Experiments with a High-Level Navigation Language

Jesús Sánchez Cuadrado[1], Frédéric Jouault[2],
Jesús García Molina[1], and Jean Bézivin[2]

[1] Universidad de Murcia
{jesusc, jmolina}@um.es
[2] AtlanMod team, INRIA & EMN
{jean.bezivin, frederic.jouault}@inria.fr

Abstract. Writing navigation expressions is an important part of the task of developing a model transformation definition. When navigation is complex and the size of source models is significant, performance issues cannot be neglected. Model transformation languages often implement some variants of OCL as their navigation language. Writing efficient code in OCL is usually a difficult task because of the nature of the language and the lack of optimizing OCL compilers. Moreover, optimizations generally reduce readability.
An approach to tackle this issue is to raise the level of abstraction of the navigation language. We propose to complement the regular navigation language of model transformation languages with a high-level navigation language, in order to improve both performance and readability. This paper reports on the initial results of our experiments creating the HLN language: a declarative high-level navigation language. We will motivate the problem, and will we describe the language as well as the main design guidelines.

1 Introduction

Model transformations are a key element for the success of Model Driven Engineering (MDE). As this discipline becomes mature, model transformations are being used to address problems of an increasing complexity, and the number of developers writing transformations is also growing. In this way, MDE is being applied to contexts such as DSL-based development, system modernization, or megamodeling. In some scenarios (e.g. system modernization), models being handled are typically large, and performance becomes an important concern.

Model transformation languages usually rely on query or navigation languages for traversing source models in order to feed transformation rules (e.g., checking a rule filter) with the required model elements. In complex transformation definitions a significant part of transformation logic is devoted to model navigation, and most of the transformation bottlenecks are located there. In this setting, performance cannot be neglected when writing navigation expressions. However, writing efficient code can be a difficult issue, and it often compromises readability.

To tackle this issue we propose to raise the abstraction level of the navigation language, so both readability and performance may be achieved at the same time. Readability is improved as language constructs are declarative and reflect better the intention of the developer, whereas compiler optimizations are easier to perform because the granularity of the constructs is coarser. Such a high-level navigation language is intended to complement the regular navigation language implemented as part of a model transformation language.

This paper reports on the initial results of our experiments with a high-level navigation language, called HLN. We have implemented the language on top of the ATL virtual machine [2] in order to interoperate with any language implemented on top of it (e.g., ATL, QVT-R). However, HLN could also be implemented on a different engine architecture. We will explain the structure and the design of the language, and we will show why it is a good complement for navigation languages.

The paper is organized as follows. Next section motivates our approach. Section 3 presents the technical context of this work. Section 4 presents the HLN language and describes how it has been designed. Finally, Section 5 presents some related works, and Section 6 gives the conclusions.

2 Motivation

Many current model transformation languages, such as ATL [8], QVT [12], and ETL [9], use some variants of the Object Constraint Language (OCL) [15] as their navigation language. OCL encourages a "functional" style based on collections, iterators, and expressions without side-effects (e.g., collections are immutable). However, despite its apparent "declarativeness", it can be considered as a low-level navigation language since all details about the navigation steps must be specified [14]. We argue that the level of abstraction of OCL is sometimes inadequate to achieve performance.

From our experience in discovering performance patterns for model navigation in model transformation [5], we have identified four issues with OCL:

- **Algorithm locality.** Algorithms are typically defined in operations attached to metaclasses. Several identical passes are often done, computing the same value several times. Writing global algorithms as global functions often improves performance.
- **Immutability.** The lack of side-effects makes optimizing compilers very important. OCL makes intensive use of collections, and rewriting inmutable operations to mutable operations when possible is a must to get performance. However, few optimizing compilers for OCL are available.
- **Verbosity.** Efficient code tends to be large and verbose.
- **Specificity.** Similar specific algorithms to solve a given navigation problem are generally implemented several times in different transformations definitions, generally at least once per different metamodel. OCL does not provide mechanisms for generalizing algorithms.

To tackle these issues we have built a high-level navigation language, called HLN. This is a declarative language, in which each language construct is intended to address a recurrent navigation problem (i.e., a navigation pattern).

As we will show during the paper, our approach has four main advantages involving performance and readability, namely:

1. **Optimization opportunities.** Possibility for the compiler to optimize, since the language constructs are declarative and their granularity is coarse (e.g., the compiler can use mutable operations internally easily).
2. **Simplicity.** The developer does not need to know how to implement the navigation expressions in an efficient way, the best algorithm is chosen by the compiler.
3. **Readability** is improved because each construct of the language reflects exactly the intention of the developer.
4. **Generality.** Code repetition is alleviated (e.g., writing a similar algorithm for different meta-types) because of the generality of the constructs.

3 Technical context

The technical context of our work is the ATL Virtual Machine (ATL VM) architecture provided by the AmmA platform. The ATL VM language is a small imperative instruction set composed of four categories of bytecodes: stack, memory, control flow, and model handling. It provides facilities to attach functions to metamodel elements, for instance to create helpers, as specified by OCL [15].

In this way, a language such as ATL is built by creating a compiler targeting the VM. An important benefit of targeting a VM architecture is that of language interoperability: operations defined in one language may be used from another. For instance, a library written in HLN can be reused in transformations written in several languages (provided they are implemented on top of the ATL VM).

As we will see, each HLN construct will implicitly yield to the creation of one or more helpers (i.e., operations attached to metaclasses). Such helpers can be called by any other language implemented on top of the VM (e.g., ATL), but also HLN interoperates via helpers with such languages. We do not intend to extend OCL, but to complement it “externally” with navigation libraries.

4 The HLN language

In this section we present the High-Level Navigation (HLN) language. It is a domain-specific language for the domain of model navigation. HLN is intended to allow model transformation developers to specify model navigation statements at a high level of abstraction.

HLN is not a general purpose navigation language in the sense that it is not possible to specify all kinds of navigation on a source model. It is intended to cover a range of common navigation problems for which it provides good performance. Other problems can be solved using the other navigation language that HLN complements (e.g., OCL as generally used for model transformation). As explained above, targeting the ATL VM allows us to fulfil this requirement. The four principles guiding the design of the language are the following:

- **Declarative.** It should be based on declarative constructs. Any construct should allow to specify the result of the navigation without detailing each navigation step. Thus, the language is designed to require the minimum possible amount of information from the developer.
- **Readable.** The syntax of the constructs should resemble natural language as far a possible. Readability and maintainability are promoted since each statement reflects an intention of the developer.
- **Useful.** Each construct of the language should be included only on the basis of experimental tests showing that it provides an improvement in performance or readability.
- **Simple.** There is no "expression language" in order to keep the language simple. We rely on the interoperability with other languages (via helpers) when conditions must be expressed.

The current version of HLN comes from our experience developing a catalog of navigation patterns, which identifies recurrent problems in model navigation [5][1]. Some of these patterns are amenable to be encoded as language constructs. In particular, we have currently implemented four of them as HLN constructs, namely: *linking* elements by some property, computing *transitive closures*, computing the *opposite* of a reference, and setting a *navigation path*.

An HLN library is composed of a header and a set of navigation statements. An excerpt of an HLN library to navigate class diagrams is shown below. The header of a library declares the source models with their reference models (i.e., metamodels) to be navigated (in the example the *CD* metamodel stands for class diagram). Then, one or more navigation statements are written, for instance, to compute the transitive closure of the *superclasses* relationship. A metamodel element is specified by prefixing the metaclass name with the corresponding metamodel name, using **!** as a separator (e.g., *CD!Class* for metaclass *Class* of metamodel *CD*). A metaclass property (possibly a helper) is specified using the dot notation (e.g., *CD!Class.superclasses*).

```
navigate IN : CD;
transitive closure allSuperclasses of CD!Class.superclasses
```

Figure 1 shows the abstract syntax of the language. The constructs inherit from the *Statement* abstract metaclass. The *HelperRef* metaclass represents a property or helper of some metaclass (note that it references *MetaElement*), but it does not say whether the property or helper exists or whether it will be created. This is part of each construct's semantics. The concrete syntax of HLN (used in examples) is textual, and is implemented with the TCS tool [7].

It is worth noting that the current implementation of the HLN compiler does not perform any static checking against the source metamodel. This means that, for instance, it is required to explicitly declare whether a property is multivalued (*multivalued* attribute of *HelperRef*), despite this information being al-

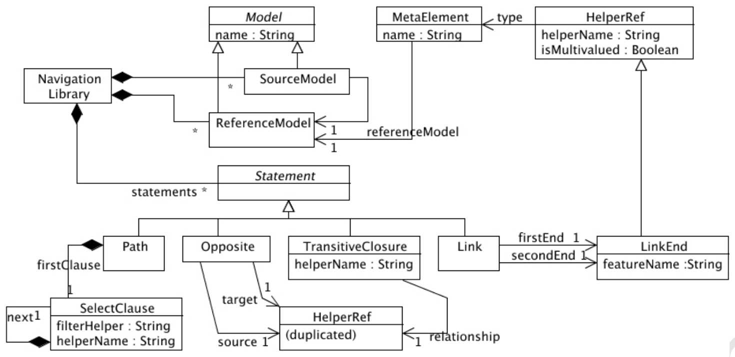

Fig. 1: Abstract syntax metamodel of HLN.

ready present in the source metamodel. At the concrete syntax level this is expressed with [*]. The *SelectClause* and *TransitiveClosure* do not use *HelperRef*, but just *helperName*, to reference a helper to be created, because the metaelement and multiplicity are implicit.

Next, each construct will be introduced by showing a piece of simple OCL code illustrating the navigation pattern. From such code, the essential parts will be identified, and an HLN construct will be derived from them.

4.1 Linking

The *linking* pattern appears when two model elements are implicitly linked because they both have some feature (possibly helpers) with the same value (i.e., this is a kind of *join*).

For instance, a class diagram can be annotated using some weaving infrastructure such as AMW. Gathering the annotations for a given class could be done in the following way. For each class, all annotations are iterated, looking for those "pointing" to the current class.

```
context CD::Class def: annotations : Sequence(AMW::ClassAnnotation) =
  AMW::ClassAnnotation.allInstances()->select(a| self.__xmiID__ = a.ref)
```

Taking into account the piece of code above we derive an HLN construct, which is typically implemented using some kind of hash join. The essential parts are the following: the two metaclasses whose instances will be matched (e.g., *CD::Class* and *AMW::ClassAnnotation*), the name of the feature of each metaclass to be compared (e.g., *__xmiID__* and *ref*), and the name of helpers to be created (e.g., only *annotations* in this case). In general, two helpers will be created, one for each linked end.

The piece of abstract syntax metamodel for this construct is shown in Figure 1 (*Link* metaclass). A linking is specified as a *link* between two *link ends*. Each link end represents a metaclass and the property used to compare. A new helper containing the result of linking one end with the other end is created. In this way, we focus on the "what", and the "how" is left to the HLN's compiler.

A piece of HLN code illustrating the concrete syntax of this construct is shown below. It is intended to be read as if it were natural language: *link each Class using __xmiID__ to each ClassAnnotation using ref*). The result is that a helper called *annotations* is attached to metaclass *Class*. The _ symbol is used to indicate that we do not want to create the helper at the other end.

```
link  CD!Class.annotations[*] on __xmiID__
  to AMW!ClassAnnotation._[*] on ref;
```

Notice that the [*] modifier could be removed, but instead of computing a collection of links for each instance, only one link would be selected (if there are more links they are just discarded).

4.2 Navigation path

Navigation of models with OCL is based on the dot notation to access model element properties. Collections are typically handled using iterators, such as *select* to filter elements, *collect* for deriving a collection from another one or *any* for getting an element satisfying a condition.

A typical pattern in OCL is navigating through multiple multi-valued references, filtering by some criteria, so that a combination of *select*, *collect* and *flatten* operations are needed to get the final collection. The following piece of OCL code shows this pattern in the case of obtaining all attributes contained by the classes of a package.

```
context CD::Package def : getAllAttributes : CD::Attribute =
  self.classifiers->select(c | c.isClass)->
                    collect(c | c.features)->flatten()->
                    select(f | f.isAttribute)
```

The evaluation of this expression implies creating a collection with all classifiers satisfying the *isClass* condition, next a collection containing collections of features is created, which is then flattened, and it is filtered again to obtain the result. As the number of navigation steps grow the evaluation is more inefficient.

This shows that several implementation-level details must be specified in OCL, in particular the *collect* and *flatten* operations are needed only to "normalize" the filtered collection before applying the next filter (i.e., the second *select* operation). The essential information includes navigated properties (e.g., *classifiers* and *features*) and filter conditions (if any).

The piece of metamodel for the corresponding construct is shown in Figure 1 (*Path* metaclass). An expression is based on nested navigation clauses that take the result of the owning clause to perform its navigation step. An example of the concrete syntax is shown below. The *when* part is optional if no filter is specified. This expression can be naturally read as: *given a package, select all classifiers satisfying the isClass condition, and, for each one, select all features satisfying the isAttribute condition.*

```
path ClassM!Package.getAllAttributes
    select classifiers   when isClass
    select features      when isAttribute
```

4.3 Opposite

Given a relationship from one model element to another, it is often necessary to navigate through the opposite relationship. For instance, in a class diagram, the opposite for the *superclasses* relationship of a *Class* metaclass is the collection of direct subclasses for a given class.

If the opposite relationship has been defined in the metamodel, then navigation in both directions can be efficiently achieved. However, such opposite relationship is not always available, so an algorithm has to be worked out.

A straightforward algorithm will involve traversing all the instances of the opposite relationship's metaclass and checking which of them are part of the relationship. For instance, to get the owning package of a class, the opposite of the package's *classifiers* relationship is computed as follows[3]:

```
context CD::Class def: owner: CD::Package =
  CD::Package.allInstances()->any(p |
    p.classifiers.includes(self) )
```

The corresponding HLN construct can be easily derived from this algorithm. The required elements are two metaclass/relationship pairs: the source metaclass and the already existing relationship, and the target metaclass with the name of the new relationship to be computed. The piece of metamodel corresponding to this construct is shown in Figure 1 (*Opposite* metaclass).

An example of the concrete syntax is shown below. It should be read in the following way: *compute the opposite relationship, called CD!Class.owner, of the relationship CD!Package.classifiers.*

```
      opposite CD!Class.owner of CD!Package.classifiers[*];
```

4.4 Transitive closure

Computing the transitive closure of a relationship is a common operation in model transformations. An example is computing the set of all direct and indirect superclasses of a class. Another example is computing the set of reachable states from a given state of a state machine.

Let us consider a piece of OCL code to compute the transitive closure of the `superclasses` relationship in a class diagram. There are several performance issues in this code. Firstly, the `union` operation is immutable, which means that collections are duplicated unless the OCL compiler is able to detect and optimize this case. Secondly, *collect* and *flatten* also imply duplicating collections. Finally, the transitive closure is computed several times for the same class. A transitive closure can be implemented in one traversal, whereas a straightforward OCL implementation such as this one performs several.

[3] MOF and Ecore provide the refImmediateComposite() and eContainer() operations respectively to get an element's container. However, the discussion still holds for non-containment relationships, and when such operations are not made available by the underlying transformation language.

```
context CD::Class def : allSuperclasses : Sequence(CD::Class) =
  self.parents->union(self.superclasses->
                              collect(c | c.allSuperclasses)->flatten())
```

The only essential information to derive the HLN construct is the name of the relationship (e.g., *superclasses*), the corresponding metaclass (e.g., *CD!Class*), and the attribute helper that will be created (e.g., *allSuperclasses*). The piece of metamodel corresponding to this construct shows that (Figure 1, *TransitiveClosure* metaclass). The details about how to perform the computation are ignored This means that the compiler may evolve to implement a more efficient version of the construct, without affecting existing HLN libraries.

The concrete syntax for this construct is the following. Notice that it is implicit that the *allSuperclasses* helper must belong to *CD!Class*, and that it is multi-valued.

```
transitive closure allSuperclasses of CD!Class.superclasses
```

4.5 Combining constructs

To cover a wider range of navigation problems, while keeping the language simple, HLN allows constructs to be combined using the helpers created as a result of one construct in another construct. Again, we rely on the use of helpers to interoperate, in this case for the interoperability of the language constructs. Notice that the order of the constructs is not important, because the compiler may keep track of the dependencies.

For instance, the *transitive closure* construct does not consider a relationship defined by means of an intermediate class, such as is the case of the *superclasses* relationship in UML 2.0, which is defined using the *Generalization* metaclass.

Instead of extending HLN, the *path* construct can be used to first get the collection of direct superclasses, going through the *generalization* relationship. Then, the helper created for this construct is used seamlessly for the *transitive closure* construct. This is shown in the following piece of HLN code.

```
path UML!Class.directSuperclasses
      select generalization
      select general;
transitive closure allSuperclasses of UML!Class.directSuperclasses;
```

Another useful example of this technique involves defining the "inverse transitive closure" of a relationship, which can be computed combining the *opposite* and *transitive closure* constructs.

5 Related work

Two main approaches to model query or navigation can be found in model transformation languages: patterns and navigation languages. Graph patterns are typically used in graph transformation languages, such as Viatra [4] or GReAT [3]. Objects patterns are available in QVT Relational [12] and in Tefkat [10]. OCL-like navigation languages are the primary navigation mechanism provided by rule transformation languages such as ATL [8], QVT Operational [12], and ETL [9].

Even though in this paper we have focused on OCL, our approach is applicable to complement other query languages. For instance, pattern languages use object properties to constraint query results. Such properties can be helpers defined by an HLN library, since the VM makes the integration seamless.

Regarding the performance of OCL, in [6] the need for developing benchmarks to compare different OCL engines is mentioned. The authors have developed several benchmarks but they are intended to compare features of OCL engines, rather than performance. In [11] the authors present several algorithms to optimize the compilation of OCL expressions. They argue that its optimizing OCL compiler for the VMTS tool can improve the performance of a validation process by 10-12%.

Finally, domain-specific query languages have been proposed as a means to enhance the query mechanism (strategies) of the Stratego program transformation tool. In particular, the implementation of an XPath-like language is discussed [13]. It behaves like a macro-system, generating Stratego code. In our case, we are able to generate efficient VM code.

6 Conclusions

In the paper, we reported on our experiments with an approach to model navigation based on a high-level navigation language providing declarative constructs. We introduced the HLN language, and we compared HLN against writing navigation expressions in OCL. The initial benchmarks we have carried out have shown performance improvements ranging from 20% to 800% with respect to a normal ATL implementation.

The contribution of this work is two-fold, on the one hand we have shown that raising the level of abstraction of a navigation language has several advantages: (1) it allows the compiler to easily optimize, which yields improved performance, (2) quality attributes such as readability and maintainability are also improved, and (3) model navigation best-practices are encoded in language constructs. On the other hand, the HLN implementation is a contribution itself. Its current implementation can be used as a complement to any language built on top of the ATL VM, and it is also useful for tool implementors to compare the performance of their navigation languages[4].

A possible extension of this work includes adding new constructs and improving performance of the current ones, comparing to other transformation languages (e.g., with an optimizing compiler), and creating a static type checker. It would also be useful to investigate how to improve interoperability of HLN with other languages (e.g., via parameters) while still keeping the language simple.

Acknowledgments

This work has been supported by Fundación Seneca (Murcia, Spain), grant 08797/PI/08.

[4] The implementation of HLN, and several benchmarks can be downloaded from http://www.modelum.es/projects/hln [1].

References

1. Benchmarks for HLN. http://www.modelum.es/projects/hln/.
2. Specification of the ATL Virtual Machine. http://www.eclipse.org/m2m/atl/doc/.
3. A. Agrawal. Graph Rewriting and Transformation (GReAT): A Solution for The Model Integrated Computing (MIC) Bottleneck. In *ASE*, pages 364–368, 2003.
4. A. Balogh and D. Varró. Advanced model transformation language constructs in the VIATRA2 framework. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 1280–1287, New York, NY, USA, 2006. ACM.
5. J. S. Cuadrado, F. Jouault, J. Garcia-Molina, and J. Bèzivin. Optimization patterns for OCL-based model transformations. In *Proceedings of the 8th OCL Workshop*, 2008.
6. M. Gogolla, M. Kuhlmann, and F. Buttner. A benchmark for OCL engine accuracy, determinateness, and efficiency. In *MoDELS*, volume 5301 of *Lecture Notes in Computer Science*, pages 446–459. Springer, 2008.
7. F. Jouault, J. Bézivin, and I. Kurtev. TCS: A DSL for the specification of textual concrete syntaxes in model engineering. In *GPCE '06: Proceedings of the 5th International Conference on Generative programming and Component Engineering*, pages 249–254, New York, NY, USA, 2006. ACM.
8. F. Jouault and I. Kurtev. Transforming Models with ATL. In *MoDELS 2005: Proceedings of the Model Transformations in Practice Workshop*, Oct 2005.
9. D. S. Kolovos, R. F. Paige, and F. A. Polack. The Epsilon Transformation Language. In *1st International Conference on Model Transformation, ICMT'08*, pages 46–60, 2008.
10. M. Lawley and J. Steel. Practical declarative model transformation with Tefkat. In J.-M. Bruel, editor, *Model Transformations In Practice Workshop*, volume 3844 of *LNCS*, pages 139–150, Montego Bay, Jamaica, Oct. 2005. Springer.
11. G. Mezei, T. Levendovszky, and H. Charaf. An optimizing OCL compiler for metamodeling and model transformation environments. In *Software Engineering Techniques: Design for Quality*, pages 61–71. Springer, 2007.
12. OMG. Final adopted specification for MOF 2.0 Query/View/Transformation, 2005. www.omg.org/docs/ptc/05-11-01.pdf.
13. J. van Wijngaarden. Code Generation from a Domain Specific Language. Designing and Implementing Complex Program Transformations. Master's thesis, Utrecht University, Utrecht, The Netherlands, July 2003. INF/SCR-03-29.
14. M. Vaziri and D. Jackson. Some Shortcomings of OCL, the Object Constraint Language of UML. In *Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS'00)*, page 555, Washington, USA, 2000. IEEE Computer Society.
15. J. B. Warmer and A. G. Kleppe. *The Object Constraint Language: Precise Modeling With UML*. Addison Wesley, 1998.