_{getName}$, are always renamed together. The situation is significantly different, however, if m_{find} is renamed, say to "y": in this case, renaming *either* of the two operations defined by the classifier Directory, o_{find} or o_{add}, to "y" also would *equally* satisfy the above constraint set, and thus restore well-

[2] We use the term *property* here to collectively denote attributes and association ends associated with an object ([10], §7.5.1 and §7.5.3). Conforming to [10], we use the dot notation *o.p* to denote the value of a property *p* of an object *o*, where *o* may be an object literal or a variable (including another property).

formedness of the diagram. However, *not both* possible renamings maintain the original meaning: if o_{add} is renamed to “y”, the model is well-formed with respect to (2), but has a different meaning, since now m_{find} binds to o_{add} and thus a different operation. The situation is even more dissatisfactory when m_{find} is renamed to “add”: in that case, the model remains well-formed without the renaming of an operator, but the binding (and thus the meaning) is definitely changed.

The problem is that the well-formedness rule (2) is insufficient for refactoring, since it is indifferent to the operation a method binds to. Given that well-formedness rules such as (2) and (1) are about well-formedness, not about meaning, this is not surprising: they express two necessary conditions for the binding of a message to an operation, namely that an operation with the same name as the message is declared in the classifier classifying the receiver of the message, and that this name is unique. They do not express which operation a method binds to, or that the notion of binding does at all exist — this is not required for well-formedness.

As it turns out, the semantic underspecification in (2) can be removed by replacing the existentially quantified variable *o* with a Skolem function [16]

$$binding: Lifeline \times Message \rightarrow Operation \tag{5}$$

mapping a lifeline and a message to the operation the method (should) bind to. Since the lifeline is functionally dependent on the message (meaning that for any given message, a lifeline is uniquely determined; cf. Figure 2), (5) can be projected to

$$binding: Message \rightarrow Operation \tag{6}$$

which lets us Skolemize (2) to

$$\begin{aligned} &\forall\, l \in lifelines \quad \forall\, m \in l.messages : \\ &\quad binding(m) \in l.classifier.operations \wedge m.name = binding(m).name \end{aligned} \tag{7}$$

Note how the first conjunct replaces for the range restriction of the existentially quantified variable *o* (representing the operation): the Skolem function *binding* must be chosen so that the operation a message *m* binds to is among the operations defined by the classifier associated with the lifeline *l* of the object to which *m* is sent.

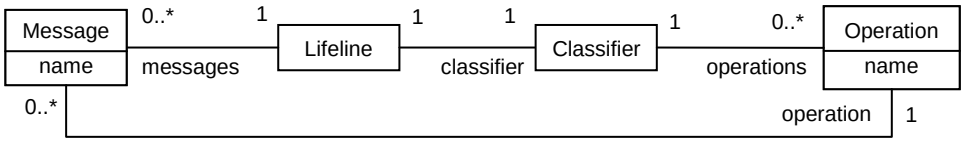

Figure 3. Metamodel of Figure 2 extended by an association between messages and the operations they bind to.

In modelling, the introduction of the binding (or lookup) function (6) corresponds to adding a metamodel association between Message and Operation as shown in Figure 3. Replacing the Skolem function *binding* with this added association allows us to rewrite (7) to

$$\forall\, l \in lifelines \quad \forall\, m \in l.messages :$$
$$m.operation \in l.classifier.operations \wedge m.name = m.operation.name \tag{8}$$

which expresses that the operation a message *m* binds to (represented by *m.operation*) must be among the operations defined by the classifier associated with the lifeline *m* is sent to, and must have the same name as *m*. Note that it was not specified before that *m* should bind to an operation with these properties: if anything, it reflected our intuitive interpretation of sequence diagrams.

Application of (8) to the model of Figure 1 gives us the constraint set

$$\{m_{find}.operation \in l_{left}.classifier.operations \wedge m_{find}.name = m_{find}.operation.name, \tag{9}$$
$$m_{getName}.operation \in l_{right}.classifier.operations \wedge m_{getName}.name = m_{getName}.operation.name\}$$

in which l_{left} and l_{right} denote the left and right lifeline, respectively. (9) lacks the disjunction of (4), removing the ambiguity which operation is to be renamed along with m_{find}. Note that, should m_{find}, and thus also o_{find}, be renamed to "add", the constraint (3) generated from well-formedness rule (1) requires that o_{add} is renamed to a different name, maintaining the unambiguousness. Any existing references to o_{add} would then be renamed as well (via other constraints generated from well-formedness rule (2); none in our example). As can be seen from this simple example, our constraint-based capture of renaming is comprehensive.

There is however a price associated with removing the existential quantification, namely the introduction of the indirection involved in the path expression *m.operation.name*. In our current RENAME example, in which the values of the properties

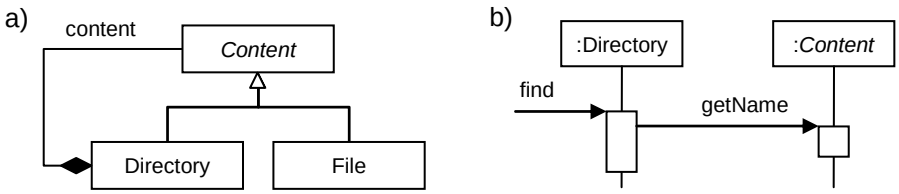

Figure 4. Model of Figure 1 refactored to nested directories: a) new superclass Content extracted from File and Directory, and change of association end content to target that class; b) generalization of classifier File to Content.

m.operation never change for any *m*, this problem can be solved by replacing m_{find}.*operation.name* with o_{find}.*name* and so on, but generally, the problem of chained properties of the form $x.p_1.p_2$ is that which property is selected by p_2 depends on the value of the qualifying property p_1: if constraint solving is allowed to change the value of p_1, the property denoted by $x.p_1.p_2$ (and thus the variable constrained by the enclosing constraint) changes with it. We will return to this problem and provide solution for it in Section 4.3; here, we content ourselves with observing that in constraint-based refactoring, some properties are variable, while others are not.

2.4 Specifying Refactorings

The above RENAME MESSAGE refactorings were performed by assigning the *name* property of a specific message a new value (the new name), and by computing (using constraint solving) which other *name* properties must change accordingly. That we changed a *name* property of a message, and that solving computed new values for other *name* properties (and not for *classifier* or other properties), is part of the nature of the RENAME MESSAGE refactoring — other refactorings are characterized by the change of other properties. Generally, refactorings can be specified by making three statements:

1. the immediate *refactoring intent*, stating the desired new values of selected properties reflecting the primary change associated with the intended refactoring;
2. the *non-fixed properties*, stating which other, complementing (or secondary) changes the refactoring is allowed to make in order to preserve the meaning of the model; and
3. the *fixed properties*, stating what the refactoring must not change.

Of these statements, only the first is specific to the concrete *application of the refactoring* and is made by the user of the refactoring (who has to select the properties the refactoring

is to change, and their new values); the second and the third are specific to the kind to the refactoring (i.e., whether it is a RENAME refactoring, a MOVE refactoring, etc.). In the case of a RENAME MESSAGE refactoring, the refactoring intent is the new name of a selected message, the non-fixed properties are the names of other model elements (more concretely: operations and other messages referring to the same operation), and the fixed properties are all others. As the following, somewhat more demanding examples show, other refactorings can be specified by making the same kinds of statements.

Suppose that the model of Figure 1 is to be refactored to allow nested directories. For this purpose, the refactoring EXTRACT SUPERCLASS [5, 24] is to be applied to classes `Directory` and `File`, creating a new superclass, `Content`, that generalizes both `Directory` and `File`. In addition, this refactoring suggests that the newly introduced `Content` be used in place of `Directory` or `File` wherever the generalization is deemed useful [5]. In our example, this is the case for the composition of directories, which can now be composed of files *and* directories, as reflected in Figure 4 (note that for this, not only the composition in Figure 4 a), but also the classifier associated with the target of the message labelled "getName" in Figure 4 b) changes). What must be made sure by the refactoring, then, is that the operations required from `Content` are defined by `Content`, so that the changed model is well-formed and keeps its meaning. In the given example, this means that `Content` must define $o_{getName}$.

As it turns out, the secondary changes necessary for the EXTRACT SUPERCLASS refactoring can be computed from (9), and thus from the same set of constraints used for the above RENAME refactoring. There are however four important differences with respect to the constrained properties:

1. the *name* properties of messages and operations are fixed for this refactoring;
2. the *operations* properties of classifiers are non-fixed for this refactoring so that constraint solving can assign them new values (corresponding to the moving of operations);
3. the domain of the *classifier* properties of lifelines is extended with a new member, $C_{Content}$, a new classifier literal representing the extracted classifier `Content`; and
4. the refactoring intent is expressed by not changing the value of $l_{left}.classifier$ and by changing the value of $l_{right}.classifier$ to $C_{Content}$, meaning that only the type of this latter lifeline is generalized to `Content`, reflecting the goal of the refactoring.

Of course, for the binding invariant to be maintained, the *operation* properties of messages must remain fixed (as in the RENAME refactoring) — this is required for meaning preservation. From these statements, it follows that

$$m_{find}.operation = o_{find}$$
$$m_{getName}.operation = o_{getName}$$
$$l_{left}.classifier = C_{Directory}$$
$$l_{right}.classifier = C_{Content}$$
$$m_{find}.name = \text{“find”}$$
$$o_{find}.name = \text{“find”}$$
$$m_{getName}.name = \text{“getName”}$$
$$o_{getName}.name = \text{“getName”}$$

which lets the constraint set (9) be reduced to

$$\{o_{find} \in C_{Directory}.operations \wedge \text{“find”} = \text{“find”},$$
$$o_{getName} \in C_{Content}.operations \wedge \text{“getName”} = \text{“getName”}\}$$

which in turn is solved by making $o_{getName}$ a member of $C_{Content}.operations$.[3]

Thus, we have that the above RENAME and EXTRACT SUPERCLASS refactorings can be performed using the exact same set of constraints, the only difference being which properties are assigned new values to reflect the refactoring intent, and which are allowed to be changed by constraint solving. In fact, changing these statements lets us define further refactorings: for instance, making the *classifier* properties of lifelines non-fixed and all other properties fixed provides the basis for the USE SUPERTYPE WHERE POSSIBLE refactoring [24], which would replace the classifiers (types) of all lifelines with supertypes, granted that these supertypes provide all operations required by the messages sent to the lifelines. A further refactoring, MOVE MESSAGE/OPERATION, which is also specified this way, will be introduced in Section 4.

As should have become clear from the above, the great flexibility in specifying refactorings makes constraint-based model refactoring very generic. Yet this genericity is not entirely without problems. More specifically, making the *operations* properties of classi-

[3] Note that the classifier literal $C_{Content}$ is both the owner of the property $C_{Content}.operations$ and the value of the property $l_{right}.classifier$. This is so because the property *classifier* has reference semantics, a notion foreign to standard constraint solvers; we will return to this in Section 4.3.

fiers non-fixed impacts the constraints generated from well-formedness rule (1): since it is unclear up front which operations will be defined in which classifiers, constraints of the form

$$o_1.name \neq o_2.name$$

(which are intended to guarantee local name uniqueness) must be replaced by families of constraints

$$(o_1, o_2 \in c.operations \rightarrow o_1.name \neq o_2.name)_{c \in C}$$

in which C is the set of classifiers of a model the operations o_1 and o_2 may end up in (the original class and the extracted superclass in the case of EXTRACT SUPERCLASS; cf. Section 3.2). How this variability in the constraint generation process can be handled in a systematic fashion will be shown in Section 4.2.

2.5 Interpreting Diagrams

The above examples showed how a well-formedness rule that contains existential quantification can be too weak for refactoring: if selection of the element whose existence is required is part of the interpretation of the diagram (i.e., contributes to its meaning), the rule misses out parts of the static semantics of the modelling language. This semantic underspecification allows refactorings that are correct with respect to the specification, but whose resulting models do not mean the same thing to the user. In these cases, existential quantification can be replaced by a Skolem function, which in turn can be translated to an (additional) association in the metamodel, in our case represented by the property *operation* of messages (cf. (8) and Figure 3). However, the value of this property cannot be read directly from the model — it has to be inferred using the rules of the modelling language's static semantics. This poses the question whether enriched well-formedness rules sufficient for correct refactoring of a model are also sufficient for interpreting it, in our example, to infer its bindings.

As it turns out, they can be, at least for our examples. When interpreting, rather than refactoring, a model such as that of Figure 1, all properties except the ones whose values need to be inferred by way of interpretation (i.e., assigning the model a meaning), are fixed. For example, when the model of Figure 1 must be mapped to an instance of the metamodel of Figure 3, the only non-fixed properties (whose values need to be inferred) are $m_{find}.operation$ and $m_{getName}.operation$ (which are precisely the properties representing

the Skolem function *binding*; cf. Section 2.3). The constraint set constraining the non-fixed properties is again (9), i.e., the same as used for all above refactorings; it is solved by the assignments

$$m_{find}.operation := o_{find} \qquad m_{getName}.operation := o_{getName}$$

which represent the derived information, i.e., desired model interpretation.

While the constraint set is the same as for refactoring, solving it is considerably more difficult, since the properties for which values need to be found are the qualifying properties of $m_{find}.operation.name$ and $m_{getName}.operation.name$ (cf. Section 2.3). Since the constraint variables used in CSPs have value semantics, constraints such as (9) with non-fixed *operation* properties cannot be fed directly to a constraint solver (cf. Footnote 3). Instead, they have to be transformed — how will be shown in Section 4.3.

2.6 Detecting Semantic Underspecification

A companion question to how underspecification is removed (Section 2.3) is how it is detected. As noted above, one way to do so is to refactor a model using the available well-formedness rules, and then assess whether the refactored model has the same meaning to the human beholder. However, given that not all possible refactorings will show the semantic underspecification, this procedure requires significant manual testing.

A fully automated approach to detecting semantic underspecification is to check whether all properties whose values are not directly given in the model (so that their values need to be obtained by way of interpretation; see above) can be assigned unique values.[4] For instance, interpreting the model of Figure 1 in terms of the metamodel of Figure 3 with the constraints (4) resulting from the application of well-formedness rule (2) to the model, the values of the *operation* properties of messages are left dangling — they can be assigned any value, and the constraints will be satisfied. Thus, the binding of messages cannot be inferred using (2) — the rule is too unspecific. By contrast, applying the well-formedness rule (8) to Figure 1, the values of $m_{find}.operation$ and $m_{getName}.operation$ are

[4] NB: It is admissible in UML that not all information that can be specified in a diagram is actually provided by the modeller. For instance, in a sequence diagram the name of a message may be omitted. This intentional underspecification is a feature of modelling rather than a bug of the modelling language; however, deliberately underspecified models are generally inept to identify the semantic underspecification of the language we are searching for.

uniquely determined by the resulting constraints — (8) is sufficient for interpreting this model.[5]

However, that a single well-formed model such as that of Figure 1 can be interpreted unambiguously is not enough to infer that the static semantics of a modelling language is sufficiently specified. For instance, in a variation of Figure 1 in which o_{find} and o_{add} both have the name "find", the value of $m_{find}.operation$ is not uniquely determined (meaning that its binding is ambiguous), so that rule (8) alone is also insufficient — it needs to be complemented by rule (1), removing the ambiguity by identifying this model as malformed, hence rejecting it as meaningless.

It follows that detecting underspecification is a search problem: if a model is found that cannot be interpreted unambiguously, we know that we have a case of semantic underspecification; if not, we know nothing. Models can be found using the usual techniques of model instance generation used, e.g., in model checking (here applied on the metalevel); whether interpretation of a generated model is ambiguous is computed as the number of solutions found by constraint solving (if it is greater than 1, the model cannot be interpreted unambiguously). Of course, the viability of this approach depends on the fact that properties whose values need to be derived by way of interpretation (in our example: the *operation* properties of messages) do exist — if not (here: if no *operation* properties are defined, as was the case for the metamodel of Figure 2), there is nothing left for interpretation, so that no underspecification can be detected (cf. Footnote 5).

2.7 Summary

Thus, we have that the constraints generated from simple well-formedness rules such as (1) and (2), the latter enhanced and rewritten as in (8), are not only sufficient for performing very different refactorings, they also serve the interpretation of the diagrams, as well as the detection of semantic underspecification of the modelling language. The only adaptation necessary for the different uses of the constraints is to specify which properties are fixed and which are non-fixed and thus can be adapted by constraint solving.

[5] Note that the metamodel of Figure 2 is insufficient to detect the underspecification automatically, by means of interpretation — since this metamodel does not provide for the binding of a message to an operation, it cannot be derived that binding is ambiguous.

3 Constraint-Based Refactoring

The technique of constraint-based model refactoring that we are proposing is based on constraint-based program refactoring as described in some detail in [18, 19, 24, 25]. To make this article self-contained, we provide a brief introduction to constraint-based program refactoring here, limiting ourselves to those aspects of it that are relevant for model refactoring also.

3.1 Constraint Rules

In constraint-based program refactoring, a program to be refactored is transformed to a constraint satisfaction problem (CSP) by application of so-called *constraint rules*, which are generally of the form

$$\frac{query}{constraints}$$

Here, *query* represents a logical expression searching for elements of the program to be refactored, and *constraints* represents the set of constraints to be generated (added to the CSP) for the program elements selected by the query. Both the queries and the constraints contain variables that are placeholders for the program elements the rule is applied to; these variables (which are not constraint variables!) are implicitly universally quantified. For instance, application of the constraint rule

$$\frac{binds(r, d)}{r.name = d.name} \tag{10}$$

to a program to be refactored searches the program for occurrences of all pairs of references r and declared entities d such that r binds to d, and generates for each found pair a constraint requiring that *name* properties of r and d equal. This constraint rule expresses a binding invariant of the underlying programming language: for r to bind to d, r and d must have the same names. Taken alone, this constraint allows it that a reference r or a declared entity d be given a new name by a refactoring as long as the name of the other changes to that name as well; however, constraints generated by the same or other constraint rules may constrain the properties (constraint variables) $r.name$ and $d.name$ further. Note that when applied to a well-formed program, the generated constraints are always satisfied with the properties set to the values reflecting the program as is before the refactoring (the *initial values*).

3.2 Specifying and Applying a Refactoring

To a certain extent, constraint rules are agnostic to the refactorings they are used for [21, 25] (see Section 3.4 for when this may not be the case), so that they can be reused across different refactorings. A particular refactoring then requires further specification, namely of what it is to change (called the *forced changes* in [19]) and what it may change (the *allowed changes*). What it is to change reflects the *refactoring intent* and is specified by the user of the refactoring with each application; what it may change divides the remaining properties into *non-fixed* and *fixed* ones (cf. Section 2.4) and specifies the kind of the refactoring (e.g., whether it is an instance of RENAME, EXTRACT SUPERCLASS, or MOVE).[6] Once these specifications have been made, the constraints can be generated.[7]

As pointed out above, the CSP thus generated is in solved form with the program as is, i.e., with all constrained properties set to their initial values. By applying the refactoring intent, that is, by assigning the selected properties their new values, the CSP may become unsolved, in which case further action is required: Only if the constraints are still satisfied with the new property assignments, the refactoring is finished — if not, a constraint solver must attempt to assign non-fixed properties new values, until a solution is found. The computed assignments constituting the solution then represent the additional (secondary) changes to the program that are required to make the refactoring work; if no solution is found, the refactoring is impossible and must be rejected.

3.3 Dealing with Multiple Solutions

If constraint solving finds more than one solution, the secondary changes required by the refactoring are not uniquely determined. In this case, it is possible to add soft constraints to the CSP that direct solving in finding a best solution [19]. For instance, the solver

---

[6] See [19] for how constraint-based refactorings can be specified formally. As has been pointed out in [22], the difference between the kind of a refactoring and its particular application dissolves when ad hoc refactorings, that is, refactorings which have not been standardized and which can perform arbitrary changes, are considered.

[7] In theory, the constraints can be generated from the constraint rules and the program to be refactored alone. However, in practice this leads to a large number of superfluous constraints (i.e., constraints that are unrelated to the intended refactoring). Therefore, special algorithms have been devised that generate only the constraints required for a specific application of a specific kind of refactoring [19, 21]. These algorithms take the specific application as part of their input.

could be instructed to minimize the number of additional changes or to prefer one kind of change (e.g., renaming a model element) over another (e.g., moving it). However, since the best solution is not the only solution, and since it may not meet the approval of the user, alternative solutions may need to be inspected. It is then important that the number of alternatives is kept small.

One particularly effective way of reducing the size of the solution space is to restrict the domains of non-fixed properties (constraint variables) as far as possible. For instance, in case of the EXTRACT SUPERCLASS refactoring, it is useful to restrict the set of possible locations of class members to their old class and the newly extracted superclass (cf. Section 2.4), in particular to exclude superclasses of the new superclass (if any), to which the members could also be moved. Similarly, for renaming it is useful to restrict the domains to the old and a (fresh) new name, rather than allowing all conceivable names. Using this technique, and also using special algorithms for constraint rule application [19, 21], the number of alternative solutions (which depends heavily on the concrete refactoring) can be kept small (see [19, 21] for empirically obtained figures for a selection of refactorings).

3.4 Constraint Rule Rewriting

It is instructive to note that to a certain extent, the queries (expressions above the bar) and the constraints (expressions below the bar) of a constraint rule are exchangeable for each other. In fact, as we have noted elsewhere [19, 21], the main difference between constraints and queries is that while a query is evaluated at rule application time, a constraint (generated by rule application) is evaluated at constraint solving time. This means that for constraints whose constrained properties are all fixed (so that they can be evaluated at rule application time), constraint rules can be rewritten to save the generation of these constraints. For instance, in the constraint rule

$$\frac{binds(r, d)}{d \in r.receiver.type.members \quad r.name = d.name}$$

if the receiver of a reference *r*, its type, and the set of members of the type are all fixed, the constraint $d \in r.receiver.type.members$ can be promoted to a query, giving us

$$\frac{binds(r, d) \quad d \in r.receiver.type.members}{r.name = d.name}$$

Since $d \in r.receiver.type.members$ is subsumed by $binds(r, d)$, it can dropped from the rule precedent, which then still matches the exact same pairs (r, d). Assuming that names are non-fixed on the other hand means that the constraint $r.name = d.name$ is needed for refactoring, and must remain in the rule consequent.

While the lack of variability of some properties allows constraints to be promoted to queries or dropped altogether, the variability of others may require the demotion of queries to rule consequents, where they serve as premises of conditional constraints. For instance, in the case of the rule

$$\frac{same\text{-}scope(d_1, d_2)}{d_1.name \neq d_2.name}$$

requiring uniqueness of names of different declared entities defined in the same scope, if declared entities may be moved between scopes by a refactoring, the constraint rule must be rewritten to

$$\frac{d_1 \quad d_2}{same\text{-}scope(d_1, d_2) \rightarrow d_1.name \neq d_2.name}$$

(cf. Section 2.4).

The possible rewriting of constraints rules due to the fixedness of properties is central to our transformation of well-formedness rules to constraint rules; a more complete account of it is given in [21].

4 From Well-Formedness Rules to Constraint Rules

Well-formedness rule checking can be viewed as a special case of constraint-based refactoring (constraint generation and subsequent constraint solving) in which all constrained properties (constraint variables) are fixed and set to their initial values. For instance, the constraint rule (10) directly translates to the well-formedness rule

$$\forall r, d, binds(r, d) : r.name = d.name$$

However, the opposite mapping, from a well-formedness rule to a constraint rule, is more difficult, since it must (a) deal with quantifiers that are nested inside well-formedness expressions; (b) separate the fixed from the non-fixed properties, making it dependent on the concrete refactoring; and (c) must map properties with reference semantics (used for

navigating a model) to properties with value semantics (which can be used by a constraint solver).

4.1 Dealing with Quantifiers

As stated in Section 3.1, a constraint rule implicitly universally quantifies over the variables representing the model elements the rule is applied to. A well-formedness rule likewise universally quantifies over the model elements it applies to (in OCL, implicitly over the context variable `self`). For well-formedness rules with universal quantifiers nested inside expressions, these must be moved to the left prior to the transformation into a constraint rule. This is a standard procedure, which may however require the renaming of variables [16].

As has been pointed out in Section 2.3, existential quantifiers may be a source of semantic underspecification that may impede meaning preservation (if not on a strictly technical level, at least on an intuitive level[8]). Therefore, one should always attempt to remove existential quantifiers contained in a well-formedness rule using Skolemization, that is, by replacing a choice of model elements (a disjunction) with the selection of a specific one that satisfies the (formerly quantified) expression.[9] If Skolemization is impossible, for instance because the language specification does not give reason to the selection of a specific model element (see Section 5 for examples of this), existentially quantified expressions have to be unrolled (as was done in Section 2.2, before Skolemization was considered) prior to conversion into a constraint rule. In that case, refactorings based on this constraint rule may still be correct in a technical sense (cf. above), but may not be meaning preserving according to the user's intuition. Note that, since the number of model elements is always finite, unrolling into a disjunction is always possible.

4.2 Separation into Queries and Constraints

Generally, a constraint rule has the form of a universally quantified implication, with the additional restriction that, as noted in Section 3.4, the premise must contain only constraints whose properties are fixed for all model elements quantified over (so that they

[8] Cf. the initial remarks of Section 2.5 concerning the correctness of refactoring under underspecification.

[9] Arguably, a semantic interpretation that contains a choice (a disjunction) is ambiguous.

Table 1: Fixed and non-fixed properties for three important refactorings (cf. Figure 3).

REFACTORING PROPERTY	RENAME MESSAGE/OPERATION	EXTRACT SUPERCLASS	MOVE MESSAGE/OPERATION
messages	fixed	fixed	non-fixed
operation	fixed	fixed	fixed
classifier	fixed	non-fixed	fixed
operations	fixed	non-fixed	non-fixed
name	non-fixed	fixed	fixed

can be evaluated at rule application time). Thus, the first step in transforming a well-formedness rule such as (8) into a constraint rule is rewriting it to the form

$$\forall\, l, m : l \in \mathit{lifelines} \wedge m \in l.\mathit{messages} \rightarrow \\ m.\mathit{operation} \in l.\mathit{classifier}.\mathit{operations} \wedge m.\mathit{name} = m.\mathit{operation}.\mathit{name} \tag{11}$$

The remainder of the transformation depends on which properties are fixed and which are non-fixed for a given refactoring, as shown in Table 1 for three sample refactorings.

For RENAME, *name* is the only non-fixed property so that (11) can be transformed to

$$\frac{l \in \mathit{lifelines} \quad m \in l.\mathit{messages}}{m.\mathit{name} = m.\mathit{operation}.\mathit{name}} \tag{12}$$

This is so because *l.messages* never changes so that the constraint $m \in l.\mathit{messages}$ can be evaluated at rule application time, after *m* and *l* have been bound to concrete model elements (recall that both *m* and *l* are implicitly universally quantified), and because neither of *m.operation*, *l.classifier*, and *l.classifier.operations* can change their values, so that $m.\mathit{operation} \in l.\mathit{classifier}.\mathit{operations}$ must remain satisfied and can be dropped (as stated in Section 3.1, all constraints are satisfied with their initial assignments).

For EXTRACT SUPERCLASS, the transformation is analogous, yielding

$$\frac{l \in \mathit{lifelines} \quad m \in l.\mathit{messages}}{m.\mathit{operation} \in l.\mathit{classifier}.\mathit{operations}} \tag{13}$$

which, applied to the example of Section 2.4, produces exactly the constraints necessary to force that $o_{getName}$ is an operation of $C_{Content}$. Finally, for MOVE METHOD [5] (which, depending on which kind of model element or diagram it is applied to, should be called MOVE OPERATION or MOVE MESSAGE), (11) transforms to

$$\frac{l \in lifelines \quad m}{m \in l.messages \rightarrow m.operation \in l.classifier.operations}$$

in which the constraint $m \in l.messages$ cannot be promoted to a query, since *l.messages* may be changed by the refactoring (it is changed for the source and target lifelines of the message to be moved, and may be changed for other messages that may have to move with it; cf. [5] for why this may be the case).

4.3 Mapping Properties to Constraint Variables

In the previous sections, we pretended that the properties involved in well-formedness rules can be directly mapped to constraint variables that can be handled by a constraint solver. Generally, however, this is not the case. Instead, we have to deal with the following mismatches:

- *Properties may have reference semantics.* Properties representing certain attributes and all association ends have reference semantics, i.e., they point to other objects. By contrast, constraint variables generally have value semantics, and their values (except for set values; see below) are unstructured.

  Solution: Map properties with reference semantics to constraint variables with value semantics, and emulate dereferencing of such variables as shown below.

- *Properties may have other than* {1} *multiplicities.* Many properties are optional, which in UML is represented by a {0..1} multiplicity. Others model links to arbitrary numbers of objects at the same time, which is represented by a {0..*} multiplicity. By contrast, constraint variables always have a single value, which may however be a set.

  Solution: Map properties to constraint variables with set domains, and transform multiplicities to constraints on the cardinalities of the values of these variables.

- *Properties may be chained.* Properties with reference semantics (cf. above) may be chained, which amounts to a navigation of properties, involving dereferencing of intermediate properties. By contrast, constraint variables cannot be dereferenced: the value of a constraint variable cannot be, or have, a constraint variable. This is particularly a problem if the properties through which is being navigated are non-fixed (meaning that their values can be changed by a constraint solver).

  Solution: Let $C(x.p_1.\cdots.p_n.p)$ be a constraint constraining property p accessed via navigation through properties p_1, …, p_n (all with reference semantics and, for uniform-

ity of presentation, all assumed to be set-valued) starting from the object represented by variable x (so that $x.p_1.\cdots.p_n$ evaluates to the set of objects that can be reached from x by navigating through p_1, …, p_n; note that C may — and usually will — constrain other properties as well). Without loss of generality, we assume x to be universally quantified and restricted by a predicate (constraint) P involving only fixed properties, so that we have

$$\forall x : P(x) \rightarrow C(x.p_1.\cdots.p_n.p) \tag{14}$$

To be able to map $C(x.p_1.\cdots.p_n.p)$ to a constraint of a constraint rule, we first have to replace $x.p_1.\cdots.p_n$ with a variable y representing the model elements reached from x via p_1, …, p_n so that we can rewrite (14) to the intermediate form

$$\frac{P(x) \quad y}{y \in x.p_1.\cdots.p_n \rightarrow C(y.p)} \tag{15}$$

which is implicitly quantified over x and y and in which $y.p$ is a constraint variable not involving dereferencing. If the p_1, …, p_n are fixed properties (i.e., if their values cannot be changed by the refactoring), (15) translates to

$$\frac{P(x) \quad y \in x.p_1.\cdots.p_n}{C(y.p)}$$

in which $y \in x.p_1.\cdots.p_n$ is evaluated as a query so that the involved properties p_1 through p_n need not be mapped to constraint variables. For instance, the constraint rule for RENAME, (12), translates to

$$\frac{l \in lifelines \quad m \in l.messages \quad o \in m.operation}{m.name = o.name}$$

whose generated constraints contain only properties that map directly to constraint variables (cf. Table 1 to see that only fixed properties appear above the bar; note that, conforming to the above, *m.operation* is assumed to be set-valued, i.e., a singleton). If a single p_i is non-fixed, (15) translates to

$$\frac{P(x) \quad x_{i-1} \in x.p_1.\cdots.p_{i-1} \quad x_i \quad y \in x_i.p_{i+1}.\cdots.p_n}{x_i \in x_{i-1}.p_i \rightarrow C(y.p)}$$

which (implicitly) quantifies over x, x_{i-1}, x_i, and y, and in which the constraint $C(y.p)$ is guarded by the condition that whatever the values assigned (by the solver) to p_i, y is reached from x via $x.p_1.\cdots.p_n$. (Note that $x_0 \equiv x$ and $x_n \equiv y$, and that queries involving

p_0 (for $i = 1$) or p_{n+1} (for $i = n$) are dropped.) For instance, the constraint rule for EXTRACT SUPERCLASS, (13), of which *l.classifier.operations* corresponds to $x.p_1.p$ and $m.operation \in l.classifier.operations$ corresponds to $C(x.p_1.p)$, and in which *classifier* and *operations* are the only non-fixed properties (see Table 1) so that $i = n = 1$, is rewritten to

$$\frac{l \in lifelines \quad m \in l.messages \quad c}{c \in l.classifier \rightarrow m.operation \in c.operations}$$

in which c corresponds to $x_1 \equiv y$ (and *l.classifier* is a singleton for all l). If two properties p_i and p_j with $i < j$ are variable, (15) translates to

$$\frac{P(x) \quad x_{i-1} \in x.p_1.\cdots.p_{i-1} \quad x_i \quad x_{j-1} \in x_i.p_{i+1}.\cdots.p_{j-1} \quad x_j \quad y \in x_j.p_{j+1}.\cdots.p_n}{x_i \in x_{i-1}.p_i \wedge x_j \in x_{j-1}.p_j \rightarrow C(y.p)}$$

and so forth. Note that in all cases, the queries involve only fixed properties so that they can be evaluated at rule application time, and the generated constraints contain no chained properties so that no dereferencing is required.

Thus, together with the transformations of Section 4.2, we are able to rewrite any well-formedness rule expressed in terms of FOPL with path expressions into a constraint rule that produces only constraints amenable to standard constraint solving.

5 Real Examples from the UML Standard

To demonstrate how our approach to transforming well-formedness rules to constraint rules suitable for constraint-based refactoring extends beyond the examples of Section 2, we have applied it to three OCL rules directly taken from the UML Superstructure specification [9]. We do not delve into the details of translating the various OCL iterators to solver constraints here; this has been dealt with, for instance, in [2].

- From the *Constraints* section of §7.3.22, "InstanceSpecification":

  The defining feature of each slot is a structural feature (directly or inherited) of a classifier of the instance specification.

  ```
  slot->forAll(s | classifier->exists (c | c.allFeatures()->includes (s.definingFeature)))
  ```

  This is a typing rule, expressing that the defining feature associated with a slot of an instance specification must be a feature of at least one classifier the specified instance is an instance of. Assuming that typing must be preserved, but that the classifier(s) as-

sociated with an instance may be changed (e.g., EXTRACT SUPERCLASS [5] or GENERALIZE DECLARED TYPE [24] applied to a communication diagram), this translates to the constraint rule

$$\frac{s \in self.slot}{\exists c : c \in self.classifier \wedge s.definingFeature \in c.allFeatures()} \tag{16}$$

in which *self* represents the context [10], the Instance Specification the rule is applied to, and in which the existential quantification must be unrolled upon rule application. Skolemization is also possible, but requires a slight adaptation: the derived property *featuringClassifier* of features ([9], §7.3.19) corresponds to a set-valued Skolem function $featuringClassifier: Feature \rightarrow \wp(Classifier)$, allowing us to rewrite the above rule to

$$\frac{s \in self.slot \quad f = s.definingFeature}{f.featuringClassifier \cap self.classifier \neq \varnothing}$$

in which the second conjunct from the consequent of (16),

$$s.definingFeature \in s.definingFeature.featuringClassifier.allFeatures()$$

has been dropped (because it is tautological).

- From the *Constraints* section of §7.3.44, "Property":

  Subsetting may only occur when the context of the subsetting property conforms to the context of the subsetted property.

```
subsettedProperty->notEmpty() implies
    (subsettingContext()->notEmpty() and subsettingContext()->forAll (sc |
        subsettedProperty->forAll(sp |
            sp.subsettingContext()->exists(c | sc.conformsTo(c)))))
```

  This rule is to express that in case a set-valued property (attribute or association end) is to subset one or more other properties, the context of the property, the owning or, in case of an end of a more than binary association, all owning classifiers, must conform to the classifier(s) of the properties that are being subset. For a PULL UP PROPERTY refactoring, it translates to the constraint rule

$$\frac{|self.subsettedProperty| > 0 \quad sp \in self.subsettedProperty \quad sc}{|self.subsettingContext()| > 0}$$
$$sc \in self.subsettingContext() \rightarrow \exists c \in sp.subsettingContext() : sc.conformsTo(c)$$

which is however ambiguous with respect to which subsetting context should conform to which, in case there is more than one. At least, [9] hints at a suitable Skolemization, by requiring conformance with the "*corresponding* element in the context of the subsetted property" (albeit without formalizing correspondence).

- From the *Constraints* section of §15.3.12, "StateMachine":

  The context classifier of the method state machine of a behavioural feature must be the classifier that owns the behavioural feature.

  ```
  specification->notEmpty() implies (context->notEmpty() and
    specification->featuringClassifier->exists (c | c = context))
  ```

  This is to express that a state machine specifying a behavioural feature (method) of a classifier must have that classifier as its context; it could be violated by a MOVE BEHAVIOURAL FEATURE refactoring, changing the `featuringClassifier` and `context` properties. The derived constraint rule for this refactoring is

  $$\frac{|self.specification| > 0 \quad s \in self.specification}{|self.context| > 0 \wedge \exists c \in s.featuringClassifier : c \in self.context}$$

  whose existential quantifier must be unrolled upon application. Note that, since *featuringClassifier* has multiplicity {0..*} (cf. above), it is not clear to which classifier the *context* property of a state machine should be set, not even intuitively — in absence of a sensible Skolem function correcting this, *context* should be given multiplicity {0..*}, too, and the constraint should be changed to

  ```
  specification->notEmpty() implies context = specification->featuringClassifier
  ```

6 Limitations of the Approach

Since an ill-formed model is meaningless, maintaining well-formedness is a necessary condition for meaning preservation and thus model refactoring. That it is not also a sufficient condition was already shown in Section 2.3: where well-formedness requires the mere existence of a model element that is depended upon, meaning preservation may additionally require that the same model element is depended upon before and after the refactoring. This observation begs the question whether this is the only limitation with respect to the sufficiency of well-formedness rules for constraint-based model refactoring.

Frankly, the answer to this question is no. Well-formedness rules, or constraints in general, are usually not used to specify the dynamic semantics of (modelling) languages, so when it comes to the refactoring of behavioural specifications, our approach of re-using available well-formedness rules to express the invariants of refactorings lacks its basis. With respect to the choice of refactorings that can be captured using our approach, this means that we are limited to ones that change the static structure of a model. As has been seen in the examples of Section 2, this may include the static structure underlying behavioural specifications (such as the name of a message or the classifier of a lifeline in sequence diagrams), but it rules out refactorings of the behaviour expressed in such diagrams (such as the order in which messages are sent). The more general question whether behavioural specifications can at all be refactored using a constraint-based approach depends on whether dynamic semantics can be expressed using constraints; so far, we have not seen such specifications.

7 Related Work

By presenting an initial set of model refactorings, and by providing formal (OCL) pre- and postconditions for some of them, Sunyé et al. set an early landmark [23]. The constraint-based refactoring approach presented here deviates from the more traditional form of specifying refactorings via preconditions and postconditions (or mechanics) by replacing preconditions with invariants that guarantee meaning preservation. Failure to meet the invariants then corresponds to inapplicability, and hence to precondition violation.

Philipps and Rumpe showed how state machine refactorings can be viewed as refinements that can be proven meaning preserving [12], but it is unclear how their approach generalizes to other refactorings. Pretschner and Prenninger let the user specify predicates that partition the state space of state machines, from which refactorings can then be computed [14]; their approach also appears to be specialized to one kind of models. by contrast, the generic model refactorings of [8] are set in the more general context of generically specifying transformations (not necessarily of models) with respect to different metamodels, therefore addressing the need to abstract from them and defining the refactorings in terms of these abstractions. However, experience with program refactoring has shown that the hard problems of refactoring lie in the details of a language specification (including its specific well-formedness rules; see, e.g., [17]), so that refactoring

specifications ignoring these details are almost inevitably incomplete. Porres specified a refactoring as a set of transformation rules relying on an action language for query and updating models, where correctness of the refactored model is guaranteed by checking conformance with the metamodel and satisfaction of applicable OCL constraints [13]. By contrast, we use metamodel and constraints as specifications of the refactorings. Gheyi et al. presented an approach for proving structural model refactorings for Alloy [6]; however, the technical scaffolding required for correct refactoring is significant, especially when compared to our approach, which re-uses pre-existing semantic specifications.

Not dealing with model refactoring, but nevertheless related to our work, Cabot et al. investigated how UML/OCL models can be transformed to CSPs that can be submitted to a constraint solver, to verify stated correctness properties of models by generating instances [2]. Our work is different in that we always start with a correct (meta)model instance (the model to be refactored) that is then temporarily invalidated by a refactoring, so that a similar (neighbouring) instance needs to be found. As has been shown elsewhere [19], this allows us to use an algorithm for constraint generation that avoids the complexity problems from which the unbounded translation of [2] suffers. Ali et al. [1] also employ OCL constraint solving, for (UML) model-based test case generation, but to address the combinatorial complexity encountered in [2], resort to a search-based approach; their search heuristics could be integrated in our approach to make constraint solving even faster. Also methodically related to ours is Egyed's work on fixing inconsistencies in models, as detected by the violation of constraints [4]: in fact, fixing inconsistencies can be seen as solving an unsatisfied CSP (with the set of solutions representing all possible repairs). With the Beanbag language [26], OCL-like consistency relations can be extended with fixing behaviour specifying how changes leading to model inconsistencies are to be compensated with other, repairing changes; however, the compensated changes are not necessarily meaning-preserving, and thus not refactorings. Even if certain refactorings could be specified as fixes in Beanbag, different refactorings would still need different fixing operations. This is in contrast to the approach presented here, for which a single set of well-formedness rules suffices for different refactorings. Finally, Correa and Werner extended the notion of model refactoring to the (co-)refactoring of OCL constraints [3]. Since OCL has well-formedness rules specified in OCL [10], our approach should be extendible to OCL refactoring also; however, we have not investigated this further.

8 Conclusion

For a modelling language without semantics, every change to a model is a refactoring. The more of the semantics of a modelling language has been specified, the fewer changes to a model result in models with the same meaning, i.e., in refactorings. By taking semantic specifications pre-existing in the form of well-formedness rules expressed in a constraint language as a starting point, we are able to transform refactoring problems as diverse as renaming, generalizing, or moving model elements, to CSPs that are amenable to standard constraint solving, which can thus be used to compute the additional changes required for a specific intended model refactoring. Using our approach, semantic underspecification is unveiled by refactored models that do not mean the same to the user; in such cases, the pre-existing constraints may be complemented with the missing semantics, for instance by extending the metamodel and adapting the constraints accordingly.

Acknowledgments

The work described in this article has been supported by the Deutsche Forschungsgemeinschaft (DFG) under grant STE 906/4-1. The author thanks Thomas Kühne for his comments on an earlier version of this article.

References

1. S Ali, M Z Iqbal, A Arcuri, L Briand “A search-based OCL constraint solver for model-based test data generation” in: *Proc. of QSIC* (2011) 41–50.
2. J Cabot, R Clarisó, D Riera “UMLtoCSP: A tool for the formal verification of UML/OCL models using constraint programming” in: *Proc. of ASE* (2007) 547–548.
3. AL Correa, CML Werner “Applying refactoring techniques to UML/OCL models” in: *Proc. of UML* (2004) 173–187.
4. A Egyed “Fixing inconsistencies in UML design models” *Proc. of ICSE* (2007) 292–301.
5. M Fowler *Refactoring: Improving the Design of Existing Code* (Addison-Wesley 1999).
6. R Gheyi, T Massoni, P Borba “A rigorous approach for proving model refactorings” in: *Proc. of ASE* (2005) 372–375.

7. WG Griswold *Program Restructuring as an Aid to Software Maintenance* (PhD Dissertation, University of Washington, 1992).
8. N Moha, V Mahé, O Barais, JM Jézéquel “Generic model refactorings” in: *Proc. of MoDELS* (2009) 628–643.
9. OMG Unified Modeling Language Superstructure Version 2.3 (May 2010) www.omg.org/spec/UML/2.3/Superstructure
10. Object Management Group Object Constraint Language Version 2.2 http://www.omg.org/spec/OCL/2.2
11. J Palsberg, MI Schwartzbach Object-Oriented Type Systems (Wiley 1994).
12. J Philipps, B Rumpe “Refactoring of programs and specifications” in: *Practical Foundations of Business and System Specifications* (Kluwer Academic Publishers, 2003) 281–297.
13. I Porres “Model refactorings as rule-based update transformations” in: *Proc. of UML* (2003) 159–174.
14. A Pretschner, W Prenninger “Computing refactorings of state machines” *Software and System Modeling* 6:4 (2007) 381–399.
15. J Reimann, M Seifert, U Aßmann “Role-based generic model refactoring” in: *Proc. of MoDELS* 2 (2010) 78–92.
16. S Russell, P Norvel *Artificial Intelligence: A Modern Approach* 2nd Edition (Prentice Hall, 2003).
17. M Schäfer *Specification, Implementation and Verification of Refactorings* (PhD Thesis, Oxford University Computing Laboratory, 2010).
18. F Steimann, A Thies “From public to private to absent: Refactoring Java programs under constrained accessibility” in: *Proc. of ECOOP* (2009) 419–443.
19. F Steimann, C Kollee, J von Pilgrim “A refactoring constraint language and its application to Eiffel” in: *Proc. of ECOOP* (2011) 255–280.
20. F Steimann “Contraint-based model refactoring” in: *Proc. of MoDELS* (2011) 440-454.
21. F Steimann, J von Pilgrim “Constraint-based refactoring with foresight” in: *Proc. of ECOOP* (2012) 535–559.
22. F Steimann, J von Pilgrim “Refactorings without names” in: *Proc. of ASE* (2012) 290–293.
23. G Sunyé, D Pollet, Y Le Traon, JM Jézéquel “Refactoring UML Models” in: *Proc. of UML* (2001) 134–148.
24. F Tip, A Kiezun, D Bäumer “Refactoring for generalization using type constraints” in: *Proc. of OOPSLA* (2003) 13–26.
25. F Tip, RM Fuhrer, A Kiezun, MD Ernst, I Balaban, B De Sutter “Refactoring using type constraints” *ACM Trans. Program. Lang. Syst.* 33(3):9 (2011).

26. Y Xiong, Z Hu, H Zhao, H Song, M Takeichi, H Mei “Supporting automatic model inconsistency fixing” in: *Proc. of ESEC/SIGSOFT FSE* (2009) 315–324.