Electronic Communications of the EASST
Volume 36 (2011)


# Proceedings of the Workshop on OCL and Textual Modelling (OCL 2011)

## Using an OCL Impact Analysis Algorithm for View-Based Textual Modelling

Axel Uhl, Thomas Goldschmidt and Manuel Holzleitner

20 pages

Guest Editors: Jordi Cabot, Tony Clark, Manuel Clavel, Martin Gogolla
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

# Using an OCL Impact Analysis Algorithm for View-Based Textual Modelling

**Axel Uhl[1], Thomas Goldschmidt[2] and Manuel Holzleitner[3]**

[1] axel.uhl@sap.com
[3] manuel.holzleitner@sap.com
SAP AG
Walldorf, Germany
[2] thomas.goldschmidt@de.abb.com
Industrial Software Systems,
ABB Corporate Research,
Ladenburg, Germany

**Abstract:** The Object Constraint Language (OCL) has become a vital part of many frameworks, tools and languages within model-driven engineering. One such application of OCL is the use for describing rules in concrete syntax definitions. Within the textual modeling framework FURCAS, OCL is extensively used for the definition of lookup and attribution rules. Based on these rules the model which is described by such a textual representation is created and updated accordingly. Changes on models over which such an expression is specified require the expression to be re-evaluated to keep the constructed model up-to-date. However, the effort for re-evaluating OCL expressions over a set of model elements grows with the number of elements, the complexity of the expressions, and the number of model changes. Thus, having large models and/or complex expressions places considerable performance costs on OCL evaluation. Techniques to reduce this effort have been presented in previous work but do not cover the full range of OCL expressions, in particular calls to operations defined in OCL, including recursive operations. In this paper, we present an approach that is applicable to the full range of OCL expressions. We validated our approach based on a large set of models and complex expressions to evaluate the performance impact of our newly introduced techniques.

**Keywords:** OCL, Impact Analysis, Change Propagation, Incremental Re-Evaluation

## 1 Introduction

The Object Constraint Language (OCL) [RG99, CW02, The10], as a standard for defining constraints on models as well as metamodels has been widely adopted in modeling approaches and tools over recent years. When constraints are evaluated they tell whether a model is in a consistent state or not. Furthermore, OCL is used as query language in many modeling tools and frameworks, such as model transformation [The05] or source code analysis [SS08]. Additionally OCL is used for lookup or attribution rules of concrete syntax definitions. The textual modeling framework FURCAS [GBU09a, GBU09b] makes extensive use of OCL for such purposes.

```
context Department::allSubDepartments() body:
    self.subDepartments->iterate(dep; subDeps = Set{self} |
        subDeps->union(dep.allSubDepartments()))
```

Listing 1: The definition of the `allSubDepartments()` operation.

For example, it uses parameterized OCL queries to express lookup rules that resolve references expressed by textual identifiers.

However, modifications to the model require such expressions to be re-evaluated as their results may have changed. Re-evaluating a large set of OCL expressions on a potentially large model can be very time consuming and therefore hinders the usability of such modelling tools significantly. A solution required here needs to find out if an expressions needs to be re-evaluated given a certain change of the model. Based on this information, the textual modeling framework can update models according to the attribution and lookup rules.

This problem has been investigated before in [CT05, CT09, AHK06]. However, the solutions presented there are incomplete especially with regard to recursive operation calls as well as loops and iterations.

The contribution of this paper is an algorithm for reliable and efficient impact analysis of OCL expressions. First, we introduce a general approach for computing the impacted context elements of an OCL expression after model changes. We show by induction that all kinds of expressions can be handled by this approach. We validate our approach using a non-trivial metamodel and a set of corresponding textual syntax definitions which include complex OCL expressions and show that the effort of computing these paths amortizes reasonably.

The remainder of this paper is structured as follows. To clarify the problem statement, we give a small motivating example in Section 2. Section 3 discusses related work. We provide an analysis on how change events affect expression values in Section 4. Our approach for tracing back to possibly affected context elements is introduced in Sections 5 and 6. Section 7 presents the results of the validation which we conducted. Finally, Section 8 concludes and outlines future work.

## 2 Motivating Example

In this section we will introduce a motivating example that clarifies the problem of OCL impact analysis within textual modeling. This example will serve as running example throughout the rest of the paper and will be referred to from the sections that introduce our approach.

Our running example is based on a metamodel that defines classes for `Department`, `Person`, `Manager` and `Project`. Figure 1 depicts this example metamodel. A `Department` may have several sub-departments, one `Manager` and several `Employees`. `Employees` work in `Projects`. The `Department` class defines an operation `allSubDepartments()`, implemented in OCL as shown in Listing 1, which returns the set of all `Departments` which are transitively contained in the `subDepartments` containment association. Listings 2 and 3 show two textual syntax definitions for this metamodel, one for the creation of `Departments` and one for `Projects`.

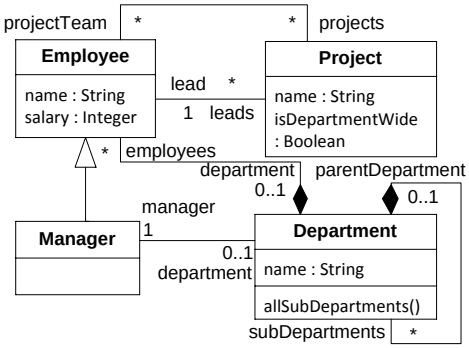



Figure 1: The department metamodel.

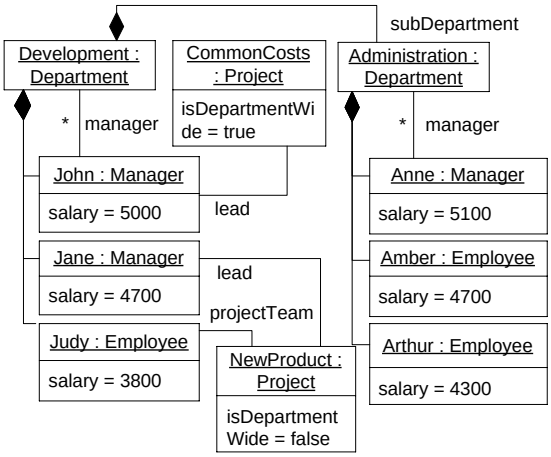



Figure 2: An example instance of the department metamodel.

FURCAS allows for the definition of multiple syntaxes as views on a single underlying model (called view types in FURCAS [GBU09b]). Therefore, it is possible to edit the same model using different views on it. We use the `Departments` view type use in this running example for the definition of departments and their managers and employees. This view type includes an example usage of the lookup functionality for which FURCAS uses OCL expressions parameterized with a question marks. In this case the expression is used to select an employee of the department by name and assign him/her as the `Manager` of the `Department`. The second view type `Projects` is responsible for providing a syntax for creating `Projects` and assign project leaders as well as `Employees` working in that project. There are two different notions of projects distinguished by its property `isDepartmentWide`. These notions are also reflected in the two different alternatives of the `Project` template. If `isDepartmentWide` is true, that means that all employees of the project leader's `department` are automatically assigned to that project (alternative 1). The second alternative lets modelers assign employees to projects by their name.

Figure 2 shows an example model which is modified through two views as shown in Listings 4 and 5. The `Department` view shows two `Departments` "Development" and "Administration" both having 3 employees. The `Projects` view shows two projects "NewProduct" and "CommonCosts" which is a department wide project. According to the first alternative of the `Projects` template, all employees of the "Development" `Department` are assigned to the "CommonCosts" `Project`.

If we consider a change that adds a new employee "Jack" to the "Development" department, this modification invalidates the result of the query of the first alternative of the `Project` template. Also, a change of the project leader from "Jane" to "Amber" would invalidate this result. Removing the "Administration" department from the sub-departments of the "Development" department would also require the employees of "Administration" to be removed from the "CommonCosts" `Project`. To be able to update the model accordingly, all OCL expressions for all context elements would have to be re-evaluated. With the approach presented in this paper, we

```
1  viewtype Departments {
2      template Department {
3          ‘‘department’’ name
4          ‘‘managed’’ ‘‘by’’ ‘‘:’’ manager {
5              query = self.employees->select(e | e.oclIsTypeOf(Manager) and e.name = ?) }
6          ‘‘{‘‘ employees {separator = ‘‘;’’} ‘‘}’’
7      }
8      template Employee {
9          ‘‘employee’’ name ‘‘:’’ salary
10     }
11     template Manager {
12         ‘‘manager’’ name ‘‘:’’ salary
13     }
14 }
```

Listing 2: Concrete syntax definition for the department metamodel.

```
1  viewtype Projects {
2      template Project {
3        ‘‘project’’ name ‘‘lead’’
4        ‘‘by’’ lead { query = Employee.allInstances()->select(e | e.name = ?) }
5        [[ —alternative 1
6          {{ isDepartmentWide = true }}
7          ‘‘departmentWide’’
8          {{ projectTeam = self.lead.department.allSubDepartments().employees }}
9         | —alternative 2
10         {{ isDepartmentWide = false }}
11         ‘‘{‘‘ employees {query = self.lead.department.employees->select(e | e.name = ? ),
12                                                                  separator = ‘‘;’’}
13         ‘‘}’’
14       ]]
15     }
16 }
```

Listing 3: Concrete syntax definition for the department metamodel.

will show that we can reduce the set expressions and context elements that has to be re-evaluated to a minimum.

## 3 Related Work

The problem which is tackled in this paper is related to the problem of incremental attribute grammar evaluation to which a solution is presented in [RTD83]. Interesting programming environments can be based on such approaches such as the Cornell Program Synthesizer [TR81]. The approach presented here handles attribute definitions expressed in OCL which includes recursive operations. Such expressions are used by the FURCAS language workbench [GBU09a, Gol10] to update features of a model that represents the program's abstract syntax tree.

There is a multitude of textual modeling frameworks such as EMFText [HJK+09] or XText [Fou10]. Many of them also rely on a declarative specification of the concrete textual syntax from which they generate parsers and pretty printer components. However, none of them facilitates the full power of OCL expressions to compute attribute values for their abstract syntaxes.

```
1 department Development managed by John {
2     manager John : 5000; manager Jane : 4700; employee Judy : 3800
3 }
4 department Administration managed by Anne {
5     manager Anne : 5100; employee Amber : 4200; employee Arthur : 4300
6 }
```

Listing 4: View on the example model using the `Department` view type.

```
1 project NewProduct lead by Jane { Judy }
2 project CommonCosts lead by John departmentWide
```

Listing 5: View on the example model using the `Projects` view type.

This part is mostly implemented in a host language such as Java. Therefore, these approaches also do not include an incremental attribute computation based on specified dependencies. See [GBU08] for an complete overview on textual modeling approaches including a classification of their capabilities.

The re-evaluation problem specific to OCL expression has already been investigated in [CT05, CT09, AHK06]. Those existing approaches distinguish between a so-called *class-scope analysis* and an *instance-scope analysis*. The class scope analysis deals with the registration of event handlers to specific types of events so that for example an OCL expression involving only a certain association only gets re-evaluated when links of that specific association are added or removed. The *instance-scope analysis* is then responsible for efficient reduction of the number of context elements for which the expression has to be re-evaluated. However, the solutions presented there are incomplete, particularly with regard to operation calls, recursive operation calls and complex iterations and loops.

EMF-INCQUERY [BHRV10] is another approach for incrementally executing queries over large scale EMF models. The approach is based on the graph pattern formalism with which a developer can formulate the queries. EMF-INCQUERY uses a caching infrastructure to store results of executed queries. The cache implementation of EMF-INCQUERY uses the so called RETE algorithm [For82] to store *partial matches* of the graph patterns which are used in the query. The cache recognizes modifications to the underlying model by utilizing the event mechanism provided by EMF. Using this approach allows for a significant improvement in query performance. However, as EMF-INCQUERY strongly relies on its internal cache to produce the performance gains, the downside of this approach is the memory overhead produced by the cache. For example, for a model size of 25k model elements the cache size is around 14 MB.

Garcia and Möller [GM08] introduce an approach for the incremental evaluation of OCL constraints. Their approach is based on the DITTO algorithm which was originally designed for a subset of Java [SB07]. DITTO is an automatic incrementalization approach for invariant checks based on functional *memoization*. In contrast to view materialization, this approach caches the inputs and outputs of a function. Whenever a function is called with the same input parameters the resulting value can be drawn from the cache. Garcia and Möller also account for actual object graph over which an expression navigates which is considered as *implicit* input parameter of the constraint. However, it remains yet to be seen how this works towards incremental re-evaluation

strategies for OCL expressions after model changes.

# 4 How Change Events Affect Expression Values

Our goal is to find out for which context elements $o$ a given OCL expression evaluates to a different value due to a given change event. We relax our goal by assuming that the overall expression may change its value when one of its subexpressions changes its value. Later, we will add specific rules to detect cases where a particular change of a subexpression's value can be proven to have no impact on the overall expression's value.

For some (sub-)expression types we understand that their value is influenced by a model change, but not always does this lead us to the `self` object based on which to re-evaluate them. In particular, an `allInstances()` call, influenced only by deletions and creations of objects of the class on which `allInstances()` is called, are not evaluated on individual objects. Instead, if an `allInstances()` call's value changes, it changes for all elements of the overall expression's context type.

For all other (sub-)expression types, it is not the existence of an object that decides about an expression's value but the values of an object's properties through which an object is connected to other objects on behalf of references. Change event notifications that signal the modification of a property's value have an impact on the value of property call expressions. Based on this, we have to identify the navigation paths along properties that may have led from the overall expression's `self` object to the source of the property call expression.

# 5 Computing `self` from Change Notifications

Let us assume we received a change event notification for a property change with the single source object $s$ which is always an element of a class type because objects of non-class types are immutable values, so no change event notifications can be emitted for those.

For each property call expression $p$ in the parse tree of the overall expression $e$ under analysis we desire to find a computable function $self_{p,e}(s)$ that for the change event source object $s$ can compute the set of context objects $o$ on which the overall expression $e$ may evaluate to a result different from the result obtained before the change. In those cases where the overall expression $e$ is obvious, $self_p(s)$ is a shorthand for $self_{p,e}(s)$.

For example, `self` is the source expression of the property call expression `self.a`. If we receive a change notification for property `a` on object $s$, the function for the (sub-)expression `self` is defined as $self_{\texttt{self.a}}(s) := s$. It computes those context objects for which `self` evaluates to $s$ and hence for which the overall expression may have changed its value due to the change indicated by the notification.

For a property change event, we can determine all `PropertyCallExp` expressions $p$ in the tree that navigate that property. For all of those, we compute $self_p(s)$. The union of all these results over all applicable expressions $p$ is the set of context objects for which the expression may have changed its value.

We now declare a function

$$traceback_{n,e}(s : \text{Element},\ t : \textbf{Sequence}(\text{Property})) : \textbf{Set}(\text{Element})$$

for class-typed objects $s$ that we will use in our definition of $self_p(s)$.

**Definition 1** ($traceback$) $traceback_{n,e}(s,t)$ computes a set of context elements $o$ such that when the overall expression $e$ is evaluated for $o$ then the objects or values to which the subexpression $n$ of the overall expression $e$ evaluates contains

$$\begin{cases} \text{the single class-typed object } s & \text{if } \texttt{t->isEmpty()} \\ \text{a tuple with s in tuple part identified by } \texttt{t->first()} & \text{if } \texttt{t->notEmpty()} \end{cases}$$

Typically, the overall expression to which the subexpression $n$ belongs is obvious. Therefore, $traceback_n(s,t)$ is short for $traceback_{n,e}(s,t)$. Like $self_n(s)$, we define $traceback_{n,e}(s,t)$ only for single objects $s$.

With the $traceback$ function we can now define the $self_{p,e}(s)$ function with $p$ being a property call expression that has $n$ as its source expression, as follows:

$$self_{p,e}(s) := traceback_{n,e}(s, \textbf{Sequence}\{\})$$

In other words, $self_{p,e}(s)$ is the set of context elements $o$ such that when $e$ is evaluated for $o$, the property call expression's source expression $n$ may evaluate to the source element $s$ on which the change occurred. This, in turn, implies the possibility for a change of the $n$ subexpression in case $e$ is re-evaluated on $o$, and this was the definition of $self_{p,e}(s)$.

# 6 Implementing the $traceback_{n,e}(s,t)$ Function

We will now inductively implement the $traceback_{n,e}(s,t)$ function for all types of OCL expressions. The implementation in most cases is recursive for those expressions that have a source or argument expressions. We define $traceback_{n,e}(s,t)$ using OCL.

In the implementation of $traceback$ it is sometimes necessary to construct an expression based on the structure of the expression node $n$ for which the $traceback$ function is being implemented. For example, if we want to navigate from an object `x` across a property to the opposite end of an property call expression `n` we write

```
x.‘n.referredProperty.getOpposite()‘
```

meaning that the expression between backquotes, which results in a property to be navigated, is first evaluated and then placed into the expression in which it is used. You may compare this with the backquote expansion performed by a Unix command line shell. In fact, the OCL expression constructed for the implementation of the $traceback$ function will just hold the name of the element resulting from the expression in backquotes, replacing the backquoted part.

## 6.1 PropertyCallExp

There are two flavors of a property call depending on the type of the source object from which the property value is to be retrieved. If the source object's type is a class, the $traceback$ function has to determine all objects of the source expression's class whose property value for the `referredProperty` is—or in the case of a multi-valued property: *contains*—$s$. If the source

object's type is a tuple type, the attribute access fetches a tuple component whose value is defined by the `value` expression of a `TupleLiteralPart` owned by a `TupleLiteralExp`.

For a property call expression $n$ the *traceback* function is therefore defined as follows:

$$
\begin{array}{l}
traceback_n(s,t) := \\
\quad \textbf{let } t' \text{: } \textbf{Sequence}(\text{Property}) = \\
\qquad\qquad \textbf{if } n.\text{source.type.oclIsKindOf(TupleType) } \textbf{then} \\
\qquad\qquad\quad t\text{->prepend}(n.\text{referredAttribute}) \\
\qquad\qquad \textbf{else} \\
\qquad\qquad\quad t \\
\qquad\qquad \textbf{endif in} \\
\quad \textbf{let } \text{sourceObjects:}\textbf{Set}(\text{Element}) = \\
\qquad\qquad \textbf{if } n.\text{source.type.oclIsKindOf(TupleType) } \textbf{then} \\
\qquad\qquad\quad \textbf{Set}\{s\} \\
\qquad\qquad \textbf{else} \\
\qquad\qquad\quad \text{`}n.\text{source.type`.allInstances()->select(} \\
\qquad\qquad\qquad \text{`}n.\text{referredProperty` = } s) \\
\qquad\qquad \textbf{endif in} \\
\quad \text{sourceObjects->collect(so } \mid \textit{traceback}_{n.\texttt{source}}(\text{so},t'))
\end{array}
$$

Note, that `sourceObject` is always a single element because a class-typed element at any point in time can be an attribute value of at most one other element; for the tuple case, $s$ is passed through unchanged, so the invariant that $s$ represents a single element is maintained trivially.

The reverse navigation here is specified using a general, yet complexity-wise expensive and scope-wise not precisely defined `all- Instances()` expression. However, an implementation can apply three major improvements:

- For containment properties, including EMF attributes, typical repository implementations – including EMF – provide efficient navigation to an element's container.
- For non-containment, non-elementary properties, a repository may provide query capabilities that can evaluate the reverse navigation more efficiently than actually evaluating an `allInstances()` expression.
- An OCL environment may allow its users to define scope and visibility rules obeyed when reverse-navigating references.

Some modeling environments allow its users to declare derived properties, e.g., by specifying an OCL expression as the derivation rule. Such a derived property declaration is equivalent to a parameterless OCL-specified operation whose handling by *traceback* is explained in Section 6.10. We therefore do not treat OCL-specified derived properties here because it does not restrict the general applicability of the approach.

## 6.2 IfExp

An `if`-expression $n$ has three subexpressions: the condition, the `then`-branch and the `else`-branch. The result of the `IfExp` is defined either by the `then`-branch or the `else`-branch. Therefore, the *traceback* function only has to go down those branches. The condition expression

may contain its own property call expressions that trace back to `self` but since it does not evaluate to the `IfExp`'s result it plays no role in the definition of the *traceback* function for the `IfExp` itself. We therefore define:

$$traceback_n(s,t) := traceback_{n.\texttt{thenExpression}}(s,t)\text{->union(}traceback_{n.\texttt{elseExpression}}(s,t)\text{)}$$

### 6.3 TypeExp, PrimitiveLiteralExp, EnumLiteralExp, NullLiteralExp, InvalidLiteralExp

These classes of expressions don't have subexpressions, are themselves not a `self` expression and evaluate to a constant value $c$. Remember that they occur somewhere in the tree of the overall expression $e$ which is defined for the context element class `C`. By definition, $s$ can only be of a class type and therefore never assume the value obtained from evaluating any of the expression types discussed here. Consequently, for all three expression types, we can define:

$$traceback_n(s,t) := \textbf{Set}\{\}$$

### 6.4 CollectionLiteralExp

Recall that the $traceback_n(s,t)$ function is defined only for $s$ being a *single* element that has a class as its type. Therefore, for a context element $o$ to be part of the $traceback_n(s,t)$ result and with $n$ being a `CollectionLiteralExp`, $s$ can not be the whole collection but may only occur in any of the literal's parts when evaluated for a context $o$. In particular, we don't have to consider `CollectionRange` parts because their elements are of type `Integer` which is not a class.

The *traceback* function for a `CollectionLiteralExp` expression $n$ is hence defined as follows:

$$traceback_n(s,t) := n\text{.parts->select(p | p.oclIsKindOf(CollectionItem))->collect(p | }traceback_{\texttt{p.item}}(s,t)\text{)}$$

### 6.5 TupleLiteralExp

By definition, $s$ can only be of class type. Therefore, if for $traceback_n(s,t)$ the list of tuple parts $t$ to extract is empty, $n$ cannot evaluate to a tuple for any context, so the empty set results.

If $t$ contains a `Property` identifying a tuple literal part by its name, the *traceback* function needs to compute the context objects for which the identified tuple literal part's `value` expression can evaluated to $s$, with the first element of $t$ removed. Therefore:

$$traceback_n(s,t) := \textbf{if } t\text{=}\textbf{Sequence}\{\} \textbf{ then } \textbf{Set}\{\} \textbf{ else } traceback_{n.\texttt{part->select(attribute.name=}t\texttt{->first().name).value}}(s, t\text{->subSequence(2,}t\text{->size()),}V\text{)} \textbf{ endif}$$

## 6.6 IteratorExp

Expressions of this type subsume predefined iterator expressions such as `collect`, `select`, `reject`, `exists`, `forAll`, `isUnique`, `sortedBy`, `collectNested`, `one` or `any`. They all define a single implicit, a single explicit or multiple explicit iterator variables and operate on the evaluation result of a source expression. This behavior causes a special propagation through the *traceback* function for `VariableExp` expressions in case the variable referred by the `VariableExp` is an iterator variable of an `IteratorExp` expression (see Section 6.9.1).

Some iterators, such as `exists` and `isUnique` result in a value of type `Boolean`. By definition, those iterators can never produce $s$ as their result because $s$ is always class-typed. Other iterators, such as `select` or `reject` compute a sub-collection of the source collection. The `collect` and `collectNested` iterators are special in that they map each combination of iterator variables over the source collection to some other object computed by the body expression.

We define $traceback_n(s,t)$ for `IteratorExp` expressions $n$ as follows:

$traceback_n(s,t) :=$ **if** n.name = 'select' **or** n.name = 'reject'
**or** n.name = 'sortedBy' **or** n.name = 'any' **then**
$traceback_{n.\texttt{source}}(s,t)$
**else**
**if** n.name = 'collect' **or** n.name = 'collectNested' **then**
$traceback_{n.\texttt{body}}(s,t)$
**else**
-- iterator with Boolean result
**Set**{}
**endif**
**endif**

## 6.7 IterateExp

The result of an `iterate` expression is computed by its body expression. Therefore, the *traceback* definition for `iterate` applies itself to the body as follows:

$$traceback_n(s,t) := traceback_{n.\texttt{body}}(s,t)$$

The more difficult part about the `iterate` construct is evaluating the *traceback* function for the "result" variable that keeps the result of the previous iteration or an initialization value (see Section 6.9.2).

## 6.8 LetExp

The value of the `let`-expression is defined by its `in`-expression. Therefore,

$$traceback_n(s,t) := traceback_{n.\texttt{in}}(s,t)$$

## 6.9 VariableExp

There are five types of variables that a `VariableExp` expression may access, each of them discussed by one of the following subsections. We will work out the *traceback* definitions for

each of them separately. It should be sufficiently obvious how to combine them into a single definition of the *traceback* function.

### 6.9.1 Iterator Variables

Iterator variables are those defined by iterator expressions such as `select` or `collect`. The values that the iterator variable can assume are taken from the values of the collection over which the iterator expression loops. This collection is computed by the `n.source` expression. Therefore,

$$traceback_n(s,t) := traceback_{n\texttt{.referredVariable.loopExp.source}}(s,t)$$

### 6.9.2 Iterate Result Variable

The result variable of an `iterate` expression can be initialized by an expression. In this case, it may be referred in the `iterate`'s body expression. In any case, at the end of each iteration, the result of evaluating the body expression is assigned to the result variable.

For the result variable to assume the value $s$ it either has to have been initialized to $s$ by its initialization expression, or the `iterate` expression's body has at some point evaluated to $s$. We can therefore define:

$$\begin{array}{l} traceback_n(s,t) := \\ \quad traceback_{n\texttt{.referredVariable.initExpression}}(s,t)\texttt{->union(} \\ \quad traceback_{n\texttt{.referredVariable.baseExp.body}}(s,t)\texttt{)} \end{array}$$

### 6.9.3 Operation Parameters

To understand the possible values an operation parameter can assume, we have to look at the expressions calling the operation to which the formal parameter belongs. We can limit the set of operation calls considered to those reachable from the root expression There is no direct link between the expression used as actual parameter and the called operation's formal parameter. Instead, the binding happens through two indirections.

First, the variable declaration referred by the variable expression is linked to the operation's formal parameter only by name equality. Second, the formal parameter is linked to the actual parameter only by the position in the parameter list. With this, we can define the *traceback* function for variable expressions referring to variable declarations representing a formal parameter of an operation $op$ whose body expression contains $n$:

$$\begin{array}{l} traceback_n(s,t) := \\ \quad \textbf{let}\ \text{params:}\textbf{Sequence}(\text{Parameter}) = op\texttt{.parameters}\ \textbf{in} \\ \quad \textbf{let}\ \text{fp:Parameter} = \text{params}\texttt{->select(name} = n\texttt{.referredVariable.name)}\ \textbf{in} \\ \quad \textbf{let}\ \text{pos:}\textbf{Integer} = \text{params}\texttt{->indexOf(fp)}\ \textbf{in} \\ \quad \textbf{let}\ \text{actualParamExprs:}\textbf{Collection}(\textbf{OclExpression}) = op\texttt{.referringExp->collect(} \\ \qquad \texttt{arguments->at(pos))}\ \textbf{in} \\ \quad \text{actualParamExprs}\texttt{->collect(ape} \mid traceback_{\texttt{ape}}(s,t)\texttt{)->flatten()} \end{array}$$

This definition computes the formal operation parameter in the `let`-expression for `p`, its position in the list of formal parameters as `pos` and with that the actual parameters of all operation

call expressions calling *op* used for the formal parameter bound to `p`. For each of those actual parameter expressions, the *traceback* function is used to determine the context objects for which the value passed as the parameter under consideration could have been *s*. The resulting collection of collections of context elements is finally flattened.

### 6.9.4 self

The `self` expression can occur in two shapes: inside or outside of an operation body definition. If it occurs outside of operations, it evaluates to *s* if and only if it is evaluated with *s* as context object.

If it occurs inside an operation body, similar to the operation parameters discussed in 6.9.3, the `source` expressions of all operation calls to the respective operation that are reachable from the root expression analyzed need to be considered.

Let's assume, the operation containing the occurrence of the `self` expression is bound to *op* which is empty in case the occurrence of `self` is outside of an operation. Then we can define:

$$
\begin{array}{l}
traceback_n(s,t) := \\
\quad \textbf{if } op\text{->isEmpty()} \textbf{ then} \\
\quad\quad s \\
\quad \textbf{else} \\
\quad\quad op\text{.referringExp.source->collect(selfSource } \mid traceback_{\texttt{selfSource}}(s,t)\text{)->flatten()} \\
\quad \textbf{endif}
\end{array}
$$

### 6.9.5 Let Expression Variables

The `let` variable's value is defined by the `LetExp` expression's variable declaration and its respective `initExpression`. Therefore, for variable expressions accessing a `let`-expression's variable, we define

$$traceback_n(s,t) := traceback_{n.\texttt{referredVariable.initExpression}}(s,t)$$

## 6.10 OperationCallExp

The operation call's value is determined by its body. The parameter variable values and the value for the `self` variable will be traced back to the argument expressions[1] We define:

$$traceback_n(s,t) := traceback_{n.\texttt{referredOperation.body}}(s,t)$$

It may be noted that this leads to a recursive definition for recursive operations.

### 6.10.1 oclAsType()

If *n* is an `OperationCallExp` where the operation called is `oclAsType()`, *traceback* is defined as follows:

[1] The way our implementation handles this, all matching argument expressions of *all* calls to the operation will be considered. While this ignores our knowledge about the particular call hierarchy, it eases caching.

$traceback_n(s,t) :=$ **if** s.oclIsKindOf('n.arguments->at(1)') **then**
$traceback_{n.\texttt{source}}(s,t)$
**else**
**Set**{}
**endif**

If $s$ is not of the type requested by the `oclAsType()` cast operation, for no context element $o$ can the cast expression evaluate to $s$. In all other cases, the cast expression may evaluate to $s$ if the cast's `source` expression may evaluate to $s$.

### 6.10.2 Calls to Standard Library Operations

The standard library defines many operations on collections, such as `including`, `excluding` and `isEmpty`. The OCL specification [The10] defines them again in terms of OCL expressions. In most cases, the result expression for the operation's return value is provided as an OCL expression.

However, in some cases the result is not defined using the = operator and a direct OCL expression but rather by a set of OCL-specified predicates that span the operation result, the value of `self` as well as the argument values. Examples of such operations are `union` and `intersection`.

The `union` operation can be specified directly in OCL as

```
context Set(T)::union(s:Set(T)):Set(T)
body:
  s->iterate(i; acc:Set(T)=self | acc->including(i))
```

The example illustrates that just because the OCL specification chooses a different way to specify some of the standard library operations on collections, this does not exclude the possibility to provide a closed specification as an OCL expression for them. It is left as an exercise to the reader to prove this for the remaining standard library operations as well.

## 6.11 Putting it All Together

Because *traceback* is defined for each specialization of `OclExpression`, it is complete for the entire OCL language specification. The standard library operations map down to standard OCL constructs that are covered by *traceback*.

We need to assemble the entire *traceback* function from the fragments provided in the sections on each of the OCL constructs. The $traceback_{n,e}(s,t)$ function is defined as a case distinction over the type of the node $n$ as follows:

$traceback_{n,e}(s,t) :=$
**if** $n$.oclIsKindOf(IfExp) **then**
$traceback_{n.\texttt{thenExpression}}(s,t)$->union($traceback_{n.\texttt{elseExpression}}(s,t)$)
**else**
**if** $n$.oclIsKindOf(TypeExp) **or** $n$.oclIsKindOf(PrimitiveLiteralExp) **or**
$n$.oclIsKindOf(EnumLiteralExp) **or** $n$.oclIsKindOf(NullLiteralExp) **or**
$n$.oclIsKindOf(InvalidLiteralExp) **then**
**Set**{}

```
    else
      ...
    endif
  endif
```

## 6.12 Application to the Motivating Example

We can now apply the traceback function to the motivating example. We will show the computation of the context element for the following change event: Removing the "Administration" department from the sub-departments of the "Development" department alters the `Department.subDepartments` feature which is used by the `self.subDepartments` subexpression of the `Department::allSubDepartments()` operation's body expression (see Listing 1) which in turn is called by the FURCAS property initialization expression for the `projectTeam` property in the first alternative of template `Project`, as shown in Listing 3. Intuitively, this change would require the employees of "Administration" to be removed from the `projectTeam` property of the "CommonCosts" `Project`. Starting from this change event the *traceback* computation works as follows:

```
self_{self.subDepartments, self.lead.department.allSubDepartments().employees}(Development) =
-- applying the definition of self:
traceback_{self, self.lead.department.allSubDepartments().employees}(
                                        Development, Sequence{}) =
-- applying rule for VariableExp (see Section 6.9) for the self case
-- for occurrences inside operation bodies. op.referringExp.source leads
-- to the self.lead.department expression.
self.lead.department->collect(selfSource | traceback_{selfSource}(
                                    Development, Sequence{}))->flatten() =
traceback_{self.lead.department}(Development, Sequence{}))->flatten() =
-- applying the PropertyCallExp rule for the non-Tuple case,
-- see Section 6.1; n.source.type is Employee:
Employee.allInstances()->select(department=Development)
          ->collect(so | traceback_{self.lead}(so, Sequence{}))->flatten()
-- Evaluating the Employee.allInstances()->select combination and applying the
-- PropertyCallExp rule for the non-Tuple case, this time for self.lead:
Set{John, Jane, Judy}->collect(so |
  Project.allInstances()->select(lead=so)->collect(so2 |
    traceback_{self}(so2, Sequence{}))->flatten() =
-- Now, self does not occur inside an operation body. We expand
-- Project.allInstances() and traceback accordingly:
Set{John, Jane, Judy}->collect(so | Set{CommonCosts, NewProduct}
  ->select(lead=so)->collect(so2 | so2))->flatten() =
-- evaluate collect/select combination and flatten; note that NewProduct is
-- part of the result because Jane from the Development department leads it:
Bag{CommonCosts, NewProduct}
```

The result contains both, "CommonCosts" and "NewProduct" which is correct from a *traceback* point of view because for both projects the expression `self.lead.department.allSubDepartments().employees` has changed its value. For both projects it evaluated to

Set{ John, Jane, Judy, Anne, Amber, Arthur} before the change and Set { John, Jane, Judy} after the change. However, the "NewProduct" project did not have its projectTeam property initialized by the self.lead.department.allSubDepartments(). employees expression because its isDepartmentWide property is false. It can be removed from the context elements by the FURCAS runtime before re-evaluation starts.

## 6.13 Further Performance Improvement Attempts

We evaluated four more approaches particularly aiming at more precise results with fewer elements in the *traceback* results for which the evaluation result has not actually changed. Due to space limitations we outline them only briefly.

### 6.13.1 Partial Evaluation

Change notifications identify the element changed, as well as the old and new value for the feature that changed. This information can be used to evaluate the subexpression immediately affected by the change for the model state before and after the change. If the subexpression happens to be the source of another expression then in some cases it is possible to evaluate that other expression, again for the state before and after the model change. For example, if a change notification describes the change of an element's name attribute, and there is an expression self.name = 'abc', then this expression can be evaluated for the old and new model state. If the name attribute changed from a value not equal to 'abc' to another value not either equal to 'abc' then the overall expression remains false.

This principle can be recursively extended along the expression tree. Partial evaluation may fail in case the values for variables are not known. However, our benchmarks show that in many cases partial evaluation results in significant performance gains.

Partial evaluation can also be applied to IteratorExp expressions that use Boolean predicates to select or reject elements. Such an expression can only produce a certain element if it passed the respective predicate. If partial evaluation can prove that the element at hand cannot have passed the predicate, the current *traceback* path can be pruned at that point.

### 6.13.2 Delta Propagation

Partial evaluation can only continue as long as the subexpression whose old and new value are known is used as the source expression of another expression. However, if the old and new value of a collect iterator's body expression are known then due to the flattening that happens for collect we can infer how the change propagates to the collect expression's value. In particular, if elements are added to the body's value, the collect value at least contains these elements, too.

We found a class of expressions that propagate such changes monotonically, including the collect expression. For example, the select iterator is monotonic in its source expression. When elements are added to a select's source then the result can at most change by adding these elements.

We propagate changes through monotonic expressions, resulting in a superset of changes possible for the overall expression. If this superset can be proven to be the empty set then we know

that the changes can not imply any changes to this overall expression.

Our benchmarks have shown the additional efforts for delta propagation to pay off significantly based on the reduction in elements for which re-evaluation needs to be triggered.

### 6.13.3 Operation Call Selection

In Section 6.9 we have specified how to trace back `VariableExp` expressions referring to operation parameters and an operation's `self` variable. Recall how we determined *all* possible calls to the operation to find all possible substitute expressions for the parameter or source expression, respectively.

This procedure can be sharpened further, particularly for the case where an operation body is traced because a single `OperationCallExp` referring to the operation is being traced back. In this case we know exactly which call to use to substitute the parameter and `self` variables. Our benchmarks show that this improvement results only in minor gains only in some cases which came as a surprise (see Figures 4a and 4b).

### 6.13.4 Unused Checks

We found out that in some special cases, based on the information delivered by the change notification, it is possible to prove that a subexpression is not used during the overall expression's evaluation. For example, if the change immediately affected an expression within a `IfExp`'s `then` expression and sufficient information was delivered to evaluate the `IfExp`'s condition to `false`, we know that the `then` expression does not contribute to the `IfExp`'s value. Therefore, the `then` expression's change has no effect on the overall expression.

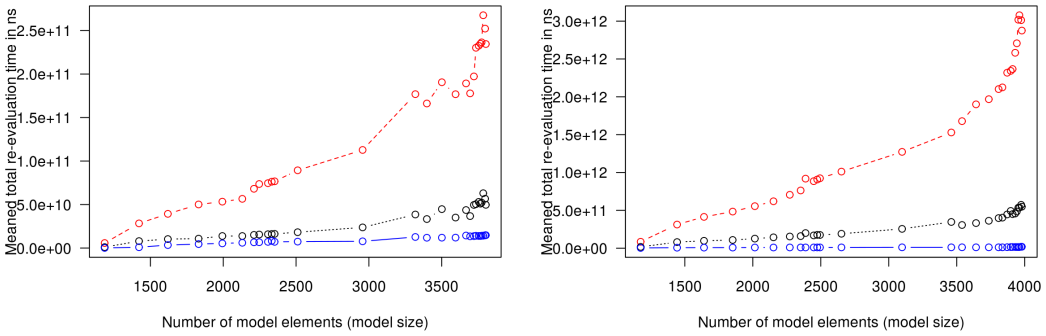
Based on this idea we defined and implemented an *unused* function which we can evaluate for any change and any subexpression. It also carries forward the values of variables inferred so far which increases the share of successful partial evaluations, such as for the `IfExp`'s condition expression.

Our benchmarks, however, have shown that the vast amount of partial evaluations necessary for trying to prove that a subexpression does not contribute to the overall expression's value on average does not pay off (see again Figures 4a and 4b). It may be possible that clever caching of partial evaluation results may change this, but we haven't investigated this any closer yet.

## 7 Validation

The most important questions we hoped to get answers for by our experiments were these:

- Would the impact analysis still provide benefits for changes that have far-reaching effects, or would it be better for those cases to just re-evaluate the expressions affected on all their context elements?
- How well would which variant of the impact analysis algorithm perform for which scenario?
- How great would the overall benefits be—if any—for the respective scenario?




(a) Central change.

(b) Large peripheral change.

Figure 3: Re-evaluation times for a central change affecting most parts of the model (Fig. 3a) and for many changes in the rims of the larger model (Fig. 3b). The upper red dashed line shows re-evaluation of all OCL expressions on all applicable contexts. The middle dotted black line shows re-evaluation of those OCL expressions whose filter condition matches the respective change notification, on all applicable contexts. The lower solid blue line shows re-evaluation of those OCL expressions whose filter condition matches the respective change notification, restricted to those context elements identified by the instance scope analysis.

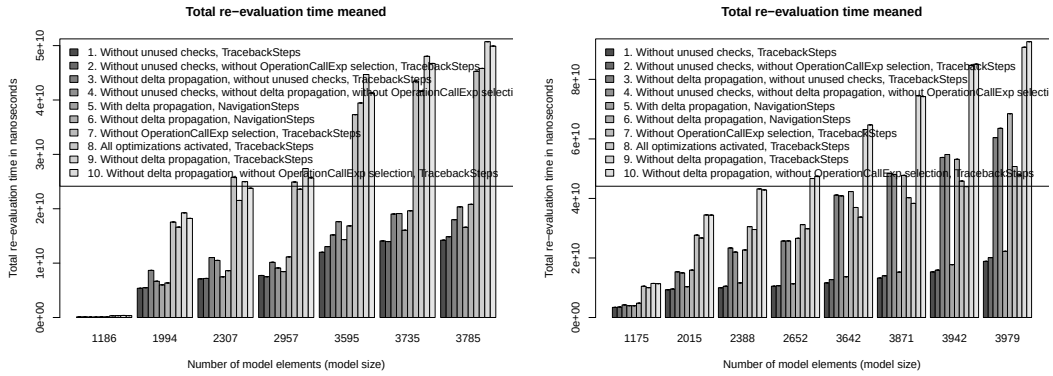
- How do the different algorithm variants such as *unused* checks, partial evaluation and delta propagation influence overall performance?

- Can the impact analysis make global OCL re-evaluation scale to large model sizes?

We performed two large benchmarks [Hol10] with EMF models of approximately 4,000 elements and 10,000 references, using 380 OCL expressions and two different change logs that we re-played. Using the technique described in [Hol10], we chop off parts of this model to obtain a series of realistic models of varying sizes. The first change log describes 150 changes in core parts of the model, causing major ripple effects during re-evaluation and therefore to be considered a "worst-case" scenario. The second change log has 1,700 changes mostly in the peripheral areas of the model where we don't expect major ripples.

We were able to compare various algorithm variants, as shown in Figure 4. Presented below are the results of the best-performing one. Figure 3a for the central changes shows the improvements using the impact analyzer as compared to only using event filtering applied by the class-scope analysis described in [CT05, CT09, AHK06], compared to the most naïve approach where all expressions on all their context objects are re-evaluated after each change. Re-evaluation with event filtering only takes approximately 2.4 times longer than when using the impact analyzer. The complete re-evaluation takes about twelve times longer.

Figure 3b shows the results for the large, peripheral change log. There, relative savings are, as expected, even greater and scalability is even better as fewer ripples occur during *traceback* evaluation. Using only event filtering instead of the full impact analyzer causes the re-evaluation




(a) Central change.

(b) Large peripheral change.

Figure 4: Re-evaluation times for a central change affecting most parts of the model (Fig. 4a) and for many changes in the rims of the larger model (Fig. 4b), comparing the effects of the different performance improvements discussed in Section 6.13.

to take approximately 20 times longer. Full re-evaluation takes about 120 times longer[2]. Note, that given the good scalability these ratios will increase even further for growing models.

More detailed analyses have shown that the use of the `allInstances()` operation affects the scalability properties. At least with a simple implementation of this operation, re-evaluation times for those expressions increase with growing model size which the impact analyzer cannot avoid. For expressions not using `allInstances()` we were able to show that there is no measurable increase in re-evaluation time for growing models when the changes are peripheral and therefore do not cause major ripple effects.

Figures 4a and 4b show the impact of the further improvements discussed in Section 6.13, again benchmarked for the two different change sets.

Regarding the algorithm's reliability, our various tests have not shown a single case where context elements for which an expression changed its value were missing from the analysis results. While lacking formal proof, this indicates that the conservative construction of the context element set is correct.

# 8 Conclusions and Outlook

We have presented an algorithm that for an OCL 2.2 expression and a model change event reliably and efficiently determines a set of context elements for which the expression may have changed its value due to the change signaled by the event. The algorithm has asymptotically optimal complexity and handles operation calls, including recursive operation calls, correctly.

Based on our implementation, we have conducted measurements with a non-trivial metamodel, many different OCL expressions of different complexities, and a number of sample

[2] We obtained these ratios from our measurements; they are hard to recognize "visually" in the figures.

models. We used the Eclipse EMF MDT/OCL 3.1.0 evaluator and based our impact analysis implementations on it. Partial evaluation, delta propagation and selective operation call traceback improved performance significantly. With this, even for worst-case scenarios with central changes causing ripples throughout the model, using the impact analyzer pays off on average, leading to a reduction in re-evaluation efforts by a factor of twelve compared to the most naïve approach, and still a factor of 2.4 compared to the event filtering-only approach. For average cases with moderate ripple effects these factors improve to 120 and 20, respectively.

Our benchmarks show that the impact analyzer makes incremental OCL re-evaluation scalable and hence practical for applications such as model validation, notification synthesis for OCL-specified derived properties as well as OCL-based attribute grammars.

Further research should be devoted to more intelligent variable value inference during computing the *traceback* function. Regarding the *unused* function, caching techniques should be considered for the many partial evaluations it requires. Last but not least, special rules for standard library operations with known semantics can be applied during *unused* evaluation.

We have made the impact analyzer described here available as part of the Eclipse MDT/OCL distribution, starting with Eclipse Indigo, in the *OCL Editors and Examples* feature in the *Modeling* category.

## Bibliography

[AHK06] M. Altenhofen, T. Hettel, S. Kusterer. OCL Support in an Industrial Environment. In *MoDELS Workshops*. Pp. 169–178. 2006.

[BHRV10] *Incremental Evaluation of Model Queries over EMF Models*. 2010. Accepted.

[CT05] J. Cabot, E. Teniente. Computing the Relevant Instances That May Violate an OCL Constraint. In *CAiSE*. Pp. 48–62. 2005.

[CT09] J. Cabot, E. Teniente. Incremental integrity checking of UML/OCL conceptual schemas. *J. Syst. Softw.* 82(9):1459–1478, 2009.

[CW02] T. Clark, J. Warmer (eds.). *Object Modeling with the OCL, The Rationale behind the Object Constraint Language*. Lecture Notes in Computer Science 2263. Springer, 2002.

[For82] C. L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence* 19(1):17 – 37, 1982.

[Fou10] E. Foundation. Eclipse XText Website. http://www.eclipse.org/Xtext/, 2010. Last retrieved 2010-07-06.

[GBU08] T. Goldschmidt, S. Becker, A. Uhl. Classification of Concrete Textual Syntax Mapping Approaches. In *Proceedings of the 4th European Conference on Model Driven Architecture - Foundations and Applications*. Lecture Notes in Computer Science 5059, pp. 169–184. Springer-Verlag Berlin Heidelberg, 2008.

[GBU09a] T. Goldschmidt, S. Becker, A. Uhl. FURCAS: Framework for UUID-Retaining Concrete to Abstract Syntax Mappings. In *Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications (ECMDA 2009) - Tools and Consultancy Track*. CTIT, 2009.
http://www.furcas.org

[GBU09b] T. Goldschmidt, S. Becker, A. Uhl. Textual Views in Model Driven Engineering. In *Proceedings of the 35th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, 2009.

[GM08] M. Garcia, R. Möller. Incremental evaluation of OCL invariants in the Essential MOF object model. In *Modellierung 2008*. GI-Edition Lecture Notes in Informatics, pp. 11–26. 2008.

[Gol10] T. Goldschmidt. *View-Based Textual Modelling*. PhD thesis, Karlsruhe Institute of Technology (KIT), 2010.

[HJK+09] F. Heidenreich, J. Johannes, S. Karol, M. Seifert, C. Wende. Derivation and Refinement of Textual Syntax for Models. In *Proc. of the 5th European Conference on Model-Driven Architecture Foundations and Applications (ECMDA-FA 2009)*. Pp. 114–129. Springer, 2009.

[Hol10] M. Holzleitner. Performance Analysis of an OCL Evaluation Infrastructure. 08 2010.

[RG99] M. Richters, M. Gogolla. On the Need for a Precise OCL Semantics. In France et al. (eds.), *Proc. OOPSLA Workshop "Rigorous Modeling and Analysis with the UML: Challenges and Limitations"*. Colorado State University, Fort Collins, Colorado, 1999.

[RTD83] T. Reps, T. Teitelbaum, A. Demers. Incremental Context-Dependent Analysis for Language-Based Editors. *ACM Trans. Program. Lang. Syst.* 5(3):449–477, 1983.

[SB07] A. Shankar, R. Bodík. DITTO: automatic incrementalization of data structure invariant checks (in Java). *SIGPLAN Not.* 42(6):310–319, 2007.

[SS08] M. Seifert, R. Samlaus. Static Source Code Analysis using OCL. In Cabot and Van Gorp (eds.), *OCL'08*. 2008.

[The05] The Object Management Group (OMG). MOF QVT Final Adopted Specification. Nov. 2005.
www.omg.org/docs/ptc/05-11-01.pdf

[The10] The Object Management Group (OMG). Object Constraint Language, Version 2.2. May 2010.
http://www.omg.org/spec/OCL/2.2/PDF

[TR81] T. Teitelbaum, T. Reps. The Cornell program synthesizer: a syntax-directed programming environment. *Commun. ACM* 24(9):563–573, 1981.