OCL support in an industrial environment

Michael Altenhofen[1] and Thomas Hettel[2] and Dr. Stefan Kusterer[3]

[1] SAP Research, CEC Karlsruhe, 76131 Karlsruhe, Germany,
michael.altenhofen@sap.com
[2] SAP Research, CEC Brisbane, Brisbane, Australia
thomas.hettel@sap.com
[3] SAP AG, 69190 Walldorf, Germany
stefan.kusterer@sap.com

Abstract. In this paper, we report on our experiences integrating OCL evaluation support in an industrial-strength (meta-)modeling infrastructure. We focus on the approach taken to improve efficiency through what we call *impact analysis* of model changes to decrease the number of necessary (re-)evaluations. We show how requirements derived from application scenarios have led to design decisions that depart from or resp. extend solutions found in (academic) literature.

1 Introduction

The MDA [1] vision describes a framework for designing software systems in a platform-neutral manner and builds on a number of standards developed by the OMG. Within this paper, we are mostly concerned with the Meta Object Facility (MOF) [2] used to define, manipulate and integrate meta-data and data in a uniform manner, and the Object Constraint Language (OCL) [3], originally designed as an extension to the Unified Modeling Language (UML) [4] to formally capture additional integrity constraints on UML models. In the meantime, the scope of OCL has been extended towards a model query language suitable for supporting model-to-model transformation tasks [5].

With its upcoming standard-compliant modeling infrastructure, SAP plans to support large-scale MDA scenarios with a multitude of meta-models that put additional requirements on the technical solution that are normally considered out-of-scope in academic environments. This may lead to solutions that may be considered inferior at first sight, but actually result from a broader set of (sometimes non-functional) requirements.

This paper focuses on one particular aspect in SAP's modeling infrastructure, namely a efficient support for OCL, both as a constraint language for meta-models and as query language for other tasks, such as model-to-model transformations or generic tool support. We will show how we have modified some of the existing approaches to better fit the requirements we're facing in our application scenarios.

The rest of the paper is organized as follows: In section 2, we will give an overview of SAP's modeling infrastructure focusing on features that are considered critical in large-scale industrial environments. Then, in section 3, we will summarize related work in the area of OCL impact analysis that has guided our work leading to a more detailed description of our approach in section 4. In section 5, we will report on first experimental experiences and conclude in section 6 by summarizing our work.

2 The SAP Modeling Infrastructure (MOIN)

Mid of 2005, SAP launched "Modeling Infrastructure" (MOIN), a development project within the NetWeaver[4] organization. The goal of the MOIN project is to implement the platform for SAP's next generation of modeling tools.

2.1 Modeling at SAP

In the past years, SAP's development teams made extensive use of modeling approaches, convinced that this shortens development cycles and improves quality of both, software designs and implementations. Especially in the area of business process platform development, modeling has proven as highly efficient.

Often without being aware of it, various teams advanced existing design-time tools into special purpose modeling solutions. In most cases, being very powerful with respect to the intended use case, these tools show deficiencies concerning interoperability. One becomes aware of this after looking at the complete suite of tools, which has developed over time.

A key to consolidation of the tool suite lies in establishing one common platform, which offers all the services required by these tools.

2.2 Overview on the Architecture and Services of MOIN

The requirements for MOIN resulted in an architecture, which consists of the components described in the following sections as major building blocks.

Repository First and foremost, MOIN is a repository infrastructure for meta-models and models. In particular it is capable of storing any MOF compliant meta-model together with all the associated models. For accessing this content, JMI compliant interfaces can be used, which are generated for the specific meta-model. This allows client applications to navigate and manipulate both, models and meta-models.

The MOF standard does not impose any concepts for physical structuring of model content onto the implementer. However, for features like locking or even the simplest read-operation, some notion of a meaningful group of model elements is required. For that, MOIN offers the concept of model-partitions, which allows users splitting up the graphs represented by model content into manageable buckets.

The MOIN repository component basically deals with storing and loading of model-partitions.

[4] SAP and SAP NetWeaver are trademarks or registered trademarks of SAP AG in Germany and in several other countries.

Query Mechanism JMI is well suited for exploring models, by accessing attributes, following links etc. However, for many use-cases more powerful means of data retrieval are needed. The MOIN query API therefore provides flexible methods for retrieving model elements, based on their types, attribute values, relationships to other model elements etc.

As part of the query API, we expose the abstract syntax of a query language, which is specific for MOIN, the so called MOIN query language (MOIN QL). This query language is tailored to the needs of the MOIN clients and supports MOIN-specific concepts like model-partitions, which are not part of OCL. However, it is planned to implement a mapping mechanism, which is capable of translating query parts of OCL statements into MOIN QL.

Eventing Framework Events can be used by MOIN clients to receive notifications for e.g. changes on models. This supports an architecture of loosely coupled components. More specifically, events are raised upon a. creation or deletion of model elements, b. attribute changes on a model element (i.e. value change, add or remove attribute), c. creation or deletion of links or d. creation, storing or deletion of model partitions, e. membership change of model elements wrt model-partitions etc.

Commands The MOIN command framework is the basis for undo/redo functionality, which will be part of most of the modeling tools. For that, modeling tools perform operations on the model content, which is bundled into a command object. Command objects can be put onto a stack, the so called undo/redo-stack. All commands implement methods for undo and redo. By calling undo for commands on the undo/redo-stack, any given number of user-actions following a save-operation can be reverted.

Model Transformation Infrastructure (MTI) The model transformation infrastructure (MTI) is planned as basis for model-to-model and model-to-text transformations, which can be defined by MOIN users for specific meta-models. MTI will provide a framework for defining and executing these transformations, where OCL is considered as option for describing query parts of transformation rules. However, since the design of MTI is not final yet, a sound assessment on the suitability of OCL for these use cases in the context of MOIN can not be made.

MOIN Core The MOIN core is the central component in the MOIN architecture, implementing and enforcing MOF semantics. It is independent from the deployment options and development infrastructure aspects and calls the other components for implementing all of MOIN's functionality.

By managing the object instances, representing model elements, the MOIN core can also be seen as in-memory cache for model content. However, it also manages the complete lifecycle of objects, triggers events, is involved in the execution of commands and uses the repository layer to read or write data.

OCL Components Apart from query and probably MTI, OCL is primarily used in meta-models to state constraints for the associated models, which cannot be expressed with means of MOF directly.

For the efficient evaluation of constraints, mainly four components are essential: a. the OCL parser, b. the OCL evaluator, c. the impact analysis and d. the MOIN core, which manages the checking of constraints by calling the impact analysis and the OCL evaluator.

The impact analysis is essential for the efficient implementation of constraint checking, as it avoids the unnecessary evaluation of constraints in specific situations. The impact analysis is described in section 4 in more detail.

3 Related Work

To our knowledge, there is not much related work in the area of optimization of OCL expression evaluation at the moment. In [6] the authors describe an algorithm to reduce the set of OCL constraints that have to be reevaluated if a model change did occur. Our solutions is based on that approach, but is more general in that it covers any sort of OCL expression, whereas [6] focuses on constraints only, which makes further optimizations possible: Assuming that all constraints are initially valid (i.e. works start with a consistent model) it is sufficient to identify only the events potentially causing constraint violation. However, there are application scenarios in MOIN where this assumption does not hold at all. Imagine a tool, which checks constraints while a user is editing, and marks elements violating constraints. Here it is important to know whether the violation was fixed so the highlighting can be removed. Sometimes, it may even be desirable, or at least tolerable to temporarily leave meta-models in an inconsistent state, like situations where the architect or designer is not yet able to provide all mandatory information. In other scenarios, like model transformations, we have to deal with any sort of OCL expressions, not only constraints, thus any modification causing the query to evaluate differently are relevant.

In a second paper [7], the same authors describe a method to reduce the number of context instances for which relevant OCL constraints have to be evaluated. It can be seen as a further optimization on top of the approach in [6]. The idea of decomposing expressions into sub-expression and building paths through the model was taken from there. However, there are two major differences: Firstly, the introduced method relies on an extension of the constrained model for computing instances. Helper associations are introduced to keep track of the applied navigations and to finally compute the affected instances, which have to be considered for reevaluation. This approach clearly violates one of our requirements that meta-models should stay intact avoiding any modifications not intended by the user. As we will see in the next section, we have taken a different approach to compute the relevant instance set which we believe is more flexible and allows further optimizations. Secondly, [7] leaves out some important details about how to handle indirect sub-expressions, esp. in the context of loop expressions (like `collect` and `iterate`), user-defined operations and attributes. Aiming at a full support of all language features, we have to provide solutions for those topics as well.

In [8], the authors go even one step further, and actively rewrite constraints for further optimizations. This may even lead to attaching a constraint to a new context. While this approach may definitely lead to a better performance than our approach, we did not consider optimizations in that direction, because this would introduce additional management overhead if we hid that transformation from the modeller and kept the two versions of constraints in sync.

In [9] a rule-based simplification of OCL constraints is introduced. These simplifications involve constant folding, removing of tautologies and unnecessary if-then-else constructs and more and are used to simplify automatically generated OCL constraints. We intentionally abandoned that approach in our work, because of usability issues: Users may get totally confused if, e.g. violations are reported on expressions they have never seen before, but merely result from silent rewrites of their expression by the underlying infrastructure.

4 OCL Impact Analysis in the SAP Modeling Infrastructure

This section presents the architecture and functionality of the OCL impact analysis and how it fits into SAP's modeling infrastructure.

4.1 Architecture

To support a wide range of different usage scenarios we decided to implement the *impact analyzer* (IA) as a general optimization add-on to applications[5], which have to deal with OCL in some way.

As indicated in Figure 1, interacting with the IA happens in two phases: Firstly, in the *analysis* phase (steps 1-3), a set of parsed OCL expressions is passed to the IA, whereupon a filter expression is returned. This filter can then be used to register with the eventing framework, so the application will only be notified about relevant model change events. Secondly, in the *filter* phase (steps 4-6), a received event can be forwarded to the IA to identify the OCL expressions affected by a change and the set of context instances per expression, for which the expression has to be reevaluated.

In fact, IA does not actually return a set of context instances, but OCL expressions evaluating to that set. This allows for quick responses and leaves further optimizations to the evaluator. Furthermore, in contrast to [7], this approach does not rely on an extension of the meta-model.

Typically, steps 1-3 are only executed once at the beginning, whereas steps 4-6 are executed often during the run-time of an application using IA.

[5] Such an application could be a constraint checker, a model transformation engine, etc.

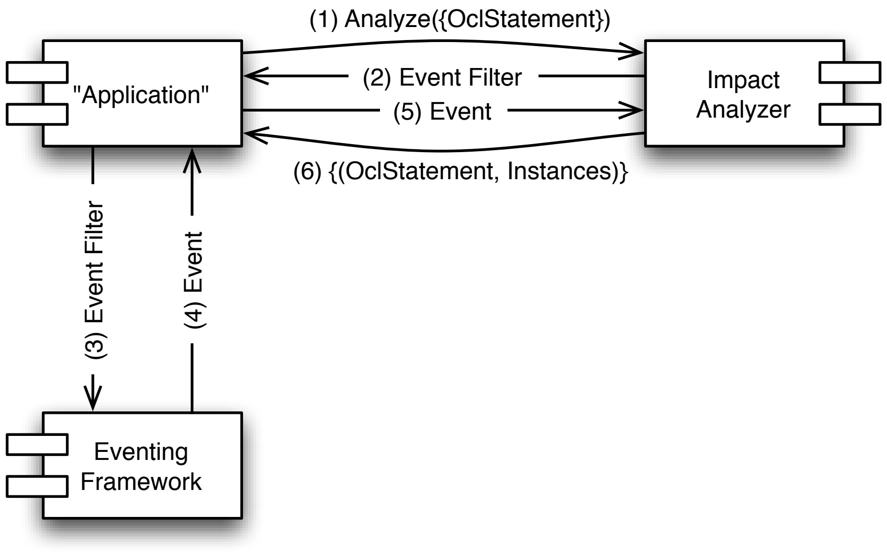

Fig. 1. Impact Analyzer Architecture.

During the analysis phase, the OCL expressions are analyzed and internal data structures are built up, which are then used in the filter phase for quick look-ups. These data structures are based on so-called *internal events* which represent classes of *model change events* provided by MOIN's eventing framework. The relationship between internal events and model change events is shown in Table 1.

The analysis phase itself is split up into a *class scope analysis* and a subsequent (optional) *instance scope analysis*. Both methods are described in the following sections.

4.2 Class Scope Analysis

The goal of the class scope analysis is to find the set of internal events (i.e., all types of model change events) which can cause the given expression to evaluate differently. At this point, it is assumed that all affected expressions have to be evaluated for all its context instances[6]. As outlined in Section 3, we use a generalized approach from [6] and walk the abstract syntax tree (AST) representing the given OCL expression in a depth-first manner, tagging each node with internal events that are relevant to it:

– Variable expressions referring to `self` → `CreateInstance(context)`, where context identifies the type of `self`

[6] Hence the name *class scope* analysis.

Internal Event	**Model Change Event**
CreateInstance(c:MofClass)	ElementAddedEvent(o:RefObject) where `o.refMetaObject = c`
DeleteInstance(c:MofClass)	ElementRemovedEvent(o:RefObject) where `o.refMetaObject = c`
AddLink(e:AssociationEnd)	LinkAddedEvent(f:RefFeatured, link:Sequence(RefObject)) where `e = f`
RemoveLink(e:AssociationEnd)	LinkRemovedEvent(f:RefFeatured, link:Sequence(RefObject)) where `e = f`
UpdateAttribute(a1:Attribute)	AttributeValueEvent(RefObject o, Attribute a2) where `a1 = a2`

Table 1. Mapping between internal events and model change events

- Operation calls `C.allInstances()` → `CreateInstance(C)`, `DeleteInstance(C)`
- Association end calls to `aE` → `AddLink(a)`, `RemoveLink(a)`, where `a` identifies the association to which the association end `aE` belongs
- Attribute call expressions to `a` → `UpdateAttribute(a)`

An OCL expression needs to be reevaluated if an event occurs which matches one of the internal events attached to a node in its AST. The IA's internal data structure therefore associates each internal event with a set OCL expressions affected by a model change event matching that internal event. Given a concrete model change event during the filter phase, IA determines the corresponding internal event and simply looks up the OCL expressions affected by that event.

For user-defined attributes and operations, the analyzer recurses into their bodies. The evaluation of a user-defined attribute or operation changes if its body is affected by a change to the model, thus affecting the evaluation of any expression referring to that user-defined operation or attribute.

```
inv: context Department
self.employee->select(e|e.age<23)->size()<self.maxJuniors
```

Listing 1.1. Example OCL expression used throughout this paper.

Example: Given the OCL expression in Listing 1.1, stating that per department only a certain number of junior employees are allowed. After having applied class scope analysis, the following internal events are relevant to the given expression: `CreateInstance(Department)`, `AddLink(employee)`, `RemoveLink(employee)`, `UpdateAttribute(age)`, `UpdateAttribute(maxJuniors)`.

4.3 Instance Scope Analysis

The goal of instance scope analysis is to reduce the number of context instances for which an expression needs to be reevaluated. Following the approach in [7], this is done by identifying navigation paths (i.e., sequences of attributes and association ends) in an expression starting at the context. If an object is changed, an OCL expression has to be reevaluated for those context instances from where the changed object can be reached by navigating along these paths. Given an element affected by a change, the set of context instances for which an expression needs to be reevaluated, can be found by following the reverse of the navigation paths. Once identified, these reverse paths are turned into OCL expressions and stored in the internal data structure. By evaluating these expressions, the set of context instances can be computed from a given changed element.

The following sections describe in more detail how sub-expressions and subsequently navigation paths can be identified and how they are reversed and translated into OCL.

Identifying Sub-Expressions The first step is to find sub-expressions. Sub-expressions start with a variable, or `allInstances()` and end in a node being the source of an operation with a primitive return type or in a node being a parameter of an operation or the body of a loop expression. Sub-expressions can also contain child sub-expressions in the body of a loop expression.

Two types of sub-expressions can be distinguished: *class* and *instance*.

Class sub-expressions start (directly or indirectly) with `allInstances()` and thus have to be evaluated for all instances of a class. There is no way to reduce the number of instances in this case.

Instance sub-expressions on the other hand start (directly or indirectly) with `self`. In this case, a subset of context instances can be identified for which the expression has to be reevaluated. The following steps only apply to instance sub-expression.

Example: Given the OCL expression in Listing 1.1, the following sub-expressions can be identified:

1. `self.employee->select()`
2. `e.age`
3. `self.maxJuniors`

Identifying Navigation Paths As per definition, sub-expressions consist only of navigation operations but do not necessarily start at the context. To get a sequence of navigation operations starting at the context, the navigation contained in a child sub-expression has to be concatenated with the navigation of the parent sub-expression. This approach only works for loop expressions calculating a subset of their source (e.g. select, reject).

```
inv: context Employee
self.employer->collect(d:Department|d.employee)->...
```

Listing 1.2. The body of `collect` has to be included in the parent's navigation path.

Example: Considering the last example, the corresponding navigation paths, relative to the context, are:

1. <employee>
2. <employee, age>
3. <maxJuniors>

As the second sub-expression does not start at the context, its navigation path has to be concatenated with the navigation path of its parent, i.e., the first sub-expression.

For loop expressions returning a completely different set compared to their source (e.g. collect, iterate), another approach has to be used. In this case the navigation contained in the body has to be included in the parent's navigation path because it contains vital information about how to get from the source type to the return type. Otherwise, there would be a gap in the navigation path.

Example: Considering the OCL expression in Listing 1.2, the following two navigation paths can be identified:

1. $<$ employer, employee, $\ldots >$ (for the parent sub-expression)
2. $<$ employer, employee $>$ (for the child sub-expression)

In this case, the `collect` operation takes a set of Departments and returns a set of Employees. Only by examining the body it can be said how to get from Department to Employee: by following the `employee` association end.

Reversing Navigation Paths Having applied class scope analysis before, each AST node has internal events, by which it is affected, attached to it. For each AST node and each internal event attached to it, the way back to the context has to be identified. This is done by reversing the navigation path which leads from the context to the currently considered AST node.

Example: Continuing the running example in Listing 1.1, we get the reverse navigation paths for each relevant internal event identified by class scope analysis as shown in Table 2.

If a new Department is created, the expression obviously has to be evaluated for that Department, therefore, the reverse navigation path is empty. If an employee is added to or removed from a department, the reverse navigation path is empty as well. More interesting is the case when the age of an employee is changed. In this case, navigating along the *employer* association end (opposite of employee) reveals the department, for which the expression has to be reevaluated.

Internal Event	**Navigation Path**
CreateInstance(Department)	<>
AddLink(employee), RemoveLink(employee)	<>
UpdateAttribute(age)	< employer >
UpdateAttribute(maxJuniors)	<>

Table 2. Internal events and corresponding navigation paths.

Translating into OCL The identified reverse navigation paths are translated into OCL and then stored in the internal data structure. Each internal event is associated with a number of expressions, which are affected by it. For each pair of internal event and affected expression a set of OCL reverse path expressions is associated with it. Evaluating that set of expressions for a given changed object results in the set of affected context instances.

For navigating along association ends, translation is straight forward: An association call expression is created referring to the reversed, i.e., opposite association end. Reversing object-valued attributes, however, is not that easy. Unfortunately, OCL does not offer a construct to find the owner of an attribute value. However, a legal OCL expression can be constructed which finds the attribute value's owner. The construct simply goes through all instances of a type `T` and checks whether it's attribute `attr` points to the given value `v`.

```
T.allInstances()->select(attr=v)
```

For performance reasons, an optimized evaluator could simply replace such a construct by a `v.immediateComposite()` call on the JMI object to determine the value's owner.

Example: Continuing the running example, in case of an `UpdateAttribute(age)` event the following OCL expression computes the Department for which the expression in Listing 1.1 needs to be reevaluated.

```
inv: context Employee
    self.employer
```

5 Preliminary results

To show the efficiency of our approach we present a theoretical best and worst case analysis, assessing the whole range of possible performance benefits. Furthermore, to see the impact on performance in practice we set up a test scenario using the MOF constraints defined in [2] with the UML meta-model as an instance of MOF.

5.1 Theoretical analysis

How much can be gained from using IA depends on four dimensions: The metamodel, the set of constraints, the set of instances and the set of events reported to an application. Note that these dimensions are not entirely orthogonal to each other.

Worst case analysis The first optimization used by IA is to reduce the number of events by removing irrelevant events. If we assume that only relevant events are reported, then there is no benefit from using IA. The next step is to reduce the number of expressions to be considered for evaluation to a subset of relevant expressions. If we further assume that all, or at least a large number of constraints use the same features, then each event is relevant to all expressions. So, there is no benefit either. The last optimization step it to reduce the number of context instances for evaluation. Assuming all expressions are class expressions (i.e., use `allInstances()`), this last optimization step does not yield performance benefits either. Hence, the worst case is that there are no performance benefits at all. Even worse, analyzing the set of expressions and computing the affected context instances adds an extra penalty to the application using IA.

Best case analysis In the very best case we assume that only irrelevant events occur. Without any optimizations lots of unnecessary work has to be done, whereas an application using IA is not even bothered – no unrelated events are reported to the application due to the filter mechanism provided by MOIN.

The second best case is that relevant events occur. If we assume that each feature in the model is only referred to by one statement, each event is only relevant to one statement. Given n statements, only $1/n$ of the statements have to be considered for evaluation. Furthermore, if we assume, that the model makes only use of one-to-one associations, for each model element affected by a change, only one context instance has to be considered for evaluation. In total, given n OCL statements and m instances per context, the work can be reduced to $\frac{1}{nm}$ of the work of an application without IA support. Depending on the number of statements and instances, this can be an enormous reduction.

5.2 UML-meta-model + MOF-constraints

To have a more realistic assessment of the performance benefits achieved by IA, we used the MOF-constraints[7] and the UML-meta-model, an instance of MOF, as a test scenario. Both, the MOF-constraints and the UML-meta-model are taken from real life and are non-trivial. As a reference, we shall compare all results to a *naive application*, i.e. an application which has no means of optimization and has to reevaluate all constraints for any change to the model.

There are two different applications using IA to different extents. Firstly, *class scope application*, which uses only the class scope analysis part. Secondly, *instance scope application*, which uses IA to its fullest extent. All three applications are exposed to the same model changes in the same order. Each model

[7] As not all of the constraints where proper OCL and our evaluator did not support type conversion at the time, we used only a subset of 38 constraints.

element in MOF is covered by exactly one event if there exists an instance of that model element in the UML-meta-model.

Reduction of expressions We consider the number of expressions which have to be evaluated after an event has been reported. In Figure 2 we compare the results from the *class scope application* to the *naive application*[8]. As the naive application has no means of optimization, it has to evaluate all (38) expressions for any event, whereas the class scope application does not have to evaluate expressions which cannot have changed due to the reported event.

For about 1/4 of the events, the number of relevant expressions could be reduced to one by applying class scope analysis. This is a reduction by 97%. For about 1/8 of the events, the number of relevant expressions could only be reduced to 12 and 11 respectively. Still, this is a reduction by 68% (71%). In average, the number of expressions to evaluate was reduced by 88%. The Median is 92%. As instance scope analysis does not reduce the number of expressions further, it is omitted.

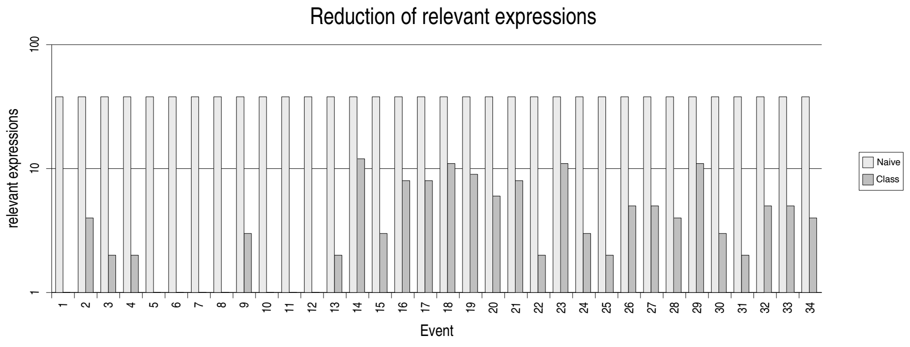

Fig. 2. Reduction of relevant expressions.

Reduction of context instances Here we consider the number of evaluator calls necessary to evaluate all affected expressions. An evaluator call is equal to the evaluation of one expression for one context instance. The numbers in Figure 3 also include evaluator calls necessary to compute the set of affected context instances. As class scope analysis reduces the number of expressions to evaluate, the number of evaluator calls is reduced as well. Therefore, the number of evaluator calls experiences about the same reduction as the number of expressions. After an already substantial reduction by class scope analysis, instance

[8] As class scope analysis does not reduce the number of expressions to be considered for reevaluation, it is not included in the chart. The results are equal to *class scope application*.

scope analysis achieves another enormous reduction: From several thousands to twenty or less for about 77% of the events (compare Figure 3). In total, the number of evaluator calls was reduced by three to four orders of magnitude, which is an enormous benefit in performance. In contrast to that, the naive application has to do some 26 000 evaluator calls per event to check the consistency of all constraints.

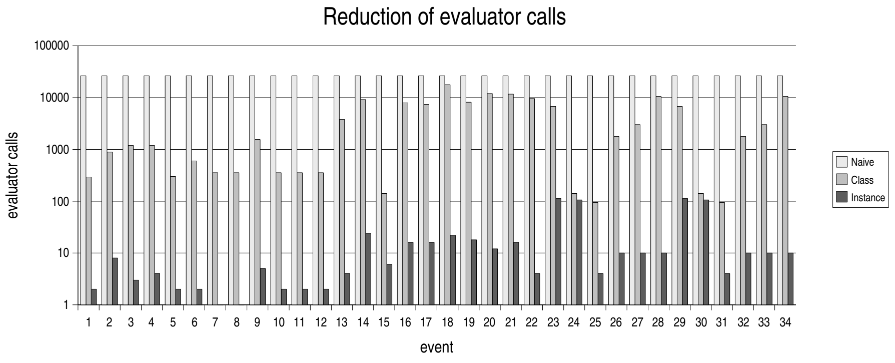

Fig. 3. Reduction of evaluator calls, including evaluator calls for computing the set of relevant context instances.

6 Conclusion

While efficient support for OCL is considered crucial in large-scale modeling environments, surprisingly little work has been published on optimizing OCL expression evaluation in case of arbitrary model changes. In this paper, we have reported on our experiences with integrating OCL into SAP's next generation modeling infrastructure MOIN.

Although some of the basic approaches from literature could be reused [6, 7], the actual implementation had to divert from these methods to cope with the (non-)functional requirements pertinent to MOIN. Most notably, we currently refused to implement any techniques that would result in silent or user-invisible changes to either the meta-models or the OCL expressions related to those meta-models. We know that this may lead to sub-optimal results in terms of performance, but preliminary experimental results show that the implemented techniques can still lead to a significant and hopefully sufficient performance gain. Further optimization techniques may be considered in the future, but they will have to be evaluated carefully on their trade-offs regarding other desired features.

Another path of optimization that we have not fully explored yet is the way how context instances are computed. The current approach is built on using

OCL as the expression language, but we plan to investigate using the internal MOIN Query Language for speeding up this computational step.

7 Acknowledgements

We would like to thank our colleagues Kristian Domagala, Harald Fuchs, Hans Hofmann, Simon Helsen, Diego Rapela, Murray Spork, and Axel Uhl for fruitful discussions during the design and the implementation of the OCL Impact Analyzer.

References

1. Object Management Group: MDA Guide. June 2003.
2. Object Management Group: Meta Object Facility (MOF) Specification. April 2002. http://www.omg.org/docs/formal/02-04-03.pdf.
3. Object Management Group: OCL 2.0 Specification (ptc/2005-06-06). June 2005.
4. Object Management Group: UML 2.0 Superstructure Specification (pct/03-08-02). August 2003.
5. Object Management Group: MOF 2.0 Query / Views / Transformations - Request for Proposal. http://www.omg.org/docs/ad/02-04-10.pdf, 2002.
6. Cabot, J., Teniente, E.: Determining the Structural Events that May Violate an Integrity Constraint. In: Proc. 7th Int. Conf. on the Unified Modeling Language (UML'04), LNCS, 3273 (2004) 173-187
7. Cabot, J., Teniente, E.: Computing the Relevant Instances that May Violate an OCL constraint. In: Proc. 17th Int. Conf. on Advanced Information Systems Engineering (CAiSE'05), LNCS, 3520 (2005) 48-62
8. Cabot, J., Teniente, E.: Incremental Evaluation of OCL Constraints. In: Proc. 17th Int. Conf. on Advanced Information Systems Engineering (CAiSE'06), June 2006.
9. Giese M., Hähnle R., Larsson, D.: Rule-based simplification of OCL constraints. In: Workshop on OCL and Model Driven Engineering at UML2004, pages 84-89, 2004.