An OCL-based bridge from concrete to abstract syntax

Adolfo Sánchez-Barbudo Herrera[1], Edward Willink[2], Richard F. Paige[1]

[1] Department of Computer Science, University of York, UK.
{asbh500, richard.paige}_at_york.ac.uk
[2] Willink Transformations Ltd. ed_at_willink.me.uk

Abstract. The problem of converting human readable programming languages into executable machine representations is an old one. EBNF and Attribute grammars provide solutions, but unfortunately they have failed to contribute effectively to model-based Object Management Group (OMG) specifications. Consequently the OCL and QVT specifications provide examples of specifications with significant errors and omissions. We describe an OCL-based internal domain specific language (DSL) with which we can re-formulate the problematic parts of the specifications as complete, checkable, re-useable models.

1 Introduction

The Object Management Group (OMG) is a consortium whose members produce open technology standards. Some of these target the Model-Driven Engineering (MDE) community. OMG provides the specifications for languages such as UML [1], MOF [2], OCL [3] and QVT [4].

The specifications for textual languages such as OCL and QVT define a textual language and an information model using

- an EBNF grammar to define the textual language
- a UML metamodel to define the abstract syntax (AS) of the language

The textual language is suitable for users and for source interchange between compliant tools. The information model facilitates model interchange between producing tools such as editors or compilers and consuming tools such as program checkers or evaluators.

The conversion between these two representations must also be specified and may make use of an additional intermediate concrete syntax (CS) metamodel whose elements correspond to the productions and terminals of the grammar[3].

[3] Modern language workbenches, such as Xtext, can automatically generate the CS metamodel from their input grammars

1.1 The OMG specification problem

The OCL [3] and QVT [4] specifications define four languages, OCL, QVTc (Core), QVTo (Operational Mappings), and QVTr (Relations). The specifications all provide fairly detailed grammars and metamodels of their respective abstract syntaxes.

Unfortunately the grammar to AS conversion is poorly specified.

In OCL, a CS is provided and the grammar is partitioned into ambiguous productions for each CS element. Semi-formal rules define the grammar to CS correspondence, the CS to AS correspondence, name resolution and disambiguation.

QVTr has a single coherent grammar to accompany its CS and similar semi-formal rules.

QVTc has a single grammar but no CS and no semi-formal rules.

QVTo similarly has a single grammar, but no CS and no semi-formal rules. Instead, notation sections suggest a correspondence between source text and AS elements by way of examples.

Since none of the conversions are modeled, tools cannot be used to check the many details in the specifications. As a result, the major omissions identified above are augmented by more subtle oversights and inaccuracies. The specifications fail to provide the complete, consistent and accurate details to help tool vendors to provide compliant implementations of the text to AS conversions.

1.2 Our solution

The intermediate CS metamodel is close to the grammar and this gap is now bridged quite satisfactorily by modern Annotated EBNF tooling such as Xtext. It is in the CS to AS conversion that greater challenges arise.

In this paper, we take inspiration from the substantial semi-formal exposition of the OCL conversions (Clause 9.3 of [3]) and introduce a fully modeled CS2AS bridge. The models can be used to variously check and even auto-generate a consistent specification and also to auto-generate compliant tooling. In addition to conventional CS and AS metamodels, we introduce new CS2AS mapping models, name resolution models and CS disambiguation models. We demonstrate how OCL itself can be used to provide a suitable internal DSL for these new models.

The paper is structured as follows. Section 2 presents an example to introduce the grammar and metamodels. Section 3 demonstrates the semi-formal solution adopted by the OCL specification. Section 4 explains the proposed solution, i.e. an OCL-based internal DSL. Section 5 describes related work and Section 6 talks about the current shortcomings of the approach. Section 7 outlines some future work, including how tool implementors can benefit from the internal DSL. Finally, Section 8 concludes.

2 Example

Our first example is a collection literal expression. This provides a simple example of the grammars and models in use. In Section 3 we show the semi-formal usage of these concepts by the OCL specification. In Section 4 we provide a contrast with our fully-modeled internal DSL solution. This example is too simple to demonstrate more than the CS2AS characteristics of our solution. We therefore introduce a further more relevant example later.

The listing in Figure 1 is an example of a *collection literal expression* comprising three comma-separated collection literal parts. The adjacent diagram shows the corresponding AS metamodel elements. *CollectionLiteralExp* contains many abstract *CollectionLiteralPart*s. *CollectionItem* and *CollectionRange* are derived to support the two cases of a single value or a two-ended integer range. The example text must be converted to instances of the AS metamodel elements.

```
1  Sequence{1, 1+1, 3..9+1}
2
3  -- equivalent to:
4
5  -- Sequence{1,2,3,4,5,
6  --     6,7,8,9,10}
```

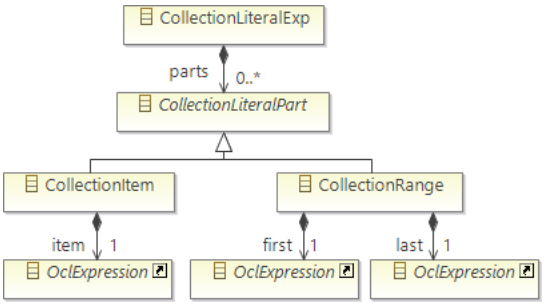

Fig. 1: CollectionLiteralPart Example and partial AS Metamodel

The listing in Figure 2 shows the EBNF grammar that parses a *collection literal part* as a *CollectionLiteralPartCS* comprising one direct *OclExpressionCS* or a *CollectionRangeCS* comprising two *OclExpressionCS*s. The adjacent diagram shows the intermediate CS model, which is similar to the AS but which omits a 'redundant' *CollectionItemCS* preferring to share a single/first expression from the non-abstract *CollectionLiteralPartCS*.

```
1  CollectionLiteralPartCS:
2    OclExpressionCS | CollectionRangeCS
3
4  CollectionRangeCS:
5    OclExpressionCS '..' OclExpressionCS
```

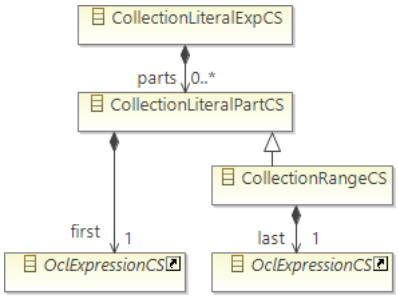

Fig. 2: CollectionLiteralPartCS Grammar and partial CS Metamodel

3 Semi-formal solution: OCL Clause 9.3

The OCL specification provides a full attribute grammar in which inherited and synthesized attributes are used to describe how the AS is computed from the CS. Figures 3 and 4 shows our first example. The specification uses OCL expressions to express how the different attributes are computed.

9.3.13 CollectionLiteralPartCS

[A] CollectionLiteralPartCS ::= CollectionRangeCS

[B] CollectionLiteralPartCS ::= OclExpressionCS

Abstract syntax mapping

CollectionLiteralPartCS.ast : CollectionLiteralPart

Synthesized attributes

[A] CollectionLiteralPartCS.ast = CollectionRange.ast
[B] CollectionLiteralPartCS.ast.oclIsKindOf(CollectionItem) and
CollectionLiteralPartCS.ast.oclAsType(CollectionItem).OclExpression = OclExpressionCS.ast

Inherited attributes

[A] CollectionRangeCS.env = CollectionLiteralPartCS.env
[B] OclExpressionCS.env = CollectionLiteralPartCS.env

Disambiguating rules

-- none

Fig. 3: OCL specification for CollectionLiteralPartCS to CollectionLiteralPart

The first section defines the EBNF production(s). The example merges two alternate productions and so many of the rules have an `[A]` or `[B]` prefix to accommodate the alternative rules.

The AS mapping declares the type of the resulting AS element as the type of a special property of the CS element: *ast*.

The synthesized attributes populate the AS element using an assignment for `[A]`. The more complex `[B]` worksaround OCL 2.4's inability to construct a *CollectionItem* by imposing constraints on a hypothetical *CollectionItem*.

The inherited attributes contribute to the name resolution by flowing down an *Environment* hierachy of all available name-element pairs from parent to child nodes using another special CS property: *env*. In this case all names visible in the parent are passed without modification to the children.

The disambiguating rules provide guidance on the resolution of ambiguities. In this simple example, there is no ambiguity.

The rules for collection range follow a similar pattern. There is now just one grammar production whose two *OclExpression*s are distinguished by `[1]` and `[2]` suffixes. The synthesized attributes have two properties to populate.

3.1 Critique

The presentation comes quite close to specifying what is needed, but uses an intuitive mix of five sub-languages without any tool assistance. In Figure 3, the typo whereby *CollectionItem::OclExpression* rather than *CollectionItem::item* is

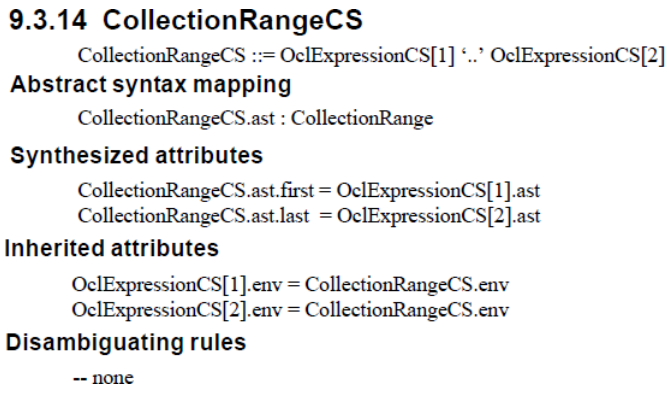

Fig. 4: OCL specification for CollectionRangeCS to CollectionRange

used in the final line of the synthesized attributes has gone unreported for over 10 years.

The lack of tooling also obscures the modeling challenge for the inheritances between *CollectionLiteralPartCS*, *CollectionRangeCS* and *OclExpressionCS*. The `[B]` grammar production in Figure 3 requires *OclExpressionCS* to inherit from *CollectionLiteralPartCS*, if *CollectionLiteralPartCS* is to be the polymorphic type of any collection literal part in the CS.

The lack of any underlying models makes it impossible for tool vendors to re-use the rules. Tool vendors must transcribe and risk introducing further errors.

4 Modeled Solution: CS2AS internal DSL

The critique of the semi-formal exposition highlights the lack of checkable or re-useable models. In this section we formalize the semi-formal approach using a DSL to declare the bridge between the CS and the AS of a language. The DSL is internal[5] and uses only facilities proposed for OCL 2.5. The DSL constrains the use of the general purpose OCL language to define a set of idioms that express CS2AS bridges.

Our rationale for choosing OCL as the host language is as follows:

- OMG specifications have a problem with bridging the CS to AS gap, so we would like an OMG-based solution.
- OCL contains a rich expression language which can provide enough flexibility to express non trivial CS2AS bridges in a completely declarative way.
- Other OMG related languages could be considered (such as one of the QVT languages), however OCL is a well known OMG language and is the basis of many others. A QVT practitioner inherently knows OCL but not vice-versa.

Instances of the internal DSL take the form of Complete OCL documents and can be maintained using Complete OCL tools [6]. Multiple documents can be used to partition the specification into modules to separate the distinct mapping, name-resolution, and disambiguation concerns of the CS2AS bridge.

4.1 Shadow Object Construction

The internal DSL uses the proposed[4] side-effect-free solution to the problem of constructing types in OCL. This avoids the need for the hypothetical objects used by the semi-formal approach. The proposed syntax re-uses the existing syntax for constructing a Tuple. The *Tuple* keyword is replaced by the name of the type to be constructed. A `Complex` number with `x` and `y` parts might therefore be constructed as `Complex{x=1.0,y=2.0}`.

4.2 CS2AS mappings

In this subsection we explain the main CS2AS mappings description language. We start by introducing an instance of the language so that the reader can have an indication of the DSL used to describe the bridge. Listing 1.1 corresponds to the CS2AS description of the OCL constructs introduced in Section 2. The listing should be contrasted with the semi-formal equivalent in Figures 3 and 4.

```
1  context CollectionLiteralPartCS
2  def : ast() : ocl::CollectionLiteralPart =
3   ocl::CollectionItem {
4        item = first.ast(),
5        type = first.ast().type
6       }
7
8  context CollectionRangeCS
9  def : ast() : ocl::CollectionRange =
10  ocl::CollectionRange {
11       first = first.ast(),
12       last = last.ast(),
13       type = first.ast().type.commonType(last.ast().type)
14      }
```

Listing 1.1: CS2AS bridge for CollectionLiteralPart and CollectionRange

The mapping is described by defining the *ast()* operation on a CS element. The 'abstract syntax mapping' and 'synthesized attributes' of the semi-formal approach are modeled by the shadow construction of the appropriate AS type and initialization of its properties. (The initialization includes the *type* property omitted by the OCL specification.)

Declarativeness: An important characteristic of the DSL is that it comprises declarative OCL constraints. The OCL constraints specify only true correspondences between AS and CS after a valid conversion. In a scenario of executing the proposed CS2AS descriptions, discovery of a suitable order in which to perform CS to AS conversions requires an implementing tool to analyze the OCL constraints and exploit their inter-dependencies. (This was also the unstated policy of the semi-formal approach.) An automated analysis is desirable since they are almost too complicated for an accurate manual formulation as a multi-pass conversion.

Operations: The CS2AS bridge is described using operation definitions. The underlying rationale is that operation definitions on a potentially complex class hierarchy of the CS can be overridden. Due to this overriding mechanism,

[4] Shadow object construction was called type construction in the Aachen report [7]

we provide some flexibility to cope with language extensions such as QVT. The operation name is not relevant, but we propose the name *"ast"* since it is aligned with the name used in the attribute grammar exposed in the OCL specification.

Shadow object construction: Shadow object constructions express how AS elements are constructed and how their properties are initialized.

Operation Calls: To compute properties of any AS element, we need to access the AS elements to determine a CS to AS correspondence. Since *ast()* is a side-effect-free query, we may call *ast()* as many times as necessary to obtain the appropriate AS element. For example, at line 4, in order to initialize the *CollectionItem::item* property, we use the *ast()* to obtain the *OclExpression* corresponding to the *first OclExpressionCS* of the context *CollectionLiteralPartCS*.

Self-contained: With the goal in mind of using the proposed internal DSL to rewrite part of the OMG specifications, the declaration of the CS2AS bridge for a particular CS element is complete and self-contained. The computations for all non-default-valued properties of the corresponding AS element are expressed directly in the shadow type expression since there is no constructor to share inherited computations.

Reusable computations: Having OCL as the host language for our internal DSL, we can factor out and define more complex and reusable expressions in new operation definitions. The operations can be reused, by just introducing operation call expressions, across the different computations of the AS element properties. At line 11, *commonType* is a reusable operation to compute the common supertype of source and argument types.

4.3 Name resolution description

In this subsection, we explain how name resolution is described when defining CS2AS bridges by the means of our OCL-based internal DSL. In a name resolution activity we can typically find two main roles:

- a producer provides a name-to-element map for all possible elements in its producing scope.
- a consumer looks up a specific element corresponding to a name in its consuming context

Our previous example had no need to resolve names, so we will now introduce a new example with a name producer and a consumer. The listing in Figure 5 is an example of a *let expression* that declares and initializes a variable named *var* for use within the *'in'* of the *let expression*. In this example the *'in'* comprises just a *variable expression* that references *var*. The adjacent diagram shows the corresponding AS metamodel elements. A *LetExp* contains the produced *Variable* and an arbitrary *OclExpression 'in'*. For our simple example the *'in'* is just a *VariableExp*. The complexity of the example lies in the initialization of the consuming *VariableExp.referred Variable* to reference the producing *LetExp.variable*.

```
1  let var : String = 'something'
2  in var
```

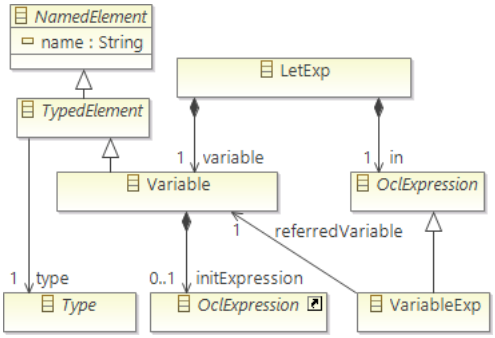

Fig. 5: LetExp/VariableExp Example and partial AS Metamodel

Figure 6 shows the corresponding grammar and CS definitions[5]. A *LetExpCS* contains a *VariableDeclarationCS* and *OclExpressionCS* which for our example is just a *VariableExpCS*.

```
1   LetExpCS:
2     'let' VariableDeclarationCS
3     'in' OclExpressionCS
4
5   VariableDeclarationCS:
6     simpleName (':' TypeCS)?
7     ('=' OclExpressionCS)?
8
9   VariableExpCS:
10    simpleName | 'self'
```

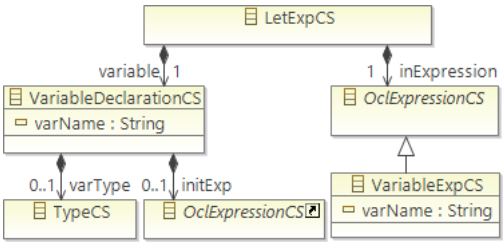

Fig. 6: LetExpCS/VariableExpCS Grammar and partial CS Metamodel

In typical programming languages every use of a variable has a corresponding declaration. The variable declaration is the producer of a name-to-variable mapping. The variable usage consumes the variable by referencing its name. Name resolution searches the hierarchy of producing contexts that surround the consuming context to locate a name-element mapping for the required name.

In our example, the required cross-reference in the AS is represented in the CS by the distinct *VariableDeclarationCS.varName* and *VariableExpCS.varName* properties. These are both parsed with the value *var* and so, when consumption of the *VariableExpCS.varName* is analyzed, the analysis must discover the corresponding *VariableDeclarationCS.varName* production.

The semi-formal approach adopted by the OCL specification re-uses the containment hierarchy of the CS as the scope hierarchy for its 'inherited attributes'. The name-to-element mappings are maintained in an *Environment* hierarchy. The mappings flow down from the root CS element to all the leaf elements which accumulate additional name-to-element mappings and/or nested environments at each intermediate CS element in the CS tree.

[5] The complexity of multi comma-separated variables has been removed, because it is not needed to explain how name resolution is described in our interal DSL

In Section 3 we saw the very simple unmodified flow-down for a *CollectionLiteralPart*. The equivalent exposition for a *LetExp* in the OCL specification is complicated by performing the CS2AS mapping of multiple comma-separated let-variables with respect to the CS rather than the AS. We therefore present its logical equivalent in Listing 1.2.

```
1  LetExpCS ::= let VariableDeclarationCS in OclExpressionCS
2
3  VariableDeclarationCS.env = LetExpCS.env
4  OclExpressionCS.env = LetExpCS.env.nestedEnvironment().addElement(
       VariableDeclarationCS.ast)
```

Listing 1.2: Semi-formal LetExpCS equivalent

The environment of the LetExpCS is passed unchanged to the VariableDeclarationCS so that name resolution within the VariableDeclarationCS initializer sees the same names as the LetExpCS.

The environment for the OclExpressionCS is more interesting. A nested Environment is created containing the name-to-variable mapping for the let-variable. The use of a nested environment ensures that the let-variable name occludes any same-named mapping in the surrounding environment.

Our modeled approach is very similar but re-uses the AS tree rather than the CS tree as the scope hierarchy. The rationale is that we are interested in looking up AS elements for which we might not have the corresponding CS (e.g OCL standard library or user model elements – classes, properties, operations, etc. –).

```
1  context OclAny
2  def : env : env::Environment
3    if oclContainer() <> null
4    then oclContainer().childEnv(self)
5    else env::Environment{}
6    endif
7
8  def : childEnv(child : OclAny) : env::Environment =
9    env
10
11 context LetExp
12 def : childEnv(child : OclAny) : env::Environment =
13   if child = variable
14   then env
15   else env.nestedEnv().addElement(variable)
16   endif
```

Listing 1.3: Name resolution producers

Listing 1.3 presents the name resolution description written in our OCL-based internal DSL. Line 2 declares an *env* property to hold the immutable *Environment* of the AS element. *env* is initialized by a containment tree descent that uses *oclContainer()*[6]. Line 5 provides an empty environment at the root, otherwise Line 4 uses *childEnv(child)* to request the parent to compute the child-specific environment.

The default definition of *childEnv(child)* on lines 8-9 flows down the prevailing environment to all its children. This can be inherited by the many AS elements that do not enhance the environment.

[6] oclContainer() returns the containing element which is null at the root.

The non-default override of *childEnv(child)* for LetExp on lines 12-16 uses the *child* argument to compute different environments for the *Variable* and *OclExpression* children. As we saw for the semi-formal approach, the environment for the *Variable* is unmodified. The environment for the *OclExpression* is extended by the addition of the variable in a nested environment.

The environment is exploited by consumers to satisfy their requirement to convert a textual name into the corresponding model element. The conversion comprises three steps

- locate all candidate elements
- apply a filtering predicate to select only the candidates of interest
- return the selected candidate or candidates

The first stage is performed by the environment propagation described above.

The filtering predicate invariably selects just those elements whose name matches a required name. It may often provide further discrimination such as only considering Variables, Properties or Namespaces. For operations, the predicate may also match argument and parameter lists.

The final return stage returns the one successfully selected candidate which is the only possibility for a well-formed conversion. For practical tools a lookup may fail to find a candidate or may find ambiguous candidates and provide helpful diagnostics to the user.

The specification is made more readable if the three stages are wrapped up in helper functions such as *lookupVariable* or *lookupProperty*[7].

List 1.4 shows the polymorphic *ast()* operation to map *VariableExpCS* to *VariableExp*. The *lookupVariable* helper function is used to discover the appropriate variable to be referenced by *referredVariable*.

```
1  context VariableExpCS
2  def : ast() : ocl::VariableExp =
3    let variable = ast().lookupVariable(varName)
4    in ocl::VariableExp {
5        name = varName,
6        referredVariable = variable,
7        type = if variable = null
8             then null
9             else variable.type
10            endif
11     }
```

Listing 1.4: CS2AS bridge for VariableExpCS to VariableExp

4.4 Disambiguation

As we commented in the introduction, CS disambiguation is another important concern which needs to be addressed during the CS2AS bridge. To explain the need of disambiguation rules, we consider the simple OCL expression `x.y`.

[7] A practical implementation may provide alternative helper implementations that exploit the symmetry of the declarative exposition to search up through the containment hierarchy examining only candidates that satisfy the filtering predicate. This avoids the costs of flowing complete environments down to every AS leaf element where at most one element of the environment is of interest.

At first glance, the *'y'* property of the *'x'* variable is accessed using a *property call expression* and a *variable expression*. However *'x'* is not necessarily a variable name. It could be that there is no *'x'* variable. Rather *'x'* may be a property of the implicit source variable, *self*, since the original expression could be a short form for *self.x.y*. Semantic resolution is required to disambiguate the alternatives and arbitrate any conflict.

The OCL specification provides disambiguation rules to 'resolve' grammar ambiguities. Clause 9.1 states : "Some of the production rules are syntactically ambiguous. For such productions disambiguating rules have been defined. Using these rules, each production and thus the complete grammar becomes nonambiguous.". Figure 7 and Figure 8 are extracted from the OCL specification. It can be seen that a *simpleNameCS* with no following `@pre` matches the `[A]` production of a *VariableExpCS* and the `[B]` production of a *PropertyCallExpCS*.

9.3.3 VariableExpCS

[A] VariableExpCS ::= simpleNameCS

Disambiguating rules

[1][A] simpleNameCS must be a name of a visible VariableDeclaration in the current env

env.lookup (simpleNameCS.ast).referredElement.oclIsKindOf (VariableDeclaration)

Fig. 7: Partial OCL Specification for VariableExpCS to VariableExp

9.3.36 PropertyCallExpCS

[B] PropertyCallExpCS ::= simpleNameCS isMarkedPreCS?

Disambiguating rules

[1] [A, B] 'simpleName' is name of a Property of the type of source or
if source is empty the name of an attribute of 'self' or
any of the iterator variables in (nested) scope. In OCL:
not PropertyCallExpCS.ast.referredAttribute.oclIsUndefined()

Fig. 8: Partial OCL Specification for PropertyCallExpCS to PropertyCallExp

The disambiguation rule for *VariableExpCS* is relatively simple delegating to the *lookup* helper and imposing a constraint that the result must be a *VariableDeclaration*. This is potentially correct, although unfortunately the specification that *VariableDeclaration* is the supertype of *Variable* and *Parameter* is missing.

The disambiguation rule for *PropertyCallExpCS* has some ambiguous wording and many details that do not correspond to the "In OCL". This requires intuition by the implementor who may also wish to consider how the rules apply to implicit opposite properties in EMOF.

Both of these disambiguation rules require semantic information which is not available when the syntactic parser requires it. The problem can be avoided by unifying the ambiguous alternatives as unambiguous productions that can be

parsed to create a unified CS tree. Once parsing has completed, semantic analysis of the unified CS can resolve the unified CS elements into their disambiguated forms.

We therefore introduce additional unifying CS elements that can be resolved without semantic information. A unifying *NameExpCS* element replaces the ambiguous *PropertyCallExpCS* and *VariableExpCS*. Figure 9 shows the unified CS definitions.

```
1  NameExpCS:
2   simpleName isMarkedPreCS?
```

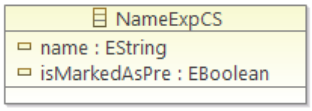

Fig. 9: NameExpCS Grammar and partial CS Metamodel

Listing 1.5 shows the definition for the CS2AS mapping of a *NameExpCS*, in which the *isAVariableExp()* at line 3 is a call of the operation providing the disambiguation rule. The return selects whether a *NameExpCS* is mapped to a *VariableExp* (lines 5-13), otherwise a *PropertyCallExp* (lines 15-23).

```
1  context NameExpCS
2  def : ast() : ocl::OclExpression =
3   if isAVariableExp()
4   then
5    let variable = ast().lookupVariable(self)
6    in oclAS::VariableExp {
7       name = name,
8       referredVariable = variable,
9       type = if variable = null
10           then null
11           else variable.type
12           endif
13   }
14  else
15   let property = ast().lookupProperty(self)
16   in oclAS::PropertyCallExp {
17      name = name,
18      referredProperty = property,
19      type = if property = null
20           then null
21           else property.type
22           endif
23   }
24  endif
```

Listing 1.5: CS2AS description for an ambiguous name expression

This approach has the benefit of localizing the disambiguation in the *isAVariableExp()* operation and associated control logic and so making *VariableExpCS* and *PropertyCallExpCS* redundant. The simple two-way disambiguation decision is shown in 1.6

```
1  context NameExpCS
2  def : isAVariableExp() : Boolean =
3   let variable = ast().lookupVariable(self),
4   in variable <> null
```

Listing 1.6: NameExpCS disambigutation rule

Simple choices such as the various forms of *CollectionLiteralPartCS* can be resolved syntactically. Semantic decisions are required for the unified name example above. The conflicts between the use of parentheses for template arguments, operation calls and iteration calls can be resolved in the same way but with a more complex semantic decision tree.

5 Related work

In this section we briefly discuss how the proposed OCL-based CS2AS bridge relates to previous work. To the best of our knowledge there does not exist a DSL approach based on OMG specifications to describe bridges between CS and AS. The Complete OCL document based approach was introduced in [8] and this paper aims to explain the whole approach (i.e. the internal DSL). Recently, [9] has been proposed as a functional transformation language to tackle model transformations. Apart from be too novel to be considered in this work, OCLT is not domain specific and it needs additional constructs (e.g. pattern matching) in order to cover more complex transformation scenarios.

We can find languages conceived to sort out the CS2AS bridges in other contexts, i.e in the context of some specific tools. We highlight two of them:

NaBL [10] & Stratego [11]: These are two separate languages for different purposes used by the Spoofax language workbench [12]. The former is used to declare name resolution and the latter to declare syntax rewrites (tree based structure transformations). As a main difference with respect to our approach, these languages are completely unrelated: whereas the former is integrated during the parsing activities in order to resolve cross-references when producing the CS tree, the latter is a general purpose program transformation language further used to obtain the potentially different AS tree. In our approach, we integrate the name resolution language into a further CS2AS activity, provided that the parsing activity first produces a CS tree. As it was commented in Section 4.3, the name lookups are performed on AS elements rather than on CS ones.

Gra2Mol [13]: Gra2Mol is an approach that is closer in objective to the approach presented in this paper. It is a domain specific transformation language conceived to define those bridges, and as our approach does, the name resolution activity is also declared as part of the transformation language. However, whilst their name resolution relies on explicitly specifying a direct search (thus, the name consumer needs to know where the name producer is located in the syntax tree), our approach for specifying name resolution is more declarative based on an independent declaration of name producers and consumer (thus, the name consumer doesn't need to know where the producer is located in the syntax tree). Another difference is that whilst we use OCL as the expression language to express the bridges, they define a structure-shy[8] query language instead. They claim that the usage of their query language is more compact and less verbose when compared to using OCL expressions. However such languages are not suitable from the point of view of OMG specifications. Besides, we can add that

[8] Xpath is an example of this kind of language

structure-shy languages are more error prone or sensitive to changes in the involved metamodels (metamodel evolution): when having a static typed language such OCL, supporting tools can better assist with metamodel evolution.

6 Limitations and shortcomings

From the point of view of the OMG specification, we do not see any limitations of the proposed internal DSL. Having OCL as the host language is a good solution for OMG specifications, because the instances of the DSL can be directly ported to those specifications in order to precisely define the corresponding CS2AS bridges. Likewise, the flexibility and modularity that Complete OCL documents provide has promise in addressing very large CS2AS gaps.

On the other hand, from the final user point of view, i.e the user of the DSL, and specially when comparing with related work, we perceive that having an external DSL fully designed to deal with concepts related to name resolution (e.g. NaBL) or disambiguation may be more convenient. We discuss this further in the next section when talking about future work.

Another shortcoming to mention is that the DSL is based on the concept of shadow type expression, which is not yet part of the OCL specification, although it is planned to be included in the next OCL version (2.5) [7][9]. The number of OCL tools which can currently be used to validate the CS2AS bridges is therefore limited (we are using Eclipse OCL[6] which prototypes some proposed OCL 2.5 features).

7 Ongoing and future work

Apart from using this OCL-based internal DSL to define CS2AS bridges, we are also producing the Java based source code responsible for obtaining AS models from CS ones. This ongoing work follows the line drawn in the introduction which highlights that the CS2AS internal DSL can be exploited by tool implementers. Although in this paper we are unable to go into further detail, we can point the reader out to some JUnit test cases[10] working on small examples, which demonstrate that the instances of the CS2AS internal DSL can be transformed to executable code and perform the CS2AS gap resolution of a language.

In terms of future work, we highlight the following.

- **Definition of CS2AS bridges for OCL and QVT.** We will apply the proposed OCL-based internal DSL to provide complete CS2AS bridge descriptions for the whole OCL and the three QVT languages. We expect these CS2AS bridge specifications to be included as part of the future OCL and QVT specifications. Likewise, we expect auto-generated code from from these

[9] It is cited in the report as type construction expression, Section 3.1

[10] http://git.eclipse.org/c/mmt/org.eclipse.qvtd.git/tree/tests/org.eclipse.qvtd.cs2as.compiler.tests/src/org/eclipse/qvtd/cs2as/compiler/tests/OCL2QVTiTestCases.java

bridge specifications to be used in future releases of the Eclipse OCL and QVTd projects. This should elimnate errors attributable to hand-written conversion source code.
- **Incremental CS2AS bridges.** Since generation of code from the declarative CS2AS bridges requires a detailed dependency analysis to identify a valid conversion schedule, we plan to exploit this analysis to synthesize incremental code for use in interactive contexts such as OCL editors. This should improve accuracy and performance dramatically since accurate efficient incremental code is particularly hard to write manually and pessimistic simplifications to improve accuracy are not always sound.
- **Creation of an external DSL.** By bringing together the good aspects of other related languages such as NaBL or Gra2Mol, we plan to create an external DSL and with a higher level of abstraction and more concise than the one presented here, to ease even more the creation of those bridges. This external DSL can embed the OCL expressions language, and the supporting tooling can include a code generator to modularly produce the instances of the internal DSL presented in this paper.
- **Integration with existing language workbenches.** As added value of the DSL and to provide more proofs about how tool vendors may benefit from it (not covered in this paper), we want to exploit the proposed DSL in the context of a modern language workbench called Xtext.

8 Conclusions

We have introduced a Concrete Syntax to Abstract Syntax bridge that is:

- **Sound**. We have shown how intuitive aspects of the current OCL specification are formalized by OCL definitions and faults corrected.
- **Executable**. We can use the dependencies behind the OCL definitions to establish an execution schedule.
- **Extensible**. We can reuse the formalization of the OCL bridge in a QVT bridge.

Our bridge modularizes and separates the specification concerns:

- **Mapping**. An OCL operation hierarchy maps CS artifacts to the AS.
- **Name Resolution**. An OCL operation hierarchy flows the visible names down to the point of access.
- **Disambiguation**. Unified CS elements, plus CS disambiguation rules, avoid the need for semantic resolution within a syntactic parser.

Our bridge is currently ready-to-go; it works on test examples. It will now be applied to replace manual tooling in Eclipse OCL and QVT by tooling generated direct from the potential OCL 2.5 specification models.

Acknowledgement. We gratefully acknowledge the support of the UK Engineering and Physical Sciences Research Council, via the LSCITS initiative,

References

1. Object Management Group. Unified Modeling Language (UML), V2.5. OMG Document: ptc/2012-10-24 (http://www.omg.org/spec/UML/2.5), 2012.
2. Object Management Group. Meta Object Facility (MOF) Core Specification, V2.4.1. OMG Document: formal/2013-06-01 (http://www.omg.org/spec/MOF/2.4.1), 2013.
3. Object Management Group. Object Constraint Language (OCL), V2.4. OMG Document: ptc/2013-08-13 (http://www.omg.org/spec/OCL/2.4), January 2013.
4. Object Management Group. Meta Object Facility (MOF) 2.0 Query/View/-Transformation V1.2. OMG Document: ptc/2014-03-38 (http://www.omg.org/spec/QVT/1.2), May 2014.
5. Martin Fowler. *Domain-Specific Languages*. Pearson Education, 2010.
6. The Eclipse Foundation. Eclipse OCL. On-Line: http://projects.eclipse.org/projects/modeling.mdt.ocl, 2005.
7. Achim D Brucker, Dan Chiorean, Tony Clark, Birgit Demuth, Martin Gogolla, Dimitri Plotnikov, Bernhard Rumpe, Edward D Willink, and Burkhart Wolff. Report on the aachen ocl meeting. In *Proceedings of the MODELS 2013 OCL Workshop*, volume 1092. CEUR Workshop Proceedings, 2014.
8. Adolfo Sánchez-Barbudo Herrera. Enhancing Xtext for General Purpose Languages. In Benoit Baudry, editor, *Proceedings of the Doctoral Symposium at MODELS'14*, volume 1321. CEUR Workshop Proceedings, 2014.
9. Frédéric Jouault, Olivier Beaudoux, Matthias Brun, Mickael Clavreul, and Guillaume Savaton. Towards functional model transformations with ocl. In *Theory and Practice of Model Transformations*, pages 111–120. Springer, 2015.
10. Gabriël Konat, Lennart Kats, Guido Wachsmuth, and Eelco Visser. Declarative name binding and scope rules. In *Software Language Engineering*, volume 7745, pages 311–331. Springer Berlin Heidelberg, 2013.
11. Eelco Visser. Program transformation with stratego/xt. In Christian Lengauer, Don Batory, Charles Consel, and Martin Odersky, editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238. Springer Berlin Heidelberg, 2004.
12. Delft University of Technology. Spoofax. On-Line: http://strategoxt.org/Spoofax.
13. Javier Luis Cánovas Izquierdo and Jesús García Molina. Extracting models from source code in software modernization. *Software & Systems Modeling*, 13:1–22, 2012.