OCL: Modularising the Language

D.H.Akehurst[1], S. Zschaler[2], W.G.J.Howells[1]

[1]University of Kent, Canterbury, UK
{D.H.Akehurst, W.G.J.Howells}@kent.ac.ukc
[2]Technische Universität Dresden, Dresden, Germany
Steffen.Zschaler@tu-dresden.de

The Object Constraint Language (OCL) was originally designed as an 'add-on' to the Unified Modelling Language (UML) in order to facilitate writing textual constraints complementing the graphical specifications. Since its original standardisation many extensions have been added to the language and many more have been proposed. The original structure of the OCL definition has not been formed, however, with a view of extensibility. Still, OCL can be redesigned in such a manner that it becomes easy to extend the language. In this paper we present a modular redefinition of OCL and illustrate how it supports extension. This new approach to the design of OCL enables you to consistently extend or customise OCL to your own needs.

1 Introduction

When developing the Unified Modelling Language (UML), its designers understood early on that visual models are not sufficiently expressive for all modelling needs. Therefore, they paired UML with the Object Constraint Language (OCL)—a textual constraint language closely linked with the concepts of UML, but providing the power of first-order predicate logic, and beyond. OCL was well accepted at least in the research community and a number of tools supporting parsing and analysing OCL as well as for generating code for various target systems have been developed. Because of its intuitive syntax and reasonable tool support, OCL came to be used in various other contexts. For example, OCL has been used in CQML [1] a language for modelling non-functional properties of component-based systems, and in the new QVT standard [20] as a means of describing model transformations. In the process, OCL has changed from a simple constraint language to a rather powerful query language. Many extensions to the original OCL have been proposed, for example for modelling temporal or geographical properties [7, 10, 22, 23]; quite a few of them have been incorporated in new versions of the OCL standard.

However, the design of OCL, from its beginnings to its current shape, has not taken into account this requirement for modification. Hence, although the language definition has grown in size, it has not improved in modularity. This is bad, because it dramatically reduces understandability of the language. Indeed, the current standard contains a number of inconsistencies as anybody attempting to provide tool support will have noticed. OCL's lack of modularity is bad for another reason: It makes it very difficult to systematically introduce extensions to the language. However, many such extensions are still being proposed. Two main sources for such proposals are:

1. Whenever OCL is being integrated with other languages (for example, for modelling non-functional properties or for describing model transformations), new concepts need to be integrated. This includes concepts at both the type and the concept level.
2. Although OCL supports first-order predicate logic and some concepts beyond that, it is still missing important logical concepts. Most importantly, OCL currently provides no support for temporal-logic specifications.

In this paper, we discuss a modular redesign of OCL. This redesign has two goals: 1) to modularise OCL so that it becomes easy to systematically *extend* it as required, and 2) to modularise OCL so that it becomes easy to customise OCL to specific project requirements or didactic objectives by *removing* features from the language. The customisability thus achieved will, in addition, simplify integrating OCL with languages other than UML.

A modular redesign of a language such as OCL must cover the language's concrete and abstract syntax, and its semantics as indicated in Fig. 1. The figure gives an overview of the different parts of what we call *modular language design.* It shows two *language modules* (a concept to be defined later) consisting of a partial grammar, a partial metamodel and a partial semantic domain, all related by corresponding transformations. When a language module extends another language module, really all its parts need to extend the respective parts of the extended language module. In this paper, we will focus on the syntactical aspects of modular language designs, aspects of semantics will be an issue for future work.

In current approaches, when a new language feature is to be added to a language like OCL, a completely new language must often be developed from scratch. This situation needs to be changed towards a more engineering-oriented approach to language design. This necessity has also been acknowledged by Klint et al. who name this field "Grammarware Engineering" [12]. The work presented in this paper is, thus, a contribution to grammarware engineering.

The remainder of this paper is organised as follows. Section 2 introduces the background to the technology, illustrating the current state of parsing technology and the current OCL standard. This is followed by Sect. 3, where the primary ideas regarding a modularised extension to OCL are presented. The extension mechanism is subsequently evaluated in Sect. 4 by applying it to one non-standard extension proposal from the OCL literature. The paper concludes by discussing an implementation of the approach and possible future work associated with the technology.

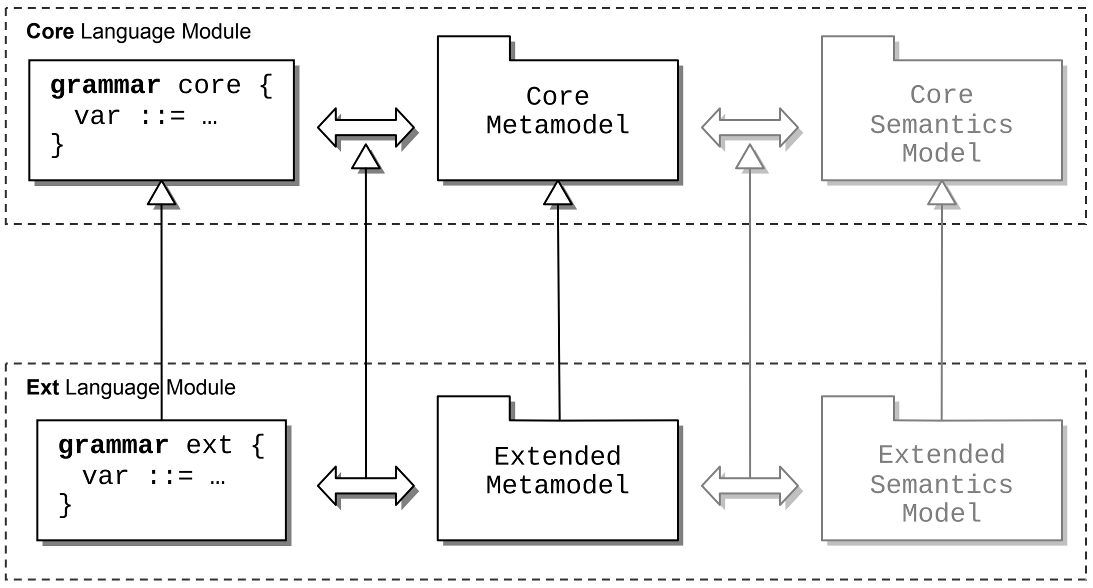

Fig. 1 A general view of modular language design

2 Background

In this section, we give a short overview of the background of this work. In particular, the first sub-section discusses the current state of the art of parsing technology, while the second sub-section gives a short overview of the current OCL standard.

2.1 Current state of parsing technology

The compiling process for textual languages defined by a metamodel can be split into a number of different stages:

1. **Scanning** – Takes the input sequence of characters and produces a sequence of tokens.
2. **Parsing** – Takes the sequence of tokens and adds structure according to a grammar producing a parse tree. The parse tree nodes represent the syntax elements interpreted by the grammar.
3. **Abstraction** – Converts the parse tree into an abstract syntax tree. Nodes in this tree are closer to the abstract concepts of the language than to the syntactical elements.
4. **Binding** – relates named elements in the abstract representation of an expression to their definition

Scanner and parser are typically generated from a grammar of the language. Most often, this is a context-free grammar [16], which is used to generate a top-down parser and scanner. Some parser generators will create bottom-up parsers, which can deal with more complex languages, but are also more complex, and often slower, themselves. The textual form of the language processed by scanner and parsers is referred to as the concrete syntax of the language. The result of scanning and parsing a piece of concrete syntax is a so-called concrete syntax tree; that is a tree-shaped internal representation based directly on the grammar of the concrete syntax.

For many post-processing tasks, dealing directly with the concrete syntax tree is inconvenient. Therefore, an additional abstraction step is performed, transforming the concrete syntax tree into an abstract syntax tree by removing nodes only relevant in the process of parsing. For example, nodes corresponding to additional non-terminals used to express precedence between operators in the grammar can be removed once the parse tree has been derived correctly. Also, nodes corresponding to terminal symbols for keywords can be removed if the same information can be derived from the type of the syntax tree node.

In addition, in modern language tools, an additional binding step resolves any references between different parts of a text (for example, between variable declarations and variable usage). This results in a directed-graph representation of the text. The structure of such a graph is normally determined by a so-called abstract syntax metamodel, the actual graph is then considered to be an instance of this abstract syntax. We sometimes refer to such a graph as an abstract syntax graph or ASG for short.

The transformation from concrete to abstract syntax still often needs to be implemented manually. However, some techniques exist that allow some or all of this to be generated. For example, many parser generators (e.g., [8, 21]) allow additional code or annotations to be used in a grammar to express how concrete syntax trees can be transformed into abstract syntax trees. Also, generators based on attribute grammars [11] allow the creation of ASGs to be declaratively expressed in the grammar.

Modern parser generators provide limited support for modularising language specifications. For example, the ANTLR parser generator [21] provides a concept for grammar inheritance. This allows grammars to inherit productions from other grammars and to extend and modify these definitions as appropriate. The JavaCC parser generator [8] has a notion of lexer states allowing the lexical analysis of different sub-languages to be combined easily. At the abstract-syntax level, mechanisms exist for combining different abstract syntaxes into one. For example, the UML Infrastructure specification [19] provides a package merge mechanism that allows to combine two metamodels into one. Still, however, most language definitions are built in a monolithic form. We believe the main reason for this to be that the existing approaches are not well connected. It is our goal with this paper, to advance the state of the art in this direction by describing one possible combination of technologies for modularising a complete language description.

2.2 The current OCL standard

The Object Constraint Language (OCL) is defined by a standard of the Object Management Group (OMG) [18]. This standard gives a monolithic definition of the OCL consisting of its concrete syntax, its abstract syntax, a model of OCL-specific types, a so-called standard library defining standard operations available on instances of OCL-specific types, a semantics, and means of connecting OCL expressions to Unified Modelling Language (UML) models. The standard is currently available in version 2.0, in which it relates to UML version 2.0.

The current OCL standard is, as mentioned, monolithically constructed. In particular, in the concrete syntax, but also in the abstract syntax and the semantics definition, many different concerns are tangled. This makes it very difficult to understand the language specification, check it for consistency, or extend it. Indeed, many inconsistencies can be found in the current standard.

Still, many extensions to OCL keep being proposed. For example, [7, 10, 23] propose different forms of temporal-logic extensions to OCL. It is, however, unclear how these extensions could be integrated into OCL in a consistent and systematic manner. At the same time, the language contains features—for example, the OclMessage construct to formulate constraints over messages sent by an operation—that are only useful for a part of the users of OCL and whose precise meaning is debated in the OCL community. For such constructs, it would be helpful to have a means of consistently removing them from the language if they are not required. We could also use some sort of style guide, restricting how the language should be used. However, in this case, those unused elements would still be in the language definition, contributing to the complexity of the language and to the difficulty of learning the language.

The OCL standard also defines a set of primitive and collection types to be used in OCL expressions. The mapping between these types and the corresponding types in a UML model or in a DSL is not entirely clear from the standard. In particular, these types define a (sometimes very narrow) set of operations that must be mapped on the corresponding operations in the corresponding types. These sets of operations defined in the standard may, furthermore, not be sufficient for writing precise and useful constraints. For this reason, it would be useful to be able to exchange the standard library types in OCL.

3 Redesigning the Standard

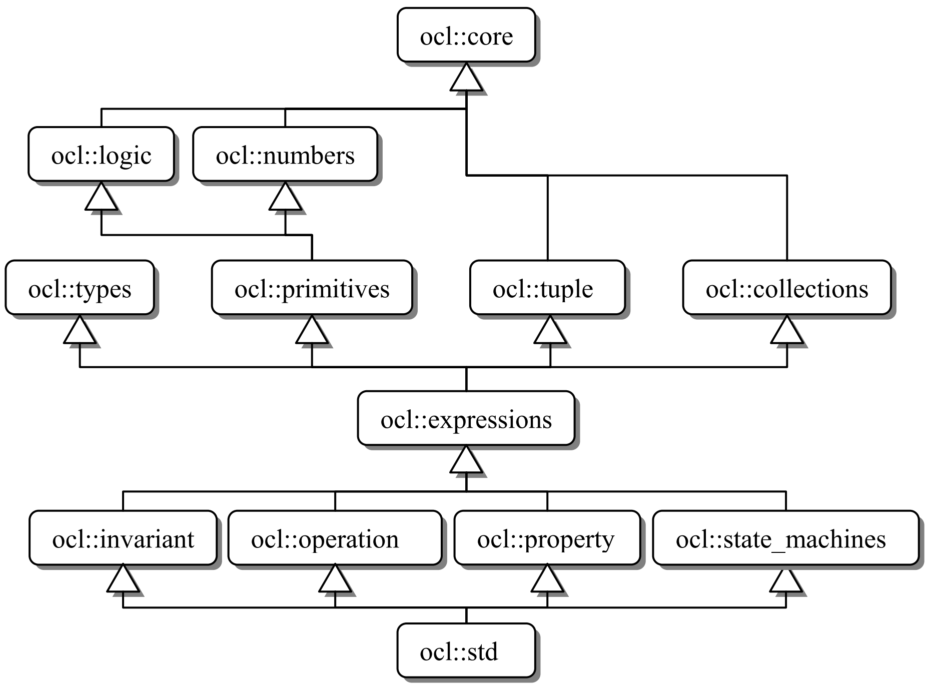

Fig. 2 OCL Language Modules[1]

Our suggested modules for a more modular redefinition of OCL are illustrated in Fig. 2. The root, *core*, module defines the fundamental part of OCL, which is the facility for navigation across object networks. The addition of the *logic*, *numbers*, *primitives*, *collections* and *tuples* modules provide the language with constant values, standard library types, and operators over those types. In the notation employed above, the inheritance arrow has its usual meaning. That is, for example, *primitives* depends on and extends the definitions in *logic* and *numbers.* The *types* module provides the facility for referencing literal types from an attached user model. These modules are brought together in a module called *expressions*, which would be the module most commonly extended by the numerous other languages that built on and around OCL. The *invariant*, *operation, property, and state_machines* modules provide the language components required for the standard OCL contexts in which expressions are usually given. Finally, the module *std*, brings together the other modules to provide a language that is equivalent to the current OCL standard.

We define a *language module* to be a module combining a grammar (i.e., a concrete syntax definition), (potentially) a metamodel (i.e., an abstract syntax definition), and some transformation rules that map syntax tree elements onto objects from the metamodel (language model). We do not prescribe that a language module must contain a metamodel, as for some languages or some modules it may be that the metamodel is better specified independently, for example, as metamodel elements may be reused in multiple modules.

A language module L_A can specialise other language modules L_0 to L_n. In this case, the language described by L_A is the language described by L_0 to L_n including the new definitions from L_A itself. The grammar of L_A is defined as a specialisation of the grammars from L_0 to L_n using some form of grammar inheritance. The metamodel is derived from the metamodel of L_0 to L_n and L_A. The set of transformation rules is the union of the transformation-rule sets from all language modules.

The concrete techniques we use for presenting the individual parts of language modules in this paper are as follows:

- **Grammar**: The grammars for our language module are written using an OO extension to EBNF that facilitates modularisation, inheritance, reuse and redefinition of rules. We do not give a definition of the extensions, but we consider them to be intuitive given an understanding of EBNF.

[1] The module code can be downloaded from http://www.ee.kent.ac.uk/~dha/OCLModularisingTheLanguage. The parser technology required to interpret the language modules can be obtained from the first author upon request by e-mail.

- **Metamodel**: The metamodel for OCL is introduced in segments as it is used by each language module. The metamodel elements are defined in the traditional manner, using class diagrams. The segments are intended to be composed by package merge.
- **Abstraction/Binding**: We define and implement the abstraction and binding steps of compilation using the MDD notion of transformations. This enables us to define sets of rules in each module that handle the mappings from syntax tree to OCL metamodel elements. There is not space in this paper to give the full specification of the rules we use; instead we show an overview of the rules and how they are related. The relationship between rules is an important mechanism in our technique for composing language modules.

3.1 ocl::core – Navigation

The fundamental or core aspect of OCL is the facility for navigation across object networks. The most obvious navigation is the use of properties (attributes and associationEnds in UML 1.X). Additionally, both qualified properties and method calls are also a form of navigation, starting from one object (or collection) and resulting in another.

Grammar

A simple grammar that supports the specification of such navigations is defined in Listing 1. This grammar defines the core navigation syntax for OCL, consisting of a starting name, followed by zero or more navigations over properties, qualified properties, or operation calls.

```
package ocl;

grammar core {

  expr = navExpr | rootExpr ;

  rootExpr = variable ;

  navExpr = propertyCall | operationCall ;

  variable = NAME ;
  propertyCall = expr '.' NAME [ '[' exprList ']' ] ;
  operationCall =  expr '.' NAME '(' [exprList] ')' ;

  exprList = (expr / ',')* ;

  NAME = "[a-zA-Z_][0-9a-zA-Z_]*" ;
}
```

Listing 1 OCL Core Grammar

Metamodel

The core elements of the OCL metamodel are illustrated in Fig. 3 and Fig. 4. In the standard definition, the OCL metamodel relies on certain definitions from the UML metamodel. Such reliance makes the assumption that OCL is going to be used to write expressions over UML models. However, as has been made apparent by many of the “miss-uses” of OCL [1, 3, 7, 10, 20, 23], the community at large may wish to use OCL over alternative metamodels. We thus define, separately to the UML metamodel, a set of “interface” types that must be supported by a metamodel, if OCL is to be used with it. This notion has been suggested in other works such as [5, 15].

The types Property and Operation are the metamodel representations of properties and operations defined within a UML user model. A VariableDeclaration is a representation of a typed name in the current context (environment), e.g. ‘self’, a parameter, or a name from a ‘let’ statement.

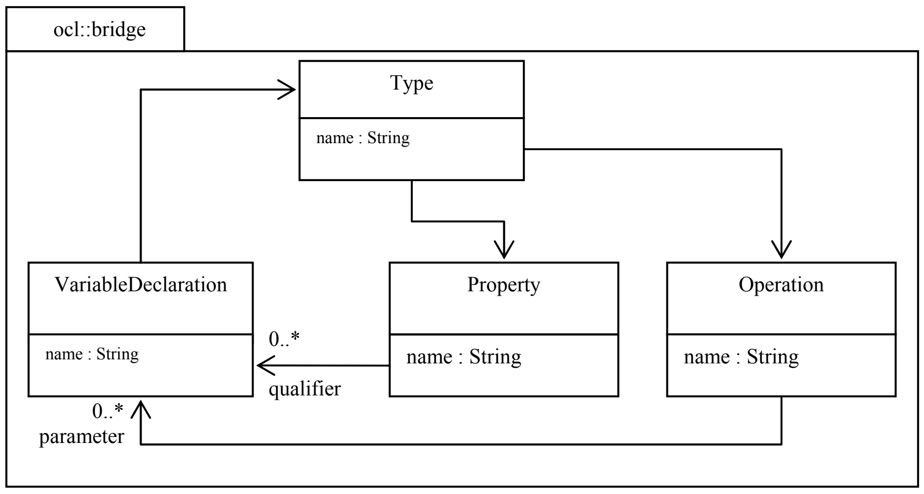

Fig. 3 Interface to metamodel

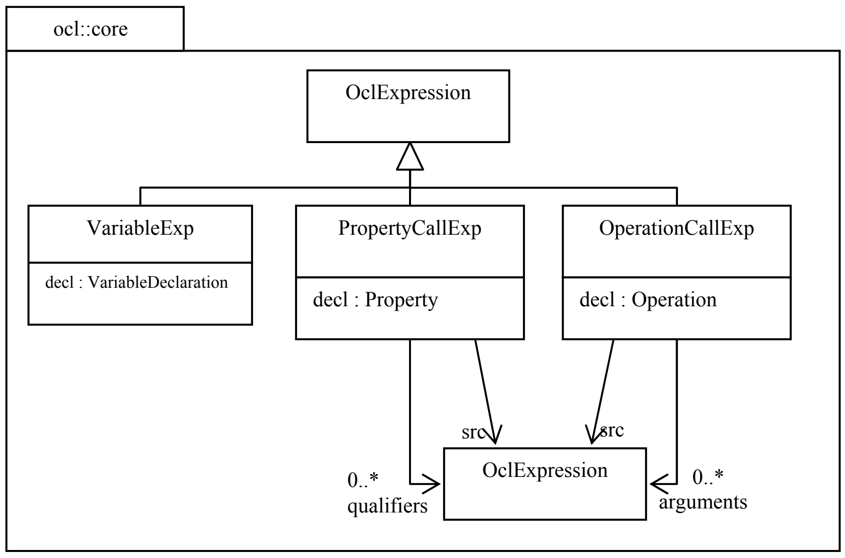

Fig. 4 Core Metamodel

Abstraction / Binding

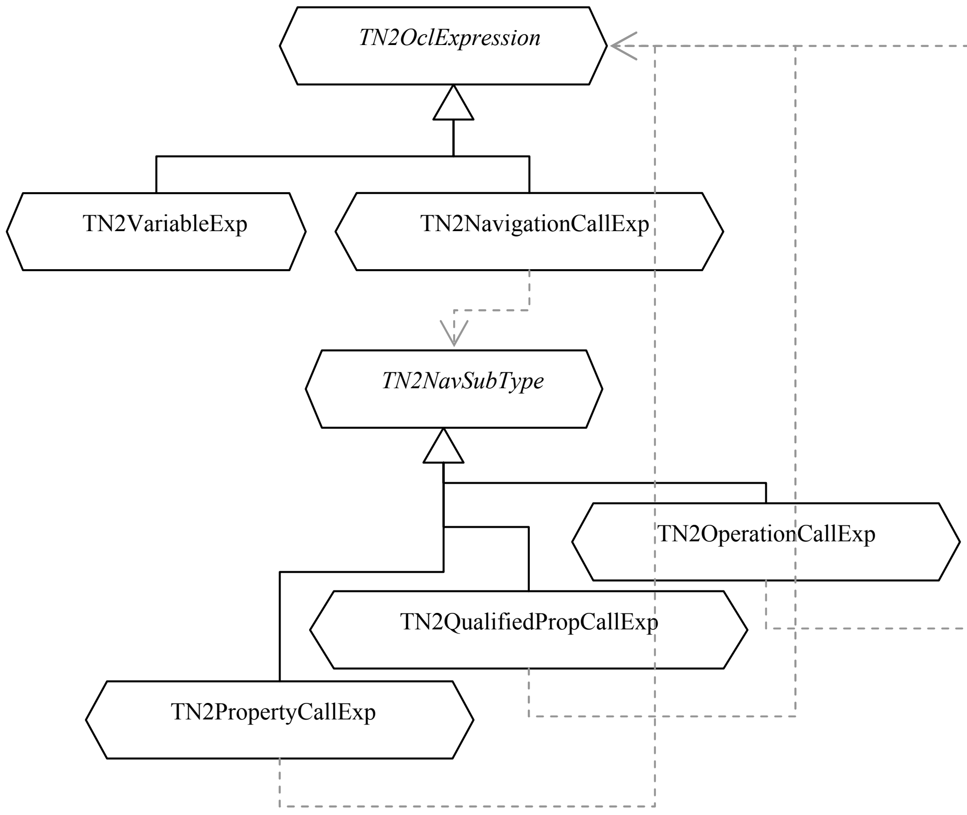

Fig. 5 Transformation Rules for ocl::core module

The transformation rules for the core module are illustrated in Fig. 5. These diagrams use our own notation, showing transformation rules as elongated hexagons. There are two types of relationship between rules illustrated:

- *Generalisation:* This relationship indicates a sub-typing relationship between rules, if a specification indicates use of a super-rule, then the actual rule used will be one of the sub-rules. The actual rule used is selected based on the domain object to be transformed and the constraints specified in each sub-rule. The rule system we used for implementing our modules actually allows super-rules to be reused by sub-rules. However, here, we make no use of this feature.
- *Dependency:* This indicates that the target rule (of the dependency) is used by the source rule to further transform some sub-part of the domain object being transformed by the source rule. For example, using the QVT notions, a rule specified in a 'where' clause.

The root rule maps an element of the syntax tree (TreeNode – TN) into an OclExpression. This rule is an abstract rule which facilitates extension in other modules. The realisation provided in the core module handles basic navigation expressions, the root of a navigation being a variable expression. In the core module there are three options for navigation, a property call, a qualified property call and an operation call. Rules for handling the abstraction and binding of each of these navigation types are provided. Note that there are two hierarchies of rules, one for managing alternative fundamental OCL expressions and the other for handling the alternative kinds of navigation expression.

The notation of Fig. 5 illustrates an overview of the transformations rules, indicating the relationship and connectivity between the different rules. Each rule can be specified in detail using a transformation language such as the QVT, or other QVT-like language. For example, details for the rule TN2VariableExp could be expressed as follows, using the QVT-based Kent Model Transformation Language [3]:

```
rule TN2VariableExp extends TN2OclExpression
src : TreeNode [ children.first.data = n ] when { src.name == 'variable' }
tgt : VariableExp [ decl = VariableDeclaration{ name = n } ]
```

We do not provide full details of the transformation rules in this paper, rather we indicate the approach. To implement the transformations, any one of the many transformation tools that are being developed could be used. Some detail on our tooling and implementation is given in a later section.

3.2 The Standard Library

The standard library in OCL consists of both the primitive types, Boolean, Integer, Real and String, and the Collection types, Bag, Set, Sequence, OrderedSet. In the standard, these types are closely tied in with the language itself. This is undesirable for at least two reasons:

1) The operations defined for standard types in the standard are insufficient for most useful expressions. The best example is the standard String type of OCL, which provides no means of inspecting the contents of a String other than comparing it—in its entirety—to some other String.
2) DSLs often define their own notions of these primitive types. When OCL is combined with a DSL different from UML it should be possible to reuse the DSL's primitive types within OCL expressions.

To solve these current problems, each of the primitive types—and thus the complete standard library—should be defined as a class in the user model as per any other class in the model. There are, however, two problems associated with removing these types from the standard definition:

1) The grammar defines the syntax for constant values that need to be mapped to instantiations of the standard library types.
2) Some of the methods defined on these classes are explicitly tied to concrete syntax elements in the OCL grammar. For example, consider the 'plus' method defined on an Integer type, this is not usually called using the core navigation syntax, rather the mathematical infix notation is used along with the '+' operator symbol

To achieve our aim of separating the standard library from the core definition of OCL, and to allow it to be 'replaceable' we need to define a customisable mechanism that links elements from the concrete syntax grammar to types supplied in the model and a mechanism for associating certain patterns of grammar elements to certain methods on a class.

We provide a definition that enables the "standard library" to be treated in the same manner as any other user model. And in such a manner, that with a little extra work, it can be replaced with an alternative 'user model'-level library.

Metamodel

Although we can treat the actual types in the standard library as 'user model' elements, it is still necessary for the OCL metamodel to understand the concepts of literal expressions, and facilitate the linkage of literal expression to the user model types. The OCL metamodel must therefore include elements for representing the notion of literal values. Fig. 6 indicates the metamodel elements for representing primitive and collection literal values. The OclExpression metamodel element has been inherited from the *core* language module (see Fig. 2). It is an abstract concept representing arbitrary OCL expressions.

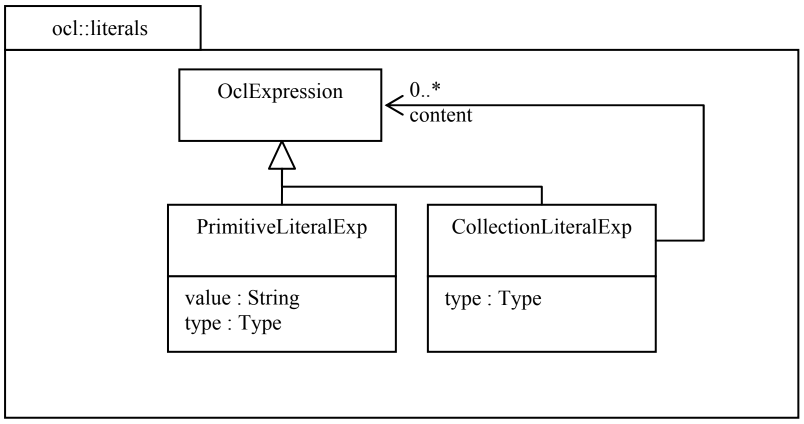

Fig. 6 OCL Metamodel - Literals

We do not have space in the paper to provide all the definitions of every language module. To illustrate the approach we show and discus the module for numbers.

ocl::numbers – Grammar

The grammar for numbers introduces literal numbers as alternative kind of root expression and extends the navigation options to allow for prefix and infix operator expressions. In addition we add parenthesis grouping for basic expressions as this is needed now that we have the operator expressions.

The ‘<’ symbol in the grammar rule for infix expressions defines (an operator) precedence between the different options.[2]

```
package ocl;

grammar numbers extends core {

  expr = core.expr
       | '(' expr ')' ;

  rootExpr = core.rootExpr | number ;

  navExpr = core.navExpr
          | infixExpr
          | prefixExpr
          ;

  infixExpr = expr '/' expr
            < expr '*' expr
            < expr '+' expr
            < expr '-' expr
            ;

  prefixExpr = '+' expr
             | '-' expr
             ;

  number = INTEGER | REAL ;

  INTEGER = "[0-9]+";

  REAL = "[0-9]+[.][0-9]+";

}
```

Listing 2 OCL Numbers Grammar

ocl::numbers – Abstraction / Binding

The binding process for the numbers module requires that we link the ‘type’ of a literal expression to a type in the user model that can be use to express the literal value. In addition we need to provide a mapping from the operators used in the grammar definition onto operations in the appropriate type.

Terminal	Type
INTEGER	ocl::stdLib::Integer
REAL	ocl::stdLib::Real

Table 1 Mapping primitive literal tokens to types from the standard library

Operator	Type	ArgTypes	Operation
/	ocl::stdLib::Integer	{ocl::stdLib::Integer}	Integer.divide()
*	ocl::stdLib::Integer	{ocl::stdLib::Integer}	Integer.multiply()
+	ocl::stdLib::Integer	{ocl::stdLib::Integer}	Integer.plus()
-	ocl::stdLib::Integer	{ocl::stdLib::Integer}	Integer.minus()
/	ocl::stdLib::Real	{ocl::stdLib::Real}	Real.divide()
*	ocl::stdLib::Real	{ocl::stdLib::Real}	Real.multiply()
+	ocl::stdLib::Real	{ocl::stdLib::Real}	Real.plus()
-	ocl::stdLib::Real	{ocl::stdLib::Real}	Real.minus()
+	ocl::stdLib::Real	{}	Real.self()
-	ocl::stdLib::Real	{}	Real.negate()

Table 2 Mapping operators to operations on standard-library types

To achieve this we provide two tables as input to the abstraction/binding rules. One that maps literal terminal names onto user model classes, and another that maps (operator, type, argumentType) tuples onto operations from a model type. Our definition of the numbers module thus requires the two tables Table 1 and Table 2. We assume that the user model includes a package named ocl::stdLib that includes primitive types such as Integer, Real, String and Boolean.

[2] Note that initial ideas along the lines of this paper have been presented in [4]. The *numbers* grammar is very close to the grammar shown there.

To make use of this information we define transformation rules that extend the core abstract rules. The numbers grammar introduces three new kinds of expression: literal numbers, infix operations, and prefix operations. The infix and prefix operations are special kinds of navigation, thus rules for the abstraction and binding of these syntax nodes can be implemented as extensions to the navigation rule. Fig. 7 illustrates the additional rules and shows how they relate to the core rules.

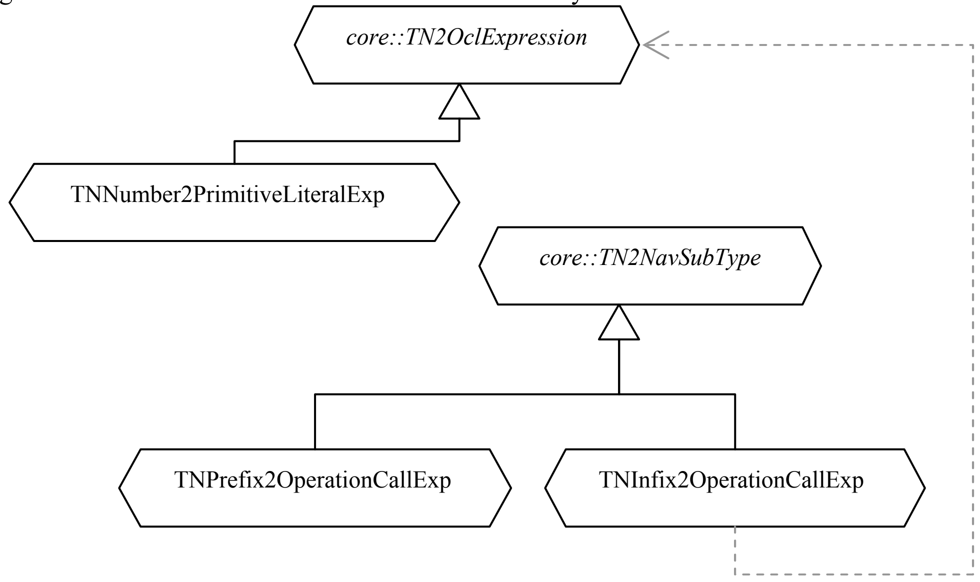

Fig. 7 Transformation Rules for ocl::numbers module

3.3 Deriving the standard language

Once all the language modules have been defined, as indicated above for selected examples, they need to be combined to derive the OCL of the current standard. As shown in the module listing in Fig. 2, we create a new language module std inheriting (directly or indirectly) from all the modules comprising the standard OCL. This ensures that the standard language contains all features provided by these modules.

This mechanism also provides for customisations of the language. If a particular feature is not required for some usage of OCL, a new language can be defined by specialising only those modules that are required. Because dependencies between modules have been made explicit through using specialisation relations between language modules, they will be respected automatically.

4 Integrating a Temporal Logic Extension

In [10] Flake et al propose a temporal logic extension to OCL. They are not the only authors to propose such an extension, [7, 23] also propose an alternative temporal-logic extensions. To illustrate the ease with which our new approach to defining OCL can be extended, we take the extension proposed by Flake et al and re-specify it using the techniques proposed in this paper.

The extension of Flake et al extends the notion of '@pre' navigation to include other temporal operations in addition to 'pre'. Such operations (e.g. 'post') can also take arguments that provide time bounds to the operator. Additionally, they add a new literal expression for traces; that is, sequences of states. To this end, they also need to extend the definition of `Sequence` to include a new operation:

`includesSequence(seq:Sequence(T)): Boolean`.

The current OCL standard does not allow such extensions easily. Flake et al invest considerable effort to describe their extension as formally as possible (which is not true for all extension proposals, alas), but still it would not be possible to integrate it directly into existing tooling. In this section, we are going to show how this extension can be formulated as a language module directly compatible with the overall approach of defining OCL as presented in this paper.

4.1 Realising the extension in our framework

To specify and implement this extension using our framework we need to:

- firstly, decide which of the base language modules to extend
- secondly, provide extensions to the grammar, metamodel and transformation rules
- and finally, provide a configuration of language modules that incorporates the new module

The new temporal operator is an extension to the basic expression language of OCL, hence we could extend the ocl::expressions module. However the extension is intended to facilitate the specification of temporal invariants, thus we extend the ocl::invariants module in order to incorporate a suitable context for writing temporal expressions. This is illustrated in Fig. 8.

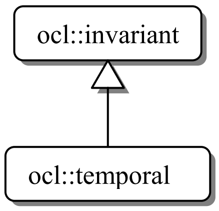

Fig. 8 OCL-T Language Module

Grammar

An extended grammar that supports the specification of temporal navigations is defined in Listing 3. This grammar extends the basic navigation expression with the syntax for a temporal navigation.[3]

```
package ocl;

grammar temporal extends invariant {

  navExpr = invariant.navExpr
          | temporalNavigation
          ;

  temporalNavigation = expr '@' NAME '(' exprList ')' ;

}
```

Listing 3 OCL Temporal Grammar

Metamodel

The metamodel extension for temporal concepts is illustrated in Fig. 9. This extension simply provides a new kind of navigation (or operation) call expression that can be semantically interpreted appropriately. In addition, it provides abstract syntax for trace literals. Notice that in contrast to [10], we do not have to use stereotypes and tagged values, but can construct a complete metamodel. This has some advantages. In particular, it allows us to create concepts that are not derived from concepts already existing in the standard language. In general, creating full metamodels is more powerful than using stereotypes and tagged values.

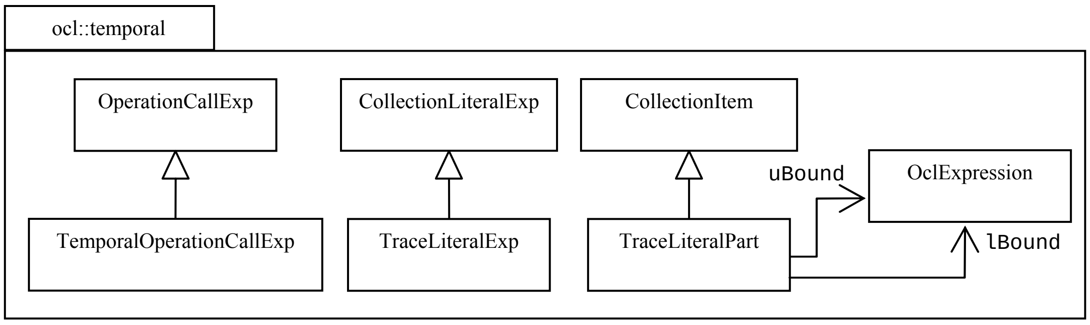

Fig. 9 Temporal OCL Metamodel extensions

Abstraction / Binding

The transformation-rule based approach to specifying the mappings from syntax tree to metamodel means that we can simply provide a new sub-rule type for the abstract rule that handles the different kinds of navigation. In addition, we need to provide a rule for translating trace literals. The transformation rules extension is illustrated in Fig. 10.

[3] We do not show the concrete syntax for trace literals, as it has not been discussed in [10] either.

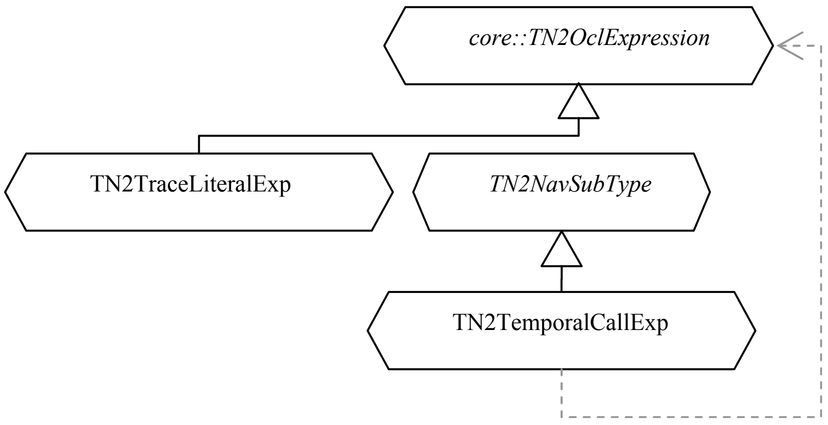

Fig. 10 Transformation Rules for ocl::temporal module

New Operation for Sequence

The extension additionally introduces a new operation for `Sequence`. This can be very easily provided by exchanging the `Sequence` type of the standard library by a new sub-type providing this additional operation. In Sect. 3.2 we have discussed how we decouple the standard library from the definition of the language. Here, we provide a new interpretation for sequences by using these mechanisms.

5 Tooling and Future Work

To evaluate our approach of modularising OCL, we have built a prototypical implementation of parsing, abstraction, and binding for OCL language modules. This current implementation uses bespoke bottom-up parsing technology for generating modular parsers for the language modules. It can be obtained upon request by email from the first author. The exemplar grammars shown above can be used directly with these parser generators. Abstraction and binding has been implemented as a transformation from the abstract syntax tree to a metamodel instance using SiTra [2] transformation rules. We have experimented with various implementations for specialising abstraction and binding, but have eventually arrived at the solution presented here because it is the simplest. The OCL and UML metamodels have been represented as KMF model repositories [13].

Basing our prototypic implementation on the bespoke parser generators allowed us to experiment quickly and efficiently with different modularisations of OCL as well as with different approaches to specialising abstraction and binding. However, some of the technology used in this experiment is quite non-standard. In particular, it uses bottom-up parsing with some additional features, makes no explicit distinction between token definitions and grammar rules, and has its own notion of grammar inheritance. Now that the module layout and rule-inheritance technology has stabilised, it will be interesting to see how these ideas can be implemented based on more standard technology. We are, therefore, planning to re-implement the Dresden OCL2 Toolkit [9] and the Kent OCL Library [17] in a modular fashion, but based on their respective technology. We would also like to encourage other providers of OCL parsing technology to adjust their tools accordingly. Eventually, this may even lead to a reformulation of the OCL 2 standard in a modular fashion.

As mentioned above, we have experimented with different ways of structuring the modules and with different ways of specialising abstraction and binding. Although we have come to a simple solution in this paper, some of the ideas we came across still deserve some further thought. For example, we have been thinking about an even more generic approach to the core OCL definition. In this approach, every OCL expression would be a navigation expression and every navigation expression would follow one of two patterns:

```
navExpr = basicNavExpr
        | prefixNavExpr
        ;
basicNavExpr = expr [ nameSpaceID ] NAME [ parameters ] ;
prefixNavExpr = NAME expr [ parameters ] ;
```

Listing 4 A different approach to defining navigation expressions

The first pattern captures the more common navigation expressions using the `.`, `->`, or `@` name spaces. Additionally, it also captures infix operators such as `and`, `or`, and (with a somewhat widened definition of `NAME`) `+` or `-`. Infix operators can be supported by using an empty name space identifier (`nameSpaceID`). The second type of navigation expression captures prefix expressions and the more convoluted `if __ then __ else __ endif` expression. All these navigation expressions have in common that there is a source expression (`expr` in the grammar) and a navigation modified by some name space identifier (`nameSpaceID` for `basicNavExpr`, for `prefixNavExpr` the system internally uses a constant `'prefixNameSpace'` to identify the name space). What name spaces are supported, and what navigation targets (identified by the `NAME` and, optionally, the parameters) are available, depends on the type of the source expression. Instead of providing generic transformation rules for abstraction and binding, language modules would then register specific navigation name spaces for specific types defining the available navigations and their translations for each type. The respective benefits and drawbacks of these different methods still need to be studied. One major benefit would be a unification of operation calls and operator use into the schema of navigation expressions. An issue to be solved is how to express precedence between operators.

So far, in this paper, we have only discussed a modularisation of OCL with respect to concrete and abstract syntax and the translation between the two. As already indicated in Fig. 1 we also need to modularise semantics. There are two approaches to semantics in the OCL standard: 1) the normative semantics based on a metamodel of values and an explicit mapping between expressions and values expressed using OCL constraints, again, and 2) a mathematical semantics based on naïve set theory. For both cases, we will also need to develop modularised versions of the semantic domain as well as modularised approaches for the relation between abstract syntax and semantic domain. In the first case, this should be quite straight-forward and very similar to the technologies used in this paper for the abstract syntax and the mapping from concrete syntax to abstract syntax. Modularisation at the level of semantics will be an important task for future research.

6 Conclusion

This paper has shown a working modular and extensible specification of the syntax of the Object Constraint Language (OCL). The extensibility of this specification has been validated by providing the definitions of a temporal-logic extension of OCL, in addition to using the extension mechanisms within the definition of the standard OCL itself. The paper reuses existing parsing technology from the University of Kent and could probably be reimplemented based on other parsing technology. Hence, the main contributions are the synchronisation of parsing and instantiation of abstract syntax in a modular language definition, and the concrete language modules defined for OCL.

This approach to defining the OCL can reduce the learning effort of newcomers to the language. This is possible because one can start with a reduced subset of OCL and include additional language modules as required and as understanding of the core concepts evolves.

Furthermore, our modular definition of OCL and the extensibility mechanism incorporated provide a clean approach to unifying the multiple OCL extensions that are proposed within the modelling community. In recent years, many extensions to OCL have been proposed, and OCL is reused as the basis for new languages. Examples are the QVT [20] and other transformation languages [14], CQML [1] for specifying non-functional properties, OCL-based template languages [6], and many more.

Because of these benefits, we propose that in the future such extensions and reuse of OCL should be defined in the manner presented in this paper.

References

1. Aagedal, J.: *Quality of Service Support in Development of Distributed Systems.* Dissertation, University of Oslo, Norway, 2001.
2. Akehurst, D.H., Bordbar, B., Evans, M., Howells, W.G., McDonald-Maier, K.D.: *SiTra: Simple Transformations in Java.* ACM/IEEE 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS), Genova, Italy, October 2006.
3. Akehurst, D.H., Howells, W.G., McDonald-Maier, K.D.: *Kent Model Transformation Language.* Proc. Workshop on Model Transformations in Practice, held at the MoDELS conference, Montego Bay, Jamaica, October 2005.
4. Akehurst, D.H., Howells, G., McDonald-Maier, K.D.: *Supporting OCL as part of a Family of Languages. In* Proc. of the MoDELS'05 Conference Workshop on Tool Support for OCL and Related Formalisms - Needs and Trends, Montego Bay, Jamaica, EPFL, October 2005.
5. Akehurst, D.H., Linington, P.F., Patrascoiu, O.: *OCL 2.0: Implementing the Standard.* Technical Report, University of Kent at Canterbury, 12-03, November 2003.

6. Akehurst, D.H., Patrascoiu, O.: *Tooling Metamodels with Patterns and OCL.* Proc. Workshop on Metamodelling for MDA, York, November 2003.
7. Conrad, S., Turowski, K.: *Temporal OCL: Meeting Specifications Demands for Business Components.* In Siau, K., Halpin, T. (eds.): Unified Modeling Language: Systems Analysis, Design, and Development Issues. Pages 151–156, IDEA Group publishing, 2001.
8. Copeland, T.: *Generating Parsers with JavaCC.* Centennial Books, 2007.
9. Demuth, B., Löcher, S., Zschaler, S.: *Structure of the Dresden OCL toolkit.* In Proc. 2nd International Fujaba Days "MDA with UML and Rule-based Object Manipulation". Technical Report, Technical University of Darmstadt, Germany, September 2004.
10. Flake, S., Mueller, W.: *An OCL Extension for Real-Time Constraints.* In Clark, T., Warmer, J. (eds.): Object Modeling with the OCL: The Rationale behind the Object Constraint Language. Pages 150–171, LNCS **2263**, Springer, 2002.
11. Hedin, G.: *Reference Attributed Grammars.* In Parigot, D., Mernik, M. (eds.): Proc. 2nd Workshop on Attribute Grammars and their Applications (WAGA 1999). Pages 153–172, INRIA Rocquencourt France, 1999.
12. Klint, P., Lämmel, R., Verhoef, C.: *Toward an engineering discipline for grammarware.* ACM Transactions on Software Engineering and Methodology (TOSEM) **14:**331-380, 2005. http://doi.acm.org/10.1145/1072997.1073000
13. KMF development team: *Kent Modelling Framework (KMF).* http://www.cs.kent.ac.uk/projects/kmf
14. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: *Merging Models with the Epsilon Merging Language (EML).* In Proc. ACM/IEEE 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS). Springer, 2006.
15. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: *Towards Using OCL for Instance-Level Queries in Domain Specific Languages.* Proc. Workshop on OCL for (Meta-) Models in Multiple Application Domains, held at MoDELS 2006. Pages 26–37, Technical Report Technische Universität Dresden, 2006.
16. Kozen, D.C.: *Automata and Computability.* Springer, 1997.
17. Kent OCL tools development team: *Kent OCL library.* http://www.cs.kent.ac.uk/projects/ocl.
18. Object Management Group: *Response to the UML 2.0 OCL Rfp (ad/2000-09-03), Revised Submission, Version 1.6.* Object Management Group, ad/2003-01-07, January 2003.
19. Object Management Group: *UML 2.0 Infrastructure Specification.* Object Management Group, ptc/03-09-15, September 2003.
20. Object Management Group: *MOF QVT Final Adopted Specification.* Object Management Group, pct/05-11-01, November 2005.
21. Parr, T.: *The Definitive Antlr Reference: Building Domain-Specific Languages.* Pragmatic Bookshelf, 2007.
22. Pinet, F., Kang M.-A., Vigier, F.: *Spatial Constraint Modelling with a GIS Extension of UML and OCL: Application to Agricultural Information Systems.* In Proc. International Symposium on Metainformatics (MIS 2004), Revised Selected Papers, LNCS **3511,** Springer, 2005.
23. Ziemann, P., Gogolla, M.: *OCL Extended with Temporal Logic.* In Broy, M., Zamulin, A., editors, 5th Int. Conf. Perspectives of System Informatics (PSI'2003), pages 351–357. Springer, Berlin, LNCS 2890, 2003.