Incremental Consistency Checking for Complex Design Rules and Larger Model Changes

Alexander Reder and Alexander Egyed

Johannes Kepler University, Linz, Austria
alexander.{reder,egyed}@jku.at

Abstract. Advances in consistency checking in model-based software development made it possible to detect errors in real-time. However, existing approaches assume that changes come in small quantities and design rules are generally small in scope. Yet activities such as model transformation, re-factoring, model merging, or repairs may cause larger model changes and hence cause performance problems during consistency checking. The goal of this work is to increase the performance of re-validating design rules. This work proposes an automated and tool supported approach that re-validates the affected parts of a design rule only. It was empirical evaluated on 19 design rules and 30 small to large design models and the evaluation shows that the approach improves the computational cost of consistency checking with the gains increasing with the size and complexity of design rules.

Keywords: consistency checking, performance, incremental checking

1 Introduction

Errors in design models range from basic well-formedness problems (e. g., syntactic violations) to more advanced, multi-view inconsistencies. While designers may be willing to tolerate these errors [10,1], the designer should nonetheless be aware of their existence. Fortunately, recent progress on consistency management has demonstrated that modeling tools can be made to detect errors in design models in real time while retaining the free customizability of design rules. Existing approaches, such as the Model/Analyzer [6], re-validate design rules only if they are affected by model changes. Empirical evaluations have shown that such approaches are very fast: they can validate the impact of a design change in milliseconds in average with the performance being unaffected by the model size.

However, the performance of state-of-the-art incremental consistency checkers decreases with 1) the quantity of model changes, 2) the complexity of design rules, and 3) the number of design rules. For example, existing approaches assume that models change in small increments only (e. g., a class is renamed, a new message is added to a sequence diagram). Most model changes are indeed small. Unfortunately, there are a range of quite common modeling activities that cause larger model changes. Examples are model transformations [13], re-factoring [19], model branching or merging (as in subversion) [18], and model repairs [7,9,15]. These activities may be arbitrary complex

and pose a challenge to incremental consistency checking because the increment becomes too large to handle it instantaneously. This problem is aggravated with the complexity of design rules. The more complex a design rule the more likely it is affected by a model change. And the problem is even further aggravated with an increasing number of design rules. The more design rules there are, the more likely are model changes to affect multiple design rules. Combined, they strongly impact the performance of incremental consistency checking.

This paper proposes a novel approach for improving the performance of incremental consistency checking. The basic idea is to not validate design rules in their entirety but to focus on the parts that are affected by model changes. Whether a part of a design rule is affected by a change is determined fully automatically based on observations of the design rule's validation which is stored as a *validation tree*. The approach does require an additional memory overhead for storing the validation tree. Complete validation trees can be voluminous but a second novel contribution is in the reduction of the validation tree to those parts of a design rule validation whose change can impact the validation result as a whole. For example, if $a = true$ and $b = false$ in the conjunction $a \wedge b$ then a change to a cannot affect the result of the conjunction and can be cut from the tree.

We evaluated our approach on 19 design rules, 30 industrial design models (approximately 130,000 elements), and roughly 1,500 random model changes. We will demonstrate through empirical evaluations that our approach achieves up to 20-fold performance for 19 designs rules we analyzed – the more complex the design rule, the larger the performance gains. While the scalability of the proposed approach is still linearly dependent on the quantity of model changes and the number of design rules, the performance gain implies that much larger model changes or quantities of design rules can be handled instantaneously than was possible to date. The memory cost is in average only 2-fold more expensive compared to state-of-the-art and increases linear with the size of the model. Our approach is fully automated, tool supported, and integrated with the modeling tool IBM Rational Software Architect. The implementation supports OCL as the constraint language and UML as the modeling language, but the approach is designed to be applicable to arbitrary modeling languages and their corresponding constraint languages.

The remainder of this paper is structured as follows. Section 2 defines the basic terms and a running example that are used in this paper. In Section 3 the main principles of our approach are shown. Section 4 shows the evaluation of our approach and Section 5 explains the threats of validity of the evaluations. An overview of the existing work on this topic is given in Section 6 and, finally, Section 7 concludes the work and gives an outlook about future work planned.

2 Definitions and Example

2.1 Basic Definitions

Definition 1. *A **model** represents the main aspects of a software project that must be implemented. It consists of **model elements** that contain **properties**, e. g., a name or a reference to other model elements. A **design rule** defines requirements that must be*

fulfilled in the model. A violation of a design rule causes an inconsistency in the model. The requirement of the design rule is expressed as a ***condition*** *that validates 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). Each validation can cause a separate inconsistency.*

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

Definition 2. *A design rule condition consist of a set of hierarchical ordered expressions where each expression consists of an operation (o), a set of 0 to * arguments (α) and a validation result (σ). The arguments of an* ***expression*** *(ϵ) are expressions itself and they are tree based ordered, i. e, each expression has exactly one parent and is in a set of arguments of an other expression except the root expression (ϵ_0).*

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

2.2 Running Example

Figure 1 introduces a small illustration to accompany the discussion in the paper. The example depicts an UML model containing two diagrams, a class and sequence diagram. The given model represents an early design-time snapshot of a video-on-demand (VOD) system. The class diagram (left) represents the structure of the VOD system: a 'User' that controls the system and watches videos, a 'Display' used for visualizing movies and receiving user input and a 'Streamer' for downloading and decoding movie streams. The sequence diagram (right) describes the process of selecting a movie and playing it. Since a sequence diagram contains interactions among instances of classes (objects), the illustration depicts a particular user invoking 'select' (a message) on the 'd' lifeline of type 'Display' which then invokes 'connect' on the 's' lifeline of type 'Streamer'. The movie starts playing once the 'play' message is issued which is followed by 'stream' and successive 'draw' messages.

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

Design Rule (1) discusses two conditions the model must satisfy: 1) whether a given message in a sequence diagram matches the direction of the class association (1.1), and 2) whether the given message has a same-named operation (1.2). The two conditions are

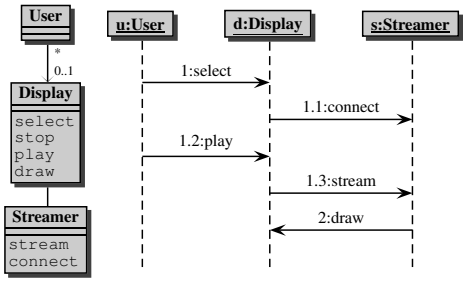

Fig. 1. UML Class and Sequence Diagram of a Video on Demand System

expressed in one design rule and linked together by a conjunction ($\wedge$). This linking into more complex design rules is common, for example, to avoid unnecessary validations. As such, the search for a matching operation name (1.2) is useful only if the operation's class is identified correctly by the association (1.1).

Typically, rules are written from the perspective of a context – a type of model element. The context for Design Rule (1) is the UML type 'Message' (see $Message\, m$ at the very top) and the rule has to be validated separately for each message in the sequence diagram in Figure 1. There are thus five validations of that rule necessary in the illustration. Each evaluation validates the correctness of its message only.

If, for example, the design rule validates the message 'connect' then, first, both ends of the message are accessed through the receive event ($l_1 \in m.receiveEvent.covered$) and the send event ($l_2 \in m.sendEvent.covered$). As it is possible in UML that a message is assigned to more than one lifeline, the design rule iterates via an existential quantifier ($\exists l_1, l_2 \ldots$) over all returned lifelines which are assigned to the variables l_1 and l_2. For message 'connect' l_1 is instantiated with the lifeline 's' of type 'Streamer' and l_2 with the lifeline 'd' of type 'Display'. Next, the owned attributes (the association ends are expressed as attributes) of the senders lifeline type ($l_2.represents.type.ownedAttribute$) are compared to those of the receiver ($a.type = l_1.represents.type$). If one is found then there must be an association that connects sender type and receiver type. The second part of Design Rule (1) accesses the receiver lifeline – lifeline 's' in this example. All types (universal quantifier $\forall$) of the lifeline must include an operation that is named after the message name. The 's' lifeline is an instance of class 'Streamer' that includes operations ('ownedOperation' property) such as 'connect' and 'stream'. The existential quantifier then validates whether at least one operation matches the name of the message (='connect'). Thus, both conditions of Design Rule (1) are satisfied and the condition validates to true (=consistent).

2.3 Problem Description

Current state-of-the-art requires the re-validation of the entire design rule [6] if a model change affects it. This is computationally increasingly expensive with the complexity of the design rule or the number of model changes, thus these approaches scale only for small quantities of model changes (individual changes) and comparative small design rules. A solution explored in [3] is thus to describe design rules from the perspective of different model changes. However, this requires manual overhead and introduces errors

if done incorrectly. Another solution would be to split up the design rule into smaller parts. However, doing so increases the number of design rules but would not cause any performance and memory advantages. This problem is aggravated by the quantity of model changes (e. g., as in model re-factoring, branching, merging, repair). To illustrate this, consider the case of a repair. Our previous work [9] demonstrated that between ten to twenty kinds of changes can resolve a typical inconsistency. This number seems small enough. However, for computing the effects of such repairs ([5] referred to them as side effects), permutations of these ten kinds of choices need to be explored where each permutations requires (incremental) consistency checking. The number of possible repair alternatives increases exponentially.

3 Model/Analyzer Approach

This section introduces an approach that improves incremental consistency checking. Our empirical evaluation shows a reduction up to 20-fold (average 10-fold) in context of 19 design rules we analyzed. Our approach automatically records the run-time behavior of design rules to reason about which parts of the design rule validation are affected by a model change. In the following, we demonstrate how to capture the validation of a design rule in form of a validation tree and how to identify which part of the validation tree is affected by a model change.

3.1 Principles

Incremental consistency checking builds a scope for each design rule – or in case of Model/Analyzer for each design rule validation. If a model element changes then all those design rule validations need re-validation that included the changed element in their scope. Take, for example, a simple conjunction $a \wedge b$. In fact, Design Rule (1) is a conjunction where a checks for the message direction and b checks for the method declaration. During the initial validation of this conjunction, the consistency checker will first validate a and if a is true then b will be validated also. If a is false then b need not be validated because the result does not depend on b. The validation of the conjunction $a \wedge b$ results in either true (=consistent) or false (=inconsistent).

A model change only then affects the validation result of conjunction $a \wedge b$ if either a or b changes. Clearly, if neither a nor b change then the validation result cannot be affected. However, not all changes to a or b affect the validation result. For example if $a = true$, $b = false$ and a changes then this change does not affect the result of the conjunction ($a \wedge b$ was false because of b and for as long as b remains false a change to a does not matter). In this case, we may well discard a from the change impact scope which means that a change to a should not trigger a re-validation of $a \wedge b$ (expressed in Table 1, row 3). We see that initially, both a and b are validated ($a = true$, $b = false$) but a is discarded from the scope (column 'initial'). If a changes (column 'change a'), no re-validation is performed. If b changes, however, we need to validate a because it may have changed since (by having discarded it from the change scope we no longer know what happened to it since). The scope stays the same unless a is false in which case b may be discarded (recall from above that b needs no validation if a is false).

Row 4 in Table 1 depicts another situation where $a = true$ and $b = true$. The validation result of the conjunction is thus true and it is clear that both a and b may affect this validation result if either one of them were to change. Thus, both need to be validated and nothing can be discarded. This is the worst case for a conjunction where there is no apparent savings in the validation time and scope (memory consumption). However, even here we find savings in how incremental validation is performed. For example, if b changes then it must have become false and no validation is necessary to determine that the validation result of the conjunction is false also, i. e., a need no re-validation and can be discarded from the scope.

	a	b	$a \wedge b$	validate/discard		
				initial	change a	change b
1	false	false	false	$a/-$	b/a	$-/-$
2	false	true	false	$a/-$	$b/-$	$-/-$
3	true	false	false	ab/a	$-/-$	$a/\begin{cases} b & \text{if } a = \text{false} \\ - & else \end{cases}$
4	true	true	true	$ab/-$	$-/b$	$-/a$

Table 1. Initial Validation and Re-validation of a Conjunction

To illustrate the benefits, consider the total number of validations and dismissals in Table 1. We see that the initial validation investigates at least a and often also b (in average 1.5 validations depending on situation). Here, our approach's performance is identical to the traditional Model/Analyzer approach. However, the advantage of our approach becomes apparent with the changes. Traditional approaches have to pay the initial validation cost for all subsequent changes. In our approach, we see that for changes to a only 0-1 (average 0.5) re-validations are necessary (instead of the 1.5). The same is true for changes to b with an even lower average of 0.25. These saving are small if we consider expressions individually but these saving accelerate with every expression. As an example, assume that a in $a \wedge b$ is another conjunction $a_1 \wedge a_2$: $(a_1 \wedge a_2) \wedge b$. If a_1 changes but a_1 was discarded from the scope (Table 1) then no re-validation is necessary. If a_1 was not discarded then it may affect $a_1 \wedge a_2$ and we need to re-validate $a_1 \wedge a_2$ to be certain. Only if the re-validation of $a_1 \wedge a_2$ shows that it indeed changes and Table 1 reveals that its change may affect $a \wedge b$ where $a = a_1 \wedge a_2$ then we re-validate $a \wedge b$ (where the validation result for a is already known because it was just validated). The more complex the design rule, the more significant must be the savings. Consider that $a_1 \wedge a_2 = true$ and $b = false$. In this case, the a branch, consisting of $a_1 \wedge a_2$, is discarded from the scope which means that neither a_1 nor a_2 can affect the design rule. Each upward re-validation step is thus a double filter: 1) to assess whether the change can impact the validation result of the step and 2) only if that validation result changes then the step above is re-validated.

To achieve these performance gains, our approach must retain some intermediate validation results from the initial validation or subsequent re-validation in form of a validation tree (will be discussed below). However, we will demonstrate that this memory consumption is moderate because significant parts of the validation tree can be discarded as shown in the small example above.

Naturally, our approach is not just limited to conjunctions. Design Rule (1) consist of a conjunction but its arguments are more complex expressions such as quantifiers. Table 2 shows the validated and discarded parts for the initial validation and re-validation for possible changes on an existential quantifier. An existential quantifier is similar to concatenated disjunctions. From this it follows that if the existential quantifier validates to true then one validation of the source elements must be kept to ensure that this quantifier can change its state only if at least this validation fails. All others can be discarded. Other logical expression, such as disjunctions, implications, negation, the universal quantifier, ..., are analogous and we omit their discussion due to brevity.

	$A = \{a_1, a_2\}$	$\exists a \in A \vert a$	validate/discard					
			initial	add a_3	delete a_1	delete a_2	change a_1	change a_2
1	{false, false}	false	$Aa_1/-$	Aa_3/a_1	Aa_2/a_1	$-/-$	$a_1/-$	$-/-$
2	{false, true}	true	Aa_1a_2/a_1	$-/-$	$-/-$	Aa_1/a_2	$-/-$	Aa_1/a_2
3	{true, false}	true	$Aa_1/-$	$-/-$	Aa_2/a_1	$-/-$	Aa_2/a_2	$-/-$
4	{true, true}	true	$Aa_1/-$	$-/-$	Aa_2/a_1	$-/-$	Aa_2/a_1	$-/-$

Table 2. Initial Validation and Re-validation of an Existential Quantifier

3.2 Filtered Validation Tree

The validation tree is a structured log of the validation of a design rule and depicts every expression performed, the order they were performed, the model elements that were accessed, and all intermediate results generated. This validation tree will be generated the first time a design rule is validated on a model element, i. e., on start-up and when new model elements that match the context of a design rule are created. Figure 2 shows a validation tree for the message 'connect'. Algorithm 1 describes how the validation tree is built and how parts of it are discarded to reduce the impact of a model change (CPU savings) and reduce memory.

At the beginning, the algorithm distinguishes (line 2) between expressions that have a Boolean result (e. g., conjunction, existential quantifier) and all other expressions (e. g., model access, string or collection manipulations). If the expression is a Boolean expression then the first action (3) is to create a node in the validation tree that points to the expression (ϵ) (e. g., the conjunction for Design Rule (1) becomes the root node in the tree). The next step is the validation of the arguments (α) of the expression (6) which is done by a recursive call of the validate algorithm for all its arguments. The validation is guarded by a condition (5) that the argument needs validation only if the result is not already in the validation tree. Of course, during the initial validation no results are in the validation tree; however, we will see later that re-validation makes use of this algorithm also and the need for this condition will be explained in the next section. An edge will be added between the node of this expression and the node created during the validation of each argument (7). The algorithm distinguishes between the different logical operation types (o). Due to brevity only the conjunction (8) and the existential quantifier (10) are given (there were discussed above in detail), but the other operations can be derived from these two operation because of the rules of the Boolean algebra.

Algorithm 1 Initial Validation and Creation of a Validation Tree

```
1  validate(Expression e)
2    if (e.operation is a boolean operation)
3      add node(reference to e)
4      for (i=1 to #e.arguments) //validate arguments
5        if (e.arguments[i] is not in validation tree)
6          validate(e.arguments[i])
7        add edge(e.node, arguments[i].node)
8        if (e is-a conjunction)
9          if e.arguments[i].result!=false next
10       else if (e is-a existential)
11         if e.arguments[i].result!=true next
12       ...
13       else next
14     e.result = e.operation(e.arguments) //compute validation result
15     if (e is-a conjunction) //filter 1 validation tree (discard)
16       if (e.arguments[1] and !e.arguments[2]) remove edge(e,e.arguments[1])
17     else if (e is-a existential)
18       if (e.result=true)
19         for (i=1 to #e.arguments)
20           if e.arguments[i]=false remove edge(e, e.arguments[i])
21     ...
22   else
23     e.result = e.operation(e.arguments)
24     if (e.operation accesses model elements)
25       add node(reference to model elements)
```

Not all arguments need to be validated to compute the validation result of the expression (9, 11 are analogous to Table 1). After the validation has finished, the result is calculated (14) and the filtering of the validation tree starts (15-21). The filtering of the validation tree is the dismissal of previously validated nodes/edges (e. g., see Table 1 third row or Table 2 second row). For example, if the expression is a conjunction and the first argument is true but the second one is false then the edge to the first argument can be discarded.

As was said, the algorithm distinguishes between Boolean and non Boolean expressions. The non-Boolean expressions are usually model accesses (e. g., retrieve the name of a message) or manipulations (e. g., remove an association from a collection). Those expressions are processed in lines 23 to 25. Essentially, our approach keeps track of all model elements accessed for which we create leaf nodes in the validation tree. The actual results computed by these expressions are discarded eventually to minimize the memory overhead of our approach. The leaf nodes typically only contain the references to the model elements through which the results were computed. In Design Rule (1), the source of the first existential quantifier is a property call: $m.receiveEvent.covered$ which includes accesses to two model elements: the 'receiveEvent' of the message 'm' and from its result the 'covered' property. This sequence of two property calls reveals the lifelines that act as message receivers. The leaf node will identify these accessed model elements and properties. If one of them should change then the existential quantifier would be (potentially) affected and may require re-validation. In our example the existential quantifier that iterates over all the operations of class 'Streamer' creates one node and edge to the model element properties that are accessed to get the operations. For each operation in the source a sub tree representing the condition of the quantifier

is created. After the sub trees are created, all the sub trees that are not needed for the validation result of the existential quantifier are discarded (refer to Table 2).

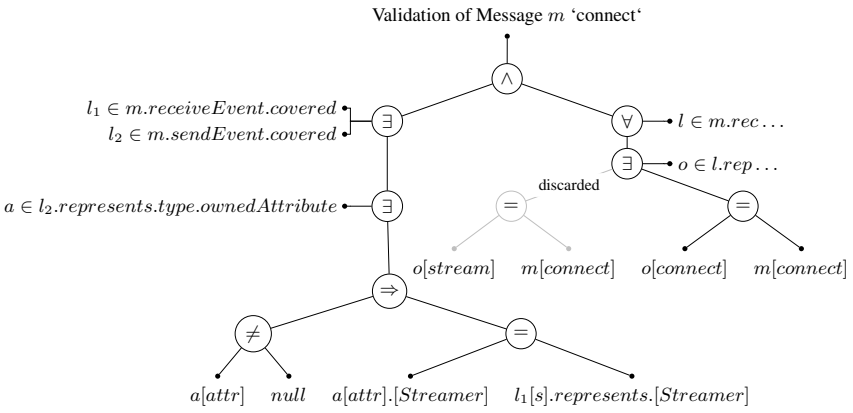

Fig. 2. Validation Tree for Message 'connect'

Since the validation of a design rule discards parts of the validation tree, we speak of a *filtered validation tree*. The filtered validation tree contains only nodes representing the Boolean expressions with their Boolean validation results. Both are cheap to maintain in terms of memory consumption. All other validation results are discarded after the validation except for the model element/properties that were accessed to compute the results. These model accesses are references to the design model and such references are also cheap to maintain in terms of memory consumption.

3.3 Impact of a Change

Once a validation tree has been created, only those parts must be re-validated that are affected by the change. The initial generation of a validation tree is strictly top down whereas the incremental re-validation is mostly bottom up. The previous section discussed one part of the benefits of our approach in that the re-validation focuses on the filtered validation tree only and ignores changes that would have affected discarded parts of the validation tree. This saves both memory and improves performance because a change becomes less likely to affect the validation tree. This section discusses another part of the benefits of our approach. It demonstrates that changes propagate upward for as long as the nodes are affected by the change only. Since the validation tree stores all intermediate Boolean results, only changes to these results must be computed anew. The model elements accessed (scopes) are referenced at the bottom of the tree (the leaves) and the impact of model element changes thus always start at the leaves and is propagated upward towards the root.

We illustrate this on two change examples and the validation tree for message 'connect'. The first change (change 1) is the renaming of the operation 'connect' to 'wait' in

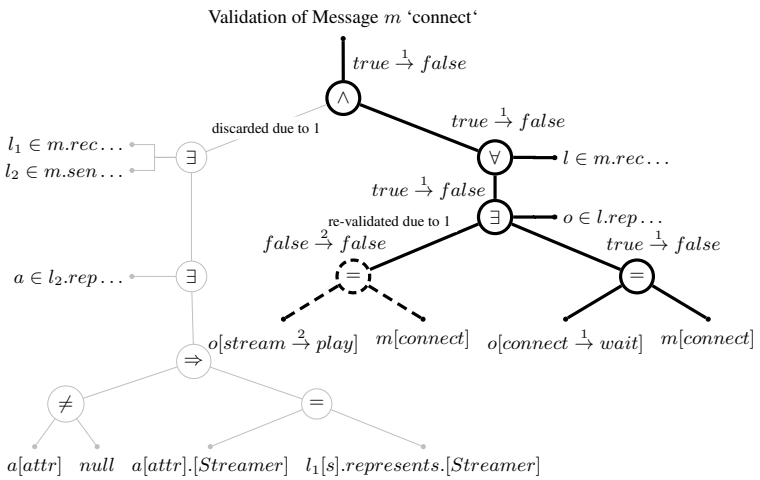

Fig. 3. Impacts of two Changes on the Validation Tree

Figure 1. The second change (change 2) is the renaming of operation 'stream' to 'play'. Their impacts on the validation tree are shown in Figure 3. Change 1 is drawn in a thick solid black line and change 2 as a thick dashed line. Algorithm 2 shows the handling of a change in pseudo code.

Change 1 affects the operation 'connect' ($o[connect \xrightarrow{1} wait]$) which is now named 'wait' (represented by the arrow with the number of the change on top of it from $connect \xrightarrow{1} wait$). Algorithm 2 first identifies all the leaf nodes that reference the changed model element. The re-validation is bottom up and starts at these leaf nodes (27, 28). For each leaf node, the re-validation saves the previous, old result (31) and re-validates the expression (32). Since the approach maintains the Boolean results in a validation tree only, it follows that no results are saved for leaf nodes and the 'oldValue' remains undefined. Leaf nodes are thus always re-validated and their results propagated upwards (33) to the parent node (35). In our example, the new name of the operation is retrieved and then propagated up to the equals expression in Figure 3. The equals expression must be re-validated using the new value from the left branch and either the old value from the right branch (if the validation tree has the value) or a computed value otherwise (note lines 5-6 in Algorithm 1). In our example, the result of the equals expression changes from true to false and as the value changed, the new value will be propagated up to the existential quantifier, the parent of the equals expression. As this was the only node that made the existential quantifier true, its change causes the re-validation of other branches. Recall that during the creation of the validation tree the branch for operation 'stream' was discarded because it did not affect the validation result of the existential quantifier. This branch may have changed since and needs to be re-validated. If there are no other elements or none of the other elements satisfy the existential quantifier (as in our example), the result changes and the new result is propagated up to the universal quantifier. This universal quantifier fails also because of the fail to the existential quantifier and the new value is propagated to the conjunction, the top node of the validation tree (it has no parent, line 34). This node will validate to false also, the overall evaluation of this design rule changes from consistent to inconsistent.

Algorithm 2 Processing a Model Change

```
26  processChange(Element elem)
27    for all (node:validation trees | node references elem)
28      revalidate(node.expression)
29
30  revalidate(Expression e)
31    oldResult = e.result // is empty if leaf
32    e.result = validate(e)
33    if (oldResult != e.result) // filter 2 stop bottom up propagation
34      if (e has parent node)
35        revalidate(e.parent)
```

As can be seen 11 out of 22 nodes (a complete re-validation) must be re-validated due to this change only. Furthermore, the left part of the validation tree (the first argument of the conjunction) will be discarded thereafter because it cannot influence the validation result after the change, cutting 11 nodes from the 22 nodes. This benefits the next re-validation because a change becomes less likely to affect the new filtered validation tree.

The second change affects operation 'connect' ($o[stream \xrightarrow{2} play]$). Without the first change this change would not have affected the validation tree because this branch was discarded during the initial validation and only added again after the first change. The operation 'connect' is thus in the change scope now. However, the re-validation of the second change stops at the equals relation because the result of this node does not change (33), i. e., it remains false. The upward propagation of changes thus stops once the re-validated result of a node is equal the previously known result ('oldResult') of that node. In this case 3 out of the 11 remaining nodes must be re-validated only.

In contrast to other approaches, where the change of one model element triggers a re-validation of the design rule in its entirety, our approach only triggers the re-validation of those nodes in the validation tree that are affected by a model change. Since a discarded argument in an expression discards the entire branch, the reductions increase with the complexity of the design rule (the number of nodes). For the non-discarded arguments, it must be noted that the re-validation is mostly upwards from leaf to root nodes and new validations (top down) are limited to sub trees only. If multiple model elements change then the same validation tree may have to be validated multiple times, but these re-validations always start at distinct parts of the tree and may only join at common roots (and only if the change affects the parent expressions). Redundancies are possible only if the changes trickle to common roots which is often not the case. Further optimizations are possible here.

4 Evaluation

We empirically validated our approach on 30 industrial UML models ranging from small to large models (127 to 67,723 model element/properties). These models were evaluated on 19 design rules. The design models are in part taken from [6] and were transformed from UML 1.4 to UML 2.1 (this explains the differences in model sizes). The design rules were also taken from [6] and converted form Java to OCL design rules (some of them could not be converted due to the limited expressiveness of OCL). The

Name	#Model Elements	#Scope Elements	#Design Rule Validations	MOH Brute f. [MB]	MOH Filtered [MB]	MOH MDT OCL [MB]
Video on Demand	90	127	63	2	2	1
ATM	220	763	304	20	10	7
Microwave Oven	290	296	138	29	13	9
Model View Controller	418	834	393	16	12	7
eBullition	513	892	341	53	18	10
Curriculum	763	1,350	595	150	43	4
Teleoperated Robot	1,115	1,969	885	97	34	5
Dice 3	1,274	1,649	599	74	14	3
ANTS Visualizer	1,282	3,119	1,225	169	93	6
Inventory and Sales	1,296	1,898	803	250	17	4
Course Registration	1,406	1,822	712	97	19	4
UML IOC F05a T12	1,453	2,441	998	67	23	6
VOD 3	1,558	4,652	1,789	175	110	7
Vacation and Sick Leave	1,658	2,681	1,084	145	65	5
Home Appliance	1,707	2,115	784	267	53	6
HDCP Defect Seeding	1,784	2,199	985	72	36	7
DESI 2.3	1,974	4,727	1,838	188	106	7
iTalks	2,212	4,049	2,289	417	130	6
Hotel Management Sys.	2,583	4,244	2,033	359	87	5
Biter Robocup	2,632	6,265	2,334	227	129	8
Calendarium	2,809	6,160	2,694	326	79	6
UML LCA F03a T1	2,983	2,912	1,243	108	53	3
<unnamed>	5,373	6,804	2,906	973	129	7
NPI	7,110	8,536	2,930	1,353	97	7
Word Pad	8,078	17,907	8,186	860	513	20
dSpace 3.2	8,761	12,994	5,869	N.A.	259	11
OODT	9,828	26,650	11,384	752	434	21
Insurance Network Fees	16,255	27,442	10,562	N.A.	172	58
<unnamed>	33,347	33,844	16,627	N.A.	382	111
<unnamed>	64,061	67,723	40,297	N.A.	724	61

Table 3. Model Size, Design Rule Validations and Memory Overhead

complete list of design rules and the Model/Analyzer tool can be found on our tools homepage http://www.sea.jku.at/tools/.

Before we evaluate the performance of the new approach we have to ensure that the validation of the design rules are correct. To validate this, we compared the results with the standard MDT (Modeling Development Tools of Eclipse) implementation of OCL. Given the large number models and over 90,000 correct design rule validations, the approach can be considered correct.

The evaluation of our approach covers the memory overhead and the performance. The evaluations were done using our implementation for the IBM Rational Software Architect (RSA) on an Intel Core 2 Quad CPU @2.83GHz with 8GB (4GB available for the RSA) RAM and 64bit Linux (3.1.9). We compare our evaluation results against incremental approaches exclusively and do not address the improvements against batch evaluation which is already done in our former work [6]. We evaluated the memory

overhead based on three criteria: 1) the memory used by a *Brute force* validation, i. e., each intermediate validation result as well as the property calls are cached, 2) the memory overhead of the filtered validation tree, and 3) the memory overhead of the MDT OCL validation which caches the scope only. For larger models the *Brute force* validation could not be completed due to out-of-memory exceptions (N.A.). The memory consumption was measured using the 'Runtime' interface of Java and before measuring the garbage collector was activated. The measured data were double checked with data from the TPTP profiler for eclipse.

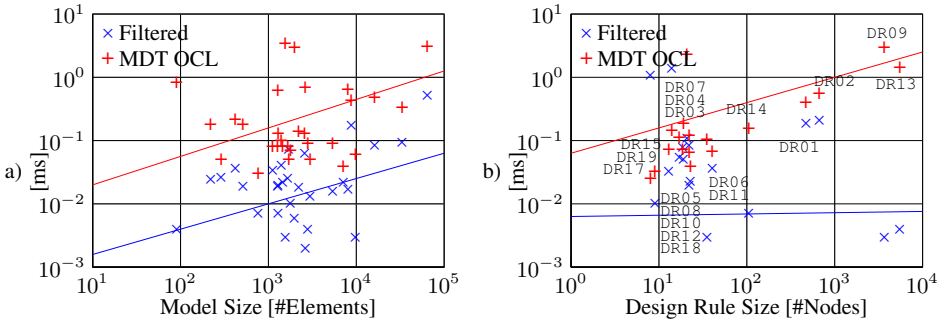

Fig. 4. Evaluation Time (in ms) of MDT OCL and the Optimized Approach

Table 3 shows the evaluation results for the memory overhead (MOH in Mega Byte) in relationship to the model sizes, the accessed model element properties, and the quantities of validated design rules. Compared to the worst case (*Brute force*), the reduction of the used memory (*Filtered*) is between 50% and 80% (except for the very small models). As can be seen, the reduction depends not only on the model size but also on the number of validated design rules and accessed model elements. The main memory overhead (*MDT OCL*) is caused by the scope that must be built, even for other incremental consistency checking approaches that are scope based.

The evaluation of the performance is done in the same environment using 'System.nanoTime()' (resolution $1\mu s$). We compared the evaluation of the validation using the MDT OCL environment with the validation of the filtered validation tree. Figure 4 shows the timing results regarding the model size (a-left) and regarding the design rule complexity (b-right). We measured the average re-validation of 50 random model changes on each model. The model changes cover the modification of model element properties, the addition of model elements and their deletion. As can be seen, the re-validation times (measured in ms) are nearly independent of the model size, but the state-of-the-art OCL validation times are about 2 to 20-fold slower than compared to our new approach. Slightly different are the results for the design rule complexity. Whereas the re-validation time remains stable for our new approach, the re-validation using the MDT OCL increases linear with the design rule complexity (please note the logarithmic scale on both axes for both diagrams).

It is intuitive to believe that our approach should perform better for larger, more elaborate rules because our approach does not require the re-validation of the entire

rule (=equivalent to the entire validation tree) but only some paths from the leaves to the root. If a validation tree has m nodes (=expressions) then the run-time complexity of the normal approach should be $O(m)$ whereas the run-time complexity of the optimized approach should be $O(log(m))$. Figure 4 (b) confirms this hypothesis for all rules. The larger the rules (x-axis, measured in the average sizes of their evaluation trees), the more significant the saving.

5 Threats to Validity

The threats to validity are mostly centered on the random testing of changes. Random changes can lead to models that may not be valid and do not conform to the UML standard. Still, they are possible changes and it may be useful to know that our approach is superiors, perhaps even with changes that are impossible. Another aspect has to do with the fact that random changes may under represent expensive changes. This assumes that there are changes that are likely and expensive. However, previous work has shown that likely changes are rarely expensive changes [8]. The reason we relied on random changes was simply our desire to perform large quantities of model changes. Second, the validation time for the state-of-the-art MDT OCL validations is measured without the generation of the scope because to do so the UML implementation must be instrumented (as on the case of our former work). Using the new approach this is not necessary any more. However, this implies that the results of our approach should be even better because computing the scope would have been higher, as would have been the memory cost.

6 Related Work

Cabot and Teniente present an event triggered approach to detect inconsistencies in UML/OCL conceptual schema [4]. They use a list of events that trigger the re-evaluation of consistency rule. An event is a modification in the model and using this events they reduce the amount of rules that must be re-validated in the model. They also use a syntax tree that is annotated with events that potentially violate the constraint expressed in OCL. In contrast to our approach the use a static analysis of the OCL constraint and as such the incremental characteristic is limit to single constraints only.

Jouault et al. [12] developed an incremental approach using ATL (AtlanMod Transformation Language) to transform OCL rules and to trigger only those rules that are affected by a model change. This approach similar to ours as it uses model element and their properties to trigger the re-evaluation of constraints. But their main focus is on a static analysis of the constraints where the trigger events are extracted whereas we are able to consider parts of a constraint only. Unfortunately, they provide no evaluation of their approach regarding the evaluation time and memory consumption.

Similar, Blanc et al. [3] achieve near instant performance thanks the re-writing of design rules for each relevant model change. This requires the engineer to re-factor consistency rules to understand the impact of model changes. If done correctly, this leads to good performance. However, since writing these annotations may cause errors, they are no longer able to guarantee the correctness of incremental consistency checking.

Bergmann et al. [2] present an query based approach. This approach is similar to the approach presented in this paper, but they use a query language (IncQuery) that is executed on EMF models. Their approach is based on the Viatra2 [20] framework and in RETE [11] networks. In their approach the queries must be stored permanently in memory and the values must be updated after each model change. In contrast, in our approach we only store references on the model elements and the Boolean values of the validation tree, which shortens the massive memory consumption problem. Unfortunately, the smallest unit in their timings are 10ms which is rather high for the problem addressed in our paper.

Nentwich et al. present xLinkIt [14], a language that evaluates the consistency of "documents", including UML design models. Design rules are expressed in a uniform manner and xLinkIt is capable of checking the consistency of models incrementally. However, it requires between 5 and 24 seconds for evaluating changes and the tool is thus not able to keep up with an engineer's rate of model changes. The approach by Reiss [16] is in principle alike xLinkIt. Rather than defining consistency rules on XML documents, Reiss defines consistency rules as SQL queries which are then evaluated on a database which may hold a diverse set of artifacts. Reiss' use of a database makes his approach certainly more incremental. However, the incremental updates in his study take about 30 seconds to 3 minutes. ArgoUML [17] was probably the first UML design tool to implement incremental design checking but it required annotated consistency rules. Their annotations were lightweight but so where their computational benefits.

In the context of pervasive computing Xu et al. optimized the re-validation of design rules [21]. The also use validation trees to for their optimization but in contrast to our approach they process modifications of the context (the location) only. However, as we address model-based software development we have to deal with more types of changes. Furthermore, we optimize the tree in the post process to achieve better results regarding reduced memory consumption and re-validation effort.

7 Conclusions and Future Work

This work introduced a novel approach to the incremental validation of design rules in design models. Empirical validation on 19 design rule has shown that our approach reduces the time to re-validate a design rule up to 95%. This observation was made on a large number of design models and we found that it outperformed the state of the art under all situations by a large margin. Indeed, we have not encountered a single design rule that would not benefit from our approach. This work paves the way for processing a much larger number of model changes and/or more complex model changes with instant or near instant response times. In our future work, we will use this re-validation approach to simulate repair actions and determine the effects that such actions have on the overall design model and on other design rules.

Acknowledgments

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

References

1. R. Balzer. Tolerating Inconsistency. In L. Belady, D. R. Barstow, and K. Torii, editors, *ICSE*, pages 158–165. IEEE Computer Society / ACM Press, 1991.
2. G. Bergmann, Á. Horváth, I. Ráth, D. Varró, A. Balogh, Z. Balogh, and A. Ökrös. Incremental evaluation of model queries over EMF models. In *Proceedings of the 13th international conference on Model driven engineering languages and systems: Part I*, MODELS'10, pages 76–90, Berlin, Heidelberg, 2010. Springer-Verlag.
3. X. Blanc, I. Mounier, A. Mougenot, and T. Mens. Detecting model inconsistency through operation-based model construction. In W. Schäfer, M. B. Dwyer, and V. Gruhn, editors, *ICSE*, pages 511–520. ACM, 2008.
4. J. Cabot and E. Teniente. Incremental integrity checking of UML/OCL conceptual schemas. *Journal of Systems and Software*, 82(9):1459–1478, 2009.
5. B. Demsky and M. Rinard. Data structure repair using goal-directed reasoning. In *Proceedings of the 27th international conference on Software engineering*, ICSE '05, pages 176–185, New York, NY, USA, 2005. ACM.
6. A. Egyed. Instant consistency checking for the UML. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 381–390, New York, NY, USA, 2006. ACM.
7. A. Egyed. Fixing Inconsistencies in UML Design Models. In *ICSE*, pages 292–301. IEEE Computer Society, 2007.
8. A. Egyed. Automatically Detecting and Tracking Inconsistencies in Software Design Models. *IEEE Trans. Software Eng.*, 37(2):188–204, 2011.
9. A. Egyed, E. Letier, and A. Finkelstein. Generating and Evaluating Choices for Fixing Inconsistencies in UML Design Models. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 99–108, Washington, DC, USA, 2008. IEEE Computer Society.
10. S. Fickas, M. Feather, and J. Kramer. Proceedings of ICSE-97 Workshop on Living with Inconsistency. 1997.
11. C. Forgy. Rete: A Fast Algorithm for the Many Pattern / Many Object Pattern Match Problem. *Artificial Intelligence*, 19:17–37, 1982.
12. F. Jouault and M. Tisi. Towards Incremental Execution of ATL Transformations. In L. Tratt and M. Gogolla, editors, *ICMT*, volume 6142 of *Lecture Notes in Computer Science*, pages 123–137. Springer, 2010.
13. M. Mens, S. Ragnhild, and M. D'Hondt. Detecting and Resolving Model Inconsistencies Using Transformation Dependency Analysis. In *The 9th Int'l Conf. MoDELS 2006*, pages 200–214. Springer-Verlag, 2006.
14. C. Nentwich, L. Capra, W. Emmerich, and A. Finkelstein. xlinkit: a consistency checking and smart link generation service. *ACM Trans. Internet Technol.*, 2(2):151–185, May 2002.
15. C. Nentwich, W. Emmerich, and A. Finkelstein. Consistency management with repair actions. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 455–464, Washington, DC, USA, 2003. IEEE Computer Society.
16. S. P. Reiss. Incremental Maintenance of Software Artifacts. *IEEE Trans. Software Eng.*, 32(9):682–697, 2006.
17. J. E. Robbins. ArgoUML, v0.32.1. http://argouml.tigris.org/, March 2011.
18. M. Sabetzadeh, S. Nejati, S. Liaskos, S. M. Easterbrook, and M. Chechik. Consistency Checking of Conceptual Models via Model Merging. In *RE*, pages 221–230. IEEE, 2007.
19. R. Van Der Straeten and M. D'Hondt. Model refactorings through rule-based inconsistency resolution. In *Proceedings of the 2006 ACM symposium on Applied computing*, SAC '06, pages 1210–1217, New York, NY, USA, 2006. ACM.

20. D. Varró and A. Balogh. The model transformation language of the VIATRA2 framework. *Sci. Comput. Program.*, 68(3):214–234, 2007.
21. C. Xu, S. C. Cheung, and W. K. Chan. Incremental consistency checking for pervasive context. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 292–301, New York, NY, USA, 2006. ACM.