OCL Support in an Industrial Environment

Michael Altenhofen[1], Thomas Hettel[2], and 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.

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 an efficient support for the OCL [3] constraint language. We will show how he 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), as development project within the NetWeaver[1] organization. The goal of the MOIN project is to implement the consolidated platform for SAP's next generation of modeling tools.

2.1 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 capable of storing any MOF compliant meta-model together with all the associated models. For accessing and manipulating this content, client applications can use JMI compliant interfaces, which are generated for the specific meta-model.

The MOF standard does not impose any concepts for physical structuring of model content onto the implementer, however, 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 loaded and stored by the MOIN repository.

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 (including a query language) therefore provides flexible methods for retrieving model elements, based on their types, attribute values, relationships to other model elements etc.

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. The event types supported by the framework will be discussed in section 4.

Model Transformation Infrastructure (MTI). The model transformation infrastructure (MTI) is planned as basis for model-to-model and model-to-text transformations. MTI will provide a framework for defining and executing these transformations, where OCL is considered as an option for describing query parts of transformation rules.

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

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

manages the complete life-cycle of objects, triggers events, and uses the repository layer to read or write data.

OCL Components. For dealing with OCL expressions, MOIN contains an OCL parser, OCL evaluator, and an OCL impact analysis component, managed by the MOIN core.

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 [5] the authors describe an algorithm to reduce the set of OCL expressions that have to be evaluated if a model change occurs. We follow that approach, but had to relax it since we have to deal with any sort of OCL expression whereas [5] only deals with OCL constraints where further optimization are possible, based on the assumption that initially all constraints are valid. However, there are application scenarios in MOIN where this assumption does not hold at all. E.g., it may 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 a second paper [6], the same authors describe a method to reduce the number of context instances for which relevant OCL constraints have to be evaluated as a further optimization on top of the approach in [5]. The idea of decomposing expressions into sub-expression and building paths through the model was taken from there. However, the approach taken in [6] violates one of our requirements that meta-models should stay intact avoiding modifications not intended by the user. Furthermore, we had to extend the algorithm to support all language features of OCL.

In [7], 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 [8] a rule-based simplification of OCL constraints is introduced, including, e.g., constant folding, and removing tautologies. We intentionally abandoned that approach in our work, again because of the additional overhead introduced.

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, which have to deal with OCL in some way.

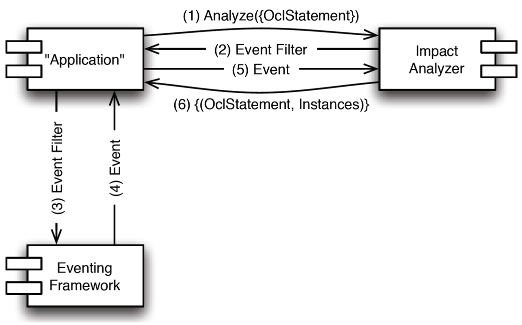

Fig. 1. Impact Analyzer Architecture

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 evaluated.

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 [6], this approach does not rely on an extension of the meta-model.

During the analysis phase, 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 affect a given expression, assumming that all affected expressions have to be evaluated for all its context instances[2].

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

Table 1. Mapping between InternalEvents and ModelChangeEvents

Internal Event	**Model Change Event**
CreateInstance(MofClass c)	ElementAddedEvent(RefObject o), c being the type of o
DeleteInstance(MofClass c)	ElementRemovedEvent(RefObject), c being the type of o
AddLink(AssociationEnd e)	LinkAddedEvent(Link l), e and l referring to the same association
RemoveLink(AssociationEnd e)	LinkRemovedEvent(Link l), e and l referring to the same association
UpdateAttribute(Attribute a)	AttributeValueEvent(RefObject o, Attribute b), a and b referring to the same attribute

As outlined in Section 3, we use a generalized approach from [5] and walk the abstract syntax tree (AST) representing the given OCL expression in a depth-first manner, tagging each node[3] with internal events that are relevant to it:

– Variable expressions referring to `self` → `CreateInstance(C)`, where `C` identifies the type of `self`
– `C.allInstances()` → `CreateInstance(C)`, `DeleteInstance(C)`
– Association end calls to `aE` → `AddLink(l)`, `RemoveLink(l)`, where `l` refers to the association to which the association end `aE` belongs
– Attribute call expressions to `a` → `UpdateAttribute(a)`

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.

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

Listing 1.1. OCL expression [5] for the running example

For the OCL expression in Listing 1.1 [4], the Class Scope Analysis returns the following internal events: `CreateInstance(Department)`, `AddLink(employee)`, `RemoveLink(employee)`, `UpdateAttribute(age)`, and `UpdateAttribute (maxJuniors)`.

[3] 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.

[4] Within a department only a certain number of junior employees are allowed.

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 evaluated. Following the approach in [6], this is done by identifying navigation paths[5]. Given an element affected by a change, the set of relevant context instances 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. *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 evaluated. 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: `self.employee->select()`, `e.age`, and `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[6].

Example: For the example in Listing 1.1 the context-relative navigation paths are: <`employee`>, <`employee`, `age`> [7], and <`maxJuniors`>.

For loop expressions with a different return type than their source (e.g. collect, iterate), the loop body contains vital information which has to be included; otherwise, the navigation path would contain a gap.

[5] I.e. the sequences of attributes and association ends, in an expression starting at the context. If an object is changed, an OCL expression has to be evaluated for those context instances from where the changed object can be reached by navigating along these paths.

[6] This approach only works for loop expressions calculating a subset of their source (e.g. select, reject).

[7] 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.

Example: Considering the OCL expression in Listing 1.2, the following two navigation paths can be identified: <employer, employee, . . . > (for the parent subexpression), and < employer, employee > (for the child sub-expression).

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

Listing 1.2. An OCL expression including a `collect` subexpression

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. For each tagged node in the AST, the way back to the context (variable) of the expression has to be identified. This is done by reversing the path from the variable subexpression to the 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.

Table 2. Internal events and corresponding navigation paths

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

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 evaluated.

Translating into OCL. Reverse navigation paths are translated into OCL and stored in the internal data structure which relates each internal event with a number of relevant expressions. For each such pair of internal event and expression a set of OCL expressions is maintained, which, when evaluated for a changed model element, 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 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 iterates through all instances of a type `T` and checks whether it's attribute `a` points to the given value `v`: `T.allInstances()->select(a=v)` [8].

Example: Continuing the running example from Listing 1.1, in case of an `UpdateAttribute(age)` event, the relevant Department instances are computed from the OCL expression `self.employer`.

5 Preliminary Results

To show the efficiency of our approach we present empirical results from a test scenario using the MOF constraints defined in [2] with the UML meta-model as an instance of MOF.

5.1 UML-meta-model + MOF-constraints

To have a more realistic assessment of the performance benefits achieved by IA, we used a subset of the MOF-constraints and the UML-meta-model, an instance of MOF, as a test scenario. We ran the tests with three types of applications: a *naive application (1)* that evaluates all constraints on any model change, a *class scope application (2)* that only uses the Class Scope Analysis part of the IA, and an *instance scope application (3)* that uses the IA to its fullest extend.

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 *(2)* to those from *(1)*[9]. The performance gains are due to the fact that the *CSA* 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%, with a Median of 92%.

Reduction of Context Instances. Here we consider the number of *evaluator calls* (the evaluation of one expression for one context instance) necessary to evaluate all affected expressions. The numbers in Figure 3 also include calls necessary to compute the set of affected context instances. As the number of expressions to evaluate is reduced in (2), 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 in

[8] 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.

[9] As instance scope analysis does not further reduce the number of expressions, it is not included in the chart.

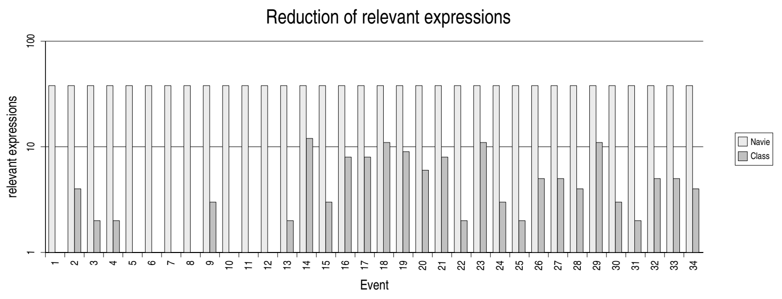

Fig. 2. Reduction of relevant expressions

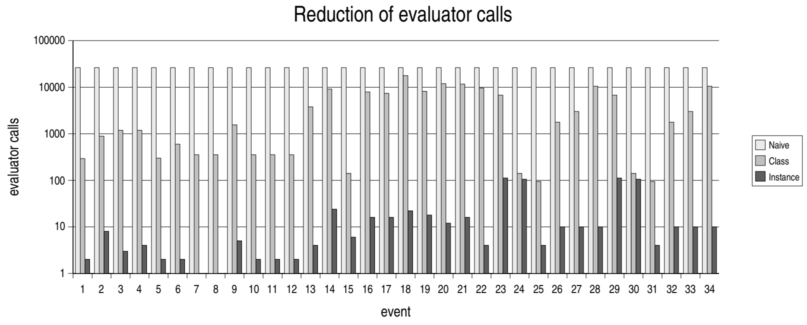

Fig. 3. Reduction of evaluator calls (including calls for computing context instances)

(2), (3) 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 compared to some 26000 calls per event in (1).

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 [5,6], 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 related OCL expressions. 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. We plan to investigate how the usage of the internal MOIN Query Language could speed up this computational step.

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. 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
6. 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
7. Cabot, J., Teniente, E.: Incremental Evaluation of OCL Constraints. In: Proc. 17th Int. Conf. on Advanced Information Systems Engineering (CAiSE'06), June 2006.
8. 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.