Computing Repair Trees for Resolving Inconsistencies in Design Models

Alexander Reder
Johannes Kepler University
Linz, Austria
alexander.reder@jku.at

Alexander Egyed
Johannes Kepler University
Linz, Austria
alexander.egyed@jku.at

ABSTRACT

Resolving inconsistencies in software models is a complex task because the number of repairs grows exponentially. Existing approaches thus emphasize on selected repairs only but doing so diminishes their usefulness. This paper copes with the large number of repairs by focusing on what caused an inconsistency and presenting repairs as a linearly growing repair tree. The cause is computed by examining the runtime evaluation of the inconsistency to understand where and why it failed. The individual changes that make up repairs are then modeled in a repair tree as alternatives and sequences reflecting the syntactic structure of the inconsistent design rule. The approach is automated and tool supported. Its scalability was empirically evaluated on 29 UML models and 18 OCL design rules where we show that the approach computes repair trees in milliseconds on average. We believe that the approach is applicable to arbitrary modeling and constraint languages.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques

General Terms

Design, Performance

Keywords

Inconsistency Management, Repairing Inconsistencies

1. INTRODUCTION

State-of-the-art on inconsistency management in model-based software development has focused on detecting inconsistencies [21]. Today, many approaches are available and they detect inconsistencies fast and correctly [14, 17, 8, 5]. While it is important to allow inconsistencies [4], they must be resolved eventually. Unfortunately, repairing inconsistencies is much harder than detecting them because the number of alternatives grows exponentially with the complexity of the design rule and the number of model elements accessed.

Existing approaches either ignore the exponential growth [18] or emphasize on selected repairs only [7, 20]. Yet other approaches pose limitations on the consistency language or focus on individual inconsistencies [24, 11, 18]. All these limitations are problematic because the designer must ultimately choose and the generated repairs either overwhelm the designer or may fail to include the one the designer wants. Since the repair of inconsistencies goes hand in hand with the creative process of modeling, we strongly advocate against heuristics that replace the role of the human designer. For example, a repair that favors the fewest model changes is frequently the same as an undo.

Our approach combines the structure of design rules (as in xLinkit [18]), the design rule's expected and observed validation results (in part through observing the rule's validation as in [9]), and basic facts about the meta model (types of fields, non-changeable model elements) to filter impossible changes. The structure is important for understanding the inconsistency and enumerating repair alternatives. For example, if an inconsistent rule's structure requires $A \wedge B = true$ then there are three repair alternatives (as generated by xLinkit): repair A, repair B, or repair A and B. This list appears reasonable, however, it may contain incorrect repair alternatives and non-minimal repair actions. For example, if B is true already then 1) repair B is incorrect because it would repair something that is not broken and 2) repair A and B is non-minimal because it would repair more than necessary. Of course, repair A may inadvertently break B which is a side effect and also detectable by our approach.

By eliminating false and non-minimal repairs, the approach vastly reduces the number of repair alternatives; however, not necessarily the exponential growth. This paper thus also introduces repair trees. A repair tree organizes the repair actions in a hierarchical manner that reflects the structure of the design rule. The repair tree is thus intuitive to understand and, more significantly, the repair tree (repair actions) grows linearly with design rule complexity (expressions that must be validated).

The main contribution of this paper is a fully automated, tool supported, scalable approach for generating a complete set of repair alternatives; where each alternative consists of sequences of model changes (repair actions) that are needed to resolve the inconsistency. Typically, there exist many alternatives for repairing inconsistencies, however, these alternatives strongly overlap in the model changes (repair actions) they require. Enumerating all repair alternatives is

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ASE '12, September 3–7, 2012, Essen, Germany
Copyright 2012 ACM 978-1-4503-1204-2/12/09 ...$15.00.

trivial in principle; however, it results in a large number of alternatives – too many for human consumption. The repair tree computed by our approach focuses on repair actions on the part of the model that caused the inconsistency and organizes those repair actions in a hierarchical manner to help the designer decide how to repair the model. Its correctness and scalability was validated on 29 mostly industrial models. The approach was designed to apply to arbitrary modeling language (our focus on three quite distinct types of UML diagrams supports this since these diagrams are often considered separate modeling languages) and constraint languages.

The remainder of this paper is organized as follows. Section 2 defines the basic terms that are used in this paper and Section 3 illustrates the main idea. The approach is discussed in Section 4 and validated in Section 5. Section 6 gives an overview of related work and Section 7 concludes this paper with an outlook on future work.

2. DEFINITIONS

This section provides some basic definitions:

DEFINITION 1. *A **model** represents the software system that must be implemented. It consists of **model elements** which contain **properties**, e. g., a name or a reference to other model elements.*

*A **design rule** defines a requirement that the model must fulfill. The requirement is expressed as a **condition** that evaluates to true (consistent) or false (inconsistent). A design rule is written for a specific **context** that can be a single model element (the design rule will be validated once) or a type of model element (the design will be validated for each instance of this model element type in the model).*

$$\begin{array}{lcl} Design\,Rule & := & \langle context,\, condition\rangle \\ condition & : & context \rightarrow \{true,\, false\} \end{array}$$

DEFINITION 2. *A design rule condition consists of a set of hierarchical ordered expressions where each expression consists of an operation (o), a set of 0 to * arguments (α), an expected (σ) and a validated result (ς). The arguments of an **expression** (ϵ) are also expressions, forming a tree analogous to that of programming languages. An expression has exactly one parent except for the root expression (ϵ_0) but as many child expressions as there are arguments.*

$$\begin{array}{lcl} condition & := & \bigcup_{i=0}^{n} \epsilon_i | \begin{cases} \exists i,j : \epsilon_j \in \epsilon_i.\alpha & \text{if } j>0,\, i\neq j \\ \nexists i,j : \epsilon_j \in \epsilon_i.\alpha & \text{if j=0, i}\neq\text{j} \end{cases} \\ \epsilon & := & \langle o,\, \alpha,\, \sigma,\, \varsigma\rangle \end{array}$$

DEFINITION 3. *A **repair action** (τ) defines a change of a model element property that resolves an inconsistency in part or full (often multiple repair actions are needed to resolve an inconsistency). A repair action contains the type of change (λ), the model element (ω), and the model element property (γ) that is affected by a change. The following types of changes are possible: **add** (adds a model element), **delete** (deletes a model element) or **modify** (modifies a model element property).*

$$\begin{array}{lcl} \tau & := & \langle \lambda,\, \omega,\, \gamma\rangle \\ \lambda & \in & \{add,\, delete,\, modify\} \end{array}$$

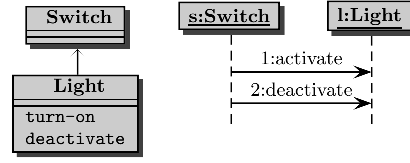

Figure 1: Inconsistent UML Class and Sequence Diagram for a Light Switch

DEFINITION 4. *A **repair tree** (R) is a hierarchical ordered set of repair actions for a single inconsistency. The nodes of a repair tree define whether the underlying repair actions are alternatives (●) or sequences (+) .*

3. ILLUSTRATION AND PROBLEM

To illustrate an inconsistency, Figure 1 introduces a small UML model. The model consists of a class diagram containing a 'Switch' and a 'Light' class. The sequence diagram reveals that the 'Switch' calls 'activate' and 'deactivate' on the 'Light', to turn on or turn off the light.

Design rule 1 is an example of a non-trivial UML design rule. It validates whether a given message in a sequence diagram matches the direction of the class association (1.1) and whether that class has an operation that matches the message name (1.2). Design rules are typically written from the perspective of a context element. In the case of design rule (1) the context is the UML type 'Message'.

$$\begin{array}{l} Message\; m : \\ \quad \left.\begin{array}{l} (\exists l_1 \in m.receiveEvent.covered, \\ \quad l_2 \in m.sendEvent.covered : \\ \exists a \in l_2.represents.type.ownedAttribute : \\ a \neq null \Rightarrow a.type = l_1.represents.type) \end{array}\right\} (1.1) \\ \qquad\qquad\qquad \wedge \\ \quad \left.\begin{array}{l} (\forall l \in m.receiveEvent.covered : \\ \exists o \in l.represents.type.ownedOperation : \\ o.name = m.name) \end{array}\right\} (1.2) \end{array} \quad (1)$$

The condition of design rule (1) is a Boolean expression that is validated on the design model. The rule reveals the sequence in which model elements are accessed and the specific properties are used. For example, rule 1 is a conjunction $\wedge$ with the first argument being an universal quantifier ($\forall$) and the second argument an existential quantifier ($\exists$). The universal quantifier iterates over a set of lifelines l. These lifelines are found by first validating the 'receiveEvent' property of the 'Message m', which references another model element of type 'MessageOccurenceSpecification'; and then by validating the 'covered' property of that 'MessageOccurenceSpecification', which references a model element of type 'Lifeline'. Details on the model elements and their properties can be found in [2]. The universal quantifier then iterates over all lifelines found to ensure that all lifelines satisfy the quantifier condition. This condition is another existential quantifier ($\exists o \in l.represents.type.ownedOperation$).

Each validation starts at a context element ('Message m' in this case) which means that the context element is always the first element accessed during the validation of a design rule. There are thus as many validations necessary as there are instances of the context element (instances of 'Message' in this case). So this design rule will be validated twice on the model shown in Figure 1 – for message 'activate' and for message 'deactivate'. Both validations are inconsistent

because neither message matches the direction of the class association (only class ‘Light‘ can access ‘Switch‘, contrary to the message call directions in the sequence diagram). Also note that the validation on the message ‘activate‘ fails because the class ‘Light‘ has no operation ‘activate‘. Both validations investigate the same design rule but because they commence at distinct model elements (method ‘activate‘ vs. ‘deactivate‘) they do not necessarily access the same model elements or yield the same validation result. Each validation thus accesses a potentially overlapping but unique part of the model. If the validation of a design rule fails (=inconsistent) then this accessed part of the model must contain an error. We say that part of the model to have caused the inconsistency. In previous work [9], it was demonstrated that often a small set of model elements causes a given inconsistency. Nonetheless, there are many choices for repairing inconsistencies because each combination of the accessed model elements could be changed – an exponential explosion. For example, to repair the inconsistent message ‘activate‘, we could add an operation ‘activate‘ to class ‘Light‘ and add a association from ‘Switch‘ to ‘Light‘ (a sequence of two repair actions); or we could rename operation ‘deactivate‘ to ‘activate‘ and change the arrowhead of the existing association between ‘Light‘ and ‘Switch‘; or we could perform variations of these and other repair actions.

Enumerating all repair alternatives would yield a list of correct choices from which the designer must choose one. However, enumerating all repair alternatives would likely overwhelm the designer. For example, during the validation of ‘activate‘, ten model element properties are accessed. By considering their combinations there are 1,023 repair alternatives (comparable with the cardinality of the power set of involved model elements minus the empty set: $|P(n) - \emptyset| = 2^n - 1$, where n is the number of model elements involved). In practice, some of these combinations are infeasible but this does not alter the exponential complexity. We will show later that by customizing the repair to the cause of this particular inconsistency, we can correctly eliminate most wrong repairs and non-minimal repairs. This reduces the repair alternatives to 72 combinations. We further organized these combinations into a repair tree containing 17 repair actions, that asks the designer to make two decisions with eight and nine choices, respectively.

We believe that by considering the immediate cause of an inconsistency as well as the hierarchical representation of the generated repairs, we now have the means of providing comprehensive error feedback in small enough chunks for a designer to comprehend. Additionally, the incremental character of this approach ensures that repair alternatives are available to the designer on demand.

4. APPROACH

The following introduces our approach for generating repair trees for inconsistencies. Our approach generates these repairs on demand and the repairs are tailored to the structure of the inconsistent design rule condition.

4.1 Principle

Consider the design rule $A \vee (B \wedge C)$ where A, B, and C are Boolean expressions. If inconsistent, any combination of these three Boolean expressions may need repairing. The repair alternatives are: $\{A\}$, $\{B\}$, $\{C\}$, $\{A, B\}$, $\{A, C\}$, $\{B, C\}$, or $\{A, B, C\}$. For example, repair $\{B\}$ suggests a way of changing the model elements referred to by B such that validating B becomes true. Recall that design rule (1) was essentially a conjunction (like $B \wedge C$). Hence, repair $\{B\}$ could imply repairing one argument of that conjunction (e. g., $\forall l \in m.receiveEvent.covered...$) which may require repairing one or more model elements accessed by that argument. A repair alternative may contain sequences of repair actions (e. g., repair $\{B, C\}$ is a sequence of repair $\{B\}$ and repair $\{C\}$) and repair alternatives may overlap in their repair actions (e. g., repair $\{C\}$ overlaps with repair $\{B, C\}$).

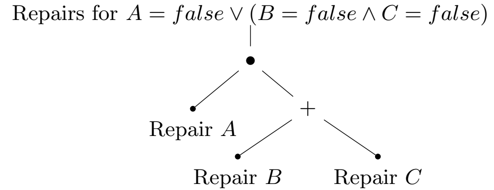

Figure 2: Repair Tree for $A = false \vee (B = false \wedge C = false)$

However, not all repair alternatives identified above are necessary. For example, repair $\{A\}$ would make $A \vee (B \wedge C)$ consistent regardless of repairs to $\{B\}$ or $\{C\}$. Hence, by considering the structure of the design rule condition we can eliminate unnecessary combinations of repair actions, such as repair $\{A, B\}$, $\{A, C\}$ and $\{A, B, C\}$. The list of still complete repair alternatives for this design rule shrinks to: $\{A\}$, $\{B\}$, $\{C\}$, or $\{B, C\}$. The approach by Nentwich et al. [18] analyzes the design rule's structure to compute repair alternatives and actions as in this example. This list applies to the design rule as a whole and thus is applicable to all inconsistencies of this design rule. However, this list is not yet customized to a particular inconsistency. While more concise, there are still incorrect repair alternatives because we did not consider how inconsistencies are caused. Let us assume that B is true already while A and C are false. $A \vee (B \wedge C)$ would be inconsistent but, in this particular scenario, only two repair alternatives out of the four above are feasible: repair $\{A\}$ or repair $\{C\}$. Repair $\{B\}$ is an incorrect repair because for as long as C is false it does not matter how B is changed. And repair $\{B, C\}$ is a non minimal repair because it forces the designer to also repair $\{B\}$ even though it is not broken.

It is important to understand that the repair alternatives for any two inconsistencies may differ. For example, if all three expressions A, B and C were false then the correct list of repair alternatives would be: repair $\{A\}$ or repair $\{B, C\}$. In fact, there is no combination of A, B and C being true or false that would require all four repair alternatives $\{A\}$, $\{B\}$, $\{C\}$, and $\{B, C\}$. We found that approaches that rely on a static analysis of the design rule condition only are rarely minimal. By focusing on the violated parts of an inconsistent design rule condition we are able to reduce the list of repair alternatives. However, the list of repair alternatives may still be large. We thus propose to describe repair alternatives in form of sequences (*and*) and alternatives (*or*): repair A *or* (repair $\{B\}$ *and* repair $\{C\}$) for $A = B = C = false$. These sequences and alternatives can be visualized as a *repair tree,* where the sequences are + nodes and alternatives are • nodes (see Figure 2).

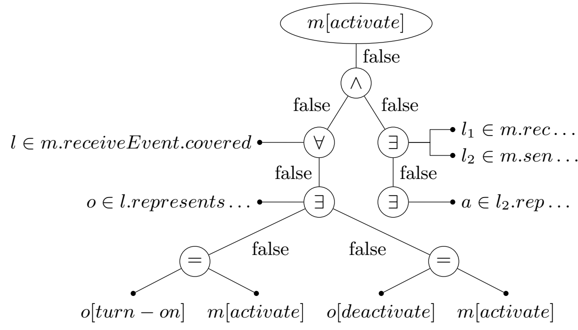

Figure 3: Validation Tree log the Validation of Design Rule (1) for Message 'activate'

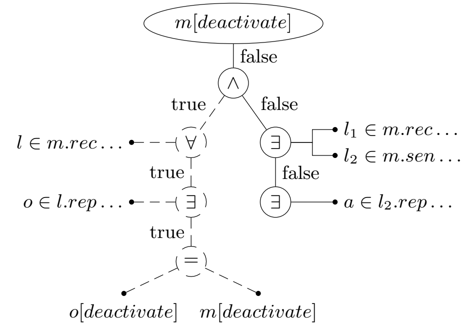

Figure 4: Validation Tree log the Validation of Design Rule (1) for Message 'deactivate'

This example of a repair tree may not appear shorter compared to the list of all repair alternatives but we will demonstrate that the repair tree grows linearly with design rule size avoiding the exponential growth of repair alternatives. Moreover, we believe that the repair tree is more informative to the designer because it is structured instead of a flat list.

4.2 Understanding the Inconsistency

Understanding an inconsistency is needed for eliminating incorrect repairs and non-minimal repairs. Our approach bases this understanding on the rule's validation and its intermediate validation results. Technically, this is achieved by instrumenting the consistency checker and logging step-by-step the design rule's validation. We refer to this log as the validation tree which contains all operations performed, their results, and the model elements accessed along the way (leaves). Computing a validation tree is analogous to **profiling**. Code profiling, a well known technique in code understanding [16, 12], logs the executing code (usually in a hierarchy that reflects method calls). Design rule profiling logs the validation of a design rule in the hierarchy that reflects the structure of the design rule (e. g., its syntax).

Figure 3 depicts the validation of rule 1 on message 'activate' as a validation tree. Recall from Section 3 that the condition of design rule (1) was a conjunction ($\wedge$) with the first argument ($\forall l \in m.rec...$) validating whether the message name has a corresponding operation name and the second argument ($\exists l_1 \in m.rec...$) validating whether the calling direction matches the message direction. The consistency checker thus first validated the conjunction and hence the top level node in the validation tree is the $\wedge$ node. To validate the conjunction, the consistency checker then validates its two arguments. The two sub nodes in the tree correspond to the two arguments – a $\forall$ node and a $\exists$ node. The first argument was a universal quantifier ($\forall$). The elements this quantifier iterated over (the source) were identified by the first argument of that quantifier ($l \in m.rec...$).

The consistency checker found a list of lifelines upon validating this source. The quantifier then validated the quantifier's condition on every lifeline found ($\exists o \in l.rep...$). The validation tree thus depicts the source as a leaf (leaves are model elements accessed) and each validation of a lifeline it depicts as a separate sub tree. Hence, a quantifier node may have a variable number of sub nodes: each describing the validation of the condition of each element in the source. Since in our example, the source identifies a single lifeline ($l : Light$), only one sub tree is needed to describe its validation on the quantifier condition in Figure 3 (starting with the $\exists$ node beneath the $\forall$ node). The existential quantifier is similar to the universal quantifier. Analogous to the universal quantifier, the existential quantifier defines its source ($o \in l.rep...$), a list of all operations owned by the lifeline's type (class 'Light'), and a condition that at least one element in the source must satisfy.

There are two UML operations in the class 'Light' (operations 'turn-on' and 'deactivate'). The existential quantifier then iterates over both operations to identify whether the operation name equals the message name. Hence, the existential quantifier node in the validation tree has a leaf denoting the source and two sub trees, one for validating each operation. Since both sub trees evaluate the same quantifier condition (comparing an operation name with a message name: $o.name = m.name$) it follows that both sub trees are identical in structure. Both sub trees thus have a top-level comparison node (=) and their respective sub nodes are the arguments of the comparison. In both cases, the arguments identify properties of model elements. For example, the first sub tree compares (=) the operation name 'turn-on' with the message name 'activate'. Hence, the validation tree depicts a = node with two leaves corresponding to the two model elements. For brevity, the validation tree omits certain data. For example, the leaf o[turn-on] uniquely identifies the model element (via the element ID), the element type, and the property name (e. g., $\langle A3242ED22, UML.Operation, name\rangle$).

The validation tree of the second argument of the top-level $\wedge$ node is similar, containing two existential quantifiers. The implication and two comparisons are not depicted in the validation tree because no attribute is found in the second existential quantifier that iterates over the source $a \in l_2.represents.type.ownedAttribute$ (note in the UML spec, outgoing associations are also attributes and class 'Light' has neither outgoing associations nor attributes). Hence, the bottom right $\exists$ node has a source but no sub trees.

It is easy to see that the validation tree mirrors exactly the validation of the design rule condition. We instrumented the consistency checker to be able to automatically monitor the evaluation step-by-step and record this information. It is important to note that the leaves of the validation tree are either constants defined in the design rule or properties of model elements accessed during the validation. Some or

all of these properties of model elements need to be changed to repair the inconsistency.

The edges between Boolean expressions in the validation tree list the validation results. A validated result ς is the result of the validated node using its sub nodes as arguments (α). For example, both = nodes are false because the strings they compare are not equal; the $\exists$ node above it is false because no sub node was true; the $\forall$ node above it is false because at least one sub node was false; and, finally, the $\wedge$ node above it is false because both arguments (including the left-handed $\forall$) are false. The validated results ς are thus simply the logged, intermediate results computed by the consistency checker during validation which are also automatically retrieved through the instrumentation of the consistency checker. We see that all nodes in Figure 3 validated to false.

As discussed earlier, the model elements accessed by an inconsistency are unique for every validation of a design rule. This is evident in the validation of design rule (1) on message 'deactivate' which yields a structurally similar validation tree (Figure 4) however with somewhat different model elements accessed and different validation results for some nodes. The left hand $\forall$ nodes and edges in Figure 4 are painted in a dashed style because this part of the tree is not violated. How to compute this is discussed below.

4.3 Generating Repair Actions

A repair action (τ) is a change to the model. This change must modify a property (ω) of a model element (γ), insert elements or values, or deletes them. Repair actions repair individual nodes in the validation tree. We will see below how to compose repair actions into a repair tree by combining the repair actions with the nodes of a tree. Since the repair actions differ according to the node operation, Table 1 discusses how to compute complete and minimal repair actions for common first order logic operations. Complete in the sense that it lists all possible variants of how to resolve an inconsistency based on the information from the design rule validation and minimal in that it avoids the generation of repair actions that do not have an effect for the inconsistency resolution.

For brevity, the table shows an excerpt of some common operations only but our tool supports a much larger set of operations (e. g., all of OCL). The first column of Table 1 lists the operations and the repair generator function used to reference it. The second column specifies the arguments of the operations and the third column lists the repair actions that need to be generated. Since the repair of a logical operation may require the repair of arguments of the logical operation, the repair generator functions are defined recursively.

Every operation has its own repair generator function $G(\epsilon, \sigma)$. This function has two parameters: ϵ is the expression containing a logical operation and arguments (e. g., $a \wedge b$) and σ is the expected validation result (Boolean). An expression is inconsistent and must be repaired if its expected result differs from the validated result (recall Section 4.2). The generator functions are thus guarded by conditions involving the expected result σ and the validation results ς observed.

Computing the expected result is simple. The expected result of the top node in the validation tree is always true because the design rule condition is expected to validate to true to be consistent. This expectation is propagated down to the sub nodes. In Figure 3, the top level $\wedge$ node is thus expected to be true and correspondingly its arguments are expected to be true also. Certain operations change the expected result. In the case of the negation ($\neg$), we see in Table 1 how the expected result is negated. Notice that the repair action for $G(\neg a, \sigma)$ results in the repair action for its argument with a negated expected value: $G(a, \neg\sigma)$. Since some other approaches fail on the negation [24] or their reasoning becomes less effective [18], our simple handling of expectations represents another improvement over the state-of-the-art which simplifies the repair generation.

Computing the repair alternatives for any given logical operation is more complex. Take, for example, the repairing of $a \wedge b$ which is described in the generator function $G(a \wedge b, \sigma)$ in Table 1. For simplicity, Table 2 enumerates the possible repair actions for the various Boolean states of the expected result σ, and for the validation results ς. If $a \wedge b$ is false ($\varsigma = false$) but it is expected to be true ($\sigma = true$), then there are three possible repair alternatives: 1) repair a, 2) repair b, or 3) repair both $a + b$. To identify the appropriate one, these repair alternatives are guarded. For example, if it is known that $a = true$ (denoted as $\varsigma_a = t$) and $b = false$ (denoted as $\varsigma_b = f$) and the expected result $\sigma = true$ then repair a is the right choice (denoted as $G(a, \sigma)$. The repair for a is then computed recursively by calling the generator function on a with the expected result $\sigma = true$ – the expected result of its parent node. However, if $a \wedge b$ is expected to validate to *false* then the repair alternatives for $a \wedge b$ change because either a or b need to be false for $a \wedge b = false$. Quite relevant are the + symbol, which denotes a sequence (repair a and repair b), and the • symbol, which denotes an alternative (repair a or repair b).

$$G(\epsilon, \sigma) \quad : \quad f(\epsilon, \sigma) \rightarrow \bigcup^{\bullet/+} \tau$$
$$R \quad := \quad \bigcup G(\epsilon_0, t)$$

Since the top level expression of the design rule (1) is a conjunction ($\wedge$), (see Figure 4) a corresponds to its first argument (branch) and is a placeholder for the universal quantifier ($\forall$). Repair a thus really means to repair the $\forall$ expression underneath in the validation tree. In our example, both arguments are false and hence we need to repair a and repair b. To understand the repair of an iterator, repair a is illustrated next.

The $\forall$ expression in the validation tree is divided into two parts: 1) the repair of the source (deletion/addition depending on the operation type and expected result), and 2) the repair of one or more violated conditions which where validated on the elements in the source. For example, the universal quantifier iterates over all elements of the source and validates each one with the given condition. The universal quantifier is false if there exists at least one element that violates the condition. If the universal quantifier is expected to validate to true (as in our example) then the deletion of an element in the source can repair it. This is encoded as deleting the union of all elements that violate the condition σ of the quantifier as depicted in Table 1 in column 3 (top). The second alternative (column 3 bottom) is to resolve the inconsistency by repairing the elements that violate the conditions rather than deleting them. This requires a recursive descend for each argument $b(a)$ that violates σ. In other words, we would then have to repair the sub trees that cor-

Table 1: Excerpt of Rules for the Generation of the Repairs

$G(\epsilon, \sigma)$	α	R
$\epsilon := \neg a$	$a : Boolean$	$R = \{\ G(a, \neg\sigma)$
$\epsilon := a \wedge b^*$	$a : Boolean$ $b : Boolean$	$R = \begin{cases} G(b, \sigma) & \text{if } \sigma = t,\ \varsigma_a = t,\ \varsigma_b = f \\ G(a, \sigma) & \text{if } \sigma = t,\ \varsigma_a = f,\ \varsigma_b = t \\ G(a, \sigma) + G(b, \sigma) & \text{if } \sigma = t,\ \varsigma_a = f,\ \varsigma_b = f \\ G(a, \sigma) \bullet G(b, \sigma) & \text{if } \sigma = f,\ \varsigma_a = t,\ \varsigma_b = t \end{cases}$
$\epsilon := a \vee b$	$a : Boolean$ $b : Boolean$	$R = \begin{cases} G(a, \sigma) \bullet G(b, \sigma) & \text{if } \sigma = t,\ \varsigma_a = f,\ \varsigma_b = f \\ G(a, \sigma) & \text{if } \sigma = f,\ \varsigma_a = t,\ \varsigma_b = f \\ G(b, \sigma) & \text{if } \sigma = f,\ \varsigma_a = f,\ \varsigma_b = t \\ G(a, \sigma) + G(b, \sigma) & \text{if } \sigma = f,\ \varsigma_a = t,\ \varsigma_b = t \end{cases}$
$\epsilon := a \Rightarrow b^*$	$a : Boolean$ $b : Boolean$	$R = \begin{cases} G(a, \sigma) \bullet G(b, \sigma) & \text{if } \sigma = t,\ \varsigma_a = t,\ \varsigma_b = f \\ G(b, \sigma) & \text{if } \sigma = f,\ \varsigma_a = t,\ \varsigma_b = t \\ G(a, \sigma) + G(b, \sigma) & \text{if } \sigma = f,\ \varsigma_a = f,\ \varsigma_b = t \\ G(a, \sigma) & \text{if } \sigma = f,\ \varsigma_a = f,\ \varsigma_b = f \end{cases}$
$\epsilon := a = b$	$a : Boolean$ $b : Boolean$	$R = \begin{cases} \tau = \langle modify,\ a.element,\ a.property\rangle & \text{if } \varsigma_b = const \\ \tau = \langle modify,\ b.element,\ b.property\rangle & \text{if } \varsigma_a = const \\ \tau = \begin{cases} \langle modify,\ a.element,\ a.property\rangle \\ \bullet \\ \langle modify,\ b.element,\ b.property\rangle \end{cases} & \text{else} \end{cases}$
$\epsilon := \forall a : b^*$	$a : Set$ $b : Boolean$	$R = \begin{cases} \tau = \begin{cases} +\bigcup_{i=1}^{n} \langle delete,\ a.element,\ a.property\rangle \mid \varsigma_{bi} = f\rangle \\ \bullet \\ +\bigcup_{i=1}^{n} G(b_i,\ \sigma) \mid \varsigma_{bi} = f \end{cases} & \text{if } \sigma = t \\ \tau = \begin{cases} \langle add,\ a.element,\ a.property\rangle \mid \varsigma_{bi} = f\rangle \\ \bullet \\ \bullet \bigcup G(b_i,\ \sigma) \mid \varsigma_{bi} = t \end{cases} & \text{if } \sigma = f \end{cases}$
$\epsilon := \exists a : b$	$a : Set$ $b : Boolean$	$R = \begin{cases} \tau = \begin{cases} \langle add,\ a.element,\ a.property\rangle \mid \varsigma = t\rangle \\ \bullet \\ \bullet \bigcup G(b_i,\ \sigma) \mid \varsigma_{bi} = f \end{cases} & \text{if } \sigma = t \\ \tau = \begin{cases} +\bigcup_{i=1}^{n} \langle delete,\ a.element,\ a.property\rangle \mid \varsigma_{bi} = t\rangle \\ \bullet \\ +\bigcup_{i=1}^{n} G(b_i,\ \sigma) \mid \varsigma_{bi} = t \end{cases} & \text{if } \sigma = f \end{cases}$
$\epsilon := a.b$	$a : Element$ $b : Property$	$R = \{ \tau = \langle modify,\ a,\ b\rangle$

* This expressions are represented by combining the other existing ones using rules from first order logic

Table 2: Repair Alternatives for $a \wedge b$ depending on Expected and Validated Results

#	a	b	σ	$\varsigma = a \wedge b$	R
1	$true$	$false$	$true$	$false$	$\{b\}$
2	$false$	$true$	$true$	$false$	$\{a\}$
3	$false$	$false$	$true$	$false$	$\{a + b\}$
4	$true$	$true$	$false$	$true$	$\{a \bullet b\}$

respond to those elements that violated the quantifier. The other generator functions are similar to the ones discussed above and not discussed for brevity.

4.4 Composing Repair Trees

Figure 5 depicts the repair tree for the inconsistent message 'activate'. It is computed if the generator functions from Table 1 are applied to the validation tree in Figure 3. The recursive descend described in Section 4.3 thus follows the structure of the validation tree step-by-step. The repair tree arranges the sequential repair actions (+) and alternative repair actions (•) from Table 1 according to the hierarchy of the validation tree. For example, we see that both branches of the $\wedge$ node in the validation tree (Figure 3) are violated and hence both branches need repairing. According to Table 1 this requires a sequential repair (+) – hence the top level node of the repair tree is a +. For each branch, the repairs are computed next for which there are eight repair alternatives each (•) according to Table 1 for existential/universal quantifiers. Each quantifier adds an alternative for each property call of the source ('.') and depending on the quantifier an alternative ($\exists$) or combination ($\forall$ – in our case this quantifier has only one branch).

Actually repairs are the model elements encountered in the validation tree during the recursive descend. We find a total of 17 repair actions in the repair tree (Figure 5) for the validation tree in Figure 3. For example, $\langle add, Class[light], ownedOperation\rangle$ is added to the repair tree during the repair of the left-hand existential node where Table 1 suggests to add an element to the source ($\exists o \in l.rep...$). The UML spec reveals that the source may only hold instances of UML type *Operation* and hence adding another operation to the class 'Light' may repair this inconsistency. Note that the repair action does not say whether this operation should be created anew or should come from elsewhere (e. g., another class). Nentwich et al. [18] refer to such repair actions as abstract repairs. The repair tree also identifies alternative repair actions such as $\langle modify, Operation[turn-on], name\rangle$ which suggests to rename operation 'turn-on'. This repair action is added during the repair of the first = node where Table 1 suggests to modify either the first or second argument. Again note that the repair action does not reveal what string to rename it to. Indeed, any existing approaches for computing concrete repairs may be used to complement our repair trees with concrete values and this will be the focus of our future work.

From this repair tree, it is possible to compute repair alternatives. The designer knows that at least one repair action from each branch must be chosen. There are thus two decisions to be made, one with eight and one with nine choices, accounting for 72 distinct repair alternatives. However, not all combinations in the repair tree make sense. For example, deleting the lifeline 'light' (done by $\langle delete, Message[activate], covered\rangle$) would make it impossible to change the type of that lifeline ($\langle modify, Lifeline[light], type\rangle$). The filtering of these combinations is simply done by detecting the overlaps of the model elements and their properties. Therefore, two actions that have the model element and property in common or if the value if the property of one action is the model element of the other action (e. g., the Lifeline 'light' is the value of the message 'activate' covered property), regardless of the action type, cannot be composed. The filtering reduces the number of repair alternatives from 72 down to 39:

$$\nexists \tau_i, \tau_j \in R \quad : \quad \tau_i + \tau_j \wedge (\omega_i = \omega_j \wedge \gamma_i = \gamma_j \vee \omega_j \in \gamma_i)$$

We say that a sub tree did not cause the inconsistency if its validation result equals the expected result. The repairing is thus a recursive descend that terminates either at nodes where the expected value equals the validated value or at their leaves which are the model elements that need repairing. In Figure 4 we painted the part of the sub tree that did not cause the inconsistency in a dashed style. Design rule (1) validated on 'deactivate' was thus partially inconsistent because only the validation of the message direction failed but not the validation of the operation name – contrary to design rule (1) validated on 'activate', which was fully inconsistent. Figure 6 depicts the corresponding repair tree for the inconsistent message 'deactivate'. This repair tree is not only different in structure but the repair actions also refer to partially different model elements. We see that there is only one decision to be made, involving nine choices, accounting for nine distinct repair alternatives. Note that the repair tree for the message 'activate' (Figure 5) is flattened because nested alternatives can be flattened. Notice in the repair tree for the message 'deactivate' (Figure 6), we chose not to flatten the tree, depicting a hierarchy of two alternatives that are the result of the two nested existential quantifiers in the design rule.

4.5 Identifying Side Effects

Note that the example $A \vee (B \wedge C)$ in Section 4.1 ignored side effects in that B and C may in fact access overlapping model elements and repair B could conceivably also repair C. It is a well known problem that repairs to inconsistencies may affect other design rules – they may solve other inconsistencies or they may cause other inconsistencies. Thus, repairs should not be seen in isolation [19]. Understanding side effects is important during the repairing of inconsistencies to get an overview of the overall effect of a repair action on the model. In our approach, potential side effects are readily computable by investigating which repair actions are shared among the repair trees of different inconsistencies. Comparing the repair trees in Figure 5 and Figure 6, we find that both share the repair action $\langle add, Class[light], ownedAttribute\rangle$. This repair action suggests adding an attribute to class 'Light' which has the potential to (partially) solve both inconsistencies. The potential

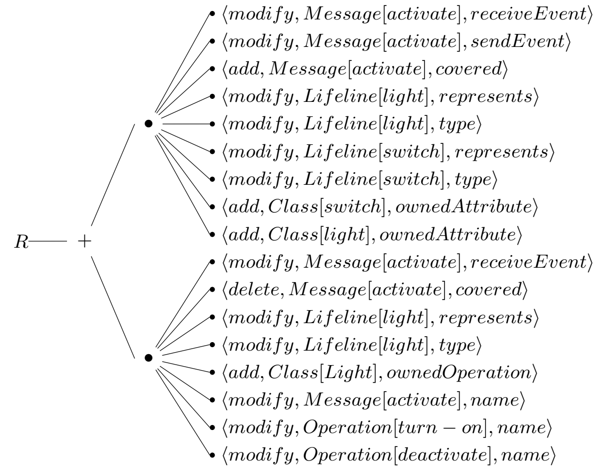

Figure 5: Repair Tree for the Inconsistent Message 'activate'

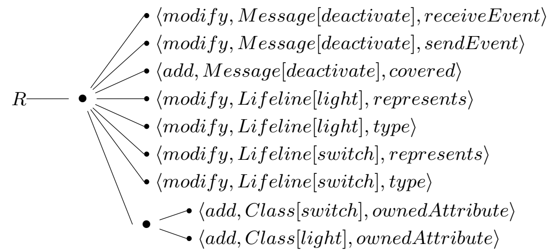

Figure 6: Repair Tree for the Inconsistent Message 'deactivate'

side effect is conservative as it identifies all possible effects (the worst case). The conservative nature is useful for the repairing of inconsistencies to understand all potential effects the repair action may have. Side effects not only exist among design rules but may also exist within design rules. Indeed, looking at the repair tree for the 'activate' inconsistency (Figure 5) we find that the repair action $\langle modify, Message[activate], receiveEvent\rangle$ appears on both sides of the conjunction. This single repair action may thus repair the entire inconsistency.

The above examples indicate positive side effects in that one repair may also repair other inconsistencies (or their parts). Side effects may also be negative in that one repair may cause an inconsistency where there is none. Such potential side effects are also detectable by checking whether a repair action of a repair tree references a model element in the validation scope (which is needed and used for the incremental re-validation of a design rule) of a consistent design rule. This was already discussed in [11] and is not explored here further.

4.6 Model/Analyzer Tool

The approach is fully automated and tool supported in the Model/Analyzer plug-in for the IBM Rational Software Architect (RSA). The tool uses a generic language, called *abstract rule language* (ARL), as the constraint language and it is possible to map arbitrary constraints languages to ARL (e. g., OCL, Beanbag, or xLinkIt). Another motivation in favor of ARL is that it provides a quite reduced set of operations and the mapping from a given constraint language, say OCL, combines these operations. For example, an op-

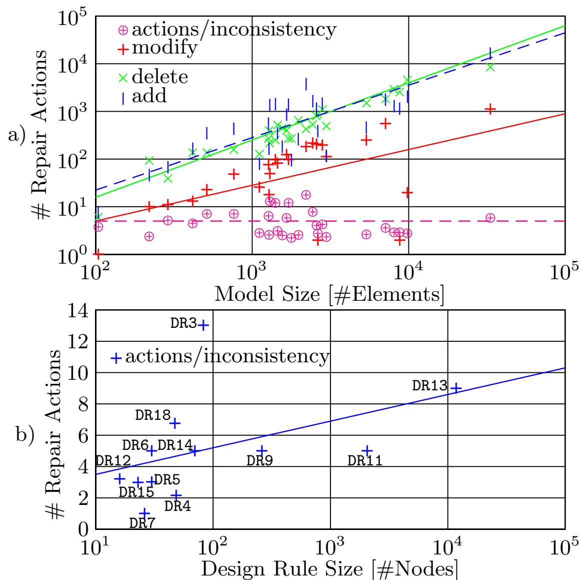

Figure 7: Repair Actions depending on the a) Model Size, b) Design Rule Size

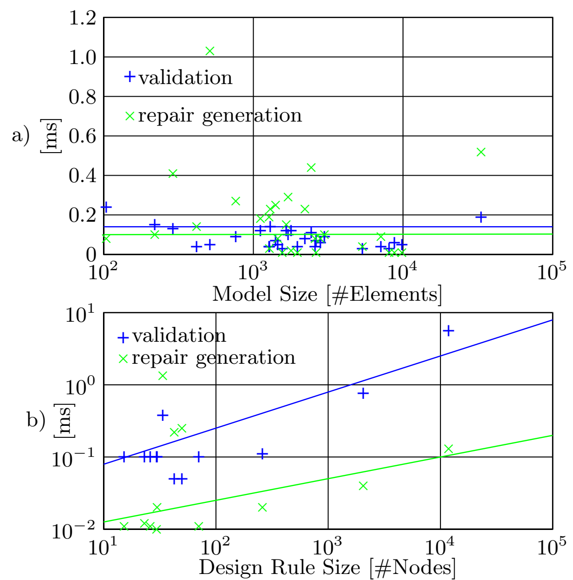

Figure 8: Computation Time depending on the a) Model Size, b) Design Rule Size

eration like ⇒ (implies) is represented in ARL as $\neg A \vee B$. This simplifies the repair language discussed earlier, yet, it must be stressed that from the designer's perspective, ARL is invisible (i. e., the design writes the design rules in, for example, OCL and the constraint is converted internally to ARL). The tool may be downloaded from `http://www.sea.jku.at/tools/`.

5. EVALUATION

We evaluated our approach on 29 mostly industrial UML design models (ranging from 104 up to 33,347 model elements) and 18 design rules written in OCL (all of them are characterized in [10] however do note differences in metrics due to UML 1.4 to UML 2.1 conversions and the fact that the design rules were rewritten from JAVA to OCL). In all, repair actions for over 6,560 inconsistencies were evaluated.

In Section 4.3 we demonstrated that repairs are generated out of the validation tree. Since all violated parts of the validation tree are considered for the generation of the repair actions, it follows that there are no other model elements which can be changed to repair an inconsistency (completeness). Since the recursive traversal of the validation tree investigates all elements involved in a violation implies that there are no actions included in a repair alternative that do not have the potential to resolve an inconsistency (minimal). That is as the filtering of the validation tree deletes all those branches that are not violated (i. e., these branches cannot cause the inconsistency) and as such no repair actions will be generated that may change model elements that are at the leaves of those branches. This informal proof of correctness was additionally confirmed empirically by rechecking the repair alternatives and their repair actions which is discussed next.

Our evaluation shows that the repair alternatives for the detected inconsistencies consist of at least one action and for each inconsistency at least one repair alternative has been generated. This demonstrates general applicability. Figure 7 a) shows the number of generated actions depending on the model size. The total number of repair actions for each model is split up into the three types of actions: add, delete and modify actions. Additionally, we show the actions per inconsistency. While the number of repair actions (alternatives) increases linear with the model size (like the number of inconsistencies), the number of repair actions per inconsistency remains stable in average. Since the repair actions are the nodes of the repair tree, this implies that the repair tree size remains constant with regard to design model size (note the logarithmic scale of both axes).

In Figure 7 b) the average number of repair actions per inconsistency is shown in relationship to the average size of the design rules (the size of a design rule was measured as the number of nodes in its validation tree). Only those design rules are shown in Figure 7b) and later in Figure 8b) that are inconsistent. As this figure shows, the number of repair actions increases logarithmically (less than linear; note the logarithmic scale of the x-axis only). The repair trees have between one and thirteen repair actions per inconsistency and design rule in average. Given that the repair trees are typically structured, their sequences and alternatives individually are small enough in size to match the humans' cognitive abilities (Miller's 7 +/- 2).

To show the usability in regard to the response time of our approach we evaluated the scalability as factor for the model size and the complexity of the design rules. The evaluations were done on an Intel Core 2 Quad CPU @2.83GHz with 8GB (3.5GB available for the RSA) RAM and 64bit Linux (2.6.37). We measured the time it takes to validate the design rules, i. e., to detect an inconsistency. Furthermore, we measured the time it takes to calculate the repair trees and their side effects for all 6,560 inconsistencies.

Figure 8 a) shows the average times it takes to validate the design rules and to calculate the repair actions and their side effects. Each spot represents the average time over all design rules in the model. As can be seen, the computation time per inconsistency is independent of the model size with 99% of the validations, repair tree/side effect computations done in less than 1ms. The time to calculate the repair trees and side effects also includes the generation of the validation tree for

design rules that are not violated but needed for computing the side effects of repair actions. In addition to the average times, we also evaluated the maximum times which is about 300ms for the validation and 500ms for the repair and side effect generation. Our approach is thus highly scalable.

Regarding the design rule size, the situation is quite similar. Figure 8 b) shows the times for validation and computing repair tree and side effect depending on the design rule size. Each spot in this diagram represents the average of a single design rule validation (total 6,560 inconsistencies/validations). As can be seen, the validation time grows linear (please note the logarithmic scale on both axes) with the design rule size whereas the time to generate the repair trees and their side effects remains nearly constant. This is likely due to the fact that the generation of repair trees and their side effects is a navigation in the validation tree and depends on the depth of the validation tree and not the breadth. The average validation times for design rules are less then 6ms in average and the worst case is 300ms. Moreover, our evaluation shows that the number of design rules to validate decreases with the complexity of the design rule, i. e., the more complex a design rule, the fewer times it tends to be validated. This fact is reflected in Figure 8 a) where the average time to validate a design rule is much less than the validation depending on the design rule's size.

The memory cost is linear to the design rule complexity and negligible because the validation tree may be discarded immediately after the computation of the repair tree.

6. RELATED WORK

Document management systems (DMS), check the consistency of interrelated documents that are processed by a team of authors. For example, Scheffczyk et al. [22] use s-DAGs [1] to represent the documents and the design rules. Repairs for inconsistent documents are derived from the s-DAG representation and not from the documents. Heuristics are used to eliminate unnecessary repairs. As this might be useful in the field of DMS this is not useful for model-based software development, because each software project has different requirements and so no generic heuristics can be derived that are applicable for all software projects. Furthermore, the generation process of repairs is independent of the inconsistency detection process and it is a multilevel process that results in increasing time consumption. With xLinkIt [17] Nentwich et al. provide an incremental approach to evaluate the consistency of arbitrary XML documents. They also presented a repair framework for inconsistent distributed documents [18]. However, as was discussed earlier, their approach is conservative and may suggest non-minimal and even incorrect repair actions if the design rule is partially violated only. Nonetheless, we consider this work as a foundation to our work. Table 1 is clearly based on their principles.

Xiong et al. [24] present an approach that combines the detection of errors and provides actions to repair them on UML models. They use their own language to define the consistency relations. This language, called *Beanbag*, has an OCL-like syntax and provides a fixing semantic for elements that are changed. However, when writing consistency relations, the designer also has to specify how this relation has to be fixed when it is violated – a manual and error prone activity without guarantee for completeness or correctness. Dam [7] analyzed and developed an approach on how OCL constraints, based on their internal structure, can be violated or resolved respectively. He distinguishes five different actions that can be taken to achieve a violation or resolution. Abstract repair plans are generated at compile time, i. e., the set of OCL constraints is statically defined in the tool, and this abstract actions are instantiated if the constraint is violated by the model. The repair plans that resolve the inconsistency are ranked and provided to the user who decides which plan to execute. The repair plans can also be modified or executed partially. This approach is designed exclusively for OCL and a proof is given that this approach is correct and complete regarding single OCL operations. However, in contrast to [18], this approach considers all inconsistencies at once which is both a scalability problem as recognized by the authors nor necessarily in the spirit of tolerating inconsistencies [4].

Van Der Straeten [23] use a knowledge base, expressed as description logic as well as the query and rule language nRQL to generate repairs for inconsistent models. The inconsistencies are detected by nRQL queries where the variables of these queries are bound to model elements. The resolutions are represented as nRQL rules that consist of statements that add or remove data from the model to resolve the inconsistency. This approach also considers all inconsistencies at once and generates a set of repair actions that transform the model from an inconsistent state to a fully consistent one – if a solution exist. As this approach must transform the model and the inconsistency rules into description logic, it has no incremental characteristic, i. e., the operation is similar to batch based approaches that are very time consuming. Moreover, the same limitation as for [7] apply.

Almeida et al. developed a prolog-based approach [3] that generates repair plans for inconsistencies. These repair plans consist of actions in Praxis notations that are needed to resolve as many inconsistencies as possible causing as few new inconsistencies as possible. As there exists an infinite number of ways to resolve inconsistencies, this approach has a configurable exploration level which reduces the number of repairs with the danger of not resolving the inconsistencies.

An incremental approach for detecting and repairing inconsistencies is presented in [8] and [9]. It uses various languages for the definition of design rules, like C# or Java and is extended to use OCL for the definition of dynamic constraints (design rules) [13] during run time. It does not need any annotations or modifications of existing languages to check the consistency of UML models. Based on this approach, [9] and [11] presents how to repair inconsistencies in models and how the generated choices are evaluated. However, this approach is overly conservative and generates repairs for all model elements accessed by the validation of an inconsistency, while often only a subset thereof causes the inconsistency. Nonetheless, this work borrows extensively from these approaches – particularly, to enable the computation of side effects as an incremental exploration of possible changes imposed by repairs.

Hegedüs et al. present an approach that is based on graph transformations which generates quick fixes for DSMLs (Domain Specific Modeling Languages) [15]. They use a graphical notation to express the model and the constraint. Based on the DSMLs the approach creates quick fixes to resolve an inconsistency using CSP(M) (Constraint Satisfaction Problems over Models). The quick fixes contain as many actions as needed to resolve a given inconsistency.

Furthermore, Czarnecki and Pietroszek [6] use OCL to define well-formedness rules for the verification of feature-based model templates which are analyzed by a SAT solver. While not directly relevant to the fixing problem, the ability to translate constraints requires detailed understanding of constraint semantics which is very relevant in this work.

7. CONCLUSIONS AND FUTURE WORK

This paper presented a novel approach for generating repairs for inconsistencies in software models. Our approach combines the syntactic and dynamic structure of an inconsistent design rule to pinpoint exactly which parts of the inconsistency must be repaired. As a result, our approach generates a concise tree of repair actions that is small, complete and correct. The approach is tool supported and designed to be applicable to arbitrary modeling and constraint languages. We demonstrated through extensive empirical studies that our approach is scalable and not affected by the model and/or design rule size. In future work, we will investigate how to compute concrete repair actions in a scalable manner to compute concrete repair actions and their side effects more comprehensively.

8. ACKNOWLEDGMENTS

This research was funded by the Austrian Science Fund (FWF): P21321-N15

9. REFERENCES

[1] *S-DAGs: Towards Efficient Document Repair Generation*. CCCT 2004, 2004.

[2] Object Management Group - UML. http://www.uml.org/, 2009.

[3] M. Almeida da Silva, A. Mougenot, X. Blanc, and R. Bendraou. Towards Automated Inconsistency Handling in Design Models. In B. Pernici, editor, *Advanced Information Systems Engineering*, volume 6051 of *Lecture Notes in Computer Science*, pages 348–362. Springer Berlin / Heidelberg, 2010.

[4] R. Balzer. Tolerating Inconsistency. In *ICSE*, pages 158–165, 1991.

[5] X. Blanc, I. Mounier, A. Mougenot, and T. Mens. Detecting model inconsistency through operation-based model construction. In *ICSE '08*, pages 511–520, New York, NY, USA, 2008. ACM.

[6] K. Czarnecki and K. Pietroszek. Verifying feature-based model templates against well-formedness OCL constraints. In S. Jarzabek, D. C. Schmidt, and T. L. Veldhuizen, editors, *GPCE*, pages 211–220. ACM, 2006.

[7] H. K. Dam and M. Winikoff. An agent-oriented approach to change propagation in software maintenance. *Autonomous Agents and Multi-Agent Systems*, 23(3):384–452, 2011.

[8] A. Egyed. Instant consistency checking for the UML. In L. J. Osterweil, H. D. Rombach, and M. L. Soffa, editors, *ICSE*, pages 381–390. ACM, 2006.

[9] A. Egyed. Fixing Inconsistencies in UML Design Models. In *ICSE*, pages 292–301, 2007.

[10] A. Egyed. Automatically Detecting and Tracking Inconsistencies in Software Design Models. *IEEE Trans. Software Eng.*, 37(2):188–204, 2011.

[11] A. Egyed, E. Letier, and A. Finkelstein. Generating and Evaluating Choices for Fixing Inconsistencies in UML Design Models. In *ASE*, pages 99–108. IEEE, 2008.

[12] B. Elkarablieh and S. Khurshid. Juzi: a tool for repairing complex data structures. In W. Schäfer, M. B. Dwyer, and V. Gruhn, editors, *ICSE*, pages 855–858. ACM, 2008.

[13] I. Groher, A. Reder, and A. Egyed. Incremental Consistency Checking of Dynamic Constraints. In D. S. Rosenblum and G. Taentzer, editors, *FASE*, volume 6013 of *Lecture Notes in Computer Science*, pages 203–217. Springer, 2010.

[14] J. Grundy, J. Hosking, and W. B. Mugridge. Inconsistency Management for Multiple-View Software Development Environments. *IEEE Transactions on Software Engineering*, 24:960–981, 1998.

[15] A. Hegedus, A. Horvath, I. Rath, M. Branco, and D. Varro. Quick fix generation for DSMLs. In *Visual Languages and Human-Centric Computing (VL/HCC), 2011 IEEE Symposium on*, pages 17–24, sept. 2011.

[16] M. Z. Malik, J. H. Siddiqui, and S. Khurshid. Constraint-Based Program Debugging Using Data Structure Repair. In *ICST*, pages 190–199. IEEE Computer Society, 2011.

[17] C. Nentwich, L. Capra, W. Emmerich, and A. Finkelstein. xlinkit: A Consistency Checking and Smart Link Generation Service. *ACM Trans. Internet Techn.*, 2(2):151–185, 2002.

[18] C. Nentwich, W. Emmerich, and A. Finkelstein. Consistency Management with Repair Actions. In *ICSE*, pages 455–464. IEEE Computer Society, 2003.

[19] A. Nöhrer, A. Reder, and A. Egyed. Positive effects of utilizing relationships between inconsistencies for more effective inconsistency resolution: NIER track. In R. N. Taylor, H. Gall, and N. Medvidovic, editors, *ICSE*, pages 864–867. ACM, 2011.

[20] J. P. Puissant, T. Mens, and R. V. D. Straeten. Resolving Model Inconsistencies with Automated Planning. In *Proceedings of the 3rd Workshop on Living with Inconsistencies in Software Development.*, pages 8–14. CEUR Workshop Proceedings, 2010.

[21] T. Schattkowsky, J. H. Hausmann, and G. Engels. Using UML Activities for System-on-Chip Design and Synthesis. In O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, editors, *MoDELS*, volume 4199 of *Lecture Notes in Computer Science*, pages 737–752. Springer, 2006.

[22] J. Scheffczyk, P. Rödig, U. M. Borghoff, and L. Schmitz. Managing inconsistent repositories via prioritized repairs. In E. V. Munson and J.-Y. Vion-Dury, editors, *ACM Symposium on Document Engineering*, pages 137–146. ACM, 2004.

[23] R. V. D. Straeten and M. D'Hondt. Model refactorings through rule-based inconsistency resolution. In H. Haddad, editor, *SAC*, pages 1210–1217. ACM, 2006.

[24] Y. Xiong, Z. Hu, H. Zhao, H. Song, M. Takeichi, and H. Mei. Supporting automatic model inconsistency fixing. In H. van Vliet and V. Issarny, editors, *ESEC/SIGSOFT FSE*, pages 315–324. ACM, 2009.