Determining the Cause of a Design Model Inconsistency

Alexander Reder and Alexander Egyed, Johannes Kepler University Linz, Austria

Abstract—When a software engineer finds an inconsistency in a model then the first question is why? What caused it? Obviously there must be an error. But where could it be? Or is the design rule erroneous and if yes then which part? The cause of an inconsistency identifies the part of the model or design rule where the error must be. We believe that the visualization of an inconsistency ought to visualize the cause. Understanding the cause is of vital importance before a repair can even be formulated. Indeed any automation (e. g., code generation, re-factoring) has to be considered with caution if it involves model elements that cause inconsistencies. This paper analyzes the basic structure of inconsistent design rules as well as their behavior during validation and presents an algorithm for computing its cause. The approach is fully automated, tool supported, and was evaluated on 14,111 inconsistencies across 29 design models. We found that our approach computes correct causes for inconsistencies, these causes are nearly always a subset of the model elements investigated by the design rules' validation (a naive cause computation approximation), and the computation is very fast (99.8% of the causes are computable in <100ms).

Index Terms—Design Tools and Techniques, Programming Environments/Construction Tools, Validation

1 Introduction

Design languages, such as the Unified Modeling Language (UML) [24], must adhere to diverse constraints. Some constraints are defined as part of the language (e. g., UML well-formedness rules) while other constraints reflect modeling philosophies, domain restrictions, or even application needs. In our work we use the term *design rule* to denote any of these constraints. A design rule is defined on the meta model level and consists of a condition that validates to either false (inconsistent) or true (consistent) to express whether the model satisfies the given design rule. Most state of the art represents design rules in first order logic and there are many approaches for validating design rules and thus for detecting inconsistencies. However, a basic question all approaches struggle with is how to communicate the cause of an inconsistency, once encountered, to the designer.

1.1 The Cause of an Inconsistency

An inconsistency is a violated condition of a design rule. It is important to not confuse the cause of an inconsistency with an error or repair. The cause of an inconsistency identifies the part of the model and design rule that contributed to the inconsistency. This part must contain an error; an error that eventually needs to be repaired. However, not the entire cause must be erroneous. For example, a design rule condition that requires a message name to be equal to an operation name would be inconsistent, if the two names were to differ. In such a case, both the operation name and message name cause the inconsistency (without the operation name there would be no inconsistency; nor without the message name). However, both names need not to be changed to repair the inconsistency. While the cause is precisely determinable (the message name and the operation name), the repair is uncertain (repair the message name, the operation name, or both?). This work focuses on computing the cause.

We define the cause of a design rule inconsistency to enumerate all model element properties (i. e., parts of the model if the model is presumed erroneous) and design rule expressions (i. e., parts of the design rule if the rule is presumed erroneous) that contributed to the inconsistency. We define 'contributed' as implying that the model element property was involved in the reasoning that led to the inconsistency, much like both the message name and the operation name contributed in the example above. For every inconsistency there is exactly one cause and this cause is neutral towards repairs of which there are typically many alternatives [26]. Previous work [8], [20] demonstrated that at least one and at most all the model element properties accessed during the design rule's validation must have caused the inconsistency. Yet, as will be demonstrated, typically not all model element properties accessed during validation of a design rule caused the inconsistency; nor do only accessed models element properties cause inconsistencies. This work is analogous to the computation of all MUS [19] for SAT models where the union of all MUS is the cause of an UNSAT. However, the focus on design modeling and rich design rule languages makes the cause computation for design models distinct from the SAT world. The concept of a cause is thus a known concept; however, how to compute the cause for design models with their rich design rule languages is a novel contribution of this work.

1.2 Application of Cause Computation

Designers can use information on the cause to assess trust in a design model. Simply said, an inconsistency is a symptom of a problem and a model element that causes an inconsistency should not be relied upon for as long as the inconsistency persists. For visualization this implies that all model element properties that caused an inconsistency should be highlighted to ensure equal emphasis (and blame). State-of-the-art tools tend to visualize a single element only in case of inconsistency, usually the design rule's context element. This is not ideal because the designer is then given a biased view. For example, returning to the "message name must be equal to the operation name" example above: If the design tool were to highlight the message name only, in case of inconsistency, then this conveys a misleading sense of error (i. e., the message name being wrong while in fact the operation name may be wrong) and a misleading sense of correctness (i. e., the operation name being correct because it is not highlighted while in fact it could be wrong). A proper visualization of the cause can thus contribute to a better design process and avoid follow-on errors (other errors caused, in part, due to the existence of errors in design models). A similar argument can be made for all automations on models. For example, model re-factoring or transformation (code generation) is less trustworthy if it involves model elements known to cause inconsistencies [10]. As such, it would be good to indicate code generated from a model element as untrustworthy if that model element also caused an inconsistency. Yet another application is for generating repairs to resolve inconsistencies [26]. While this paper purposely does not address repair issues, it is obvious that we must first understand what caused an inconsistency before we can think about how to repair it. In better understanding the cause, we can help focus the repair to those parts of the model that contributed to the inconsistency. At least one and at most all elements of the cause need to be changed to repair an inconsistency. This application would also improve current state-of-the-art which often focuses on a too large or too small set of model element properties during repairs.

1.3 Goal of this Paper

The main contribution of this paper is an algorithm for computing a complete set of model element properties and a complete set of design rule expressions that caused a given inconsistency. This work thus contributes to the fundamental best practice of software engineering and designers can use the information on the cause to better understand inconsistencies. This work combines the design rule's structure and its behavior during validation. This work thus expands on earlier work [9] where we investigated inconsistencies by observing and analyzing their behavior during validation. While a range of different languages are available for describing design rule conditions, this work focuses on OCL [23] as it is the most pre-dominant design rule language today. Nonetheless, we believe that this paper is also applicable to other constraint languages because most follow the principles of predicate logic and set-based reasoning. Similarly, this work focuses on UML as the modeling language. Yet, this paper is not biased towards any aspect of UML and should be applicable to any modeling language that is based on a defined syntax and semantics (e. g., meta model). Indeed, past work has demonstrated that our basic framework can be applied to product line languages [30], mechatronics design languages, and others.

1.4 Paper Structure

The remainder of this paper is organized as follows. Section 2 introduces a running example and defines the basic terms used in this paper and demonstrates an example of a cause of an inconsistency. Section 3 discusses the state-of-the art in consistency checking, and presents existing approximation concepts for computing the cause of inconsistencies. Section 4 lays out the main principle of our approach and presents an algorithm for computing the cause of an inconsistency. Section 5 discusses the evaluation of the approach and Section 6 concludes the paper with an outlook on future work.

2 Illustration, Background and Definitions

In this section we introduce a simple UML model and OCL design rule. The notation used in this section and in the rest of this paper is as follows: Type information from the meta model is written in *Slanted* font. Operation types (from design rule conditions), model elements and properties are typed in `Typewriter` font and the values of model element properties as well as constants are written in 'quotation' marks. Letters in formulas that refer to the model are written in Roman letters (M, a, b, ...) and letters that refer to design rules are written in Greek letters (σ, ζ, γ, ...). Tuples of elements are written in angle brackets ($\langle t, \gamma \rangle$,...).

2.1 Introductory Example

To illustrate the problem in context of a design model, consider the UML model in Figure 1. The given UML class diagram (Figure 1) describes a simple inheritance hierarchy of a `Light`. The `Light` is an `Activatable` and has an *Attribute* that indicates if it is activated. A `Light` can be a `LED`, `ElectricBulb` or a `Candle`. The *Class* `LED` itself can be a `Torch`. All these classes have further attributes. When we abstract the model visualization we detect that a model consists of elements that have properties. For example, the class `Light` has the property `name` which is of type *String* and has the value 'Light'.

Definition 1. A **model** (M) consists of **elements** ($e \in M$), where the elements can have **properties**

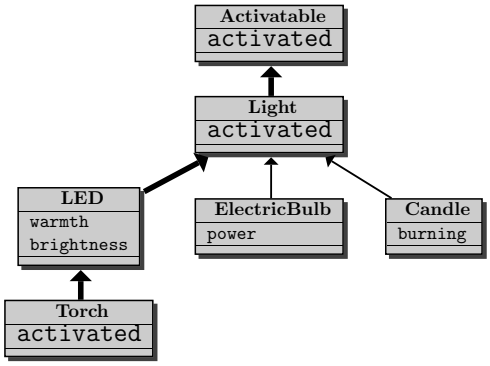

Figure 1. Class Hierarchy of a Light

(p). The property types can be any simple types like Boolean, Integer, Float, String,... or references to other elements ($e.p \mapsto M \cup$ 'any value'). The elements of a system are instances ($\Uparrow$) of a specific **type** (t) defined by the **domain language** ($t \in DL$). In the case of the UML, the domain language is the UML Meta Model defined by the OMG [24].

$$\begin{array}{rcl} e & \in & M \\ e.p & \mapsto & M \cup \text{'any value'} \\ t & \in & DL \\ e & \Uparrow & t \end{array}$$

2.2 Design Rule

A design rule describes a constraint that the design model must satisfy. Most commonly used design rule languages are based on first order logic [29].

Definition 2. A **design rule** (R) defines a Boolean condition that the model must satisfy. A violation of a design rule causes an inconsistency in the model. A design rule is expressed as a tuple consisting of a **condition** (γ) that validates to a Boolean ($\mathcal{B}$) result, 'true' (consistent) or 'false' (inconsistent) and a specific **context**, a model element type (t) that specifies the model elements on which the condition must be validated.

$$\begin{array}{rcl} R & := & \langle t, \gamma \rangle \\ \gamma & : & e \mapsto \mathcal{B} | e \Uparrow t \end{array}$$

Design Rule 1 shows a typical design rule expressed in OCL that validates whether an *Attribute* of a *Class* is not defined in a parent *Class*. A design rule is written for a specific context (Line 1). In our example rule the context is an UML type *Class* which implies that this design rule is applied on every model element of type *Class* in the model. The word `inv` defines an invariant, i. e., this design rule must hold always. In this paper we presume that all design rules are invariants. Illustrated on the example model in Figure 1, this design rule will be validated six times, once for each *Class* in the class diagram.

Design Rule 1 Design Rule to Validate if an *Attribute* is not Defined in a Parent *Class*

```
1  context Class inv:
2    let attrNms:Bag(String)=self.attribute->
         collect(p:Property|p.name) in
3    self.allParents()->forAll(c:Classifier|
4      c.attribute->forAll(p:Property|
5        not attrNms->exists(s:String|s=p.name)))
```

To illustrate the validation of this design rule, we explain the validation of the design rule on *Class* `Torch`. Line 2 defines the variable `attrNms`, which holds the names of the context *Class*' *Attribute*s. The `collect` operation collects the names (`name` property) of the *Class*' *Attribute*s (type *Property*). This variable now contains all the names of the `Torch` *Attribute*s ({'activated'}). After the variable declaration, the condition of the design rule that must be satisfied is defined between Line 3 and Line 5. Line 3 is a universal quantifier (`forAll`) that iterates over all the super *Class*es of the context *Class* (`self.allParents`). The property `allParents` is a recursive call of the UML properties `generalizable` and `general` of the context *Class* and all its following super classes. The result is a collection containing all super classes (UML type *Classifier*) of the context *Class* ({`LED`, `Light`, `Activatable`}). The condition of the universal quantifier (`c:Classifier|c.attribute...`) validates if no *Attribute* in the super class exists where the `name` equals an *Attribute* `name` of the context *Class*. This is done by an iteration over the attributes (UML type *Property*) of the super class (Line 4). The condition of the second universal quantifier checks if the actual *Attribute* `name` does not exist in the collection of attribute names (variable `attrNms`) from the context class (Line 5). The condition starts with a negation (`not`) of the existential quantifier (`exists`). The source of the existential quantifier are the names from the context *Class*' *Attribute* `name`s. The condition of the existential quantifier is an equality relation (=) that compares the *Attribute* `name` from the context *Class* with the *Attribute* `name` of the super *Class* (`s:String|s=p.name`). The overall validation result of a design rule condition is a Boolean value. In our example the validation fails (inconsistent) because the *Attribute* `activated` is defined in the parent *Class*es `Light` and `Activatable`.

2.3 Design Rule Operation Structure

As can be seen in Design Rule 1, a rule is composed of hierarchically ordered operations with specific arguments where the hierarchy decides which results of operations are arguments to others. For example, the `self.attribute property call` operation is an argument to the `collect` operation that follows. In the following, we refer to this hierarchical arrangements of operations as *expressions* where the hierarchy of expressions reflects the order on how operations are called. Starting with a root expression that is the

starting point of the design rule validation and ending at leaf expressions that reflect constants and property calls. The root expression is then synonymous to the design rule condition which must validate to a Boolean result with the result being expected to be 'true' for the design rule to be consistent. The expected result and the actually validated results will be explained in more detail in the approach but due to completeness of our definitions they are introduced here.

Definition 3. A design rule condition consists of a set of hierarchically ordered (tree-based) expressions where each expression consists of an operation (o), a set of 0 to $*$ arguments (α) and a validation result (ς) and an expected result (σ). The arguments of an **expression** (ϵ_x) are expressions itself, i. e., each expression has exactly one parent (ρ) where this expression is a child thereof except for the root expression (ϵ_0). The root expression has 'true' as expected result ('true' is equivalent to *consistent*). A property call expression (ϵ_p) is a special kind of expression that accesses a model element property. Property call expressions access properties of model elements. They do not have arguments nor expected results and hence they are leaves in the tree structure of expressions. This is similar to constant expressions (ϵ_c).

$$
\begin{aligned}
\epsilon_{x>0 \wedge x \neq p} &:= \langle o, \alpha, \rho, \sigma \in \mathcal{B}, \varsigma \in \mathcal{B} \rangle \\
\epsilon_0 &:= \langle o, \alpha, \varsigma \in \mathcal{B} \rangle \\
\epsilon_p &:= \langle p, e, \rho, \varsigma \in M \cup \text{'any value'} \rangle \\
\epsilon_c &:= \langle \rho, \varsigma \in \text{'any value'} \rangle \\
\forall i > 0 \exists j \geq 0 \wedge j \neq p &: \epsilon_i \in \epsilon_j.\alpha \wedge \epsilon_i.\rho = \epsilon_j
\end{aligned}
$$

Table 1 shows some commonly used operations in OCL. For each operation, the argument types and the result types are given. The negation, for example, has only one argument of type Boolean and the result is also of type Boolean. The conjunction, disjunction and implication have two arguments of type Boolean and a result of type Boolean. The equality relation can have any type of arguments but its result is a Boolean. The quantifiers (`forAll` and `exists`) have two arguments where the type of the first argument is a collection of elements of any type over which the quantifier will iterate whereas the second argument is a Boolean condition that applies to the elements of that collection. The result types of both quantifiers are Boolean.

Design rule languages such as the OCL also define non-Boolean operations, mostly to access and navigate the design model. For example, the `property call` provides the value of a model element property, e. g., the `name` of a *Class*. A call of a model element property might be a recursive process, as a property call might refer to another model element and this model element may also have properties that can be called. Hence, a property call on a model element can be cascading calls of property calls in the form of $m.p_1.p_2 \ldots p_x$. The validated value of the property call expression is the value of the last called property in that chain.

Table 1
Common First Order Logic Operations and OCL Extensions used by Design Model Consistency Checkers

operation	OCL	argument types		result type
		arg_1	arg_2	
$\neg$	`not`	Boolean	-	Boolean
$\wedge$	`and`	Boolean	Boolean	Boolean
$\vee$	`or`	Boolean	Boolean	Boolean
$\Rightarrow$	`implies`	Boolean	Boolean	Boolean
$=$	`=`	any	any	Boolean
$\forall$	`forAll`	collection	Boolean	Boolean
$\exists$	`exists`	collection	Boolean	Boolean
constant		-	-	any
property call	`.`	element	property	any
collect	`collect`	collection	property	collection
select	`select`	collection	Boolean	collection
variable	`let`	reference	any	any

In our example Design Rule 1 such a property call is shown in Line 2 and Line 5, where the `name` property of a *Class Attribute* is accessed. These property calls provide a single element value. The call of the `attribute` property (Line 2 and Line 4) provide a set of element values, the *Attributes* of a *Class*. The `collect` operation type iterates over a set of elements (first argument) and collects the results of a property call (second argument) on a source element. The cardinality of the result set is equal the cardinality of the source set: $|Source| = |Result|$. In contrast, the `select` operation type filters elements from the source where the condition, given as second argument, is satisfied. The elements in the result are a subset of the source elements: $Source \supseteq Result$. A variable declaration references to a value provided by a model element that can be accessed anywhere in the design rule after the variable has been declared (declaration in Line 2, used in Line 5).

The `constant` operation type let's the rule writer provide any value that can be used in a design rule. This can be, for example, a Boolean constant 'true' or 'false', any string, float or integer value.

2.4 Cause of an Inconsistency

To illustrate the cause of an inconsistency, consider again the example design rule and UML model. The context of this design rule is an element of type *Class*, i. e., the rule is validated on every *Class* in the class diagram. First, we consider the cause of model element properties. The validation of the design rule 1 on the *Class* `Torch` failed because the *Attribute* `activated` is defined in the super *Class* `Light` and `Activatable`. The following twelve model element properties caused this inconsistency:

1) `Torch.attribute`
2) `activated(Torch).name`
3) `Torch.generalization`

4) `Torch.generalization.general` (a generalization returns an unnamed element, so we leave the initial property call of this property call chain)
5) `LED.generalization`
6) `LED.generalization.general`
7) `Light.generalization`
8) `Light.generalization.general`
9) `Light.attribute`
10) `activated(Light).name`
11) `Activatable.attribute`
12) `activate(Activatable).name`

The first model element property is the `attribute` property of the *Class* 'Torch'. This model element property must be in the cause of model elements because there would be no inconsistency if the *Class* `Torch` would not own the *Attribute* `activated` (this property is accessed in the `let` expression in Line 2 via the `attribute` property of the `self` variable). The next model element property in the cause is the `name` of the *Attribute* `activated`. The name is accessed in Line 2 as part of the `collect` expression. The name also causes the inconsistency because there would be no inconsistency if the attribute had a different name.

The next model element properties that must be in the cause are the parents of the context class (element properties 3 to 8, design rule Line 3). The parents contribute to the cause because there would be no inconsistency if, say, the *Class* `Light` were not a parent of the *Class* `Torch`. Similarly, the `attribute` properties are in the cause also. However, do note that the `attribute` property of the *Class* `LED` is not in the cause while the `attribute` property of the *Class* `Light` is. This is explained in that none of the attributes of the *Class* `LED` caused the inconsistency. Hence, while the `attribute` properties are called for all parent classes, only the ones that failed to satisfy the inner condition of the design rule caused the inconsistency (element properties 9 and 11, design rule Line 4). Consequently, the `name` property of only those attributes are in the cause (element properties 10 and 12, design rule Line 5).

In state-of-the-art it is commonly assumed that the design rule is correct and inconsistencies must be caused by model elements and their properties (the list above). However, it is also possible that the design rule is incorrect (e. g., if the design rule is the result of a constraint imposed by another model [6]). In this case, the cause of the inconsistency could also be expressions of the design rule itself. Recall that the cause computation is neutral as to the eventual repair and as such we simply enumerate causes but make no assessment as to which is more likely or plausible. If the design rule condition itself causes the inconsistency then some or all its expressions must be part of the cause. Consider, for example, that the writer of the design rule accidentally uses an universal quantifier (`forAll`) instead of an existential one (`exists`) in Line 4 of our design rule. The meaning of this design rule now changes such that, if there exist attributes in a class, at least one of the parent class attributes must not be named as the attributes of the context class. From this it follows that the cause of an inconsistency contains parts of a design rule condition.

Definition 4. A **cause** (ζ) of an inconsistency consists of two parts: 1) The expressions where the validation result does not equal the expected result (the cause of expressions ζ_ϵ), and 2) the model elements properties that immediately influence the validation result of any given expression (the cause of model element properties $\zeta_{e.p}$). Each expression that is in the cause must have a parent that is in the cause too (except for the root expressions which has no parent). The cause of an inconsistency is then the union of all the model element properties and of all expressions where the validated result does not equals the expected result.

$$\begin{aligned} \zeta_\epsilon(\gamma) &:= \bigcup \epsilon_x \in \gamma(m) | \epsilon_x.\varsigma \neq \epsilon_x.\sigma \wedge \\ &\quad (\epsilon_x \neq \epsilon_0 \Rightarrow \epsilon_x.\rho.\varsigma \neq \epsilon_x.\rho.\sigma) \\ \zeta_{e.p}(\gamma) &:= \bigcup \epsilon_p.e.p \in \gamma(m) | \epsilon_p.\rho.\varsigma \neq \epsilon_p.\rho.\sigma \end{aligned}$$

2.5 Usability of the Cause

The main contribution of this paper is to help the user to understand what causes an inconsistency in a design model by identifying what parts of a model or design rule (or both) contributed to it. Most inconsistency management technologies to date attach inconsistencies to the context element they were evaluated on. However, this conveys a misleading sense of error. As the example above shows, the context element is but one of many elements involved in the inconsistency. It may be part of the cause but it may not necessarily be the error. More problematic is the misleading sense of correctness that is communicated if not the entire cause is identified because the designer is wrongly left to believe that other parts of the model are correct. Considering the example, every element above that is missing from the cause misleads the designer. Since the larger models we validated that have more than 33K model elements, the designer would be faced with the hard near impossible task of trying to find erroneous elements from a large pool of elements (the missing needle in a haystack) where it is likely that the designer fails which may lead to incorrect or sub-optimal repairs. Another weakness with current state of the art is that they presume the model to be wrong in case of inconsistencies. This is generally true for generic well-formedness constraints (e. g., such as the ones provided with UML) but not necessary true for constraints derived from other sources, say, requirements. The cause computation thus should also compute which part of the design rule contributed.

In previous work, we argued that inconsistencies should be associated with all elements that are accessed during an inconsistent design rule's validation. Doing so is better in that most/all potentially erroneous elements are highlighted, however, there may still be missing elements (false negatives) and incorrectly tagged elements that were accessed but are not erroneous (false positives). This work remedies this problem by computing

a correct and complete cause. Section 1.2 enumerates several applications of a correct and complete cause computation.

3 Related Work

To the best of our knowledge, no work has ever investigated how to compute the causes of inconsistencies for design models. Nonetheless, the computation of causes is sometimes done implicitly in work that focus on how to repair inconsistencies. Moreover, understanding the cause of inconsistencies in design models has some similarity to program understanding in source code (why does a program fail where the program here is a design rule condition in first order logic) or cause computations for other reasoning engines such as MUS in SAT.

The closest analogy of our work is the (H)UMUS [22] ((High-level) Unions of Minimal Unsatisfiable Sets) work for SAT models. An MUS [19] (Minimal Unsatisfiable Set) identifies a minimal unsatisfiable set and the union of all unsatisfiable sets identifies the clauses (elements) that caused an UNSAT (UNSAT is analogous to inconsistent). Of course, it is possible to transform a design model and its constraints to a SAT model, followed by applying (H)UMUS. However, doing so has problems and we argue that it is beneficial to identify causes directly in design models. First, transforming a design model to SAT is straightforward in principle (e. g., Alloy [16] or Czarnecki-Petroszek [3]), however, it is computationally intensive and we are not aware of any instant, incremental mechanisms for doing so that could compete in performance with our work. Second, computing the cause in SAT is NP complete in complexity. Third, the cause once determined in the SAT model then needs to be interpreted back to the context of the design model which requires extensive traceability and reverse transformation. Both the SAT model and the traceability would require extensive memory overheads not needed by our approach. Points 1 and 3 are merely technical challenges. However, (H)UMUS is not incremental and this work thus contributes a fast, incremental mechanism that, to the best of our knowledge, presently does not exist.

In our approach we are concentrating on analyzing design rules that are expressed in first order predicate logic [18], [29]. Our approach adopts some of these ideas and includes them to achieve an approach that is able to detect the cause of an inconsistencies based on the structure and behavior of a first order predicate logic expression. What is unique about our work is its combination of the structure of the design rule and its validation behavior. Most approaches to consistency checking in design models are based on design rules in predicate logic [20], [32]. Our approach should thus be applicable to all of them. As a proof of concept, the empirical evaluation focuses on OCL only.

Most of the consistency checking approaches that exist for model based software development treat the design rules that must be satisfied by a model as black boxes. Furthermore, they use transformations to validate the model against the design rules. Winkelmann et al. [31], for example, presents an approach that translates meta models and OCL [23] design rules into graphs so that the design rules can be checked during the instance generation process. Furthermore, Czarnecki and Pietroszek [3] use OCL to define well-formedness rules for the verification of feature-based model templates which are analyzed by a SAT solver. The ability to translate design rules requires detailed understanding of design rule semantics which is very relevant in this work. However, as already discussed above, this transformation does not provide a computation of a cause but merely a different formal basis for reasoning about the cause. As this paper will show, design models and design rules are perfectly adequate to support such reasoning directly without the computational and memory overhead of having to maintain models/rules in another, formal language and the transformations back and forth.

The use of an intermediate representation is not a pre-requisite for consistency checking. Indeed, it is possible to write design rules that directly compare design models rather than transforming them first [13], [15], [20], [28]. Moreover, relevant to this paper are approaches to incremental consistency checking because these approaches are able to detect and track inconsistencies when they occur. Cabot et al. present an incremental approach [2] that is applicable on UML [12] and OCL [23]. Based on the design rule defined in OCL they generate a set of actions that, if they are executed on the model, violate the design rule. During the validation process the design rule is modified in a way that the best context is found for an efficient re-validation. Since the validated design rule changes its behavior dynamically, this approach cannot provide any information that would help to reduce the set of accessed model elements during its validation. Xu et al. showed in their paper [33] how design rules can be optimized for the re-validation. They use also a run-time observation of the design rule validation to filter out parts of the validation that do not contribute immediately to the overall validation result. The filtering is not, like in this paper, optimized to detect the cause of an inconsistency but to optimize the memory consumption and re-validation time for the pervasive context where the resources are limited. While all these approaches are able to detect inconsistencies, none of these approaches are able to compute the cause of an inconsistency. Indeed, we would argue that our approach could complement any of these approaches in computing the cause. Blanc et al. introduced an incremental approach that is based on the model changes that can be made [1]. In their approach a design rule will be re-validated only if a certain change in the model affects the result of the design rule. They pointed out that the performance of re-validating a design rule depends on

the complexity of the design rule condition. In [27] we showed how the performance and the memory used for the re-validation of a design rule can be optimized.

More relevant, however, are approaches that compute repairs for inconsistencies of which there are several. Nentwich et al. present an approach that is type triggered [20]. If a model element of a specific type has been modified, all design rules that match the type of the modified element are validated. During the validation process a set of links are generated. The links can be consistent links, if the validation of the design rule detects no violation or the links are inconsistent links, if the validation detects a violation. These links are used for the re-validation of the rules. Since only these links are collected during the validation process, this is a black box validation. However, in their later work about generating repair actions [21] they use a white box analysis of the violated design rule to determine repair actions for these violated design rules. Nonetheless, the run-time behavior of a violated design rule is not considered in their approach. Rather they investigate a design rule's structure only. Consequently, they cannot identify which part of a design rule caused a given inconsistency (e. g., in case of $a \wedge b$ their approach would suggest to repair either a, b, or both even if, say, a is not violated). Their conservative generator for repair actions thus presumes that inconsistencies are always violated in their entirety, which implies that the model elements they recommend for repairing is a conservative approximation of the cause we are assessing more precisely in this paper.

Dam et al. [5], [4] developed an approach on how OCL design rules can be violated or resolved respectively, based on the internal structure of the design rules. Based on this analysis, abstract repair plans are generated at compile time, i. e., the set of OCL design rules is statically defined in the tool, and this abstract actions are instantiated if the design rule is violated by the model. A major difference to our approach is that they generate the repair plan at compile time of the tool so they can not detect the concrete model elements that will cause the violation. Their approach is designed exclusively for OCL and a proof is given that this approach is correct and complete regarding the single OCL operations. Furthermore, this approach, in contrast to [21], typically considers all inconsistencies at one time whereas our approach is able to consider single inconsistency as well as arbitrary groupings thereof.

In the ontologies domain there also exist mechanisms for detecting the cause of inconsistencies . Kalyanpur et al. [17] presented an approach to debug unsatisfiable classes in OWL to detect the causes of errors in web ontologies. Deng et al. [7] developed an approach that measures the Shapley value (know from game theory) to find axioms in ontologies that cause an inconsistency in the input. These work show the importance of detecting the cause of inconsistencies, however, these techniques follow a different approach than our work and are not readily applicable to the design modeling domain because they are working with axioms and assertions that are not based on first order logic. Therefore transformations are needed and the incremental characteristics of our approach might get lost.

We base this work on our previous work on incremental consistency checking [8], [14], where the defined design rules are treated as black boxes. The internal structure of the design rules is invisible to the user and can be defined in any language. The validation is triggered by a context element and the re-validation is based on a scope of model elements that are accessed during the validation of the design rule. Furthermore, this scope is used to generate repair actions for violated rules [9], [11]. The scope of this approach is similar to the cause of model element properties. However, the scope can be 1) incomplete due to short cut validation (the validation stops when the first violation is found) and 2) not minimal because of all accessed elements are in the scope. For example, the attributes of the *Class* `LED` would be in the scope but, as explained in the last section, they do not cause the inconsistency.

4 Approach

4.1 Principle

To illustrate the determination of the cause of an inconsistency in principle, consider a simple conjunction $a \wedge b$ as a design rule condition. From the definitions, we know that the conjunction is expected to validate to 'true' to be consistent (recall that the expected result for design rules is 'true' always). The conjunction validates to 'true' if a and b are both 'true' and there are obviously four possible scenarios of a and b being 'true' or 'false' (columns 1 and 2 in Table 2). Both a and b must be Boolean expressions but during validation both must access model elements. For example, $attribute_1.name \neq$ 'x' $\wedge$ $attribute_2.name \neq$ 'x' is a possible example for $a \wedge b$ where a accesses the name of $attribute_1$ to ensure that its name is not 'x' and b accesses the name of $attribute_2$ to ensure that that's name is not 'x' either. So, if both a and b are 'false' then the model elements accessed by a and the model elements accessed by b must cause ($\zeta_{e.p}$) the inconsistency, i. e., $attribute_1.name$ and $attribute_2.name$ (see row 2/column 3 of Table 2). However, if b is 'false' only and a is 'true' then clearly a cannot have caused the inconsistency – the cause is b ($attribute_2.name$ as in row 3/column 4 in Table 2).

However, multiple expressions may access the same model element(s). For example, a and b may both access the same model element(s) as in the example $attribute_1.name \neq$ 'x' $\wedge$ $attribute_1.name \neq$ 'y'. Imagine now that the current name of $attribute_1$ is 'y' in which case $attribute_1.name \neq' x'$ validates to 'true' whereas $attribute_1.name \neq' y'$ validates to 'false'. Hence $attribute_1.name$ causes the inconsistency even though it is also accessed by another expression that

Table 2
Causes of a Conjunction, Disjunction and negated Conjunction

a	b	$\zeta_{e.p}(a \wedge b)$	$\zeta_{e.p}(a \vee b)$	$\zeta_{e.p}(\neg(a \wedge b))$	$\zeta_{e.p}(\neg(a \vee b))$
$false$	$false$	$\zeta_{e.p}(a) \cup \zeta_{e.p}(b)$	$\zeta_{e.p}(a) \cup \zeta_{e.p}(b)$	$\emptyset$	$\emptyset$
$true$	$false$	$\zeta_{e.p}(b)$	$\emptyset$	$\emptyset$	$\zeta_{e.p}(a)$
$false$	$true$	$\zeta_{e.p}(a)$	$\emptyset$	$\emptyset$	$\zeta_{e.p}(b)$
$true$	$true$	$\emptyset$	$\emptyset$	$\zeta_{e.p}(a) \cup \zeta_{e.p}(b)$	$\zeta_{e.p}(a) \cup \zeta_{e.p}(b)$

does not cause the inconsistency. A model element property thus causes an inconsistencies if it is involved in at least one expression that causes the inconsistency.

Table 2 shows the causes for a conjunction in the third column and the causes for a disjunction in the fourth column. It is interesting to see that the cause for $\zeta(a \vee b)$ is the same as for $\zeta(a \wedge b)$ if both a and b are 'false' – because both cause it. However, they differ otherwise. We can see that the cause computation depends on the kind of operation (conjunction vs. disjunction) but it also depends on the validation results of their arguments a or b. It follows that the cause computation requires both knowledge on the operations (obtainable through the design rule structure) and their validation results (obtainable through the validation behavior). Obviously, no causes exist whenever there is no inconsistency.

Indeed, in the context of $\zeta(a \vee b)$ the distinction between the cause of an inconsistency and repairs is easy to see. If both a and b are 'false' in $a \vee b$ then either a or b needs repairing (but not necessarily both) – an uncertainty that makes the enumeration of repair alternatives hard because of the combinatorial explosions in more complex design rules [26]. However, clearly both a and b caused the inconsistency because if either had been 'true' then the inconsistency would not have happened.

The example of the conjunction and the disjunction shows that not all accessed elements form the cause. Moreover, the example shows that simply enumerating accessed model element properties may also fail to identify all model elements that are part of the cause if the validation algorithm is optimized to avoid unnecessary computations, which is common (for example [33] or [27]). The conjunction and disjunction are but two examples. Yet the observations made above are evident in other operations and multiply as the design rules become more complex. In extreme cases, the design rule may access a very large number of model elements but inconsistencies may be caused by few model elements only. For example, a design rule that ensures that no two classes are named the same needs to access all class names (of which there could be many). However, typically few classes are named the same and the cause of such an inconsistency are only those few classes that share names. The degree of reduction is thus highly dependent on the design rule as our evaluation (Section 5) will show. Table 2 also shows the causes for a negation of a conjunction and disjunctions to illustrate the effect of operation hierarchies. These are discussed next.

As discussed above, for computing the cause we require knowledge on operations (hierarchy) and knowledge on validation results. However, we also require knowledge on expected results. The example above assumed that $a \wedge b$ was expected to validate to 'true'. Indeed, all design rules are expected to validate to 'true'. However, for example, a negation in a design rule inverts the expected result of its sub-expressions. Table 2 (row 5) shows the effect of a negation on the conjunction $a \wedge b$. For example, in column 5 we see that if both b and a are 'true' then the conjunction may be 'true' but because of the negation the overall result is 'false'. Both arguments cause the inconsistency. Quite similar is the situation for a negated disjunction. For example, if $b = true$ and $a = false$ then only b causes the inconsistency because a is 'false' already.

Quantifiers are special. A quantifier normally has two arguments – the first one being the source which identifies a set of elements and a second one being a condition that must hold for certain elements from the source. During validation, the condition of the quantifier will be validated for each element or combination of elements if there are multiple elements identified in the source. Table 3 shows the cause for an universal and existential quantifier. The condition of the quantifiers are single values or sub expression that validate to a Boolean result (a_1, a_2).

The source of the elements (A) is in the cause always whereas only those validations of the quantifier's condition are in the cause that violate it. In the case of the universal quantifier all validations of the quantifier condition must be 'true' and as such all those validations are in the cause that validate to 'false'. The existential quantifier is in essence a negation of the universal quantifier ($\forall a \in A \equiv \neg\exists\neg a \in A$).

Aside of the model elements, expressions may also be in the cause of their inconsistency (ζ_ϵ). This is important if the design rule were erroneous and we like to know which part of the design rule caused the inconsistency. To illustrate this we extend the abstract example by replacing the a and b with some more complex expressions to get a design rule that consists of more hierarchical levels. Consider the following expanded example: If a and b access other expression like disjunctions or conjunction, for example, $a := \neg(c \wedge d)$, $b := e \wedge \neg d$, $c := false$, $d := o.name = m.name \mapsto true$ (e. g., compares the message name to an operation

Table 3
Cause of an Universal and Existential Quantifier

$A = \{a_1, a_2\}$	$\zeta_{e.p}(\forall a \in A : a)$	$\zeta_{e.p}(\exists a \in A : a)$
$\{true, true\}$	$\emptyset$	$\emptyset$
$\{true, false\}$	$\zeta_{e.p}(A) \cup \zeta_{e.p}(a_2)$	$\emptyset$
$\{false, true\}$	$\zeta_{e.p}(A) \cup \zeta_{e.p}(a_1)$	$\emptyset$
$\{false, false\}$	$\zeta_{e.p}(A) \cup \zeta_{e.p}(a_1) \cup \zeta_{e.p}(a_2)$	$\zeta_{e.p}(A) \cup \zeta_{e.p}(a_1) \cup \zeta_{e.p}(a_2)$

name and in our example we assume that both are equal), $e := true$. Please note that behind c and e can also be sub trees that contain accesses to model element properties but due to simplification for a better understanding we use constants instead. Each of the expressions is expected to be 'true' and as such the cause (ζ_ϵ) of this inconsistency are the expressions $a \wedge b$ and $e \vee \neg d$ but not the expression $\neg(c \wedge d)$ because $\neg(c \wedge d)$ validates to 'true'. If c, d and e are accesses to model element properties then the cause ($\zeta_{e.p}$) for this inconsistency contains the model element properties of d only (`o.name` and `m.name`). c is 'false' indeed, but due to the negation in front of $c \wedge d$ which inverts the expected result from 'true' to 'false' and as such this part of the design rule does not cause the inconsistency and as such it is not in the cause of an inconsistency.

In summary, the cause of an inconsistency is calculated based on a comparison of an expected and a validated result of an expression where the expected result for the design rule condition (which is the root expression) is always 'true'. The expected results of the sub expression are determined top down and they depend on the expression types (a negation, for example, inverts the expected result of its sub expressions). Our approach computes the causes for inconsistent design rule conditions in a two-step process:

1) generate a validation tree for the inconsistency to compute the expected and validated results

2) compute the cause by traversing the validation tree

4.2 Generating the Validation Tree

Whereas the design rule condition defines the basic syntactical structure (similar to a program written in a programming language [29]), the validation tree represents the execution of the design rule (similar to a log of the execution of a program) with all its intermediate validation results. The hierarchy of the validation tree reflects the hierarchy of the syntax of the design rule and the validation tree contains a node for each validated expression from the design rule condition and the nodes are annotated with their expected and validated results.

Algorithm 1 shows how a validation tree is build up during the validation of a design rule. Indeed, the algorithm is a simplified validation algorithm with additional lines needed for creating nodes/edges in the validation tree (marked with '+') and the computation of expected results (marked with '-'). The remaining (unmarked) parts that are the same for the normal validation. The first expression for which the algorithm is called is the root expression of the design rule (=the root of the syntax tree as was discussed in Section 2.3) which is always a Boolean expression that has 'true' as its expected result. The algorithm first adds a node to the validation tree that represents this validated expression. The next step is the calculation of the arguments of that expression which are sub expressions explored recursively. The number of arguments depends on the operation of the expression (discussed in Section 2.3). For each argument the validate algorithm is called recursively do validate the arguments of the expression. Is the argument a Boolean expression (with a Boolean validation result) then we compute an expected result which is equal to the parent expression's expected result unless the expression's operation is a negation. The loop also gathers the validation results of the arguments and adds edges to the validation tree from the parent expression to all its arguments (=sub expressions for which the recursive descend will add nodes). Once the results of the arguments have all been computed, the expression's operation is performed and the validation result is stored. The end, a validation tree exists that mirrors one-to-one the validation of the design rule for which all validated and expected results are known.

Algorithm 1 Computing the Validation Tree during Validation of Design Rule

```
validate(designRule, contextElement)
  self=contextElement
  designRule.root.expectedR = true
    validate(designRule.root)

validate(expression e)
    set argumentResults = {}
+   add node(e)
    foreach argument of e.arguments
        validate(argument)
-       if (argument.validatedR is Boolean)
-           if (e.operation is Negation)
-               argument.expectedR = not e.
    expectedR
-           else
-               argument.expectedR = e.expectedR
        argumentResults += argument.validatedR
+       add edge(e, argument)
    end for
    e.validatedR = e.operation(argumentResults)
```

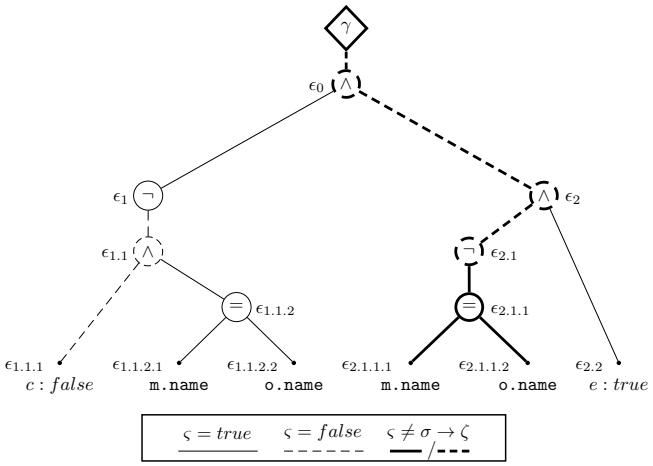

Figure 2. Validation Tree for $\gamma = \neg(c \wedge d) \wedge (e \vee \neg d)$, where $c = false$ and $d = e = true$

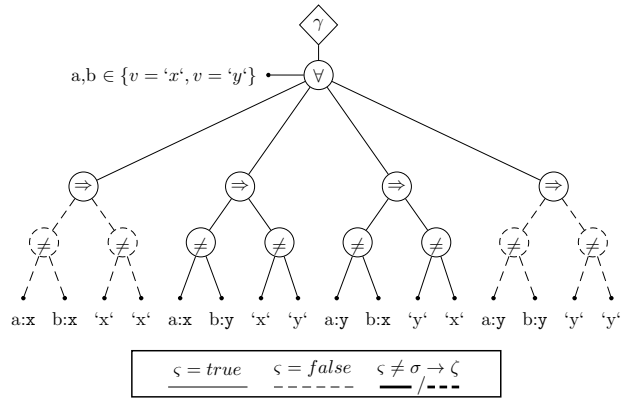

Figure 3. Validation Tree for an Universal Quantifier

In the following we illustrate the validate algorithm on the example we introduced earlier (Section 4.1).

$$
\begin{aligned}
\gamma &= \neg(c \wedge d) \wedge (e \vee \neg d) \\
c &:= false \\
d &:= m.name = o.name \mapsto true \\
e &:= true \\
\epsilon_0 &= \langle \wedge, \{\epsilon_1, \epsilon_2\}, false \rangle \\
\epsilon_1 &= \langle \neg, \{\epsilon_{1.1}, \epsilon_0, true, true\rangle \\
\epsilon_{1.1} &= \langle \wedge, \{\epsilon_{1.1.1}, \epsilon_{1.1.2}\}, \epsilon_1, false, false\rangle \\
\epsilon_{1.1.1} &= \langle \epsilon_{1.1}, false\rangle \\
\epsilon_{1.1.2} &= \langle =, \{\epsilon_{1.1.2.1}\epsilon_{1.1.2.2}\}, \epsilon_{1.1}, true, true\rangle \\
\epsilon_{1.1.2.1} &= \langle name, m, \epsilon_{1.1.2}, \text{`}x\text{`}\rangle \\
\epsilon_{1.1.2.2} &= \langle name, o, \epsilon_{1.1.2}, \text{`}x\text{`}\rangle
\end{aligned}
$$

$$
\begin{aligned}
\epsilon_2 &= \langle \wedge, \{\epsilon_{2.1}, \epsilon_{2.2}\}, \epsilon_0, true, false\rangle \\
\epsilon_{2.1} &= \langle \neg, \{\epsilon_{2.2.1}\}, \epsilon_2, true, false\rangle \\
\epsilon_{2.1.1} &= \langle =, \{\epsilon_{2.1.1.1}, \epsilon_{2.1.1.2}\}, \epsilon_{2.1}, true, true\rangle \\
\epsilon_{2.1.1.1} &= \langle name, m, \epsilon_{2.1.1}, \text{`}x\text{`}\rangle \\
\epsilon_{2.1.1.2} &= \langle name, o, \epsilon_{2.1.1}, \text{`}x\text{`}\rangle \\
\epsilon_{2.2} &= \langle \epsilon_2, true\rangle
\end{aligned}
$$

The numbers of the expression represent the sequence of how the design rule is validated. The numbers after the '.' represent the hierarchy of the design rule's validation in terms of recursive calls on the validate function. The first expression that is validated is the root expression ϵ_0. The next is ϵ_1 followed by $\epsilon_{1.1}$ (one hierarchy lower), $\epsilon_{1.1.1}$, $\epsilon_{1.1.2}$, $\epsilon_{1.1.2.1}$, and so on. The validation is in post order traversal, i. e., starting with the left branch, followed by the right branch (argument) and all other branches (arguments) and then the application of the operation (to root of the expression) onto the branches (arguments).

As simpler and more intuitive representation is given in Figure 2 which shows the validation for this example. We call this representation a *validation tree*. The diamond node of the validation tree defines the design rule condition and the starting point for the validation. This node is expected ($\sigma = true$) to be 'true' always to be consistent. Solid lines in the validation tree show the parts of the tree that validated to 'true' and dashed lines show the ones validated to 'false'. Thick lines indicate the cause, i. e., these parts of the validation tree where the expected result (ς) differs from the expected result (σ). The circular nodes are the operations of the expressions.

The condition starts with a conjunction ($\wedge$) where the expected result is 'true' also (ϵ_0). So that a conjunction validates to 'true', both arguments must validate to 'true', i. e., the expected result for both arguments is 'true'. The left branch of the conjunction is a negation ($\neg$) that inverts the expected result (ϵ_1) for the following conjunction ($\epsilon_{1.1}$) and as such the expected result for both branches of the conjunction is 'false' also. Once a leaf node has been reached, the validated results are calculated. The leaf node c is validated to 'false' ($\epsilon_{1.1.1}.\varsigma = false$), the equality ($=$) node to 'true' ($\epsilon_{1.1.2}.\varsigma = true$) because of the equality of the two property call expressions $\epsilon_{1.1.2.1}$ (model element `m` and property `name`) and $\epsilon_{1.1.2.2}$ (model element `o` and property `name`). Hence, the conjunction validates to 'false' ($\epsilon_{1.1}.\varsigma = false$), the negation inverts this result and validates to 'true' ($\epsilon_1.\varsigma = true$).

Now, the left branch of the top conjunction has been validated but to validate the complete condition, the right branch must be validated also. The expected result from this conjunction is propagated down to the expression on the right branch, i. e., the conjunction (ϵ_2) on the right side is expected to be 'true' also. The branches of this conjunction are the negation ($\epsilon_{2.1}$) and the leaf node e ($\epsilon_{2.2} = \langle \epsilon_2, true\rangle$) which validates to 'true'. The negation follows the equality node $\epsilon_{2.1.1}$ which compares two model element properties $\epsilon_{2.1.1.1}$ and $\epsilon_{2.1.1.2}$ (the same expression as $\epsilon_{1.1.2}$. This equality is also 'true' ($\epsilon_{2.1.1}.\varsigma = true$) and the negation inverts this result and validates to 'false' ($\epsilon_{2.1}.\varsigma = false$) and as a consequence the conjunction also ($\epsilon_2.\varsigma = false$). The result from conjunction ϵ_2 is propagated to conjunction ϵ_0 which validates to 'false' and hence the overall con-

dition validation fails, i. e., the design rule condition is inconsistent ($\gamma \mapsto inconsistent$).

As the validation tree of the simple logical expressions follows strictly the syntactical structure, the validation tree of a quantifier depends on the number of variables used in the condition and the number of elements in the source of the quantifier. During the validation of a quantifier the condition will be validated for each variable/element combination that exists in the quantifier's condition. A separate sub tree in the validation tree is created for each validation of an element. The validation tree for a quantifier consist of one branch that represents the source of the quantifier and i branches of validation conditions (b_i).

$$
\begin{array}{lll}
i & := & n^{|A|} \\
i & \ldots & \text{number of branches} \\
n & \ldots & \text{variables referencing elements from the source} \\
A & \ldots & \text{source of elements}
\end{array}
$$

To illustrate an universal quantifier, we take a set containing two elements $\{v = `x`, v = `y`\}$ and check that the values in this set must be different: $\forall a, b \in A : a \neq b \Rightarrow a.v \neq b.v$. Figure 3 shows the validation tree for this expression. It shows a non violated validation, i. e., there is no cause in this validation tree. How a partially violated quantifier looks like is illustrated in our running example shown in Figure 4. As can been seen, the condition has been validated for each permutation of the possible variable assignments. Dependent of the quantifier type (universal or existential) the operation node acts similar as a conjunction (in the case of an universal quantifier) or an disjunction (existential quantifier).

4.3 Calculate the Cause

Once the validation tree of an inconsistency is generated completely, the cause is computed by traversing the tree from top to bottom to determine which expressions are violated. The traversing starts at the top node of the validation tree, the conjunction ϵ_0. The validated result of this expression differs from the validated result and as such this expression will be included in the cause ($\zeta_\epsilon = \{\epsilon_0\}$). The expression ($\epsilon_1$) on the left branch of ϵ_0 validates to the expected result ($\epsilon_1.\varsigma = \epsilon_1.\sigma$) and as such the left sub tree can be omitted for the cause calculation because the expected result equals the validated result. However, the expression (ϵ_2) on the left branch of ϵ_0 does not validate to the expected result ($\epsilon_2.\varsigma \neq \epsilon_2.\sigma$) and will be included in the cause ($\zeta_\epsilon = \{\epsilon_0, \epsilon_2\}$). The expression ($\epsilon_{2.2}$) on the right branch of ϵ_2 validates to the expected result and as such $\epsilon_{2.2}$ will be not included in the cause. However, the expression on the left branch ($\epsilon_{2.1}$) will be included in the cause because the validated result differs from the expected result ($\zeta_\epsilon = \{\epsilon_0, \epsilon_2, \epsilon_{2.1}\}$). The expression on the branch of $\epsilon_{2.1}$ is also in the cause of this inconsistency ($\zeta_\epsilon = \{\epsilon_0, \epsilon_2, \epsilon_{2.1}, \epsilon_{2.1.1}\}$) but the two leaf expressions (the property call expression) are not included in the cause of the design rule. Rather, these two expressions are used to calculate the model elements that cause inconsistencies ($\zeta_{e.p}$). Recall from Definition 4 that the cause of an inconsistency is defined as 1) the expressions where the validated and expected results differ because they identify which part of the design rule causes the inconsistency (if the design rule should be erroneous) and 2) the model elements accessed by expressions because they identify which part of the model causes the inconsistency (if the model should be erroneous). As the parent of this two expression is $\epsilon_{2.1.1}$ which is in the cause of expressions (i. e., the validated and expected results are different), the two property calls to model elements are in the cause: $\zeta_{e.p} = \{m.name, o.name\}$.

Algorithm 2 Computing the Cause of an Inconsistency

```
1  computeCause(expression e, set causeE, set
       causeP)
2  /*causeE is the cause of expressions*/
3  /*causeP is the cause of model element
       properties*/
4    foreach argument of e.arguments
5      if (argument is Boolean)
6        if (argument.expectedR <> argument.
             validatedR)
7          causeE += argument
8          computeCause(argument, causeE, causeP)
9        else
10         remove edge(e, argument)
11       end if
12     else if (argument is Constant)
13       causeE += argument
14     else if (argument is Property Call)
15       causeE += argument
16       causeP += model element(s) accessed by
             argument
17     else if (argument is Let)
18       causeE += argument
19       computeCause(argument, causeE, causeP)
20     else if ... /* Table lookup */
21     end if
22   end for
```

Table 4 shows some of the common operations used in OCL and how their causes are calculated. The cause calculation is recursive. The cause calculation always starts at a Boolean expression (the top node is always a Boolean expression for a valid OCL constraint, namely the root expression ϵ_0) and navigates through its children and children's children until the leaf expressions are reached (leaf nodes always access model elements or are constants for a valid OCL constraint), i. e., the cause of expressions and the cause of model element accesses can be affected.

Algorithm 2 shows how to compute the cause out of a validation tree. The algorithm represents the guarded cause calculation from Table 4. In the algorithm the most common expression types are illustrated and the last `else if` branch represent the additional operations shown in Table 4 and provided by OCL. The cause computation starts at the root which is an expression with a Boolean result. The compute cause algorithm has additional two parameters representing the cause

of expressions (ζ_ϵ- `causeE`) and the cause of model element properties ($\zeta_{e.p}$- `causeP`). The algorithm starts with the root expression of design rule validation that detected an inconsistency and the cause of expressions is initialized with the root expression. For each argument of this expression it is determined if it validates to a Boolean result and if yes, then the expected and validated results of the argument will be compared. If they differ, the compute cause algorithm is called recursively and the expression is added to the cause of expressions (ζ_ϵ). If they do not differ, the branch to the argument will be cut off from the tree as it cannot have caused the inconsistency. If the argument is a `constant` expression, the expression will be added to the cause of expressions (a constant is a part of the design rule condition). In the case of a `property call` expression, all the model element accessed in this expression are added to the cause of model element properties ($\zeta_{e.p}$). If the expression is a variable declaration (`let` expression), the `computeCause` algorithm will be recursively called for both arguments of this expression.

Most of the expressions in Table 4 are guarded by a condition that specify for which part of the expression the cause must be calculated. This guard consist of an expected result of the expression itself and the validated values from their arguments. As was shown in Table 2, a conjunction has four different cases that need to be considered to calculate the cause – three cases where the expected result is 'true' and one where the the expected result is 'false'. Very similar is the cause calculation of a disjunction and implication.

In the case of the quantifier we differ between two guard conditions only but as it will be iterated over all sub trees generated during the validation, the other guard conditions are used implicitly (a cause of a sub tree will be calculated only if the validated result of the sub tree differs from expected result: $\epsilon.\sigma \wedge b_i(a_i)$ where b_i is the sub tree for the variable assignments a_i). If such a quantifier has been violated the cause calculation can be done in the same way as for simple logic operations.

In the last example we started with a conjunction (ϵ_0) and calculated the cause of expressions (ζ_ϵ). The expression itself was added to the cause and the cause was calculated for its violated arguments. For which arguments the cause must be calculated depends on the expected result of the expression and the validated results of the arguments because of the expected result, the expected result of the arguments can be determined and the cause for an argument must be calculated only if the validated result differs the expected result. The condition, if for an argument the cause must be calculated is expressed in a guard condition next to the expression (already explained earlier in this section). In our example only one argument was violated for which the cause has to be calculated. Every time a violated expression was visited, this expression was added to the cause of expressions. When an expression was reached that has no arguments that validate against a Boolean result, the cause for the model element accesses was calculated. In our example this was the case at the equality expression $\epsilon_{2.1.1}$. This expression has two `property call` expressions as its arguments. One expression that accesses the name property of a method (`m.name`) and the other that accesses the name property of an operation (`o.name`). In this example we assume that both are equal. The same properties are accessed by the `equality` expression $\epsilon_{1.1.2}$. If, for example, this expression would also be violated, the cause of model element accesses would not change, only expression would be added to the cause of expressions.

The `property call` expression is the only one expression type that adds a model element access to the cause (e for the model element and p for the accessed property). In the case of quantifiers, `collect`, `select` or `size`, the source of these expressions (A) are potential model element property accesses and therefore the cause will be calculated. If a `let` expression (variable declaration) is in a violated path of an condition for both, the variable declaration ($decl$) and the condition where it is uses (in) the cause will be calculated. As both can be Boolean expressions, the cause calculation for expression is started which may add elements to the cause of expressions and to the cause of model element accesses.

4.4 Algorithm Illustrated

The following illustrates the workings of our approach on the inconsistency of the *Class* `Torch` from the example introduced in Section 2. First, the validation tree of the inconsistent validation is build. Figure 4 shows the complete validation tree. The root expression is the `let` expression which has as its left branch the declaration of the variable `attrNms`. This variable is of the OCL type *Bag* which is an unsorted collection of the attribute names of the context class, the UML *Class* `Torch`. The horizontal branch of the `collect` expression is the source of elements, analogous to all other operations that are based on collections (e. g., quantifiers). In our case these are the attributes of the `Torch`. The `collect` operation accesses the name properties of the source elements that are added to the collection `attrNms`. As the *Class* `Torch` has only one attribute, only one branch is created for the `collect` expression. This variable is visible in the condition given in the right branch of the `let` expression.

The condition of the `let` expression starts with an universal quantifier. The source of this universal quantifier are the super classes of the *Class* `Torch` which are accessed by the `allParents` property (how the `allParents` property is validated was already explained in Section 2). Because of the recursive validation of the `allParents` property, the source of the universal quantifier consist of seven model element properties. First, the `generalization` and `general` property of the *Class* `Torch` is called, then the same properties for the

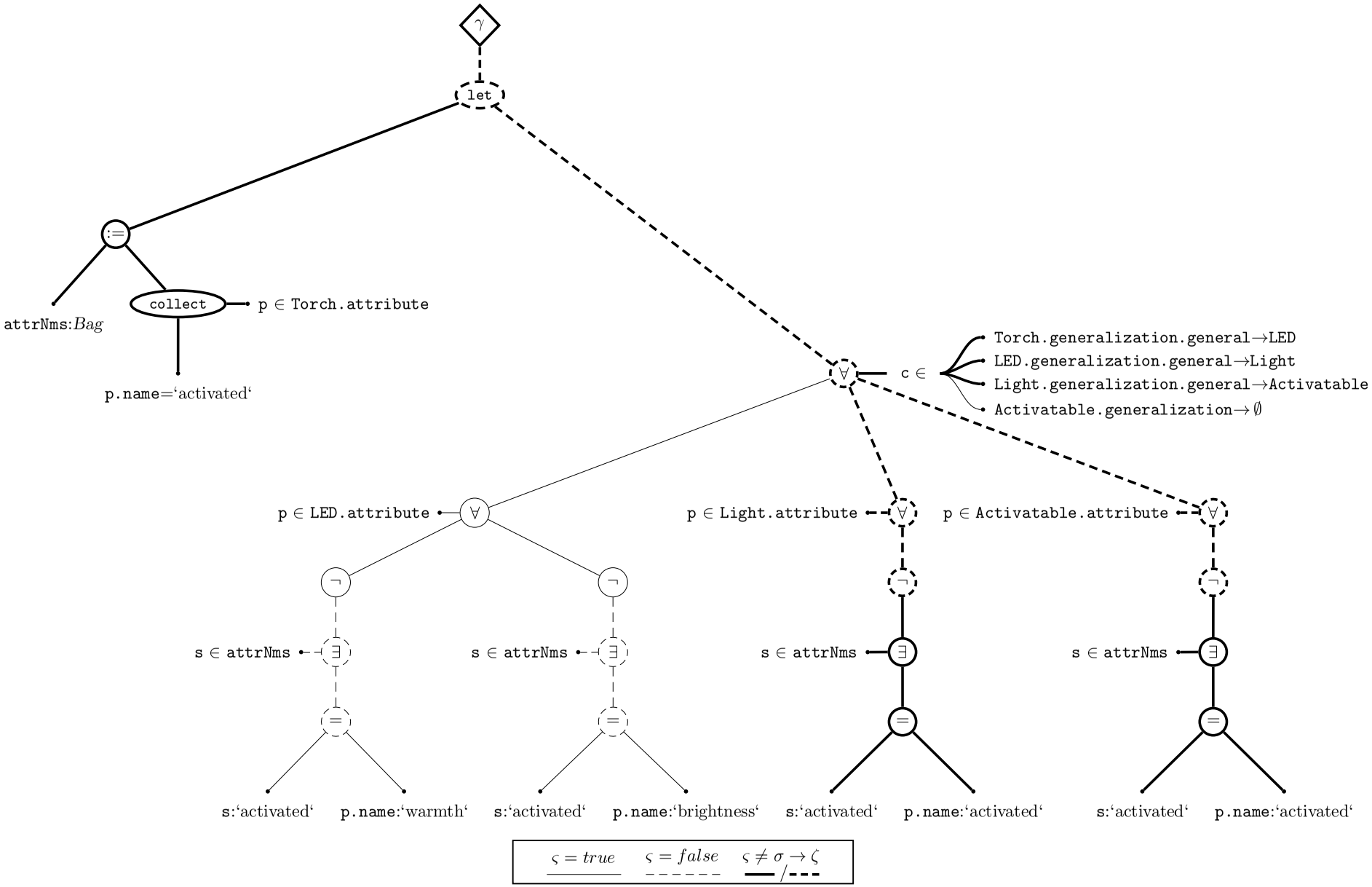

Figure 4. Validation Tree for the *Class* ‘Torch‘

Table 4
Excerpt of Cause Calculation for Different Expression Types

$\epsilon.o$	$\epsilon.\alpha$	Cause of Expressions ζ_ϵ	Cause of Model Element Properties $\zeta_{e.p}$
$\neg$	$\{a\}$	$\zeta_\epsilon = \{\epsilon\} \cup \zeta_\epsilon(a)$	
$\wedge$	$\{a, b\}$	$\epsilon.\sigma \wedge a.\varsigma \wedge \neg b.\varsigma \;:\; \zeta_\epsilon = \{\epsilon\} \cup \zeta_\epsilon(b)$ $\epsilon.\sigma \wedge \neg a.\varsigma \wedge b.\varsigma \;:\; \zeta_\epsilon = \{\epsilon\} \cup \zeta_\epsilon(a)$ $\epsilon.\sigma \wedge \neg a.\varsigma \wedge \neg b.\varsigma \;:\; \zeta_\epsilon = \{\epsilon\} \cup \zeta_\epsilon(a) \cup \zeta_\epsilon(b)$ $\neg\epsilon.\sigma \wedge a.\varsigma \wedge b.\varsigma \;:\; \zeta_\epsilon = \{\epsilon\} \cup \zeta_\epsilon(a) \cup \zeta_\epsilon(b)$	
$\vee$	$\{a, b\}$	$\epsilon.\sigma \wedge \neg a.\varsigma \wedge \neg b.\varsigma \;:\; \zeta_\epsilon = \{\epsilon\} \cup \zeta_\epsilon(a) \cup \zeta_\epsilon(b)$ $\neg\epsilon.\sigma \wedge a.\varsigma \wedge \neg b.\varsigma \;:\; \zeta_\epsilon = \{\epsilon\} \cup \zeta_\epsilon(a)$ $\neg\epsilon.\sigma \wedge \neg a.\varsigma \wedge b.\varsigma \;:\; \zeta_\epsilon = \{\epsilon\} \cup \zeta_\epsilon(b)$ $\neg\epsilon.\sigma \wedge a.\varsigma \wedge b.\varsigma \;:\; \zeta_\epsilon = \{\epsilon\} \cup \zeta_\epsilon(a) \cup \zeta_\epsilon(b)$	
$\Rightarrow$	$\{a, b\}$	$\epsilon.\sigma \wedge a.\varsigma \wedge \neg b.\varsigma \;:\; \zeta_\epsilon = \{\epsilon\} \cup \zeta_\epsilon(a) \cup \zeta_\epsilon(b)$ $\neg\epsilon.\sigma \wedge \neg a.\varsigma \wedge \neg b.\varsigma \;:\; \zeta_\epsilon = \{\epsilon\} \cup \zeta_\epsilon(a)$ $\neg\epsilon.\sigma \wedge \neg a.\varsigma \wedge b.\varsigma \;:\; \zeta_\epsilon = \{\epsilon\} \cup \zeta_\epsilon(a) \cup \zeta_\epsilon(b)$ $\neg\epsilon.\sigma \wedge a.\varsigma \wedge b.\varsigma \;:\; \zeta_\epsilon = \{\epsilon\} \cup \zeta_\epsilon(b)$	
$=$	$\{a, b\}$		$\zeta_{e.p} = \zeta_{e.p} \cup \zeta_{e.p}(a) \cup \zeta_{e.p}(b)$
$\forall$	$\{A, b\}$	$\zeta_\epsilon = \{\epsilon\} \cup \begin{cases} \bigcup \zeta_\epsilon(b_i) \vert a_i \in A \wedge \epsilon.\sigma \wedge \neg b_i(a_i).\varsigma \\ \bigcup \zeta_\epsilon(b_i) \vert a_i \in A \wedge \neg\epsilon.\sigma \wedge b_i(a_i).\varsigma \end{cases}$	$\zeta_{e.p} = \zeta_{e.p} \cup \zeta_{e.p}(A)$
$\exists$	$\{A, b\}$i	$\zeta_\epsilon = \{\epsilon\} \cup \begin{cases} \bigcup \zeta_\epsilon(b_i) \vert a_i \in A \wedge \epsilon.\sigma \wedge \neg b_i(a_i).\varsigma \\ \bigcup \zeta_\epsilon(b_i) \vert a_i \in A \wedge \neg\epsilon.\sigma \wedge b_i(a_i).\varsigma \end{cases}$	$\zeta_{e.p} = \zeta_{e.p} \cup \zeta_{e.p}(A)$
collect	$\{A, b\}$		$\zeta_{e.p} = \zeta_{e.p} \cup \zeta_{e.p}(A) \cup \left\{ \bigcup \zeta_{e.p}(b(a_i) \vert a_i \in A) \right.$
select	$\{A, b\}$	$\zeta_\epsilon = \{\epsilon\} \cup \left\{ \bigcup \zeta_\epsilon(b_i) \right.$	$\zeta_{e.p} = \zeta_{e.p} \cup \zeta_{e.p}(A)$
size	$\{A\}$		$\zeta_{e.p} = \zeta_{e.p} \cup \zeta_{e.p}(A)$
property call	$\{p, e\}$		$\zeta_{e.p} = \zeta_{e.p} \cup \{e.p\}$
let	$\{decl, in\}$	$\zeta_\epsilon = \{\epsilon\} \cup \zeta_\epsilon(a) \cup \zeta_\epsilon(b)$	
constant		$\zeta_\epsilon = \{\epsilon\}$	

Class `LED` and the *Class* `Light`, and at the end only the `generalization` property of the *Class* `Activatable` is called.

The first branch of the universal quantifier is another universal quantifier that has the attributes of the *Class* `LED` as its source from which two branches are created. The branches start with a negation followed by an existential quantifier that has the attribute names from the context class as its source (the variable `attrNms`). For each of the attribute names (only one is present in the context class, i. e., only one branch is created) is checked if it equals one of the attribute names (the attributes of the second universal quantifier) of the class from the first universal quantifier. The equality expression has the name of the attribute of the *Class* `Torch` as its left branch and the name property of the attribute of the *Class* `LED` as its right branch. The same validation is done for the other super classes of the *Class* `Torch`. The number of branches of the second universal quantifier corresponds the number of attributes of the super class and the number of branches of the existential quantifier corresponds the number of attributes of the *Class* `Torch`.

As can been seen, the validation fails, because the *Class*es `Light` and `Activatable` contain an *Attribute* named ‘activated‘. Furthermore, in the validation tree the violated parts can be easily detected. Branches that validate to ‘true‘ or to a non Boolean value are drawn as a solid line and branches that validate to ‘false‘ are drawn as a dashed line. The branches that contribute to the inconsistency (i. e., cause the inconsistency) are drawn in thick style. The validation on the *Class* `LED` does not contribute to the inconsistency, because all the validated result correspond to the expected results. In contrast, the validation for the other two super classes cause the inconsistency. From this view we can easily detect the model elements that caused the inconsistency as what we have done manually in Section 2. Moreover, we can also detect the parts of the design rule validation that are involved in this inconsistency. Something that

Table 5
29 Evaluated UML Models

	Name	#Model Elements
1	Video on Demand	104
2	ATM	220
3	Microwave Oven	290
4	Model View Controller	418
5	eBullition	513
6	Curriculum	763
7	Teleoperated Robot	1,115
8	Dice 3	1,274
9	ANTS Visualizer	1,282
10	Inventory and Sales	1,296
11	Course Registration	1,406
12	UML IOC F05a T12	1,453
13	VOD 3	1,558
14	Vacation and Sick Leave	1,658
15	Home Appliance	1,707
16	HDCP Defect Seeding	1,784
17	DESI 2.3	1,974
18	iTalks	2,212
19	Hotel Management Sys.	2,583
20	Biter Robocup	2,632
21	Calendarium	2,809
22	UML LCA F03a T1	2,983
23	<unnamed>	5,373
24	NPI	7,110
25	Word Pad	8,078
26	dSpace 3.2	8,761
27	OODT	9,828
28	Insurance Network Fees	16,255
29	<unnamed>	33,347

is hardly possible when considering the syntactic structure of a design rule condition only. Through the recursive use of the expressions in Table 4 the calculation of the cause can be fully automated which is implemented as a plug-in for the IBM Rational Software Architect. The plug-in is available as a download on our homepage http://www.sea.jku.at/tools/.

5 Evaluations

We evaluated our approach with regard to correctness (causes are complete and minimal), effectiveness (reductions compared to existing approaches that could be used to approximate cause), and computational scalability. To evaluate this we use 29 UML models shown in Table 5. The table lists the names of the models (from which to domain can be inferred) and their respective sizes. Furthermore, the 29 UML models are validated against a set of 20 OCL design rules listed in Appendix A in the supplemental material.

5.1 Correctness

For correctness, calculated causes must be complete in that they identify all elements that cause the inconsistency and minimal in that they do not contain any elements that do not cause the inconsistency.

For the proof of correctness, we can make the following assumptions which are true for commonly used modeling languages and design rules: 1) the design rule is well formed (i. e., syntactically correct) and thus can be evaluated, 2) the design rule starts its evaluation always at a context element provided, 3) all model elements accessed during that rule's evaluation must be reached by navigating from this context element (i. e., there must not exist any "floating" model parts that are not connected to other parts but are accessible), and 4) the design rule is expected to validate to 'true'. All design rules we encountered to date satisfy these conditions. It is also generally true that design rules are hierarchical ordered expressions. We basically distinguish between Boolean expressions, where the result is Boolean, and expressions that access/manipulate model element properties (e. g., strings, collections). If we think of the syntactic structure as a tree then its leave expressions are typically model property calls and the Boolean expressions are above. Since an inconsistency can be caused by either an incorrect model or an incorrect design rule, algorithm 2 computes both : 1) the part of the design rule (the Boolean expressions) that causes a given inconsistency (lines 5-11) and the part of the model (the model element properties) contributing to this cause (lines 12-21). Each model element has properties and a model is essentially a collection of model element properties. The manner in which a model element property may cause an inconsistency is if this property is an argument in some Boolean expression above it in the tree – and only then if that Boolean expression causes the inconsistency, i. e., if the Boolean expression is in the cause. We first observe that a model element property only then causes an inconsistency if it is an argument of a Boolean expression that causes the inconsistency. The correctness of this statement is easy to follow. A model element property can only cause an inconsistency if it somehow negatively influences the Boolean expressions that make up the inconsistency.

$$\exists e.p \in \zeta_{e.p}, \exists \epsilon \in \zeta_\epsilon | \epsilon_{e.p} \in \epsilon.\alpha$$

In the following, we thus merely need to show that our approach correctly computes which Boolean expressions cause an inconsistency. The model element properties that cause an inconsistency are then simply all model element properties in the arguments of these Boolean expressions. Line 16 of algorithm 2 thus only adds model element properties to the cause if it also adds the expression that accessed it to the cause (line 15).

$$\zeta_\epsilon \rightarrow correct \quad \Rightarrow \quad \zeta_{e.p} \rightarrow correct$$

To determine whether an expression causes an inconsistency, algorithm 2 first computes for every Boolean expression whether its validation result differs from the expected result (Line 6). A Boolean expression is always expected to validate to true. Thus starting from the root, the expected result must be 'true' unless the expression uses a Boolean operator that negates (as in the logical 'not'). It is easy to see that the expected

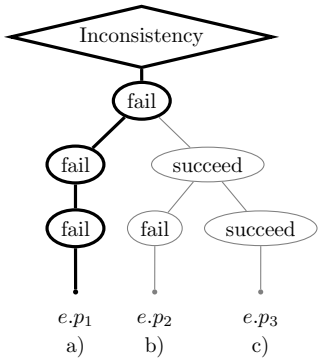

Figure 5. Combinations of Expression Validations in an Inconsistent Validation Tree

result is correctly computable top down under the assumption that the root expression is always expected to validate to 'true' (which is so defined as mentioned in our assumptions above). Likewise, it is easy to see that the computation of the validation result is correct because it is so observed during the validation of the design rule. The computation of the validation result is bottom up and simply transcribes the intermediate, Boolean results encountered.

We say that a Boolean expression *succeeds* if its validated result is equal the expected result. Otherwise it *fail*s. If a Boolean expression succeeds then the Boolean expression returned the result that was expected for consistency. Hence, a Boolean expression that succeeds cannot cause an inconsistency. Indeed, if the root Boolean expression succeeds then there is not even an inconsistency. An inconsistency can only happen if some Boolean expression(s) fail – including the root expression. However, even in case of inconsistency some Boolean expressions may succeed (recall example in Figure 4). (Sub) Expressions that succeed simply point to parts of design rules that were not inconsistent and hence did not cause the inconsistency. It follows that Boolean expressions that succeed cannot cause inconsistencies. Only Boolean expressions that fail may. Line 6 if the algorithm 2 thus only explores expressions that failed.

Unfortunately, not all Boolean expressions that fail cause inconsistencies. Figure 5 lays out all three possible scenarios how failed/succeeded Boolean expression may be composed of in case of inconsistency. Recall that a design rule typically forms a tree hierarchy of Boolean expressions. The root Boolean expression must have failed for there to be an inconsistency. Under a Boolean expression that fails must either be another Boolean expression that failed or a property call expressions the caused the Boolean expression to fail. This is true always because a Boolean expression can only fail if one of its arguments provided the 'incorrect' input. This failed child/argument expression thus caused the parent expression to fail and if the parent expression caused the inconsistency and the child expression caused the parent expression then clearly the child expression also caused the inconsistency.

However, underneath a failed expression there may also be any number of Boolean expressions that succeeded. Similar to before, a Boolean expression can only succeed if it has at least one Boolean child expression that succeeds or property call expressions. We already know from above that succeeded expressions cannot cause inconsistencies. They are thus ignored (filtered by the algorithm). However, as Figure 5 shows there may also be failed expressions underneath succeeded expressions. These failed expressions cannot cause the inconsistency because though the expression failed, this failure did not cause the parent expression to fail and thus the failure had no influence on the inconsistency. So, failed child expressions do not cause inconsistencies if they have some parent expressions (recursive) that succeeded.

Concluding, the top node in Figure 5 must be marked 'fail' always. Some of its child expressions must have failed for any parent to have failed but not all child expressions have to. The right child (b, c) in Figure 5 is marked 'succeed' (imagine the root expression is a logical 'and' that failed because the left child failed even if the right child succeeded). The right child expression, even though it failed, does not cause the parent to fail and hence no inconsistency could have been caused by it. Further note that the right child has two more child expressions of its own where one succeeded (c) and the other failed (b). We assert that even though the right child has children that failed none of them can cause the overall inconsistency. The correctness of this statement lies in the fact that if the failed sub expression did not cause its parent expression to fail and this parent thus did not cause its parent (the root in this case) to fail. There is no transitive effect in a validation. If an expression fails only those immediate child expressions that caused it, contributed to this failure. A failure that is hidden below a successful child expression can safely be ignored. We observe the following:

An expression causes an inconsistency if it fails and its parent expression causes the inconsistency (recursive except for parent).

$$\forall \epsilon_x \in \zeta_\epsilon | \epsilon_x.\sigma \neq \epsilon_x.\varsigma \wedge \epsilon_x \neq \epsilon_0 \Rightarrow \epsilon_x.\rho.\sigma \neq \epsilon_x.\rho.\varsigma$$

We see that Algorithm 2 correctly implements this, because it starts the calculations at the root expression (ϵ_0) for which the algorithm is called. The cause of expression is initialized including the root expression. At the begin, for each argument of the root expression it is checked what type of expression the argument is (Line 5, 12, 14, 17). For Boolean expression it is checked if the expected and validated result differ (Line 6), and if yes, the root expression is added to the cause of expressions (Line 7) and the cause calculation algorithm is repeated recursively for each child expression that failed. The recursive call of the algorithm

stops if a leaf (property call / constant expression) has been reached which are added to the corresponding causes always. If the expected and validated result are equal, the corresponding branch is removed from the tree (Line 10). Therefore, any Boolean expressions in this removed sub-tree where the expected result might differ its validated result won't be reached and the calculated cause. We conclude that all expressions and model elements properties identified by the algorithm must have caused the inconsistency (correctness) and no expression/model element property can be missing (minimal).

Please note that there may be very well some model element properties in the cause of an inconsistency that occur in branches that are not in the cause of expressions because a model element property can be accessed by more than one expression . For correctness, we thus define further:

A model element property causes an inconsistency if it is accessed by at least one expression that causes the inconsistency.

$$\forall \epsilon_{e.p} \in \zeta_{e.p} \exists \epsilon_x \in \zeta_\epsilon | \epsilon_{e.p}.\rho = \epsilon_x$$

But because of the strict top-down navigation in the validation tree (from expression to model element properties and *not* from model element properties to the expressions) for generating the cause, it is guaranteed that the accessed model element properties from the cause can be uniquely associated to the detected inconsistency.

5.2 Effectiveness

In order to understand the performance of the approach, we empirically evaluated its computational scalability and its effectiveness. This validation was done by analyzing 20 commonly-used OCL design rules on 29 UML models. The OCL design rules are all UML well-formedness rules that are applicable on all UML models. Unfortunately, model specific rules are hardly available due the lack of support by the tools used in productive process. However, the used design rules are representative enough to show the performance of the proposed approach and, as the design rules are very simple, it can be assumed the results would be better with more complex design rules than they already are.

For computational scalability, we measured the time needed for our approach to compute the cause for arbitrary inconsistencies. For effectiveness, we measured the ratio of the cause size compared to the worst-case assumption that all model elements that were accessed during the validation of an inconsistent design rule must contribute to the cause. This worst case computation is conservative and computable by state of the art [8] which displays all, during a design rule validation accessed model elements, as potential cause of an inconsistency. Nonetheless, this comparison is interesting because it reveals whether these existing approximations would already have been effective enough for computing causes also (thus invalidating the need for our approach). The complete list of design rules is listed in Appendix A in the supplemental material but only nine of them cause inconsistencies which are listed here:

Design Rule 13: An interface can only contain public operations and no attributes.
Design Rule 16: The type of operation parameters must be included in the name space of the operation owner.
Design Rule 8: At most one association end may be an aggregation or composition.
Design Rule 6: The connected classifiers of the association end should be included in the name space of the association.
Design Rule 5: A message direction must match class association.
Design Rule 15: Operation parameters must have unique names.
Design Rule 10: A class may use unique attribute names.
Design Rule 12: The elements owned by name space must have unique names.
Design Rule 14: No two class operations may have the same signature.

These design rules were validated on 29 UML design models with varying sizes – the smallest UML modeling having merely 104 model elements and the largest 33,347 elements, with a total of 82,279 elements. A total of 14,111 inconsistencies were detected among these models and for each inconsistency we computed the cause. Figure 6 depicts the average as well as the maximum and minimum sizes of the cause in relationship to the number of model elements accessed by its inconsistency. The numbers on the x-axis are the corresponding number of the design rule. Note that in the worst case, every model element accessed by that inconsistency contributed to the cause. Yet, in context of these diverse design rules, models, and numerous inconsistencies we found that inconsistencies rarely involve all model elements looked at during validation. We observed that the cause is in average 19-81% (maximum of 91%) of these elements only. This provides a strong indication that the entire scope of accessed elements is rarely involved in the cause of an inconsistency – and thus the motivation that our approach provides improvements to the designer in helping understand the cause. Since this table is based on 14,111 observations, it is statistically highly significant.

To understand how many elements typically cause an inconsistency in absolute numbers, we depict the sizes of causes in Figure 7, again grouped by design rules (the number corresponds the number of the design rule). We see that in average 12 model elements are involved in causing design rule 5 (with as few as 8 and as much as

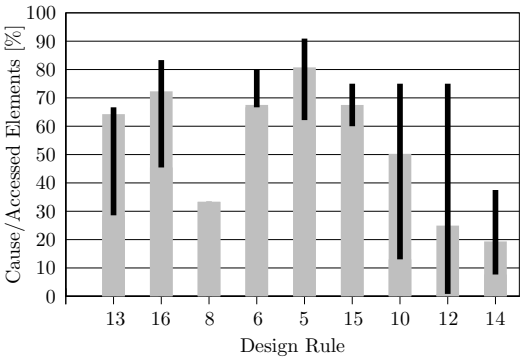

Figure 6. Ratio of Cause and Accessed Elements

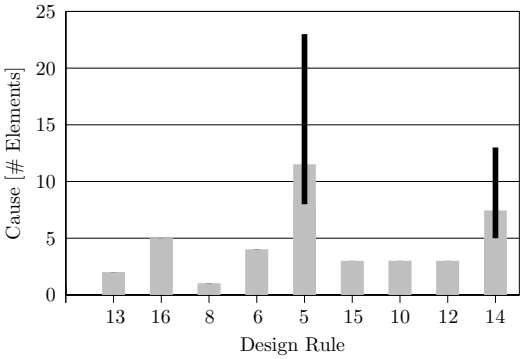

Figure 7. Size of the calculated Causes

23). As can be expected, the sizes of causes varies with design rules; however, only one design rule consistently has causes above a dozen elements. This is also good in that the causes are typically small quantities that could be investigated manually if needed. However, it is important to distinguish a fact that has been explored little in related work. Often, validations that lead to inconsistencies are incomplete validations. That is, consistency checkers terminate their validation if it becomes obvious that the design rule is no longer satisfiable. For example, if a is 'false' in $a \wedge b$ then b is no longer validated. The cause can thus be computed for both a complete validation (e. g., one were both a and b are validated in $a \wedge b$ even if a is 'false') and a partial validation (e. g., one were only a is validated in $a \wedge b$ if a is 'false'). Our approach can be used to compute the cause for partial or complete validation; however, it must be understood that partial validations lead to partial causes only. Therefore, some existing approaches in state of the art that might be perceived as approximations of causes would in fact reveal incomplete causes only [9], [11]. That is, if both a and b are 'false' in $a \wedge b$ then a partial validation would fail to include the cause of a.

Figure 8 shows the average size of the cause depending on the design rule size. The number next to the data points are the number of the design rule. The design rule size is the number of nodes in the validation tree and is an indicator for the complexity of the design rule. It

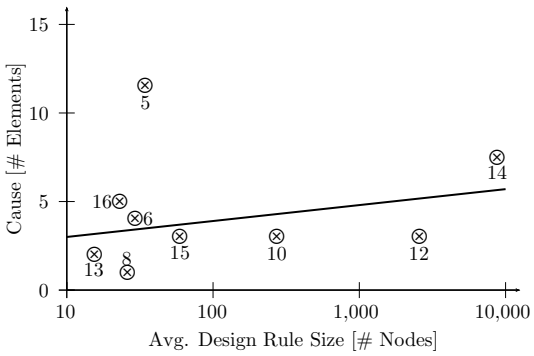

Figure 8. Size of the Cause depending on the Design Rule Size

is worth mentioning it that the most complex design rule is not necessarily the design rule with the largest cause. Design Rule 12, for example, has the complete name space in its scope but if only two elements are named equally, only three elements are in the cause, but probably several hundred are in the scope of the design rule's validation. This circumstance is reflected in figure 6 where only about 25% of the scope is in the cause of an inconsistency.

As can been seen in this empirical evaluation, the proposed approach scales even on larger and more complex design rule validation regarding appropriate feedback to the user. Based on the experience of that evaluation it can be assumed that the approach will perform still better with increasing design rule complexity (also shown in [27]).

5.3 Computational Scalability

The first part of the evaluation has shown that our approach is correct and effective in that it provides smaller causes than could be computed thus far. Last but not least, we demonstrate that our approach is also very fast and scalable. We evaluated the performance of our approach by measuring that time it takes to instantiate an inconsistency (through validation) and calculating the cause. We did so for all 14,111 inconsistencies on a Intel Core 2 Quad CPU (Q9550), running at 2,883Mhz with 8GB Ram on a 64bit Linux (2.6.34) system. Figure 9 shows the average validation times of the design rules applied on the models. The x-axis shows the average number of nodes for the different design rules and the y-axis the average time in milliseconds that it takes to calculate the cause.

We see that the computation of a cause is linear to the size of the design rules (as proposed in Section 5.3). Seven of the nine design rules take in average less than a millisecond and is thus quite fast to compute. Overall, the causes of 97% of all inconsistencies could be computed in less than 1ms each, 99.8% in less than 100ms.

While the size of the cause does not depend on the size of the design rule, the design rule's validation

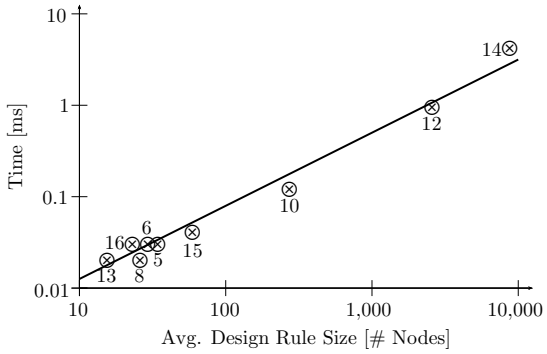

Figure 9. Validation Times versus Design Rule Size

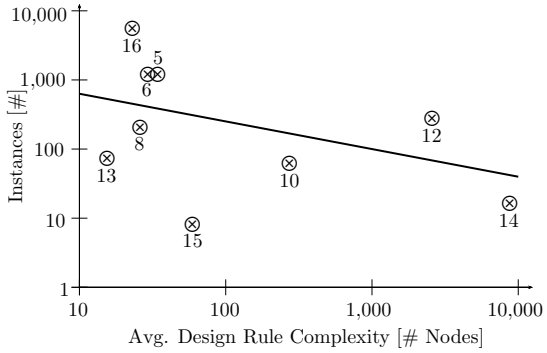

Figure 10. Number of Design Rule Instances depending on the Design Rule Size

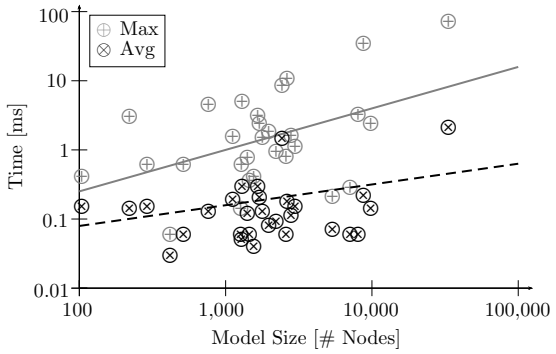

Figure 11. Validation Times versus Model Size

time depends on the design rule complexity, i. e., the validation time increases linear to the complexity of the design rule. In figure 10 we analyzed the frequency of the design rules, how many instances are validated of each design rule. As we can see, the number of instances decreases with the complexity of the design rule, so the effect that more complex design rules need more time to validate is weakened by the fact that the number of design rules that must be validated is lesser than the number of less complex design rules.

While the charts above, depicted in scalability in terms of design rule complexity, the following depicts scalability as a factor of the 29 model sizes. There we see that the computation of causes does not exhibit scalability problems with larger model sizes but rather remain nearly constant. Figure 11 shows the average time to validate a design rule and calculate the cause depending on the model size. There is only a small increase of the validation time identifiable that comes from the increased overhead of our approach. But even the highest validation and calculation time is less than 100ms.

To conclude the computational scalability of our approach, we provide an overview about the computational complexity. The computational complexity of this approach depends mainly on the complexity of the design rule and on the structure of the model. We will show this based on the example design rule from Section 2. Let p be the number of parents that a class has, f_1 is the number of features of the class that is checked and f_2 is the number of features of the parent class. We analyze the design rule structure from the inside to the outside, beginning where it is checked if a feature of a parent class is in the set of features of the class that will be checked. The selection of the feature names of the classes features can be omitted because this will be a constant offset, so the resulting complexity is $O(f_2)$. This must be instantiated for each parent class. The resulting complexity of this particular design rule is: $O(p \times f_2)$. The traversal must be done twice, where the second one is not necessarily a complete traversal (not violated branches are cut off). But as this number of traversals is a constant number it can be omitted for the complexity discussion.

Similar behavior can be discovered for the memory consumption. The base memory consumption corresponds the number of leave nodes of the validation tree ($O(p \times (f_2))$). Additionally the height h of the validation tree must be considered. In the best case for a balanced tree the memory consumption is $O(log(h) \times p \times f_2)$ and the worst case where each node contains only a single branch the memory consumption is $O(h \times p \times f_2)$. The evaluations on real world models have shown that both cases are not realistic and the reality lies somewhere between (highly dependent of the design rule structure). It must be noted that the memory can be freed after the computation of each inconsistency. Thus the number of inconsistencies does not affect memory consumption.

6 Conclusions

This paper presented a new and novel approach for identifying how design rules cause a given inconsistency and what model elements are involved. We demonstrated that our approach computes causes correctly, is very fast, and fully automated/tool support. We demonstrated through empirical evidence that the causes of inconsistencies are almost always a subset of the model elements involved in the computation of inconsistencies. Thus, the approach gives advice on where and how to start repairing an inconsistency. This work is useful for better understanding and visualizing inconsistencies

but it is also useful for assessing trust in models – that is, model elements that are causing inconsistencies are clearly less trustworthy than the ones that do not. In future work, we will explore how to repair inconsistencies (in part already explored in [26]) since any correct repair must eliminate the cause. It is also future work to assess the usefulness of the approach to software engineers in better coping with inconsistencies and avoiding potentially costly errors caused by an otherwise incorrect or incomplete understanding.

7 Acknowledgments

This research was funded by the Austrian Science Fund (FWF): P 25289-N15

References

[1] X. Blanc, A. Mougenot, I. Mounier, and T. Mens. Incremental Detection of Model Inconsistencies Based on Model Operations. In P. van Eck, J. Gordijn, and R. Wieringa, editors, *CAiSE*, volume 5565 of *Lecture Notes in Computer Science*, pages 32–46. Springer, 2009.

[2] J. Cabot and E. Teniente. Incremental Evaluation of OCL Constraints. In E. Dubois and K. Pohl, editors, *CAiSE*, volume 4001 of *Lecture Notes in Computer Science*, pages 81–95. Springer, 2006.

[3] 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.

[4] H. K. Dam and M. Winikoff. Supporting change propagation in UML models. In *ICSM*, pages 1–10. IEEE Computer Society, 2010.

[5] K. H. Dam and M. Winikoff. Cost-based BDI plan selection for change propagation. In L. Padgham, D. C. Parkes, J. P. Müller, and S. Parsons, editors, *AAMAS (1)*, pages 217–224. IFAAMAS, 2008.

[6] A. Demuth, R. E. Lopez-Herrejon, and A. Egyed. Constraint-Driven Modeling through Transformation. In Z. Hu and J. de Lara, editors, *ICMT*, volume 7307 of *Lecture Notes in Computer Science*, pages 248–263. Springer, 2012.

[7] X. Deng, V. Haarslev, and N. Shiri. Measuring Inconsistencies in Ontologies. In E. Franconi, M. Kifer, and W. May, editors, *ESWC*, volume 4519 of *Lecture Notes in Computer Science*, pages 326–340. Springer, 2007.

[8] A. Egyed. Instant consistency checking for the UML. In Osterweil et al. [25], pages 381–390.

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

[10] A. Egyed, A. Demuth, A. Ghabi, R. E. Lopez-Herrejon, P. Mäder, A. Nöhrer, and A. Reder. Fine-Tuning Model Transformation: Change Propagation in Context of Consistency, Completeness, and Human Guidance. In J. Cabot and E. Visser, editors, *ICMT*, volume 6707 of *Lecture Notes in Computer Science*, pages 1–14. Springer, 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] A. Evans, R. B. France, K. Lano, and B. Rumpe. The UML as a Formal Modeling Notation. In J. Bézivin and P.-A. Muller, editors, *UML*, volume 1618 of *Lecture Notes in Computer Science*, pages 336–348. Springer, 1998.

[13] A. Finkelstein, D. M. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh. Inconsistency Handling in Multi-Perspective Specifications. In *ESEC '93: Proceedings of the 4th European Software Engineering Conference on Software Engineering*, pages 84–99, London, UK, 1993. Springer-Verlag.

[14] 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.

[15] 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.

[16] D. Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, April 2002.

[17] A. Kalyanpur, B. Parsia, E. Sirin, and J. A. Hendler. Debugging unsatisfiable classes in OWL ontologies. *J. Web Sem.*, 3(4):268–293, 2005.

[18] R. Kowalski. Logic for Problem-solving. *DCL Memo 75*, 1974.

[19] M. H. Liffiton and K. A. Sakallah. Algorithms for Computing Minimal Unsatisfiable Subsets of Constraints. *J. Autom. Reasoning*, 40(1):1–33, 2008.

[20] 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.

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

[22] A. Nöhrer, A. Biere, and A. Egyed. Managing SAT inconsistencies with HUMUS. In U. W. Eisenecker, S. Apel, and S. Gnesi, editors, *VaMoS*, pages 83–91. ACM, 2012.

[23] OMG. OCL 2.3.1 Specification. http://www.omg.org/spec/OCL/2.3.1/, 2012.

[24] OMG. UML 2.1 Specification. http://www.uml.org/, 2012.

[25] L. J. Osterweil, H. D. Rombach, and M. L. Soffa, editors. *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006*. ACM, 2006.

[26] A. Reder and A. Egyed. Computing repair trees for resolving inconsistencies in design models. In *ASE*, pages 220–229. ACM, 2012.

[27] A. Reder and A. Egyed. Incremental Consistency Checking for Complex Design Rules and Larger Model Changes. In *MoDELS*, Lecture Notes in Computer Science, pages 202–218. Springer, 2012.

[28] J. E. Robbins. ArgoUML, v0.32.1. http://argouml.tigris.org/, March 2011.

[29] M. H. Van Emden and R. A. Kowalski. The Semantics of Predicate Logic as a Programming Language. *J. ACM*, 23:733–742, October 1976.

[30] M. Vierhauser, D. Dhungana, W. Heider, R. Rabiser, and A. Egyed. Tool Support for Incremental Consistency Checking on Variability Models. In D. Benavides, D. S. Batory, and P. Grünbacher, editors, *VaMoS*, volume 37 of *ICB-Research Report*, pages 171–174. Universität Duisburg-Essen, 2010.

[31] J. Winkelmann, G. Taentzer, K. Ehrig, and J. M. Küster. Translation of Restricted OCL Constraints into Graph Constraints for Generating Meta Model Instances by Graph Grammars. *Electr. Notes Theor. Comput. Sci.*, 211:159–170, 2008.

[32] 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.

[33] C. Xu, S.-C. Cheung, and W. K. Chan. Incremental consistency checking for pervasive context. In Osterweil et al. [25], pages 292–301.

Alexander Reder received his master's degree in software engineering and PhD degree in computer science from the Johannes Kepler University Linz in Austria, in 2009 and 2013 respectively. He is currently working as researcher at the Johannes Kepler University, Institute for Systems Engineering and Automation. His research interests include model driven engineering and consistency management in model based development.

Alexander Egyed is a Full Professor at the Johannes Kepler University (JKU), Austria. He received his Doctorate degree from the University of Southern California, USA and worked for Teknowledge Corporation, USA (2000-2007) and the University College London, UK (2007-2008). He is most recognized for his work on software and systems modeling – particularly on consistency and traceability of models. Dr. Egyed's work has been published at over a hundred refereed scientific books, journals, conferences, and workshops, with over 3000 citations to date. He was recognized as the 10th best scholar in software engineering in Communications of the ACM, was named an IBM Research Faculty Fellow in recognition to his contributions to consistency checking, received a Recognition of Service Award from the ACM, a Best Paper Award from COMPSAC, and an Outstanding Achievement Award from the USC. He has given many invited talks including four keynotes, served on scientific panels and countless program committees, and has served as program (co-) chair, steering committee member, and editorial board member. He is a member of the IEEE, IEEE Computer Society, ACM, and ACM SigSoft.