Integrating OCL and Textual Modelling Languages

Florian Heidenreich, Jendrik Johannes, Mirko Seifert,
Michael Thiele, Christian Wende, and Claas Wilke

Institut für Software- und Multimediatechnik
Technische Universität Dresden
D-01062, Dresden, Germany
{florian.heidenreich,jendrik.johannes,mirko.seifert,
michael.thiele,c.wende,class.wilke}@tu-dresden.de

Abstract. In the past years, many OCL tools achieved a transition of OCL from a language meant to constrain UML models to a universal constraint language applied to various modelling and metamodelling languages. However, OCL users still experience a discrepancy between the now highly extensible parsing and evaluation backend of OCL tools and the lack of appropriate frontend tooling like advanced OCL editors that adapt to the different application scenarios.
We argue that this has to be addressed both at a technical and methodological level. Therefore, this paper provides an overview of the technical foundations to provide an integrated OCL tooling frontend and backend for arbitrary textual modelling languages and contributes a stepwise process for such an integration. We distinguish two kinds of integration: *external* definition of OCL constraints and *embedded* definition of OCL constraints. Due to the textual notation of OCL the second kind provides particularly deep integration with textual modelling languages. We apply our approach in two case studies and discuss the benefits and limitations of the approach in general and both integration kinds in particular.

1 Introduction and Motivation

The Object Constraint Language (OCL) [1] has originally been developed to constrain models defined with the Unified Modeling Language (UML) [2]. Its standardised textual syntax and formal semantics promoted the implementation and adoption of OCL by different tool vendors which in turn supported practical adoption of OCL in combination with UML. Beyond its application to UML, OCL advanced to a constraint language applicable for various modelling languages. This includes support for modelling [3–8] and metamodelling languages [9, 10] like the Meta-Object Facility (MOF) [11] or Ecore [12]. The request for language-independent reuse of OCL led to extensible and adaptive approaches for OCL parsing and evaluation [7, 10, 13, 14].

Today, we experience a discrepancy between the technical facilities and their application in practice. While the Eclipse Modeling Framework (EMF) [12] is

equipped with an extensible OCL parser and evaluator [10], EMF (meta)models that use OCL constraints for well-formedness rules or integrate it as a constraint language are hard to find. We argue that one reason for this observation is a lack of adequate end-user tooling. While parsers and evaluation engines [10, 13] already provide means to apply OCL on various languages, OCL users still experience a lack of adequate tooling to write constraints in the first place [15]. Advanced OCL editors that provide name resolution, code navigation, auto indentation, code macros, code completion, or debugging are highly required. However, their implementation faces a special challenge, which results from the manifold applications of OCL. All the above mentioned editor functions are based on a structural and semantic evaluation of OCL expressions that are strongly influenced by the language OCL is combined with or applied to.

Our goal is to provide such advanced editing functionality for OCL in combination with arbitrary textual (meta)modelling languages. Achieving this goal does not just require an appropriate technical infrastructure, but in particular a systematic process to apply such infrastructure to corporately customise the OCL backend and frontend for different application scenarios. In this paper we provide an overview of the technical foundations to integrate OCL with arbitrary textual modelling languages and contribute a stepwise process for such integration. There, we distinguish two kinds of integration: *external* definition of OCL constraints and *embedded* definition of OCL constraints. We evaluate the benefits and limitations of both types of integration on exemplary case studies.

This paper is structured as follows: In Sect. 2 we motivate OCL integration using a simple example language, which will later be used to explain the general integration process in Sect. 3. Here, we systematically present the steps needed to derive a customised tooling frontend and backend to efficiently apply OCL in combination with arbitrary textual modelling languages. To show the genericity of the approach, its application to Ecore is presented in Sect. 4. In Sect. 5, we report on lessons learnt during application and elaborate current limitations of our integration technique. Related work is investigated in Sect. 6 and Sect. 7 concludes our contributions.

2 Running Example—The Forms Language

To demonstrate the integration of OCL into a textual modelling language, we integrate OCL with the *Forms Language*[1]—a language to model simple forms in a textual manner. Such form models can be transformed into PDF or HTML forms or interpreted by an Eclipse-based wizard providing the form's fields. The Forms language was designed using EMFText [16].

Figure 1 shows a pizza order form specified in the Forms language. The `Form` consists of two `Group`s containing multiple `Item`s to enter the order information. As shown, `Item`s can have different types like `FreeText`, `Number` or `Choice`. Although the model is fully specified, no integrity checks can be performed.

[1] http://www.emftext.org/language/forms/

pizzaOrder.forms

```
FORM "Pizza Order"
  GROUP "Your Personal Pizza"
    ITEM "dough" "Select your dough" :
      CHOICE ( "american", "italian", "cheesy crust" )
    ITEM "topping" "Select your topping" :
      CHOICE multiple ( "ham", "salami", "ananas", "onions",
        "mushrooms", "tuna" )
    ITEM "extraCheese" "Do you want extra cheese?" :
      DECISION ( "yes", "no" )
    ITEM "amount" "How many pizzas do you want?" : NUMBER
  GROUP "Your Order"
    ITEM "name" "Enter your name" : FREETEXT
    ITEM "dateOfBirth" "Enter your date of birth" : DATE
    ITEM "telephoneNo" "Enter your telephone number" : FREETEXT
```

Fig. 1. Example Pizza Order Form Specification.

To overcome this limitation, we like to extend the Forms language with support for specifying constraints on instances of form models (i.e., completed forms). To formulate such constraints we want to use OCL. In principle using OCL in this context can be achieved by embedding OCL directly into form definitions or by employing external OCL constraints that refer to the form of interest.

Before we actually start the discussion of the integration process, we first want to sketch the desired result of this process. Figure 2 shows an excerpt of an OCL file defining integrity constraints ensuring that undesired compositions of pizza toppings are not allowed. Furthermore, the entered telephone number of a pizza order form is checked for correctness. This is one possible style of integration—the external use of OCL.

Figure 3 shows the same constraints embedded into the pizza order form. This second integration example corresponds to the second style of integration—the embedding of OCL. As can be seen, `package`s and `context`s of OCL constraints have not to be declared as they can be derived implicitly from the given specification. Besides the advantage of having shorter specifications, OCL is now

pizzaOrder.ocl

```
package PizzaOrder

context YourPersonalPizza
inv noTunaWithSalami: self.topping->includes(ChoiceTopping::tuna)
    implies self.topping->excludes(ChoiceTopping::salami)

context YourOrder
inv correctTelephoneNo: let digits = self.telephoneNo.characters() in
    digits->first() = '+' and
    digits->subSequence(2, digits->size())->forAll(not toInteger().oclIsInvalid())

endpackage
```

Fig. 2. Example Pizza Order with External Constraints.

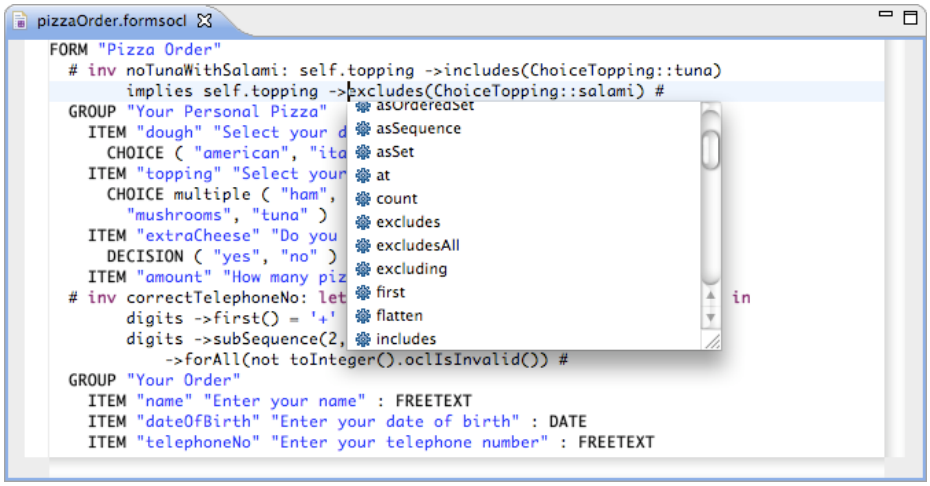

Fig. 3. Example Pizza Order with Embedded Constraints.

specified in the same document as the constrained model. This improves the readability and comprehensibility of constraints since the developer does not need to switch between multiple views to understand constraints and their contexts. Furthermore, if model elements are modified, the constraints' context is modified implicitly and no invalid states of constraints referring to non-existing model elements can occur.

3 Integrating OCL with Textual Modelling Languages

In the previous section we have shown that either the external or embedded definition of OCL constraints can be used to enrich the Forms language. In the following we present an integration process applicable for both approaches and textual modelling languages in general.

3.1 OCL Integration Process

We have developed an integration process to use OCL with different modelling languages. The process is built around small specifications out of which all necessary artefacts are created. The process itself is tool independent and thus is not bound to the tools used in our case studies. The presented process consists of the five steps depicted in Fig. 4. We used Ecore for metamodel definition and *EMFText* [16] for textual syntax specification. *DresdenOCL* [17] was used for OCL parsing and evaluation. Although the use of other tools should be possible we did not evaluate further tools. Static semantics integration was realised using an attribute grammar based on the *Scala* [18] library *Kiama* [19].

During *Metamodel Integration* (1), the metamodels of OCL and the textual modelling language are combined. The resulting metamodel (`FormsOCL.ecore`) is

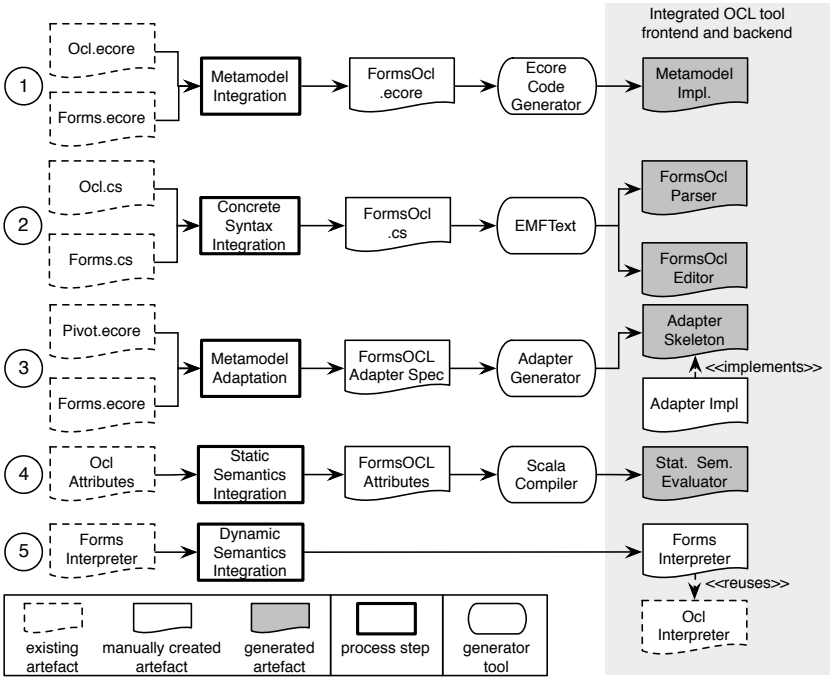

Fig. 4. The Generative OCL Integration Process (on the example of Forms language).

used by the EMF code generator to generate a Java metamodel implementation. Next, during *Concrete Syntax Integration* (2) the textual syntax of both languages is integrated and used for the generation of a textual parser/editor using EMFText. Since only an embedded OCL integration requires a new parser/editor, these first two steps are only required for embedded OCL definitions. The step *Metamodel Adaptation* (3) is required for both approaches. The creation of a Pivot Model representation of the DSL's model enables DresdenOCL to parse OCL constraints that refer to DSL model elements. *Static Semantics Integration* (4) results in a combined static semantic analysis for integrated languages of step (2). The additional attributes are only necessary with the embedded approach as the static semantic analysis has to be extended to refer to DSL model elements in that case. For external OCL definitions the metamodel adaptation from step (3) is sufficient to allow semantic analysis of constraints defined on DSL model elements. The last step (5), the *Dynamic Semantics Integration* is necessary to evaluate integrated OCL constraints. Since evaluation is required for all OCL integrations, both approaches require this step.

3.2 Integration Steps

After presenting the integration process as a whole, we now dive into detailed descriptions of the individual steps using the Forms language integration as exam-

ple for explanation where necessary. We will particularly highlight the problems that are accompanied with each step and how we solved them.

(1) Metamodel Integration (for embedded integration only) We used Ecore metamodels to describe the abstract syntax of languages. To create an integrated language, one has to create a new Ecore metamodel that imports both the metamodel of OCL and of the language to integrate with. As Ecore is an implementation of Essential MOF [11], which in turn promotes a plain object-oriented metamodelling language, the options for metamodel integration are delegation and inheritance. Thus, one can either subclass one or more metaclasses or add references to metaclasses of the involved modelling languages. By creating subclasses the integrated metamodel will allow to reference new types (i.e., to store new kinds of objects in existing references). This can be used to allow elements of the OCL metamodel (e.g., invariants or expressions) in places where the embedding language did not do so before. Alternatively, one can "frame" the embedding language by introducing a new root metaclass that points to this language as well as to OCL metaclasses.

To extend the Forms language, a new metaclass called `GroupWithOcl` that extends `Group` and has a reference to multiple `DefinitionOrInvariant`s is created (cf. Fig. 5(a)). By subclassing `Group`, `Form`s can reference either `Group`s—as before—or reference groups with OCL expressions.

Conceptually, metamodel extension by inheritance and delegation was sufficient to embed OCL in the Forms language. This is due to the fact, that we reused large portions of OCL (invariants and expressions) as a whole. In other cases of language integration, dedicated metamodel extension facilities may be more appropriate (cf. Sect. 5).

(2) Syntactic Integration (for embedded integration only) After the abstract syntax integration, the textual syntax of the integrated language needs to be specified. In EMFText, textual syntax is defined by specifying one EBNF-like grammar rule for each metaclass (cf. [16] for details). The integrated syntax can import the existing rule sets of the textual modelling language and OCL to reuse them. For the new metclasses, new rules have to be specified. In the case of the Forms language, a new rule for metaclass `GroupWithOcl` was required, which is shown in Fig. 5(b). EMFText puts the existing and new rules into relation by considering the inheritance and reference relations between the corresponding metaclasses that were established in Step (1).

In general the integration of the textual OCL syntax and other textual modelling languages is not as easy as observed for the Forms language. Context-free grammars are not closed under composition, which is why adaptations of either the embedding language or of OCL itself can be required. Such adaptations can be performed by overriding imported syntax rules.

Moreover, even if the syntax definition of OCL and the embedding language are theoretically compatible w.r.t. composition, we experienced problems with EMFText and its underlying parser technology. Parsers generated by EMFText use a scanner to split the input document into tokens, which then control the

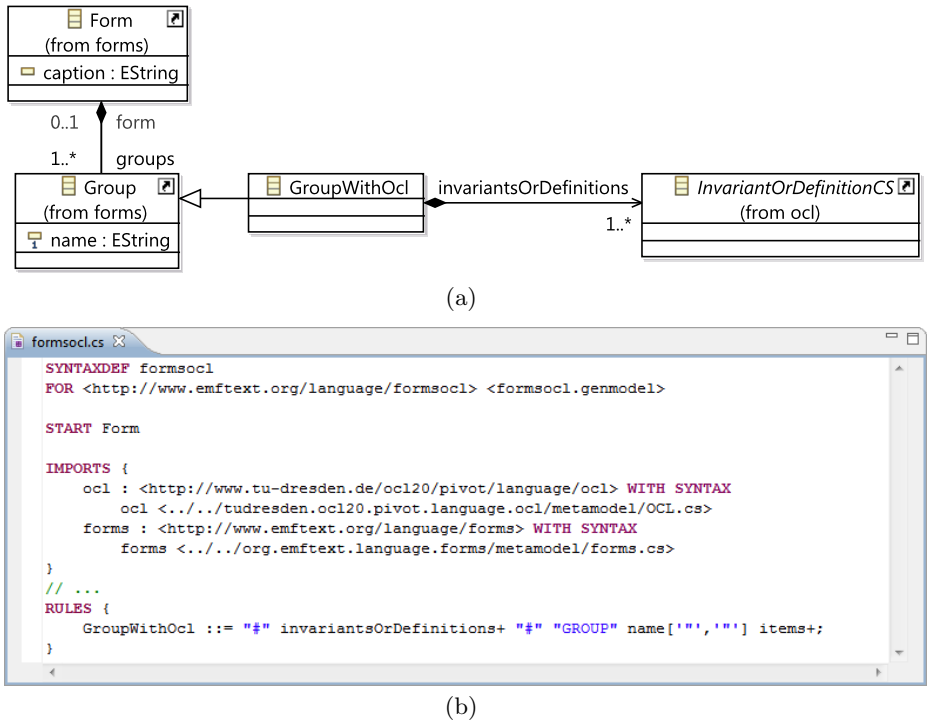

Fig. 5. (a) Metamodel and (b) Syntax Integration.

derivation of a syntax tree. If tokens of OCL conflict with tokens of the embedding language, no parser can be generated. For the Forms language, prioritising tokens was sufficient to resolve conflicts, but for more complex host languages, the situation can be more difficult.

(3) Metamodel Adaptation (for both integration styles) OCL can not be parsed, typed and evaluated in the context of a language without reasoning on the elements of the language's metamodel. As mentioned above, we use DresdenOCL for parsing and evaluation. DresdenOCL was designed to be independent of a concrete target metamodel. That is, it can be connected to arbitrary metamodels as long as they contain concepts that can be mapped to the basic concepts of object-oriented languages like types, namespaces, properties and (optionally) operations. DresdenOCL works on standardised interfaces (a *Pivot Model* [8]) which define these concepts and all operations necessary for DresdenOCL to reason on them (e.g., to get all operations defined on a type). For each modelling language that shall be connected to DresdenOCL, an adapter has to be created that maps the concepts of the language's metamodel to the pivot model concepts. To allow OCL evaluation on forms we provided a pivot model adapter for the forms metamodel. It adapts `Groups` as `Types` since they contain typed `Items` that

can be adapted to `Property`s of their `Group`s. This allows the definition of OCL constraints on `Group`s using their `Item`'s values for integrity checks. The `Type`s of `Item`s are adapted to `Type`s as well. For instance, the `FreeText` type is adapted to `String`, `Choice`s are adapted to `Enumeration`s. The enclosing `Form` is adapted to a `Namespace`. Since a language's adaptation to the pivot model contains parts that are similar for all adapters, DresdenOCL provides an adapter generator that allows adapter skeleton code generation for Ecore-based metamodels [20, Chapter 8].

(4) Semantic Analysis Adaptation (for embedded integration only) The adaptation of semantics analysis is required for the integration of OCL with modelling languages both by the backend and frontend of an OCL tool. Features like name resolution, type inference and checking enable advanced editor functions like code completion and sophisticated error reporting and are also required for static and dynamic OCL evaluation. In the current EMFText-based DresdenOCL parser, an attribute grammar [21] is used to implement the static semantics of OCL. The attribute grammar rules are specified with Scala using the library Kiama [19].

The semantic analysis is integrated in the tooling frontend by helper classes generated from EMFText that call the attribute grammar. During reference resolving, references between OCL statements are resolved. E.g., the `self` variable must be bound to the type of its context to allow code completion for property and operation calls. Since OCL itself is not modified by the integration and type bindings to the integrated language are realised by the pivot model, large parts of the attribute grammar can be reused for every OCL integration. Nevertheless, many of the reused resolving operations require context information given by the concrete language concepts OCL constraints are embedded in. Context computation has to be modified for every embedded OCL integration. Therefore, a new attribute grammar specifying the context computation is mixed into the OCL attribute grammar exploiting the *mixin composition* capabilities of the Scala language [18, Chapter 27].

(5) Dynamic Semantics Integration (for both integration styles) After integrating OCL into a DSL it should be possible to evaluate OCL constraints on DSL-based models. Thus, an OCL evaluation tool has to be integrated as the last step of the process. In general, two different approaches exist to evaluate OCL constraints: the *interpretative* and the *generative approach* [22]. The first one interprets OCL constraints whereas the second one generates check code that has to be integrated into the modelling language's interpreter implementation (e.g., a Java program). DresdenOCL allows both approaches. Using the template-based OCL-to-Java code generator [23], an adaptation to the implementation language can be achieved by modifying these templates. They describe a transformation of OCL expressions into the implementation language and the instrumentation of the constraint evaluation into the implementation code.

In addition, DresdenOCL provides an OCL interpreter [13], which can be used with various runtime objects (e.g., Java objects, XML nodes or even SWT-based widgets). In the case of the interpreter of the Forms language, runtime objects

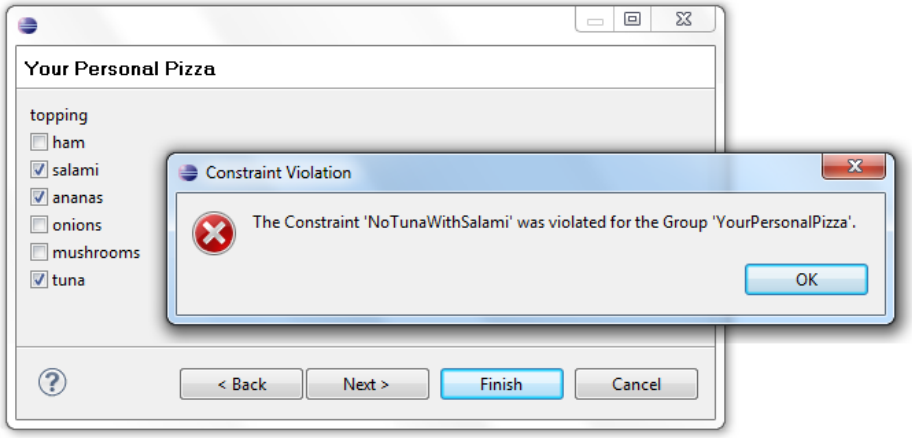

Fig. 6. Form Interpretation with Constraint Violation.

are SWT widgets, which are embedded into a wizard dialog. The evaluation of OCL constraints is triggered when the user hits a *next* or a *finish* button. Figure 6 shows a screenshot from a wizard page that belongs to the pizza order example. As can be seen, the pizza topping constraint has been violated and, thus, an error message is displayed.

4 eOCL - Integrating OCL into Textual Ecore

In addition to the integration of OCL into the Forms language, we implemented an integration of OCL into a textual variant of the Ecore metamodelling language that was developed with EMFText.[2] The aim of this integration is to lower the barrier of using OCL in metamodelling. Although OCL is well suited to define constraints for metamodels [9, 10], there seems to be still little usage of OCL in metamodelling in general. For instance, in the AtlanMod metamodel Zoo [24]—a collection of around 300 metamodels—no OCL constraints are delivered with any of the metamodels. We believe that this is to a high degree a tooling issue, since many people that create metamodels are also familiar with OCL. We address this issue with our integration of Ecore and OCL, named eOCL, that allows to describe both a metamodel and OCL well-formedness rules defined on this metamodel using an integrated textual syntax as shown in Fig. 8. The respective metamodel integration is shown in Fig. 7 and similar to the extension performed for the Forms language.

This complex case study showed that the above introduced integration steps were sufficient for the integration of OCL into a more complex textual modelling language than the Forms language. The major difference between the OCL integration into the Forms language and textual Ecore is related to dynamic se-

[2] http://www.emftext.org/language/textecore/

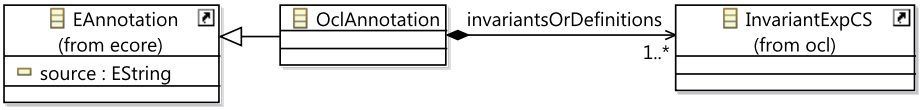

Fig. 7. Metamodel Integration of Ecore and OCL.

```
forms.eocl
package forms forms "http://www.emftext.org/languages/forms" {

    # inv: self.items->isUnique(text) #
    class Group {
        attribute EString name (1..1);
        containment reference Item items (0..*);
    }

    # inv: not self.text.oclIsUndefined() #
    class Item {
        attribute EString text (1..1);
        reference ItemType itemType (1..1);
    }

    abstract class ItemType {}

    # inv: self.isMultiple implies self.options->size() >= 2 #
    class Choice extends ItemType {
        containment reference Option options (1..*);
        attribute EBoolean isMultiple (1..1);
    }
    ...
```

Fig. 8. Forms Language Specification in eOCL.

mantics integration. For the Forms language, we integrated the DresdenOCL interpreter to evaluate the OCL constraints. For Ecore, a generative integration is more applicable since Ecore models are typically used for Java code generation. The desired Ecore/OCL integration generates check code for all OCL constraints and instruments the Java code generated from Ecore for runtime evaluation of this check code.

Besides the variation of the OCL evaluation technique, the same integration steps had to be performed for the Forms language and textual Ecore and similar problems were experienced.

5 Discussion

In this section we conclude the limitations of the introduced process w.r.t. its application to the different case studies, motivate potential solutions for future work.

Metamodel Integration Our integration approach applied inheritance and delegation for the composition of the languages to integrate. This approach was sufficient for OCL integration as we reused the OCL (invariants and expressions)

as a whole and worked with compatible metamodels. However, module composition with inheritance is discussed controversially for object orientation [25, 26] and language engineering [27]. It breaks the principle of information hiding between modules, since inherited properties can be accessed and altered in arbitrary ways. Furthermore, structural conflicts of combined metamodels (e.g., equally named attributes in classes to integrate) can not be handled appropriately. In future work we therefore plan to combine our suggestions for modular language engineering [27] with the approach presented in this paper.

Syntax Integration The problems experienced w.r.t. the syntactic integration of OCL in both case studies are originated from the fact that context-free grammars are not closed under composition. Syntactic ambiguities and token overlaps occur for languages that use token definitions in their concrete syntax that are used in OCL as well (e.g., numeric literals, string literals, or operators like `+`, `-`, `*`). In our case studies manual adaptations of the integrated syntax w.r.t. token prioritisation and reuse were sufficient to handle such conflicts. However, a more general solution for this problem could be the application of different grammar formalisms that are less restricted w.r.t. syntactic overlaps (e.g., lexer states [28], delegating compiler objects [29], scannerless parsers [30], context-aware scanning [31], Parsing Expression Grammars (PEGs) [32, 33]).

Metamodel Adaptation The implementation of a pivot model adapter to integrate arbitrary languages with OCL is the standard mechanism to couple the backend of DresdenOCL to arbitrary languages. Equivalent mechanisms can be found for other OCL tools [7, 10]. Future work has to investigate how we can extend the presented approach to enable completely specification-driven adapter generation. As pivot model adaptation can be considered a concrete technique for metamodel integration we also plan to examine the applicability of our modular language engineering approach [27] in this context.

Semantics Analysis In Sect. 3.2 we illustrated how the OCL attribute grammar can be reused for multiple OCL integrations using mixin composition. Unfortunately, the current design requires some boilerplate code that is required to integrate the mixin configuration into the EMFText-generated parser. Actually, five classes and two Eclipse extension points are necessary for each language integration. For the Forms/OCL integration these classes contain about 80 lines of code. We plan to improve the language integration process by generating most of this infrastructural code.

Dynamic Semantics Integration To reuse the same OCL interpreter for various languages it is necessary to adapt model instance objects. In [13] we presented an approach to address this issue. Furthermore, the invocation of OCL interpretation has to be included manually into the tooling for the language OCL is integrated with (e.g., the invocation of OCL interpretation when the *finish* button in an SWT form is pressed). Currently, we do not see how the proposed process could be improved in this regard.

Generative approaches for semantics integration share the same limitations. The generation of OCL check code can be reused for different integrations if their code generation relies on the same target language. However, code instrumentation or the adaptation of code generation to a new target language still requires manual effort.

6 Related Work

Integrating OCL with different languages has been investigated in various scenarios before. For example, in [3] a report on the integration of OCL with Triple Graph Grammars (TGGs) [34] can be found. The integration of OCL and the RAISE Specification Language (RSL) has been investigated in [4]. OCL has been integrated with Fujaba [5], business rules [6] and a profile for the railway domain [35]. While [3] and [5] embed OCL in graphical languages, the host language was textual in [4] and [35]. In [6] OCL was not embedded into another language, but rather integration was performed by transforming OCL to SQL. In addition, the integration of arbitrary textual languages into graphical languages has been presented in [36]. The diversity of the approaches to integrate OCL with other languages shows the necessity for general guidelines on how to achieve such integration.

As a first step to ease the application of OCL to arbitrary languages, the adaptation of the query and navigation facilities of OCL has been evaluated in [37]. This adaptation is part of the overall process to integrate OCL with other languages as described in this paper. However, we restricted ourselves to the integration of textual modelling languages and OCL. The semantical aspects of this integration, which have been the main subject of the works mentioned above can not be answered here as these highly depend on the host language. Nonetheless, we tried to provide some best practises to achieve practical language integration. For the integration of visual languages with OCL, one may consult [38, 39], where a graphical variant of OCL—Visual OCL—is proposed.

Integrating OCL with Ecore, which served as a case study in this paper, has recently been performed by the Eclipse MDT OCL project.[3] Here, the OCLinEcore Editor was released—a very similar approach to integrate OCL and Ecore more tightly. However, while the result of this integration is close to ours, no general procedure to accomplish such a coupling is available. In contrast, the goal of this paper is to outline the steps that are necessary to perform such an integration for arbitrary textual languages.

Other constraint languages, besides OCL have also been subject to integration with modelling languages. For example, the Epsilon Validation Language (EVL) [40], which is part of the Epsilon tool suite, is based on OCL and extends the language with guarded constraints (i.e., constraints which are evaluated only for certain model elements), constraint dependencies and constraint composition (i.e., to compose complex constraints from sequences of simpler constraints).

[3] http://www.eclipse.org/modeling/mdt/?project=ocl

From the perspective of integration, EVL is loosely coupled with its target languages. While this enables to reuse constraints across multiple metamodels—given these metamodels share concepts with equal names—it implies that no static checks are applied to the constraints. For example, the binding of constraints to concrete metaclasses or features they navigate on is not achieved at development time as we do in both integration styles.

7 Conclusion

In this paper we presented a tool-supported process to integrate OCL with arbitrary textual modelling languages. Our five step process supports two integration kinds: *external* definition of OCL constraints that point into a textual model and *embedded* definition of OCL constraints that are directly defined inside a textual model. Only two of the five steps are required for the first integration kind, while performing all steps yields support for both kinds. We showed the applicability of the full process on two examples: an integration of OCL into a textual modelling language for forms and an integration of OCL into a textual variant of Ecore.

The embedded integration is specific to textual modelling languages and takes advantage of the fact that both the modelling language and OCL have a textual notation. It provides integrated end-user tooling that is directly generated from the specifications defined during the process using EMF, EMFText and DresdenOCL. Such a generative approach to develop integrated tooling is required to increase the willingness of tool vendors to integrate OCL into new textual modelling languages as well as the acceptance of OCL by end-users through the deeper integration of OCL tooling. In the future we plan to extend the generative component of the tool support for our process—in particular by providing adapter generators for the semantic analysis adaptation.

Acknowledgement

This research has been co-funded by the European Commission within the 6th Framework Programme project MODELPLEX #034081, the 7th Framework Programme project MOST #216691 and by the German Ministry of Education and Research within the projects feasiPLe and CoolSoftware.

References

1. Object Management Group Object Constraint Language. Version 2.2 (February 2010)
2. Object Management Group Unified Modeling Language: Superstructure Version 2.2. Final Adopted Specification formal/2009-02-02 (February 2009)
3. Dang, D.H., Gogolla, M.: On Integrating OCL and Triple Graph Grammars. In Chaudron, M.R.V., ed.: Models in Software Engineering, Workshops and Symposia at MODELS 2008. Reports and Revised Selected Papers. Volume 5421 of Lecture Notes in Computer Science., Springer (2008) 124–137

4. Debnath, N., Funes, A., Dasso, A., Montejano, G., Riesco, D., Uzal, R.: Integrating OCL Expressions into RSL Specifications. In: IEEE Int'l Conf. on Electro/Information Technology (EIT-2007), IEEE Catalog Number: 07EX1665. (May 2007) 182–186
5. Stölzel, M., Zschaler, S., Geiger, L.: Integrating OCL and Model Transformations in Fujaba. ECEASST **5** (2006)
6. Demuth, B., Hußmann, H., Loecher, S.: OCL as a Specification Language for Business Rules in Database Applications. In Gogolla, M., Kobryn, C., eds.: 4th Int'l Conf. on Unified Modeling Language, Modeling Languages, Concepts, and Tools (UML 2001). Volume 2185 of Lecture Notes in Computer Science., Springer (2001) 104–117
7. Akehurst, D., Patrascoiu, O.: OCL 2.0 - Implementing the Standard for Multiple Metamodels. Electron. Notes Theor. Comput. Sci. **102** (2004) 21–41
8. Bräuer, M., Demuth, B.: Model-Level Integration of the OCL Standard Library Using a Pivot Model with Generics Support. In Akehurst, D.H., Gogolla, M., Zschaler, S., eds.: Ocl4All: Modelling Systems with OCL Workshop at MoDELS 2007, Berlin, Germany, Technische Universität Berlin (October 2007)
9. Loecher, S., Ocke, S.: A Metamodel-based OCL-compiler for UML and MOF. Electron. Notes Theor. Comput. Sci. **102** (2004) 43–61
10. Eclipse Model Development Tools. http://www.eclipse.org/modeling/mdt/
11. Object Management Group Meta-Object Facility (MOF) Core Specification. Version 2.0 (January 2006)
12. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: Eclipse Modeling Framework, 2nd Edition. Pearson Education (2008)
13. Wilke, C., Thiele, M., Wende, C.: Extending Variability for OCL Interpretation. In: 13th Int'l Conf. on Model Driven Engineering Languages and Systems (MoDELS 2010). (October 2010) To be published in Springer LNCS series.
14. Kolovos, D., Paige, R., Polack, F.: Detecting and Repairing Inconsistencies across Heterogeneous Models. In: 2008 Int'l Conf. on Software Testing, Verification, and Validation, IEEE Computer Society (2008) 356–364
15. Chimiak-Opoka, J., Demuth, B., Silingas, D., Rouquette, N.: Requirements Analysis for an Integrated OCL Development Environment. OCL 2009 Workshop - The Pragmatics Of OCL And Other Textual Specification Languages (2009)
16. Heidenreich, F., Johannes, J., Karol, S., Seifert, M., Wende, C.: Derivation and Refinement of Textual Syntax for Models. In Paige, R.F., Hartman, A., Rensink, A., eds.: Model Driven Architecture - Foundations and Applications. Volume 5562 of Lecture Notes in Computer Science., Berlin/Heidelberg, Springer (Juni 2009) 114–129
17. TU Dresden: Software Technology Group DresdenOCL. http://dresden-ocl.sourceforge.net/ (2010)
18. Odersky, M., Spoon, L., Venners, B.: Programming in Scala. 1st edn. Artima Press, Mountain View, CA, USA (2008)
19. Kiama - A Scala library for language processing. http://code.google.com/p/kiama/ (2010)
20. Wilke, C., Thiele, M.: DresdenOCL - Manual for Installation, Use and Development. Technische Universität Dresden, Software Technology Group, Dresden, Germany. (2010)
21. Knuth, D.: Semantics of context-free languages. Theory of Computing Systems **2**(2) (1968) 127–145

22. Demuth, B., Wilke, C.: Model and Object Verification by Using Dresden OCL. In: Proceedings of the Russian-German Workshop Innovation Information Technologies: Theory and Practice, Ufa, Russia, July 25-31, 2009, Ufa, Bashkortostan, Russia, Ufa State Aviation Technical University (July 2009)
23. Wilke, C.: Java Code Generation for Dresden OCL2 for Eclipse. Großer Beleg (minor thesis), Technische Universität Dresden, Dresden, Germany (February 2009)
24. AtlanMod Metamodel Zoo. http://www.emn.fr/z-info/atlanmod/index.php/Zoos
25. Bracha, G., Lindstrom, G.: Modularity Meets Inheritance. In: Int'l Conf. on Computer Languages, IEEE Computer Society (1992) 282–290
26. Taivalsaari, A.: On the Notion of Inheritance. ACM Computing Surveys **28**(3) (1996) 438–479
27. Wende, C., Thieme, N., Zschaler, S.: A Role-based Approach Towards Modular Language Engineering. In van den Brand, M., Gasevic, D., Gray, J., eds.: 2nd Int'l Conf. on Software Language Engineering (SLE 2009), Revised Selected Papers. Volume 5969 of LNCS., Springer (March 2010) 254–273
28. Clark, C.: Newlines and Lexer States. SIGPLAN Notices **35**(4) (2000) 18–24
29. Bosch, J.: Delegating Compiler Objects: Modularity and Reusability in Language Engineering. Nordic J. of Computing **4**(1) (1997) 66–92
30. Heering, J., Hendriks, P.R.H., Klint, P., Rekers, J.: The Syntax Definition Formalism SDF – Reference Manual. SIGPLAN Notices **24**(11) (1989) 43–75
31. Wyk, E.R.V., Schwerdfeger, A.C.: Context-Aware Scanning For Parsing Extensible Languages. In: 6th Int'l Conf. on Generative Programming and Component Engineering (GPCE'07), ACM (2007) 63–72
32. Ford, B.: Parsing Expression Grammars: A Recognition-Based Syntactic Foundation. In: 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'04), ACM (2004) 111–122
33. Grimm, R.: Better Extensibility through Modular Syntax. In: ACM SIGPLAN 2006 Conf. on Programming Language Design and Implementation (PLDI'06), ACM (2006) 38–51
34. Schürr, A.: Specification of Graph Translators with Triple Graph Grammars. In Mayr, E.W., Schmidt, G., Tinhofer, G., eds.: Proceedings of 20th Int'l Workshop on Graph-Theoretic Concepts in Computer Science, Herrsching, Germany. Volume 903 of Lecture Notes in Computer Science., Springer Verlag (1994)
35. Berkenkötter, K.: OCL-based Validation of a Railway Domain Profile. In Kühne, T., ed.: Models in Software Engineering, Workshops and Symposia at MoDELS 2006. Reports and Revised Selected Papers. Volume 4364 of Lecture Notes in Computer Science., Springer (2007) 159–168
36. Scheidgen, M.: Textual Modelling Embedded into Graphical Modelling. In Schieferdecker, I., Hartman, A., eds.: 4th European Conf. on Model Driven Architecture - Foundations and Applications (ECMDA-FA 2008). Volume 5095 of Lecture Notes in Computer Science., Springer (June 2008) 153–168
37. Kolovos, D.S., Paige, R.F., Polack, F.: Aligning OCL with Domain-Specific Languages to Support Instance-Level Model Queries. ECEASST **5** (2006)
38. Kent, S.: Constraint Diagrams: Visualizing Assertions in Object-Oriented Models. In: OOPSLA. (1997) 327–341
39. Bottoni, P., Koch, M., Parisi-Presicce, F., Taentzer, G.: A Visualization of OCL Using Collaborations. In Gogolla, M., Kobryn, C., eds.: 4th Int'l Conf. on Unified Modeling Language, Modeling Languages, Concepts, and Tools (UML 2001). Volume 2185 of Lecture Notes in Computer Science., Springer (2001) 257–271
40. Kolovos, D.S., Rose, L., Page, R.F.: The Epsilon Book. Available online at http://www.eclipse.org/gmt/epsilon/doc/book/