Model-Level Integration of the OCL Standard Library Using a Pivot Model with Generics Support

Matthias Bräuer and Birgit Demuth

Dresden University of Technology
Software Technology Group
01062 Dresden, Germany
{matthias.braeuer, birgit.demuth}@inf.tu-dresden.de

Abstract. OCL 2.0 specifies a standard library of predefined types and associated operations. A model-level representation of the library is required to reference its elements within the abstract syntax model created by an OCL parser. Existing OCL engines build this model in the implementation code which severely limits reusability, flexibility and maintainability. To address these problems, we show how a common pivot model with explicit support for template types can help to externalize the definition of the standard library and integrate it with instances of arbitrary domain-specific modeling languages. We exemplify the feasibility of our approach with a prototypical implementation for the Dresden OCL2 Toolkit and present a tailored EMF editor for modeling the OCL types and operations. We limit our discussion to the model level, i.e., we do not consider an implementation of the standard library for an execution engine.

1 Introduction

The Object Constraint Language (OCL) [1] specifies a standard library of types and associated operations. This includes primitive types such as `Integer` or `String`, collection types like `Set` or `Bag` as well as a number of special types which are important for the OCL type system (`OclAny`, `OclVoid`, and `OclType`). Among the predefined operations on these types are arithmetic, boolean and set-theoretic operations. All types in the standard library are instances of abstract syntax classes. These are located one level above the type definitions in the four-layered meta hierarchy of the OMG MOF architecture [2]. Since OCL allows querying of both metamodels and models [3, p. 97], the standard types exist either on the M2 or the M1 layer.

Now, when building the abstract syntax model from a textual OCL expression, an OCL parser needs to have access to the elements of the standard library. This is necessary, for instance, to properly locate operations defined for the implicit supertype `OclAny` or to create user-defined collection and tuple types. Existing OCL engines, such as the current release of the Dresden OCL2 Toolkit [4] or the Kent OCL Library [5], usually build an internal representation of the standard library programmatically, e.g, using the API of a model repository.

Hiding the structure of the standard library inside the implementation code of the engine triggers a number of problems. Firstly, the *reusability* of the library definition is severely impaired, because it is tied to a particular implementation language and platform. Thus, porting the OCL engine to another programming language requires an entire rewrite of the code that creates the library. Further, the model of the standard library cannot conveniently be validated, altered, extended, or modularized. This is disadvantageous if the underlying execution platform (i.e., an interpreter or code generator) does not support some of the standard library types and operations. In this case, adapting the library definition on the model level by removing the corresponding elements would be helpful. In essence, the *flexibility* of the "in-code" approach is relatively low. Lastly, the implementation of the OCL engine tends to become fairly complex leading to decreased *maintainability*. For instance, the Dresden OCL2 Toolkit in its current release contains a helper class with more than 400 lines of code alone to manage the model of the standard library.

As an answer to these problems, we propose the novel approach of creating the OCL standard library as an instance of a so-called *pivot model*, which can be viewed as a "universal language covering a certain domain" [6]. For example, Milanovic et al. [7] employ the REWERSE Rule Markup Language (R2ML) as a "pivotal metamodel" to map between OWL/SWRL and UML/OCL. In this paper, we define a pivot model as an intermediate metamodel used for aligning the metamodels of arbitrary domain-specific modeling languages (DSL) with that of OCL. By directly supporting *generics* in this metamodel, modeling all of the template types and operations in the OCL standard library becomes possible. We have implemented this approach using the Eclipse Modeling Framework (EMF) [8] which allowed us to build a highly functional editor for the pivot model and employ EMF's default XMI serialization capabilities. Providing the predefined OCL types within an OCL engine therefore reduces to a simple model file import. Concrete collection types can be created from the corresponding templates by binding their type parameters with the required element type.

The remainder of this paper is structured as follows: In Sect. 2, we briefly review the challenges for a model-level integration of the OCL standard library in the light of two existing OCL engines. We continue by describing the design of a suitable pivotal metamodel addressing these issues in Sect. 3. In Sect. 4, we present a practical evaluation of our approach. We highlight the visual editor used for modeling the standard library and describe an illustrative example. A brief account of related work is provided in Sect. 5. Finally, Sect. 6 concludes on our work and shows up further research.

2 Background

Based on observations from two well-known implementations of the OCL standard, namely the Dresden OCL2 Toolkit and the Kent OCL Library, we can identify two major challenges for a model-level integration of the standard library in an OCL engine. In brief, these are:

1. Operations and parameters in the standard library instantiate the corresponding metaclasses from the UML metamodel [9]. When OCL is integrated with domain-specific modeling languages developed within so-called language workbenches [10] or via UML profiles, we cannot rely on a common format for the library any more.
2. The standard library contains template types and operations that are parameterized with a type parameter. Most modeling languages do not support a declarative definition of these generic elements.

In the following, we will discuss these two issues in greater detail.

2.1 OCL for Domain-Specific Modeling Languages

In recent years, the importance of domain-specific languages (DSLs) for describing systems has increased and a convergence with model-driven approaches such as the OMG MDA initiative [11] can be witnessed [12]. As a result, the original scope of OCL being an add-on to UML [13] has widened to support constraints and queries over object-based modeling languages in general [14].

An obvious solution to these new challenges is the introduction of a pivotal metamodel that abstracts over the metamodels of arbitrary domain-specific languages and provides exactly those features required for an integration with OCL. Both of our reference OCL engines work this way. The Dresden OCL2 Toolkit in its current version employs a so-called *Common Model* [15] to adapt the metamodels of UML 1.5 as well as MOF 1.4, while the Kent OCL Library supports UML 1.4, Ecore (the metamodel used by EMF), and Java via a central *Bridge* model [16].

Unfortunately, both solutions fail to decouple the model of the OCL standard library from the adapted metamodel. In the Dresden OCL2 Toolkit, the predefined library operations and their parameters are instances of the corresponding UML or MOF metaclasses, while in the Kent OCL Library they instantiate metamodel-specific adapter classes. Both approaches demand a programmatic creation of the standard library types and operations. Consequently, to model the standard library externally, we need to find a way to instantiate these elements independently from any adapted metamodel.

2.2 Generics in the OCL Standard Library

The predefined collection types in the standard library are actually *template types* with the type parameter `T` [1, p. 144]. As an example, consider the `sum` operation of the OCL `Collection` type whose return parameter is typed with the element type of the collection. We say that a concrete type `Collection(Integer)` is created from the template `Collection(T)` by substituting, or *binding*, `T` with the type `Integer`. Since element types may be nested, there is an infinite number of collection types which have to be dynamically created when parsing a particular OCL expression.

However, not only types can have type parameters. Consider the product operation of `Collection(T)` which returns the cartesian product of two collections: `product(c2:Collection(T2)):Set(Tuple(first:T,second:T2))`. Note that the concrete signature of this operation (in particular, its return type) not only depends on the binding of the type parameter `T`, but also on the type of the argument `c2`. This is an example of a so-called *generic operation* [17]. Further note that the return type of the product operation is itself a template type, namely `Set(T)`, whose type parameter `T` is bound with the generic type `Tuple(first:F, second:S)`. The actual type of the type parameters `F` and `S` is determined at runtime, based on the binding for `T` and `T2`, respectively. In this case, we call `T` and `T2` *type arguments* for the generic tuple type.

Finally, some of the predefined operations in the library have return types that depend on the object they are invoked on. Examples are `OclAny::asSet` (returning a singleton set containing the object) and `OclAny::allInstances` (returning the set of all instances of a type). Both operations have `Set(T)` as their return type, but the concrete binding for `T` cannot be determined until the source type of the operation call is known.

To remove the definition of the standard library from the implementation code and specify it declaratively, a mechanism to model generic types and operations is required. Moreover, the engine needs to support the binding of generic elements at runtime to dynamically create concrete types while parsing an OCL expression.

3 The Design of a Pivot Model with Generics Support

We are currently reengineering the Dresden OCL2 Toolkit to increase its reusability and flexibility and to provide the foundations for future research into the integration of OCL with arbitrary domain-specific languages. To this end, we have redesigned and reimplemented large parts of the toolkit's infrastructure [18]. The new architecture features a more flexible model repository adaptation mechanism. It is based on a pivot model that results from a careful analysis of previous approaches (cf. Sect. 2.1) and the `Core::Basic` package of UML 2.0. So far, we have integrated both EMF and the Netbeans Metadata Repository [19] and implemented bindings for Ecore, MOF and UML. The main elements of the new pivot model are shown in Fig. 1.

A comprehensive discussion of the new architecture is outside the scope of this paper. However, for a better understanding of the following paragraphs we would like to draw attention to one noteworthy feature that sets it apart from existing OCL implementations: A layered architecture now eliminates any dependencies from the pivot model to the OCL metamodel. Thus, the support for model-level generics, which we will describe below, is only an enabling technology for modeling the template types in the OCL standard library. All necessary functionality

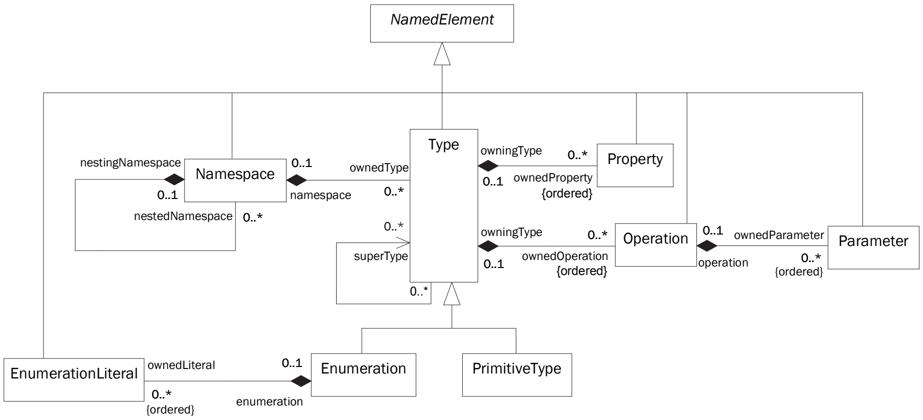


Fig. 1. The main elements of the pivot model

is already contained in the implementation of the pivot model metaclasses and can easily be leveraged for alternative model querying languages.

Figure 2 summarizes how the pivot model introduces template types and operations as first-class model entities. The design is loosely based on the generics support in EMF 2.3 [20] which closely mirrors the generic capabilities of Java 5 [17]. The key idea is to introduce a new abstraction called `GenericElement` which classifies elements that may contain one or several `TypeParameter`s.

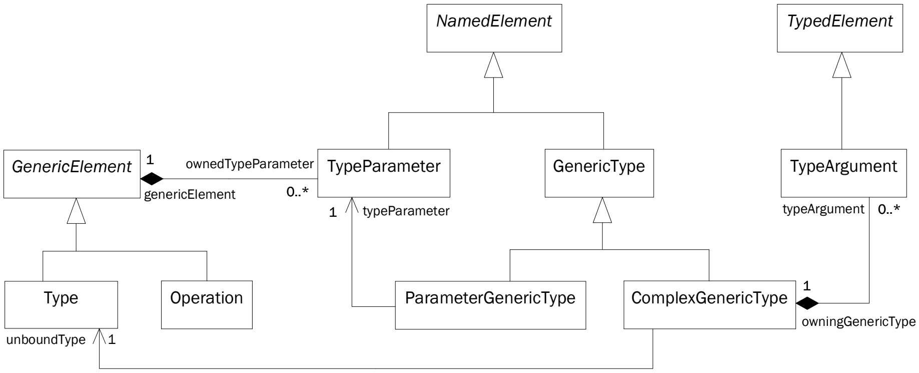


Fig. 2. Generics in the pivot model

The type parameters of a generic element may be *bound* with a concrete type, which means that all occurrences of the parameter in the definition of the generic element are replaced with this type (Fig. 3). In the case of a `Type` instance, this will affect all properties and operations (including their parameters) declared for this type.

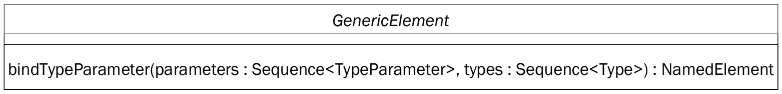

GenericElement
bindTypeParameter(parameters : Sequence<TypeParameter>, types : Sequence<Type>) : NamedElement

Fig. 3. Binding the type parameters of generic elements

In line with other metamodels, the pivot model generalizes properties, operations and parameters with an abstract metaclass `TypedElement` that declares a reference to a type. Now, as illustrated in Fig. 4, we allow typed elements to alternatively reference a `GenericType`. Generic types exist in two flavours (cf. Fig. 2). A `ParameterGenericType` simply references a `TypeParameter`, as in the case of the return parameter of the `sum` operation mentioned in Sect. 2.2. A `ComplexGenericType`, on the other hand, references another `Type` with unbound type parameters as well as a number of `TypeArguments` that will replace the type parameters during binding. In the example of the `product` operation, the return parameter contains a complex generic type referencing the unbound type `Tuple(first:F,second:S)` and defining two type arguments `T` and `T2`. This example shows nicely that type arguments, being typed elements themselves, can have a generic type as well. Through this design, an unlimited nesting of generic types becomes possible.

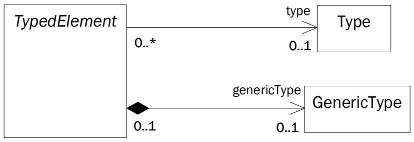


Fig. 4. Typed elements and generic types

It turns out that supporting generic types for typed elements does not suffice yet. Consider the OCL collection type `Sequence(T)`. This template type extends `Collection(T)`. Intuitively, binding type parameter `T` of `Sequence(T)` with a concrete type, say `String`, should result in `Sequence(String)` extending `Collection(String)`. Yet, the design developed so far does not cover this special case. The key observation here is that the two type parameters `T` are not the same. In fact, it is perfectly legal to label the type parameter of the sequence type with `S` instead of `T`. Correctly binding both subtype `Sequence(S)` and supertype `Collection(T)` requires `S` to be a `TypeArgument` of `Collection(T)`. This intuition leads to the introduction of a new association between `Type` and `GenericType` denoting the generic supertypes of a type (Fig. 5). Then, binding a type will cause all generic supertypes to be bound as well. If all type parameters of a generic super type are bound (i.e., it is not generic any more), it can be safely added to the regular `superType` reference list (cf. Fig. 1).

On a side note, it is worth highlighting that in contrast to EMF, our pivot model does not know the notion of a *raw type*, i.e., a “fallback” type that is as-

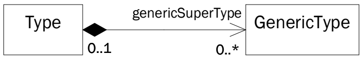


Fig. 5. Generic supertype

sumed to exist for any type parameter in an unbound generic type. This directly stems from the fact that we aimed to avoid any dependencies from the pivot model to the OCL metamodel. Otherwise, `OclAny` as the root of the OCL type system would have been a logical choice. To guarantee proper type conformance checking, we have suitably extended the implementation of the OCL collection metaclasses instead.

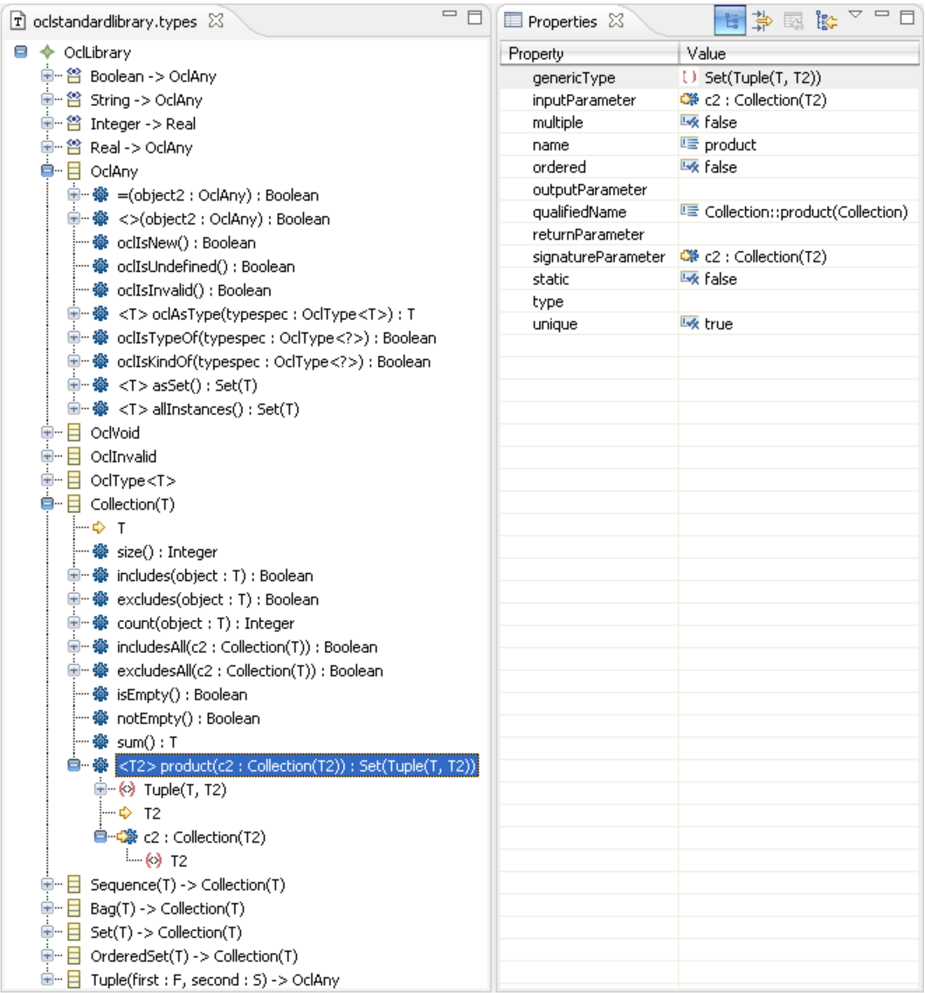
4 Practical Evaluation

The previous section presented the design of a pivot model with explicit support for generics. Now, we can proceed with showing its application. We have realized the new infrastructure of the Dresden OCL2 Toolkit as a set of Eclipse plug-ins. To create implementation classes for the pivot model elements, we employed the metamodeling and code generation facilities of the Eclipse Modeling Framework. This yielded the following advantages:

1. Except for some behavioral features that have to be implemented manually, the pivot model implementation generated by EMF is already fully functional and can be instantiated. Contrary to previous approaches, we do not depend on an integration with a particular DSL to create an instance of the OCL standard library. By realizing the same interfaces, our standard library model is compatible with any metamodel binding that is created for the pivot model.
2. The XMI serialization capabilities of EMF enable us to effectively save and load the standard library which improves reusability.
3. EMF can generate a highly customizable tree editor for a metamodel. In the next section, we show how a heavily adapted version of the default pivot model editor allows the user to conveniently view, edit and alter the model of the standard library.

4.1 Visually modeling the OCL standard library

Figure 6 shows the model of the standard library in the adapted pivot model editor. This model, which contains all types and operations defined in the OCL 2.0 specification, is part of the new toolkit infrastructure. Users may, however, replace the default library with a modified version when integrating a new domain-specific language with the engine. For instance, if a DSL does not know the concept of an ordered set, the OrderedSet type can be safely removed from the library model. This ensures that all valid abstract syntax models created by a parser will indeed execute on the domain-specific target platform.



```
OclLibrary
    Boolean -> OclAny
    String -> OclAny
    Integer -> Real
    Real -> OclAny
    OclAny
        =(object2 : OclAny) : Boolean
        <>(object2 : OclAny) : Boolean
        oclIsNew() : Boolean
        oclIsUndefined() : Boolean
        oclIsInvalid() : Boolean
        <T> oclAsType(typespec : OclType<T>) : T
        oclIsTypeOf(typespec : OclType<?>) : Boolean
        oclIsKindOf(typespec : OclType<?>) : Boolean
        <T> asSet() : Set(T)
        <T> allInstances() : Set(T)
    OclVoid
    OclInvalid
    OclType<T>
    Collection(T)
        T
        size() : Integer
        includes(object : T) : Boolean
        excludes(object : T) : Boolean
        count(object : T) : Integer
        includesAll(c2 : Collection(T)) : Boolean
        excludesAll(c2 : Collection(T)) : Boolean
        isEmpty() : Boolean
        notEmpty() : Boolean
        sum() : T
        <T2> product(c2 : Collection(T2)) : Set(Tuple(T, T2))
            Tuple(T, T2)
            T2
            c2 : Collection(T2)
                T2
    Sequence(T) -> Collection(T)
    Bag(T) -> Collection(T)
    Set(T) -> Collection(T)
    OrderedSet(T) -> Collection(T)
    Tuple(first : F, second : S) -> OclAny
```

Property	Value
genericType	Set(Tuple(T, T2))
inputParameter	c2 : Collection(T2)
multiple	false
name	product
ordered	false
outputParameter	
qualifiedName	Collection::product(Collection)
returnParameter	
signatureParameter	c2 : Collection(T2)
static	false
type	
unique	true

Fig. 6. The model of the OCL standard library

The look and feel of the pivot model editor resembles that of the EMF Ecore editor. However, we have simplified the modeling of generics to hide complexity from the user. When creating typed elements (properties, operations, and parameters), declared type parameters of the containing generic element show up in the list of possible types. The editor automatically creates the necessary `ParameterGenericType` instance in this case. If a template type is selected (e.g., for the `c2` parameter of the `product` operation), a complex generic type and corresponding type arguments are added. Similarly, the editor allows to specify the type arguments when extending generic supertypes.

The root of the model is an instance of a special facade interface called `OclLibrary`. Its definition is outlined in Fig. 7. The `OclLibrary` interface provides the necessary means for an OCL parser to retrieve the predefined standard library types when building the abstract syntax model from an OCL expression.

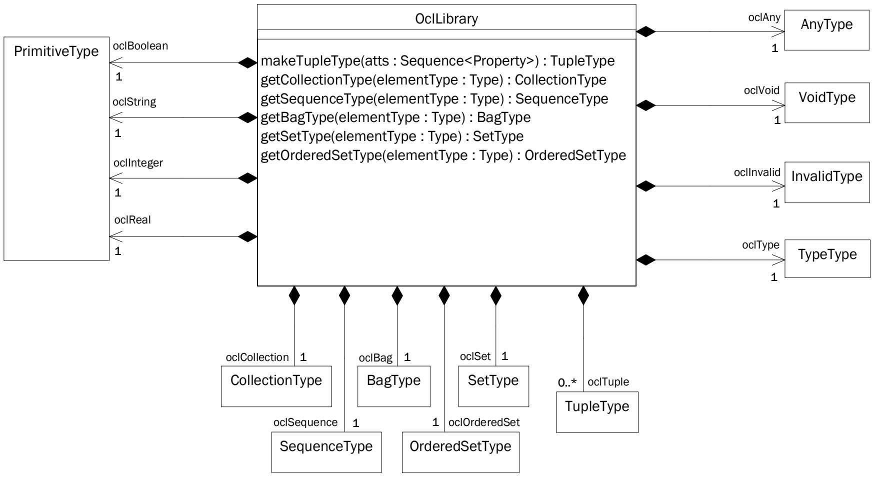

OclLibrary
makeTupleType(atts : Sequence<Property>) : TupleType
getCollectionType(elementType : Type) : CollectionType
getSequenceType(elementType : Type) : SequenceType
getBagType(elementType : Type) : BagType
getSetType(elementType : Type) : SetType
getOrderedSetType(elementType : Type) : OrderedSetType
PrimitiveType
oclBoolean
1
oclString
1
oclInteger
1
oclReal
1
oclAny
1
AnyType
oclVoid
1
VoidType
oclInvalid
1
InvalidType
oclType
1
TypeType
oclCollection
1
CollectionType
oclBag
1
BagType
oclSet
1
SetType
0..*
oclTuple
TupleType
oclSequence
1
SequenceType
1
oclOrderedSet
OrderedSetType

Fig. 7. The facade interface for the OCL standard library types

4.2 Binding template types during OCL parsing

In the following, we demonstrate the feasibility of our approach with a simple example that involves the binding of generic OCL collection types while parsing an OCL expression. Figure 8 depicts the model we will base the example scenario on.

Now, consider the following OCL expression which specifies the derived attribute `totalBalance` in class `Person`. Note that the second invocation of the dot operator (accessing the property `balance` of all elements in the `accounts` reference list) represents an implicit *collect* iterator.

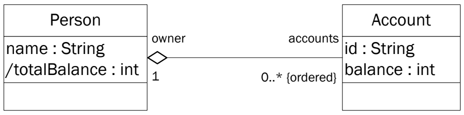
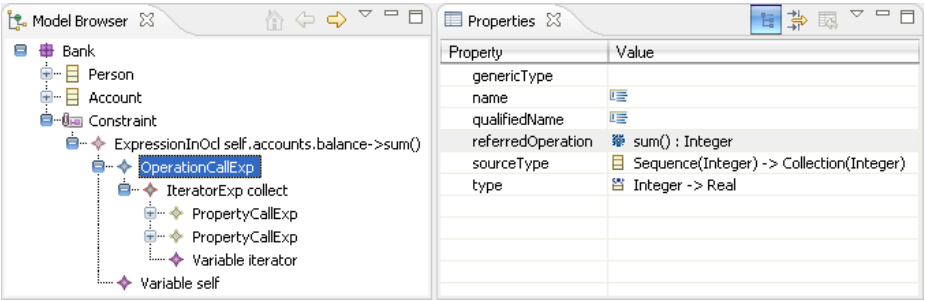


Fig. 8. The example model

```
context Person::totalBalance : int
derive: self.accounts.balance->sum()
```

Parsing this expression triggers the following template type bindings: First, the type of the property call expression referring to `accounts` is evaluated to be `OrderedSet(Account)`. This directly stems from the multiplicity specification and declared type of the property. The actual binding of the `OrderedSet` template with the element type `Account` is facilitated within the `getOrderedSetType` operation of the `OclLibrary` facade. As shown in Sect. 3, this solely requires a call to the `bindTypeParameter` operation implemented in the `Type` metaclass of the pivot model.

Similarly, the type of the *collect* iterator expression, which returns the list of individual balance values, results from binding `Sequence(T)` with `Integer`. The engine automatically maps the domain-specific `int` type to the corresponding OCL standard library type. As a result, the return type of the `sum` operation becomes `Integer`. To sum up this discussion, Fig. 9 shows the abstract syntax model of the example expression as it is visualized in the DSL-agnostic model browser that is part of the new toolkit infrastructure. Notice that not only the `Sequence` template has been bound, but also its generic supertype `Collection(T)`.



```
Bank
  Person
  Account
  Constraint
    ExpressionInOcl self.accounts.balance->sum()
      OperationCallExp
        IteratorExp collect
          PropertyCallExp
          PropertyCallExp
          Variable iterator
        Variable self
```

Property	Value
genericType	
name	
qualifiedName	
referredOperation	sum() : Integer
sourceType	Sequence(Integer) -> Collection(Integer)
type	Integer -> Real

Fig. 9. The abstract syntax model of the example expression

It is worth highlighting here that the method presented in this paper solely addresses the static structure of the OCL standard library. To realize the dynamic semantics and actually execute the expression in Fig. 9, we still rely on an *instance-level* (M0) implementation of the predefined types and operations.

To this end, we have redesigned the existing Java library of the Dresden OCL2 Toolkit to support a flexible integration of arbitrary DSLs via a set of factory interfaces. Currently, an OCL interpreter based on the new infrastructure is being developed to complement the components on the model level.

5 Related Work

To the best of our knowledge, there is no published work that deals with the model-level integration of the OCL standard library as described in this paper. Akehurst et al. [14] hint at this possibility, but simply suggest to import a UML package containing the standard library types. Therefore, they do not address the problems outlined in Sect. 2. However, they propose a mechanism to detach the implementation of the standard library types and operations on the *instance level*. The ideas from this work may complement our approach and further simplify the integration of different domain-specific languages.

Another technique that aims at aligning OCL with custom domain-specific languages on the instance level has been presented in [21]. Unfortunately, the authors employ a custom expression language that is akin but not equal to OCL [22]. Furthermore, they build on top of a model management framework and execution engine which does not support a model-level integration of the standard library.

Lastly, the latest release of the Eclipse MDT OCL project [23] features a highly innovative way of integrating OCL with different modeling languages. Instead of a pivot model, a generic environment interface defines type parameters for all metamodeling concepts required by OCL. Unfortunately, this otherwise elegant approach necessitates a concrete specialization of the entire OCL metamodel as well as the OCL standard library for each custom DSL binding. The predefined operations of the standard types have to be created within the implementation code yielding the disadvantages highlighted in Sect. 1.

6 Conclusions and Future Work

In this paper, we have presented a novel technique for integrating the OCL standard library on the model level. Contrary to previous approaches, we support a declarative rather than a programmatic definition of the predefined types and operations thereby improving reusability, flexibility, and maintainability. In addition, our method eases the integration of different domain-specific languages with OCL, because the pivot model provides an intermediate abstraction layer for a variety of metamodels. Therefore, instantiating elements of the library model is independent of a particular DSL binding and solely requires a suitable implementation of the pivot model interfaces. We have demonstrated the feasibility and usefulness of our approach through an example that was realized using newly developed components of the Dresden OCL2 Toolkit.

We are currently working on porting the tools of the Dresden OCL2 Toolkit to the new infrastructure. Our aim is to leverage the increased flexibility provided by our approach for other OCL-based languages defined by the OMG. Examples are the *Query/View/Transformation* (QVT) [24] language and the upcoming *Production Rule Representation* (PRR) [25] standard. This may open up interesting perspectives for areas as diverse as model transformation and business rule execution.

Finally, our solution still faces some limitations that are worthwhile to address. For instance, our pivot model currently lacks the expressive power to model the dynamic semantics of iterator expressions for the OCL collection types. In fact, detaching the definition of iterators requires a different approach altogether since the corresponding well-formedness rules for the abstract syntax are currently heavily intertwined with the concrete syntax. Similarly, we have not yet found a satisfying answer to the problem of binding generic operations whose return type depends on contextual information (e.g., `allInstances` and `asSet` in `OclAny` or `flatten` in the collection types). Even though we are able to model the signature of these operations, we still have to check for them explicitly in the code. Thus, the implementation of the OCL abstract syntax elements (M2) still contains a few details of the standard library structure (M1).

Acknowledgment: The authors would like to thank Florian Heidenreich, Christian Wende and Steffen Zschaler for their fruitful discussions and contributions to the Dresden OCL2 Toolkit.

References

1. Object Management Group (OMG): Object Constraint Language, Version 2.0. (2006) `http://www.omg.org/docs/formal/06-05-01.pdf`.
2. Object Management Group (OMG): Meta Object Facility (MOF) Core Specification, Version 2.0. (2006) `http://www.omg.org/docs/formal/06-01-01.pdf`.
3. Stahl, T., Völter, M.: Model-Driven Software Development: Technology, Engineering, Management. 1st edn. Wiley & Sons (2006)
4. Technische Universität Dresden, Department of Computer Science: (Dresden OCL Toolkit) `http://dresden-ocl.sourceforge.net`.
5. University of Kent at Canterbury, Department of Computing: (Kent Object Constraint Language Library) `http://www.cs.kent.ac.uk/projects/ocl`.
6. Kappel, G., Kapsammer, E., Kargl, H., Kramler, G., Reiter, T., Retschitzegger, W., Schwinger, W., Wimmer, M.: On Models and Ontologies - A Layered Approach for Model-based Tool Integration. In Mayr, H.C., Breu, R., eds.: Proceedings of Modellierung 2006, GI-Edition, Lecture Notes in Informatics, Innsbruck, Austria, 22-24 March. (2006)
7. Milanovic, M., Gasevic, D., Giurca, A., Wagner, G., Devedzic, V.: On Interchanging Between OWL/SWRL and UML/OCL. In: Proceedings OCLApps 2006: OCL for (Meta-) Models in Multiple Application Domains, MoDELS/UML 2006, Genova, Italy (2006) 81–95
8. Budinsky, F., Steinberg, D., Merks, E., Ellersick, R., Grose, T.J.: Eclipse Modeling Framework. 1st edn. Eclipse Series. Addison Wesley Longman, Amsterdam (2003)

9. Object Management Group (OMG): Unified Modeling Language: Superstructure Specification, Version 2.0. (2005) http://www.omg.org/docs/formal/05-07-04.pdf.
10. Fowler, M.: Language Workbenches: The Killer-App for Domain Specific Languages? White Paper (2005) http://www.martinfowler.com/articles/languageWorkbench.html.
11. Object Management Group (OMG): MDA Guide Version 1.0.1. Technical report (2003) http://www.omg.org/docs/omg/03-06-01.pdf.
12. Bézivin, J., Jouault, F., Kurtev, I., Valduriez, P.: Model-Based DSL Frameworks. In: Companion to the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, Portland, Oregon, USA, ACM Press (2006) 602–616
13. Warmer, J., Kleppe, A.: The Object Constraint Language Second Edition: Getting Your Models Ready for MDA. Object Technology Series. Addison-Wesley Longman, Amsterdam, The Netherlands (2003)
14. Akehurst, D., Howells, W., McDonald-Maier, K.: UML/OCL – Detaching the Standard Library. In: Proceedings OCLApps 2006: OCL for (Meta-) Models in Multiple Application Domains, MoDELS/UML 2006, Genova, Italy (2006) 205–212
15. Loecher, S., Ocke, S.: A Metamodel-Based OCL-Compiler for UML and MOF. Electronic Notes in Theoretical Computer Science (2004) 43–61
16. Akehurst, D., Patrascoiu, O.: OCL 2.0 – Implementing the Standard for Multiple Metamodels. Electronic Notes in Theoretical Computer Science (2004) 21–41
17. Bracha, G.: Generics in the Java Programming Language. Technical report, Sun Microsystems (2004) http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf.
18. Bräuer, M.: Design and Prototypical Implementation of a Pivot Model as Exchange Format for Models and Metamodels in a QVT/OCL Development Environment. Technical report, Technische Universität Dresden (2007) http://dresden-ocl.sourceforge.net/gbbraeuer/.
19. Netbeans.org: (Metadata Repository (MDR)) http://mdr.netbeans.org.
20. Merks, E., Paternostro, M.: Modeling Generics With Ecore. In: EclipseCon 2007, Santa Clara, California, USA, IBM Corp. (2007) Tutorial Slides. http://www.eclipsecon.org/2007/index.php?page=sub/&id=3845.
21. Kolovos, D.S., Paige, R.F., Polack, F.A.: Towards Using OCL for Instance-Level Queries in Domain Specific Languages. In: Proceedings OCLApps 2006: OCL for (Meta-) Models in Multiple Application Domains, MoDELS/UML 2006, Genova, Italy (2006) 26–37
22. Kolovos, D.S., Paige, R.F., Polack, F.A.: The Epsilon Object Language (EOL). In: Proceedings European Conference in Model Driven Architecture (EC-MDA) 2006, Bilbao, Spain. Volume 4066 of LNCS., Springer (2006) 128–142
23. Eclipse.org: (Model Development Tools (MDT) OCL component) http://www.eclipse.org/modeling/mdt/?project=ocl.
24. Object Management Group (OMG): Meta Object Facility (MOF) 2.0 Query/View/Transformation (QVT) Specification. (2005) http://www.omg.org/docs/ptc/05-11-01.pdf.
25. Object Management Group (OMG): Production Rule Representation Request For Proposal Draft 1.0. (2003) http://www.omg.org/docs/bei/03-08-04.pdf.