Separation of Concerns in Translational Semantics for DSLs in Model Engineering

Thomas Cleenewerck
PROG, Vrije Universiteit Brussel
Brussels, Belgium
tcleenew@vub.ac.be

Ivan Kurtev
ATLAS Group, INRIA, France (until Nov 2006)
SE Group, University of Twente, the Netherlands
ivan.kurtev@gmail.com

ABSTRACT
Development of Domain Specific Languages (DSLs) in the context of Model Driven Engineering is gaining more and more popularity. As evolution lies in the heart of every software system, the major requirement for DSLs is that they should be modular and resilient to changes. MDE-based DSL frameworks should enable a modular specification of language translational semantics and the composition of the modules into languages. Ultimately, the availability of such techniques should make the DSL development faster. Separation of concerns is a sound software engineering principle used to obtain better modularity, reusability, and adaptability of systems. However, this principle must be supported by proper tools that allow the separation achieved at a conceptual level to be preserved in the language specification. In MDE, the mainstream tools for specifying translations are model transformation languages. In this paper we evaluate a class of model transformation languages regarding their applicability for capturing the translational semantics of DSLs in a modular way. We found that the concepts in the domain of translational semantics significantly mismatch with the language constructs of the transformation language. We suggest that this problem may be better approached by a domain-specific transformation language.

Categories and Subject Descriptors
D.3.2 [**Language Classifications**]: Specialized Application Languages – *model transformation languages*. D.3.3 [**Programming Languages**]: Language Constructs and Features – *modules, packages*. D.2.13 [**Reusable Software**]: Reusable Libraries.

General Terms
Design, Languages

Keywords
Modular translational semantics, Model-based DSLs, Separation of concerns, model transformations, Model engineering

1. INTRODUCTION
Model Driven Engineering (MDE) is organized around the unifying concept of *model*. A model is expressed in a modeling language: a formal notation that allows automatic or semi-automatic manipulation of models, for example, by applying model transformations. Models may be expressed in a general-purpose modeling language such as UML or in a well-scoped domain-specific modeling language (DSML or just DSL). Since various aspects of different nature need to be modeled there is a need for variety of DSLs. Therefore, implementing domain-specific modeling languages in an efficient way is of a primary importance in MDE.

One particular problem in such an implementation is dealing with changes in the language. The changes are driven by the dynamic nature of the underlying problem domain. Emerging domains tend to change over time reflecting the dynamic nature of the real world phenomenon under study. Therefore, the DSL definition must follow these changes and needs to be adapted accordingly.

In the context of MDE we perceive a DSL as a set of coordinated models as proposed in [16]. One model specifies the domain concepts and it is known as *domain ontology* or *metamodel*. Another model may specify a particular concrete syntax for the DSL. The semantics of DSLs may be specified in multiple ways. There is no commonly agreed formalism for semantics specification of DSLs. In this paper we are interested in a specific type of semantics known as *translational semantics*. The translational semantics specifies how a sentence in a language may be translated to a sentence in another language.

It is clear that the evolution of a DSL affects all of its components. For example, adding new domain concepts affects both the metamodel and the translational semantics. Furthermore, a language is hardly developed from scratch. Existing language fragments may be reused across multiple DSLs. A DSL specification or a DSL fragment should be reusable and adaptable.

In this paper we focus on the problem of specifying a translational semantics for DSLs in MDE. The major requirement for such a specification is that it should be modular and resilient to changes. The ultimate goal is to develop a technique that allows modular incremental development of DSLs in which new constructs are added and composed with the existing ones.

Separation of concerns (SOC) [7] is a powerful technique to tackle the complexity and improve the reusability and evolvability of systems. Applied to DSL translational semantics this technique requires the identification and the specification of the translation for each language concern in a

separate module and a mechanism to integrate the loosely coupled modules in a complete language.

The most commonly used technique for specifying translations in MDE is by using model transformation languages. Several languages were proposed based on different paradigms [1][4][11][17][24]. The research question we aim to answer in this paper is whether currently proposed model transformation languages can be used to express a modular specification of translational semantics based on separation of concerns. It is not possible to explore all the languages in a single paper. Therefore, we focus on a class of model transformation languages that possess certain set of common features. We consider a case study that illustrates the typical problems when modularizing translational semantics. We use ATL [11] as a representative transformation language to implement the example.

This paper is organized as follows. Section 2 presents background information on DSLs, the principle of separation of concerns and its application to obtain modular translational semantics. Section 3 presents known obstacles encountered in the existing transformation systems when separating language concerns in DSL specification. Section 4 presents a case study of translating tuple calculus to SQL. Section 5 gives an overview of the related work. Section 6 concludes the paper and outlines future research.

2. BACKGROUND

Domain-specific languages have been studied in the context of Grammarware [13] for years. Recently, they gained a lot of attention by the MDE community. We present a possible view on DSLs in the context of MDE. Furthermore, we briefly summarize the well-known principle of separation of concerns and how it may be applied to translational semantics.

2.1 DSLs in the Context of MDE

A DSL is a language designed to be useful for a delimited set of tasks, in contrast to general-purpose languages that are supposed to be useful for much more generic tasks, crossing multiple application domains. Regardless the domain specific or general nature of languages they share a common structure:

- They usually have a concrete syntax;
- They have an abstract syntax;
- They have a semantics, implicitly or explicitly defined;

There are several ways to define these syntaxes and semantics. The most commonly used way for defining the concrete syntax is via grammar-based systems. In contrast, there are multiple semantics specification frameworks but none has been widely established as a standard.

In MDE all the components in a DSL definition are captured in models. We recognize the following components:

Domain Definition Metamodel. DSLs have a clearly identified problem domain. Programs (sentences) in a DSL represent concrete states of affairs in this domain, i.e. they are models. A conceptualization of the domain is an abstract entity that captures the commonalities among the possible state of affairs. It introduces the basic abstractions of the domain and their mutual relations. Once such an abstract entity is explicitly represented as a model it becomes a metamodel for the language. We refer to this metamodel as *domain definition metamodel* (DDMM). Such a DDMM plays the role of the abstract syntax for a DSL.

Concrete Syntax. A DSL may have different concrete syntaxes. Each one is defined by a transformation model that maps the DDMM onto a "display surface" metamodel. Examples of display surface metamodels may be SVG or GraphViz, but also XML. Often, the concrete syntax is captured by a grammar. In that case the relation between the grammar constructs and the metamodel elements is of primary importance in order to handle the necessary bidirectional translation between the textual representation and the model structure. Some approaches to solve this are described in [2][8][12].

Semantics. A DSL may have an execution semantics definition. This semantics definition is also defined by a transformation model that maps the DDMM onto another DSL having by itself a precise execution or even to a GPL.

In addition to canonical execution, there are plenty of other possible operations on programs based on a given DSL. Each may be defined by a mapping represented as a transformation model. As we already mentioned a particular type of semantics is translational semantics: a mapping of the DDMM to another DDMM. In that way models in one language are transformed in another language that may possibly be executable.

2.2 Separation of Concerns in Translational Semantics

In this section we detail the SOC for language implementations to be able to understand better the challenges language implementations pose to separate the different concerns. We distinguish two kinds of concerns: *basic concerns* and *special-purpose concerns*.

A basic language concern comprises a single language construct accompanied by its translational semantics. In the context of our view on DSLs a basic concern is related to an element from the DDMM and specifies how it is translated to elements in the target DDMM. Special-purpose concerns capture every kind of involvement of another language concern (ranging from a direct reference to other concerns, or any implementation decision that is imposed by, or stems from, another concern). As such we can study the implementation of the special-purpose concerns separately.

We define a *language module* as a language construct accompanied by its translational semantics that constitutes an important design decision in the language. The translational semantics is the semantics of the language construct that a module hides. The interface should reveal as little as possible about its translational semantics, in accordance to the definition of modules given by Parnas [22].

Often, a language concern needs to use or compute additional information extracted from other concerns or to produce results used by another concerns. We consider these issues not as an integral part of the basic concern since they all involve other concerns. In our approach these issues are treated as separate concerns. We refer to these concerns as the special-purpose concerns. We adopted this terminology from Walter L. Hursh et.al. [10], who first introduced this distinction in the context of aspect languages.

3. CHALLENGES IN SEPARATING CONCERNS IN TRANSLATIONAL SEMANTICS

Various systems were proposed to specify modular translational semantics (see section 5). The study of this domain in the context of Grammarware systems led to the identification of several challenges in separating and integrating basic and special-purpose concerns.

In this section we present the challenges on the base of a case study. Later on in section 4 we implement the case study in a model transformation language and evaluate how the problems are handled.

3.1 Case Study: Tuple Calculus to SQL

Assume we have to give a translational semantics of the Tuple calculus language by specifying transformation to the SQL language. Both languages are defined by their metamodels. Figure 1 shows the metamodel of the Tuple calculus. This language defines expressions that are evaluated to sets. An expression is instance of class *Set*. Every expression enumerates header attributes and specifies a condition over the header attributes (class *BooleanExpression*). To save space some classes are omitted from the figure: the specializations of class *Comparator*, the binary operations *And* and *Or*, and the classes for existential and universal quantifiers that specialize class *Quantifier*.

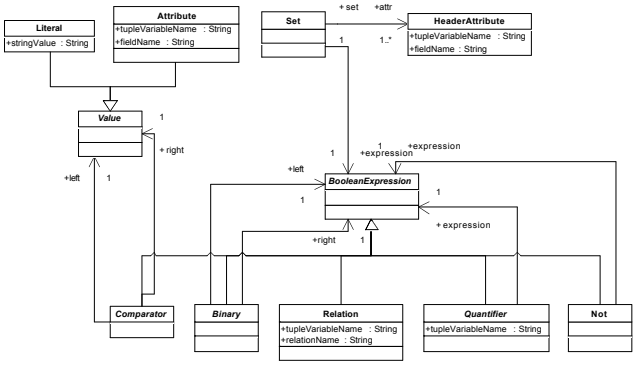

Figure 1 Metamodel of the Tuple calculus

Figure 2 shows the metamodel of the SQL language. Again, some classes are omitted. The main construct is the *Select* class that extracts number of columns (clause SELECT) from a set of tables (clause FROM). A condition may be imposed on the column values (clause WHERE).

Some of the informal translation rules are:

- A set expression in Tuple calculus is transformed to a select SQL query;
- Header attributes are transformed to columns;
- Relation predicates are transformed to table declarations enumerated in the FROM clause;
- Conditions are transformed to expressions specified in the WHERE SQL clause;

For example, the following set expression in tuple calculus

```
{p.name | employee(p) and p.salary = '50000'}
```

will be translated to the following SQL query:

```
SELECT P.name
FROM EMPLOYEE as P
WHERE P.salary = '50000'
```

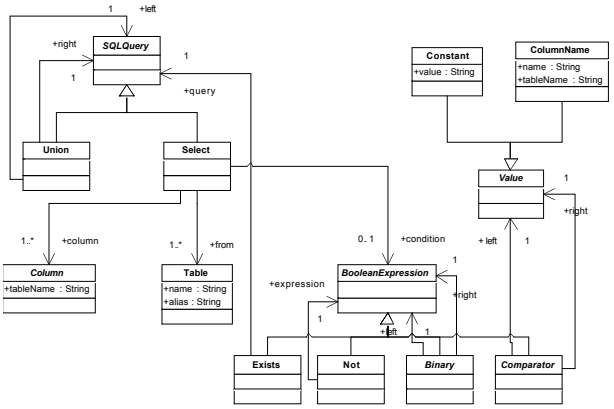

Figure 2 Metamodel of the SQL language

Furthermore, the header attributes may be unbounded. The translation process must identify the bounding expressions for such attributes. For example, the set expression

```
{t.supplier, t.article |
(∃ s)(supplier(s) and s.ssname=t.supplier and
(∃ p)(product(p) and p.pname=t.article and
(∃ a)(supplies(a) and s.s#=a.s# and
a.p#=p.p#)))}
```

contains one unbound variable *t*. Unbound in this case means that the variables are not predicated by a relation. However, the header attributes are bound in the condition: *t.supplier* is bound to *s.sname* and *t.article* is bound to *p.pname*. During the translation these bindings must be identified and the header attributes must be translated to the proper columns: *s.sname* and *p.pname* respectively.

The resulting SQL query will be:

```
SELECT S.sname as supplier, P.pname as article
FROM PRODUCT as P, SUPPLIER as S, SUPPLIES as A
WHERE ……..
```

3.2 Challenges

In this section we identify the concerns in the translational semantics and outline the challenges that must be met in order to comply with the principle of separation of concerns.

First, we choose the major source language constructs as basic concerns: *Set*, *HeaderAttribute*, *Relation*, *Attribute*, *Quantifier*, etc. In the ideal situation the translation of each construct will be isolated from the rest. However, this is possible only if the structures of the source and target languages are congruent. This is hardly possible in complex real life cases. Usually, we observe structural mismatches that lead to concern interactions and therefore to special-purpose concerns.

We have two types of structural mismatches that correspond to two types of transformations identified by Wijngaarden and Visser [27]. These types are called *global-to-local* (G2L) and *local-to-global* (L2G) transformations. We clarify them below.

Global-to-local transformations. In this type of transformation the source language construct is not enough to produce the output. Additional information is needed that may

be either external configuration information or information that resides in other language constructs, that is, in other concerns. In our example, to translate unbound header attributes we have to look in the condition expression which forms another concern. This requires coordinated effort among more than one concern to produce the result.

Local-to-global transformations. In this type of transformation multiple results are produced by a single source construct that is used by other concerns. In our example the source construct *Relation* produces two target constructs: one is part of the expression in which the relation is used and the second is a table declaration that is integrated in another target construct i.e. the *from* feature of class *Select*. The *Select* class is part of another concern. The table declaration construct is known as a *non-local* result since it pertains to another concern. Again, we have concern interaction. Moreover, some of the produced tables may be discarded depending on the expression context. This context dependency requires a coordinated effort among more than one concern to correctly integrate the non-local result. This is exemplified in the next section.

In summary, in both types of G2L and L2G transformations we are faced with interactions among concerns: querying for additional information and integrating non-local target constructs produced by different concerns. A major quality requirement to achieve a good separation of concerns is that we specify the translation in loosely coupled modules. Therefore, the major challenge is to handle the concern interactions in a way that does not increase the degree of coupling and does not break the separation among the modules.

In the next section we apply this principle on the presented example by using a model transformation language.

4. TRANSLATIONAL SEMANTICS EXPRESSED IN MODEL TRANSFORMATION LANGUAGE

In this section we implement the case study in a model transformation language and evaluate how the problems are handled.

This paper does not aim at exploring transformation languages one by one. Instead, we focus on a class of languages that share common features and select one language to illustrate the case study. The basic assumptions about the class of model transformation languages are detailed in section 4.1. Section 4.2 presents the major point of interest of the implementation according to the challenges described in Section 3.2. The last section lists our findings concerning the separation of concerns.

4.1 Basic Assumptions about Model Transformation Languages

A number of model transformation languages have been proposed. Some of them are based on graph transformation techniques and others emerged as an answer to the OMG QVT RFP [19]. In this paper we focus on the second type of languages. We assume that the following features are supported by the language:

- Transformation rule is the basic language construct. A rule has a left-hand side that matches over source models and a right-hand side that specifies metamodel elements to be instantiated in the target model;
- The language is hybrid. Both declarative and imperative styles are allowed.
- A rule may access the results produced by other rules. Two forms of access are possible. The first is via traceability links in which no explicit rule call is involved. In this case the resolution algorithm of the language is used. The second is by calling a rule by its name and parameter passing. We assume that the reader is familiar with these basic mechanisms. More info is presented in [6][17][15];
- Navigation over models is done by using OCL expressions [20];
- It is possible to separate navigation functionality in helpers (e.g. ATL [11] and QVT Operational mappings [21] support this feature). Usually, only source models can be navigated;

4.2 Implementation of the Case Study in ATL

The space limitation does not permit us to present the complete implementation of the case study. It may be downloaded from the link given in [9]. Here we focus on parts of the transformation that illustrate the solutions to the problems presented in section 3.2.

4.2.1 Local-to-global transformation

As we already mentioned an example of such a transformation is the generation of SQL table declarations from the relational predicates. This generation is captured by the two rules shown below.

The rules produce two output elements: one table and the Boolean constant *true*. The table is used by another target element and is therefore considered as a non-local element. Since the predicate relations participate in Boolean expressions they must be translated to parts of the target expression. In SQL every variable is bound to a table so the relation is always evaluated to true.

```
rule RelationTranslation
        from s : RelationalCalculus!Relation
        to table : SQL!Table (
                    name<-s.relationName,
                    alias<-s.tupleVariableName
                   )
}

rule  Relation2True
        from s : RelationalCalculus!Relation
        to true : SQL!True
```

The rule that uses the non-local table result specifies the translational semantics of the *Set* language concern (the line that sets the list of table declarations is underlined):

```
rule  Set2SelectQuery  {
        from s : RelationalCalculus!Set
        to t : SQL!Select(
                 condition<-s.expression,
                 column<-s.attr,
                 from<-s.expression.getRelations
                )

}
```

The generated tables must be collected and they become value of the property *table*. However, the integration of the tables is not that straightforward. Some tables may be discarded depending on the expression in which the relation is used. We will give one example of such situation. The full explanation of the algorithm for discarding tables and integrating is related to the semantics of expressions in tuple calculus and is beyond the scope of this paper.

Consider the following expression:

```
{w.name | (∃w)(not works_on(w) and manager(w)... )}
```

Two tables will be produced from this expression: *works_on* and *manager*. However, the *works_on* relation may be discarded since it does not change the result of the query. Existentially quantified negated relations are expressions that are always true. This is due to the semantics of the tuple calculus according to which there is always at least one element in the universe that does not belong to a given relation. Therefore, the table *works_on* should be discarded and the table *manager* should be kept. It is not possible to judge if a table is discarded only on the base of the source relation predicate. The whole expression must be analyzed.

Here we are faced with a non-trivial concern interaction. In general, there are two ways to solve the integration problem depending on the "active" part. In the first way the concern that needs the non-local result finds it and then the integration is performed. In the second way the non-local result takes the "active" role and locates the target elements in whose context it has to be integrated. The identification may be done in many ways, for example, by traversing the source model, by using a global table, etc.

In model transformation languages usually the target element that needs the non-local result takes the responsibility to find and integrate it. The non-local result may be obtained by explicit rule call or by using the resolution algorithm. The resolution algorithm requires that the source element that produces the element is found first and then it is resolved to the target element. In this process it is not necessary to know the rule that has produced the element. In our implementation we opt for the second option: using the resolution algorithm. The challenging part is to implement the navigation over the source model.

Our implementation traverses the source Boolean expression and applies the rules for detecting the relations that must be discarded. Only the selected relations are used to generate table declarations. Navigation over the source model in ATL can be encapsulated in OCL helpers. We define helper per every element type used in Boolean expressions. Every helper collects relations for that type. Relations are kept in a sequence of tuples where every tuple contains the source relation and two flags that indicate if the relation is discarded and required. Two of the helpers are shown below:

```
helper context RelationalCalculus!Relation
  def: getRelations : Sequence(TupleType(
                              relation :
RelationalCalculus!Relation,
                           discard : Boolean,
                           required : Boolean)) =
  Sequence{Tuple{relation = self,
                 discard = false,
                 required = true}};

helper context RelationalCalculus!Exists
  def: getRelations : Sequence(TupleType(
                              relation :
RelationalCalculus!Relation,
                              discard : Boolean,
                              required : Boolean)) =
  self.expression.getRelations->select(r|

not(r.relation.tupleVariableName=self.tupleVariableName
          and r.discard)
  );

helper context RelationalCalculus!Not
  def: getRelations : Sequence(TupleType(
                              relation :
RelationalCalculus!Relation,
                           discard : Boolean,
                           required : Boolean)) =
  self.expression->collect(r |
      Tuple{relation = r.relation,
            discard = true,
            required = r.required}
  );
```

The helpers are named *getRelations*. In ATL, helpers with the same name and type may be associated to different source metamodel elements. In that way a polymorphic behavior is achieved.

It can be seen that the helper defined in the context of *Not* metaclass indicates that a relation may be discarded by turning the flag *discard* to *true*. Furthermore, the helper associated to *Exist* metaclass performs discarding of the negated relation predicates. The remaining helpers are skipped in order to save space.

This helper is invoked from the rule *Set2SelectQuery* in the line

```
from<-s.expression.getRelations.
```

The right-hand side of the expression returns a set of source relations. At that moment the resolution algorithm is applied and the table declarations produced from the relations are collected and assigned as value of the *from* feature.

4.2.2 Global-to-local transformation

To illustrate this type of transformation we take the example of unbound header attributes. To produce the target element we have to inspect the source model and to find a *Comparator* of type *Equal* and to check if one of the operands is the same as the header attribute. The following code snippet shows that:

```
helper context RelationalCalculus!HeaderAttribute
  def: Binding : TupleType(tableName : String,
                           columnName : String,
                           alias : String) =
  let name : String =

self.set.expression.getTableName(self.tupleVariableName) in
  if name = '' then
    RelationalCalculus!Equal.allInstances()->
    asSequence()->collect(e |
      if e.left.oclIsKindOf(RelationalCalculus!Attribute)
and
         e.right.oclIsKindOf(RelationalCalculus!Attribute)
      then
        if e.left.tupleVariableName =
self.tupleVariableName and
           e.left.fieldName = self.fieldName
        then
          Tuple{tableName = e.right.tupleVariableName,
            columnName = e.right.fieldName,
            alias = self.fieldName}
        else
          if e.right.tupleVariableName =
self.tupleVariableName
              and e.right.fieldName = self.fieldName
          then
            Tuple{tableName = e.left.tupleVariableName,
                  columnName = e.left.fieldName,
                  alias = self.fieldName}
          else
.....
endif;
```

The query is defined in a helper, separating it from the *Set2SelectQuery* rule. The result produced by the helper is used in another helper that is not shown in the paper. Ultimately, the name and the alias of the columns are assigned properly.

4.3 Discussion

In this section we evaluate the presented implementation. Several points are of interest: mapping the conceptual elements in the problem domain to the language syntactical constructs, and achieving proper separation and integration of concerns for both types of transformations: L2G and G2L.

4.3.1 Correspondence between Problem Domain Concepts and Language Constructs

We have presented several concepts from the domain of translational semantics: basic and special-purpose concerns, local and non-local elements, etc. It is interesting to see how they are mapped to the constructs provided by the transformation language. In this discussion, the domain of translational semantics is the problem domain, and the domain of transformation languages is the solution domain.

ATL and other languages are general purpose transformation languages whereas the transformations for translational semantics are special purpose transformations i.e. they have distinct characteristics (basic vs. special-purpose concerns, local vs. non-local results). It is thus not a surprise that there is a mismatch between the problem domain concepts and the solution language constructs. First, there is no concept of a language module (see section 2.1). A basic concern may be expressed in one or in multiple rules. There is no distinction among rules. They are equal from the point of view of the transformation language whereas from the problem domain point of view they represent different language concerns. Second, there is no distinction between local and non-local elements. Both are represented as targets in transformation rules. Third, some of the transformation functionality is encoded in helpers. Again, it is not possible to identify membership of a given helper to a given concern. This membership is important in order to reuse the basic concerns in a changing language definition.

The identified mismatches do not allow representing a single language concern in a single transformation construct. An important requirement for achieving good separation of concerns is separate specification and encapsulation of the concern. The lack of distinction between local and non-local elements prevents us from specifying clear interface of every concern.

4.3.2 Local-to-global Transformations

We considered the translation of relation predicates as an example of local-to-global transformation. Here we evaluate the degree in which the implementation achieved separation of concerns.

It was possible to capture the translation of relations in a single construct and to separate the generation of the local and the non-local results in separate rules. Although in the example we gave the separation seems quite successful, there are some subtle issues that may have an impact on the degree of separation of concerns.

First, the target elements and properties where we integrate the non-local results are statically defined and tangled within the rules that produce these target elements. As these rules represent different concerns, they therefore violate the separation of concerns. Let us revisit the Set2SelectQuery. It contains a query to collect the table declarations from the sub-expressions (`from<-s.expression.getRelations`). This query is thus statically selected and tangled with the language concern to handle relations. It is statically selected because according to the metamodel we need to set up certain properties of the target element. We know that the table declarations must be in the select query. However, in cases of multiple integration locations, statically deciding where the results should be integrated is not the most elegant solution.

Second, as we mentioned before the challenge in L2G transformations is how we integrate the non-local result. This is achieved by defining helpers that perform navigation from the element that needs the non-local results to the elements that produce the result. We define several helpers, one per each type of expression. In that way helpers are attached to other language constructs and the integration algorithm as a whole can be perceived as scattered across several concerns. It is not clear how to interpret the helpers: as belonging to the basic concern of the element to which they are attached or as belonging to the special-purpose concern that integrates tables in the query.

Another possibility is to make the non-local result responsible for finding the place where to integrate itself. This would lead to specification of the algorithm in a single construct. Unfortunately this is not possible because helpers can only be attached to source elements and target elements cannot traverse the source model.

Another subtle issue is scheduling. A scheduling issue arises when a non-local result has not yet been computed by a rule and another rule tries to obtain it. A proper scheduling algorithm is necessary to solve this problem. In ATL, the scheduling algorithm has two steps: the first step instantiates all the target model elements, the second step sets up all the target properties. The target model cannot be navigated. This ensures that the required information is always there. However, other transformation languages may have different scheduling algorithm that may pose problems.

4.3.3 Global-to-local Transformations

The problem of identification of bindings for unbound header attributes demonstrates a global-to-local transformation. The main challenge is to locate and query the required additional information. ATL presents a good mechanism to encapsulate this logic in a helper. However, the changes in the source structure leads to changes in the navigation logic. To minimize this effect a special type of queries was proposed in the context of Adaptive programming and Demeter method [18] called structure-shy queries.

A structure-shy query contains only the essential knowledge about the structure that is navigated. Irrelevant structural information is not included and therefore changes in it do not affect the query.

In our implementation the navigation and query language is OCL. OCL has not been designed with structure-shyness as a requirement so we cannot claim for it that it is structure-shy language as we can do for XPath, for example. However, it

contains some features that allow specification of such queries. Consider the navigation to all the assignments that can serve as bindings. Instead of traversing the whole expression tree we can locate them by the following query:

```
RelationalCalculus!Equal.allInstances()
```

This is an example of structure-shy query that remains unchanged when the structure definition of Boolean expressions changes.

Another point is how one can reuse the basic concern in another context to provide other query logic. A basic concern embeds the name of the helper function. So to reuse the basic concern one needs to implement that specific helper. This basic scheme is vulnerable to name clashes and prohibits the refinement of existing helpers.

5. RELATED WORK

Each contemporary language implementation technique offers built-in mechanisms to confine the impact of the implementation of G2L transformations and L2G transformations on the other language concerns. The mechanisms reduce the involvement of the other concerns, hereby improving the separation of the basic concerns. We refer to these mechanisms as language implementation *strategies*. The most prominent strategies are to the best of our knowledge propagation rules, attribute forwarding, monads, structure-shy queries, traversals, dynamic rewrite rules, and symbol tables. Let us briefly discuss these.

Attribute grammars [14] provide attribute propagation rules or attribute copy rules. To be able to communicate attribute values across larger tree fragments, all the intermediate terms must be explicitly aware of that value. This requires copy rules for each of the intermediate terms. In response to that problem, propagation rules have been proposed to alleviate the developer of the tedious copy rules [3].

Intentional Programming [23] provides attribute forwarding. The strategy was also added later on to attribute grammars. When an attribute is requested and it is not available in the term nor it is explicitly provided by the implementing concern, the request is forwarded to the results produced by the term's translational semantics. Similar mechanism is available in model transformation languages where a target element may be obtained from a given source on the base of the resolution algorithm in a way that is blind for the actual rule that produces the result.

Functional programming languages provide monads [25]. Monads allow the programmer to build computations using sequential building blocks, which can themselves be sequences of computations. The monad determines how combined computations form a new computation and frees the programmer from having to code the combination manually each time it is required.

Structure-shy queries stem from structure-shy paths of the Demeter method [18]. A structure shy query does not detail the actual path that must be followed to reach a distant term, it describes the path by a series of basic operators. The operators improve the SOC as they reduce the involvement of the terms between the requesting concern and the concerns that provide the information. To the best of our knowledge the need for structure shy queries is not studied in model transformation languages.

Rewrite rule formalisms provide traversals [26]. Traversals have been added to the rewrite rule paradigm to alleviate the programmer of the cumbersome programming needed to distribute context information. The techniques proposed in [26] allow a rewrite rule to descend into a subtree. During the descend, information can be accumulated and/or terms can be rewritten at an arbitrary depth. With traversals, the compilation of a language construct requesting information only needs to interact with the terms that influence the information. Dynamic rewrite rules are rules which are locally created during the execution of a rule. With these rules locally defined effects can be exported. However, these rules need to be carefully scheduled during the execution of the system.

Most language implementation techniques lack an explicit concept to define a language module except Intentional Programming and Delegating Compiler Objects [5]. The strategies to reduce the involvement of other concerns are mainly targeted to collect or retrieve information, not to distribute information. Hence, local-to-global transformations are not well supported. The dynamic rewrite rules is the sole strategy which offers direct support for local-to-global transformations.

6. CONCLUSIONS AND FUTURE WORK

In this paper we studied the applicability of model transformation languages to specify translational semantics for DSLs in the context of MDE. The major requirement for such a specification is to apply the principle of separation of concerns in order to achieve a modular and composable translational semantics specification.

The major contribution of this work is the study of the problems identified for Grammarware systems in the context of MDE and how model transformation languages can cope with them.

The specification of modular semantics is a topic extensively studied in the context of Grammarware systems. As the related work overview showed every language implementation technique proposes its own way to cope with the main challenges in this effort. We expect that the development of DSLs along the principles of MDE will become more and more a common task. Therefore, it is important to apply similar techniques for reusing and composing language modules in order to achieve fast DSL development.

General purpose model transformation languages are the main tool for specifying translations in MDE. The results of the presented case study revealed that the concepts in the problem domain of modular translational semantics and the available transformation language constructs demonstrate a significant mismatch. This leads us to the idea that the problem of translational semantics in MDE is better to be approached with a domain-specific transformation language instead of with a general purpose one. Gathering requirements for such a language and its initial prototyping is the major goal for future research. We are currently experimenting with a prototype of a transformation system based on meta-object protocol.

This work helps for getting an insight about the problems that must be solved by such a language. We have presented several concepts from the domain of translational semantics: basic and special-purpose concerns, local and non-local elements, which do not match the language constructs of the

transformation language. This prevents us from specifying clear interface of every concern i.e. a language module. Furthermore, we saw that the two major challenges are related to two classes of transformation problems: global-to-local and local-to-global transformations. For the first problem we need to investigate the need for structure-shy navigation and query languages over models. For the second problem we need to study various mechanisms that address the problem of composition of language modules.

Similarly to the existing Grammarware systems current transformation languages provide a fixed set of constructs that support separation and composition of concerns. It is clear that these constructs will work for a limited set of problems. To provide a more open and flexible approach for defining integration strategies we plan to investigate the applicability of reflection in model transformation languages. Two types of computational reflection are necessary: structural and behavioral reflection. Structural reflection allows strategies to reason about the models at a meta-level. Behavioral reflection allows strategies to reason and intervene on the semantics of the transformation. The subject of reflection and the details exposed to the user remain open issues.

The conclusions we draw are based on certain assumptions about the transformation languages that are used. However, there exist other types of languages not explored here, for example, graph transformation languages. To extend the scope of the results of our study it is necessary to perform experiments with other languages as well.

7. ACKNOWLEDGMENTS

This work has been partially supported by MODELPLEX IST Project FP6-IP 034081.

8. REFERENCES

[1] Agrawal, A., Karsai, G., Kalmar, Z., Neema, S., Shi, F., and Vizhanyo, A. *The Design of a Language for Model Transformations*, Journal of Software and System Modeling, 2005

[2] Alanen, M., Porres, I. *A Relation Between Context-Free Grammars and Meta Object Facility Metamodels*. Technical Report 606, Turku Centre for Computer Science, ISBN 952-12-1337-X, 2003

[3] Backhouse, K., de Moor, O., Swierstra, D. *First class attribute grammars*. Informatica: An International Journal of Computing and Informatics, 24(2):329–341, June 2000. Special Issue: Attribute grammars and Their Applications

[4] Balogh, A., Varro, D. *Advanced Model Transformation Language Constructs in the VIATRA2 Framework*, ACM SAC2006, Dijon, France, 2006

[5] Bosch, J. *Delegating Compiler Objects: Modularity and Reusability in Language Engineering*. Nordic Journal of Computing, N. 1, vol. 4, pp. 66-92, 1997

[6] Czarnecki, K., Helsen, S. Classification of model transformation approaches. OOPSLA2003 Workshop on Generative Techniques in the Context of MDA, Anaheim, CA, USA, 2003

[7] Dijkstra, E. *Executional Abstraction*. Prentice-Hall, 1976

[8] Gargantini, A., Riccobene, E., Scandurra, P. *Deriving a textual notation from a metamodel: an experience on bridging Modelware and Grammarware*. European workshop on Milestones, Models and Mappings for MDA. Bilbao, Spain, 2006

[9] GMT web site at Eclipse project. ATL Transformations: http://www.eclipse.org/gmt/atl/atlTransformations/

[10] Hursch, W., Lopes, C.V., *Separation of concerns.* Technical Report NU-CCS-95-03, College of Computer Science, Northeastern University, Boston, MA, February 1995.

[11] Jouault, F., and Kurtev, I., *Transforming Models with ATL*, In proceedings of Model Transformations in Practice Workshop, October 3rd 2005, part of the MoDELS 2005 Conference

[12] Jouault, F., Kurtev, I., Bezivin, J. *TCS: a DSL for the Specification of Textual Concrete Syntaxes in Model Engineering*. GPCE2006, Portland, Oregon, USA. To appear

[13] Klint, P., Lämmel, R. Kort, J., Klusener, S., Verhoef, C., Verhoeven, E.J. *Engineering of Grammarware*. http://www.cs.vu.nl/grammarware/

[14] Knuth, D. *Semantics of context-free languages*. Mathematical Systems Theory, 2(2):127–145, 1968.

[15] Kurtev, I., van den Berg, K., Jouault, F. *Evaluation of Rule-based Modularization in Model Transformation Languages illustrated with ATL*. ACM SAC 2006, Model Transformations Track, Dijon, France, April 2006

[16] Kurtev, I., Bezivin, J., Jouault, F., Valduriez. *Model-based DSL Frameworks.* OOPSLA Companion, Portland, Oregon, USA, 2006

[17] Lawley, M., Steel, J. *Practical Declarative Model Transformation with Tefkat*. MoDELS Satellite Events 2005: 139-150

[18] Lieberherr, K.J. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns.* PWS Publishing Company, 1996.

[19] OMG, MOF 2.0 *Query/Views/Transformations RFP*, OMG document ad/2002-04-10 (2002)

[20] OMG. *Object Constraint Language (OCL)*, OMG Document ptc/03-10-14

[21] OMG, *Revised Submission for MOF 2.0 Query/View/Transformations RFP* (ad/2002-04-10), OMG Document ad/2005-07-01 (2005)

[22] Parnas, D.L. *A technique for software module specification with examples*. Communications of the ACM, 15(5):330–336, May 1972.

[23] Simonyi. C. *The death of computer languages, the birth of Intentional Programming*. In The Future of Software, Proceedings of the Joint International Computers Limited/University of Newcastle Seminar, University of Newcastle, 1995.

[24] Taentzer, G., Ehrig, K., Guerra, E., de Lara, J., Lengyel, L., Levendovsky, T., Prange, U., Varro, D., and Varro-Gyapay, S., *Model Transformation by Graph Transformation: A Comparative Study*. In Proc. Workshop Model Transformation in Practice, Montego Bay, Jamaica, October 2005

[25] Wadler, P. *Comprehending Monads*. Mathematical Structures in Computer Science, 2(4), 1992. (Special issue

of selected papers from 6'th Conference on Lisp and Functional Programming.)

[26] van den Brand, M. G. J., Klint, P., Vinju, J.J. *Term rewriting with traversal functions*. ACM Trans. Softw. Eng. Methodol., 12(2):152–190, 2003.

[27] van Wijngaarden, J., Visser, E. *Program Transformation Mechanics*. Technical Report UU-CS-2003-048, Universiteit Utrecht, 2003.