Model-driven Tool Interoperability: an Application in Bug Tracking[1]

Marcos Didonet Del Fabro, Jean Bézivin, Patrick Valduriez

ATLAS Group, INRIA and LINA University of Nantes
2, rue de la Houssinière, BP 92208, 44322, Nantes cedex 3, France
{marcos.didonet-del-fabro, jean.bezivin}@univ-nantes.fr, patrick.valduriez@inria.fr

Abstract. Interoperability of heterogeneous data sources has been extensively studied in data integration applications. However, the increasing number of tools that produce data with very different formats, such as bug tracking, version control, etc., produces many different kinds of semantic heterogeneities. These semantic heterogeneities can be expressed as mappings between the tools metadata which describe the data manipulated by the tools. However, the semantics of complex mappings (*n:1*, *1:m* and *n:m* relationships) is hard to support. These mappings are usually directly coded in executable transformations using arithmetic expressions. And there is no mechanism to create and reuse complex mappings. In this paper we propose a novel approach to capture different kinds of complex mappings using correspondence models. The main advantage is to use high level specifications for the correspondence models that enable representing different kinds of mappings. The correspondence models may be used to automatically produce executable transformations. To validate our approach, we provide an experimentation with a real world scenario using bug tracking tools.

Keywords: complex mappings, semantic heterogeneities, tool interoperability, MDE (Model Driven Engineering)

1 Introduction

A software tool, e.g., text editing, bug tracking, needs to manipulate data that may be persistent (e.g., stored in a relational database) or transient (e.g., the execution state of the tool). Today, many different tools can be used to solve similar problems. As a result of increased collaboration between organizations and to rapidly changing environments, it is often necessary that one tool uses the data produced by another tool. However, the data produced by distinct tools are often heterogeneous with very different data formats, thus making tool data integration difficult.

The integration of heterogeneous data sources has been studied for a long time in data integration applications [21, 26, 1, 10]. In order to integrate the data of different tools, it is necessary to identify the semantic heterogeneities. The format and semantics of tool data is typically specified as tool metadata. Semantic heterogeneities

[1] This work is partially supported by ModelPlex project

can be expressed as mappings which specify the relationships between elements of tools metadata.

Many solutions have proposed different kinds of mappings ranging from 1-to-1 correspondences [26, 27, 24] to ontology bridges [22, 13, 11]. However, they typically provide a limited set of semantic relationships, e.g., equality and equivalence. They do not provide support to explicitly define the semantics of complex kinds of mappings such as mapping expressions. Mapping expressions are manipulations over tool elements that involve *1:m*, *n:1* or *n:m* relationships, e.g., splitting an element *Address* into *Street* and *Number*. Most solutions implement complex mappings directly in executable transformations using generic arithmetic expressions, e.g., *project_duration = end_date – start_date*, *name = first_name + last_name*. In this case, the semantics of the entire mapping (e.g., *"name concatenation"*) is not defined in the mapping specification, but in the mapping expression itself. Therefore, it is difficult to create and reuse these expressions. The lack of explicit semantics also hardens the task of deriving these mappings into executable transformations. The transformations are responsible to translate the data produced by a tool into a different format that can be understood and consumed by another tool.

In this paper, we propose a practical solution based on Model Driven Engineering together with data integration techniques. Our approach is useful to specify and capture complex semantic heterogeneities, and to automatically produce executable transformations. In our approach, the data manipulated by a tool is a model, called a tool model. A model conforms to a metamodel which is a formal description of the model.

We classify different kinds of tool semantic heterogeneities according to their complexity, and we propose a practical solution to express the mapping semantics in a correspondence metamodel, i.e., at the specification level. The metamodel elements are created with a vocabulary close to their semantic meanings, e.g., *override*, *concatenate*, *split*. A correspondence model conforming to this metamodel contains the mappings between the tool metamodels.

The correspondence models are used to generate executable transformations. A transformation is also a model, so the heterogeneities (e.g., mapping expressions) are translated into constructs of specific transformation languages, e.g., XSLT. We generalize the process of producing transformations into a pattern that is automatically executed. This pattern may be incrementally modified to handle different semantic heterogeneities. This is a frequently executed operation in model driven engineering which can be encapsulated in a *TransfGen* operation.

The main contributions of this paper are the following. First, we develop correspondence metamodel extensions that fully capture different kinds of semantic heterogeneities between tool metamodels. We emphasize the creation of complex mapping expressions. Second, we provide a generic pattern to automatically generate transformations based on correspondence models. Third, we consider all entities as models. This allows us to apply the same principles to manipulate the tool, correspondence and transformation models. To validate our approach, we provide an experimentation with a real world interoperability scenario using bug tracking tools and our AMW (Atlas Model Weaver) prototype [9].

This paper is organized as follows. Section 2 describes a motivating example in bug tracking tool interoperability. Section 3 presents the base concepts and the definition of tool correspondences. Section 4 explains how these correspondences are translated into a transformation model. Section 5 presents our experimental validation. Section 6 discusses related work. Section 7 concludes.

2 Motivating Example

We illustrate different kinds of semantic heterogeneities in tool interoperability with a bug tracking scenario. Bug tracking tools manage the bugs (reporting, fixing, etc.) of a given application. Today, many bug tracking tools are available, e.g., GNATS, Mantis, Bugzilla, and many others [15]. Consider two autonomous software development companies, C_A and C_B, and a set of N bug tracking tools. Company C_A uses tool T_i and company C_B uses tool T_j. They need to collaborate without aligning their software development practices. This is due to pragmatic reasons, e.g., the companies already participate in other cooperative projects.

We illustrate this situation using two bug tracking tools, Bugzilla [6] and Mantis [23]. Bugzilla is a general purpose, open source bug tracking tool. It provides features such as error tracking and quality assurance management. The metadata of Bugzilla is illustrated in Figure 1.

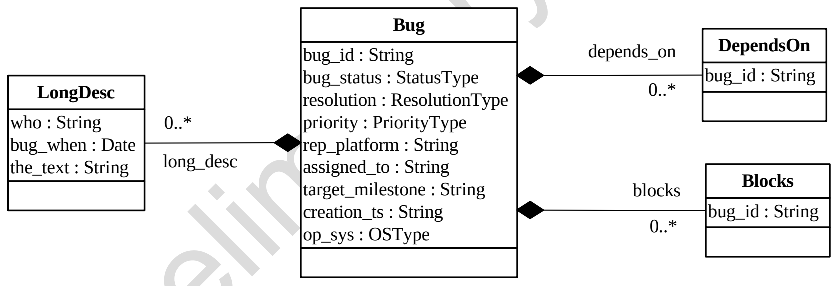

Fig. 1. Bugzilla metadata

Mantis is another bug tracking tool. It differs from Bugzilla as a light weight tool which allows adding new modules. The metadata of Mantis is illustrated in Figure 2.

We observe that it is possible to establish different kind of mappings between the elements of the tools metadata. The most common kind of mappings is equality, where two concepts are said to be equal. For example, a software bug is represented by *Bug* in Bugzilla and *Issue* in Mantis. As another example, the date a bug is created is represented by *creation_ts* and *date_submitted*. There are also elements representing equivalent data, but not the same, e.g., *target_milestone* is the version where a bug will be fixed, and *fixed_in_version* is the version where a bug was fixed.

There are also more complex kinds of mappings. For example, Bugzilla has two kinds of relationships between bugs: *depends_on* and *blocks*. In *Mantis*, bugs are related to each other using the element *relationships*, which points to *Relationship*. The relationship type is stored in the element *type*. As another example, *assigned_to* contains the responsible to solve a given bug in Bugzilla. In Mantis the relationship *assigned* points to element *Person* (that contains elements *login*, *value* and *id*).

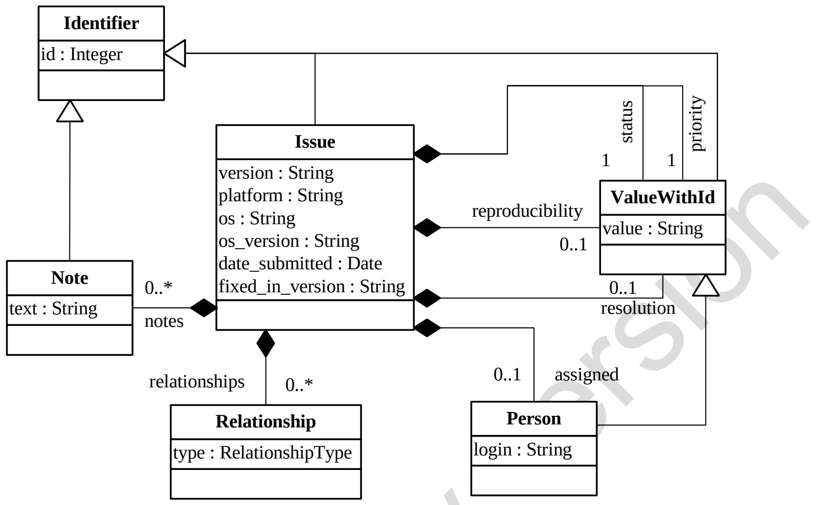

Fig. 2. Mantis metadata

In addition, there are semantic heterogeneities at the data level. For instance, the element *bug_status* in Bugzilla and relationship *status* in Mantis (that points to *ValueWithId*) contains the bug state (e.g., a bug was included in the database, a bug was solved, etc.), and the element *priority* contains the priority to solve a given bug (e.g., immediate, urgent). Each tool has its own set of status and priorities. For example, it is necessary to take into account that the *priority* with value "*P_1"* in *Bugzilla* is translated into the value "*urgent"* in Mantis*.* The same analogy applies to the element *status.* Different kinds of heterogeneities and the other elements not explained here are discussed later in the paper.

Traditional data integration applications usually create mappings to capture similarity heterogeneities (e.g., equality, equivalence). These mappings can be used to produce transformations that execute the translations from Bugzilla to Mantis. However, complex mapping expressions and data level heterogeneities are coded either in some element in the mappings, either in the produced transformations. For example the developer must code how to translate between the enumerations values in one specific language. The lack of explicit semantics for complex expressions hardens the creation of mappings because there is no domain information about the possible mappings. The possible mappings are virtually unlimited when using generic arithmetic expressions. This way is not possible to understand all the mappings without analyzing the entire expression in the produced transformations. This also reduces the reusability of these expressions. In addition, there is not enough semantic information to automatically produce the transformations, which is a frequently executed operation in model management. The mappings and produced transformations must be kept synchronized.

In order to efficiently achieve tool interoperability, all these kinds of mappings must be explicitly specified. These mappings must be derived into executable transformations. This process must be efficient, such that new transformations between other tools can be rapidly developed.

3 Tool Heterogeneity

In this section, we motivate the use of correspondence metamodel extensions to capture different kinds of tool semantic heterogeneities. First, we define what is a model which is the basic concept underlying our solution. Second, we define the tool heterogeneity problem and a core correspondence metamodel. Finally, we classify different kinds of tool semantic heterogeneities and we propose a set of metamodel extensions to express these heterogeneities.

3.1 Models

We abstract implementation and representation issues by using an integrated modeling platform. We present the model definition below (following [17]).

Definition 3.1 (Directed graph). A directed multigraph $G = (N_G, E_G, \Gamma_G, v)$ consists of a finite set of nodes N_G and a finite set of edges E_G, a mapping function $\Gamma_G : E_G \rightarrow N_G \times N_G$ and a labeling function $v : N_G \cup E_G \rightarrow A$. The type *A* is of any data type, such as characters, integers or classes.

Definition 3.2 (Model). A model $M = (G, \omega, \mu)$ is a triple where:

– $G = (N_G, E_G, \Gamma_G)$ is a directed multigraph,

– ω is itself a model (called the reference model of M) associated to a graph $G_\omega = (N_\omega, E_\omega, \Gamma_\omega)$,

– $\mu : N_G \cup E_G \rightarrow N_\omega$ is a function associating elements (nodes and edges) of G to nodes of G_ω.

The relation between a model and its reference model is called *conformance*. This definition allows an indefinite number of levels. However, we observe from different domains (XML, RDBMS, ontologies) that only three levels are needed. We call these three levels metametamodel (M3), metamodel (M2) and terminal model (M1). A *metametamodel* is a model that is its own reference model. A *metamodel* is a model such that its reference model is a metametamodel. A *terminal model* is a model such that its reference model is a metamodel.

3.2 Tool Heterogeneity

The tool data and metadata are represented as models and metamodels. Thus, the tool heterogeneities are expressed as mappings between tool metamodels. The mappings types are specified in a correspondence metamodel. We define tool, mappings and correspondence metamodel below.

Definition 3.3 (Tool). A tool T is a tuple $<M_t, S_t>$, where:

– $M_t = (G, MM_t, \mu)$ is the tool model. M_t is the data that is manipulated by T,

– MM_t is the reference model (metamodel) that represents the tool metadata,

– $S_t = \{s_i;\ i = [1..n]\}$ is the set of services (querying, updating, inserting, etc.) provided by T. Every service $s \in S_t$ must respect the constraints specified in MM_t.

Consider a bug tracking tool $T_a = <M_{ta}, S_{ta}>$. The metamodel MM_{ta} specifies how the bugs are organized, the properties of a bug, the possible states of a bug during its life cycle, etc. The model M_{ta} has the value of the bugs, e.g., that a given bug "B" has a status of "in correction" to a developer called "Joseph". The set S_{ta} contains miscellaneous services: the inclusion a new bug in the database, the update of a bug status, the query of a set of bugs, and so on.

Consider another bug tracking tool, $T_b = <M_{tb}, S_{tb}>$ with a different model, reference model and set of services. The semantic heterogeneities between metamodels MM_{ta} and MM_{tb} are expressed as mappings. The mappings between tool metamodels have different types, structures and semantics. However, intuitively, they depict the notion of typed-links between (meta) model elements.

Definition 3.4 (Mapping). Given two models M_{ta} and M_{tb}, a mapping *M* is a tuple $<S_a, S_b, T>$, where:

– S_a is a set of elements from the model M_{ta},

– S_b is a set of elements from the model M_{tb},

– T is the type of mapping between the sets S_a and S_b.

There are many different kinds of mappings, for instance *equality*, *equivalence* or, *generalization*. These are simple kind of mapping that express element similarity, usually denoting 1-to-1 links. Complex mappings have multiple cardinalities and semantic meaning. These kind of mappings abstract commonly used mapping expressions, e.g., the average between a set of elements, or the concatenation of strings. We specify the different mapping types in a correspondence metamodel.

Definition 3.5 (Correspondence metamodel). A correspondence metamodel is a model $M_C = (G_C, \omega_C, \mu_C)$ that define mapping types, such that:

– G_C has two basic types of nodes: *links* and *link endpoints*,

– *link* denote the mapping type, and refers to multiple *link endpoints,*

– *link endpoints* refer to the mapped elements.

Consider the mapping expression $t = s_1 + s_2 + s_3 + s_4 / 4$. The mapping language contains the addition and subtraction operators, plus the tokens (the model elements). The language does not explicitly specify that it is possible to create average expressions. The semantic is only known if we analyze the expression itself. In our solution, we create a link type *average* that abstracts the semantics provided by the combination of operations "+" and "/". This process is the *promotion* of the mapping semantics into the correspondence metamodel. The link type refers to a link endpoint with cardinality N (the source elements), and to a link endpoint with cardinality 1 (the target element). The mapping expression (the link between the elements) is created in a correspondence model conforming to the correspondence metamodel.

3.2.1 Core correspondence Metamodel

We create a core correspondence metamodel based on Definition 3.5. The metamodel is illustrated in Figure 3. The core metamodel has elements with information about link type, link endpoints and element identification. Element identification is a practical solution for saving unique identifiers for the linked elements.

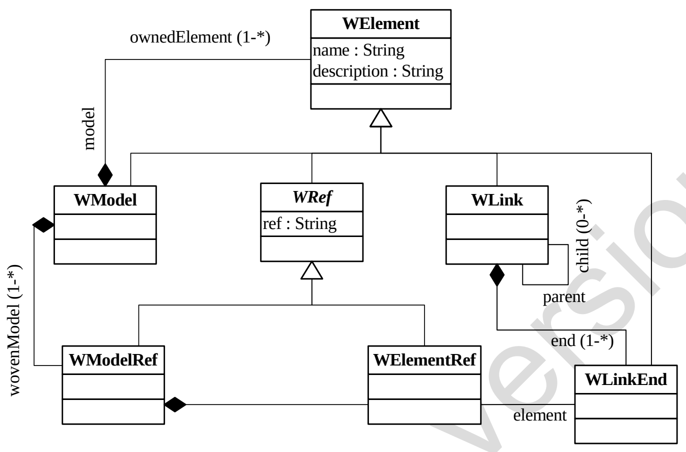

Fig. 3. The core correspondence metamodel

WElement is the base element from which all other elements inherit. It has a name and a description. *WModel* represents the root element that contains all model elements. *WLink* denotes the link type. *WLink* has a reference *end* to associate it with a set of link endpoints (*WLinkEnd*). *WLink* can have children links (*child* reference). Every *WLinkEnd* references one *WElementRef*. The attribute *ref* contains a unique identifier of the linked elements. *WElementRef* is not referenced directly by *WLink* because it is possible to refer to the same model element by different link endpoints, e.g., one model element may participate in more than one mapping expression. *WModelRef* is similar to *WElementRef*, but it contains references to the models as a whole. The *WLink* element must be extended to create different link types, e.g., equality, average and others. Different link types and link endpoints are added using metamodel extensions [9].

3.3 Metamodel Extensions for Tool Interoperability

As already stated, it is not possible to create a metamodel containing all types of links for tool interoperability. We propose to create metamodel extensions to capture different types of links. We classify them in three major groups according to the complexity of the links semantics.

The link types are defined in a simplified version of the KM3 metametamodel (see the complete syntax in [17]). KM3 is formed by classes. Classes may inherit from other classes, and are formed by attributes and references (attributes and references have a type and a cardinality). The syntax of KM3 is similar to object notations.

3.3.1 Similarity Expressions

Similarity expressions represent resemblance links between metamodel elements. These expressions are the link types encountered in most semantic-based mapping solutions. There are different kinds of similarity expressions. We describe them below.

Equality: a pair of element represents exactly the same information. For example the platform of the application the bug was detected is represented by *rep_platform in Bugzilla and platform* in Mantis. The link type does not specify the exact data type (*String*, *Class*, etc.). The data type is specified when deploying the solution (as extensions of *WLinkEnd*).

class *Equal* ***extends*** *WLink {* ***ref*** *source : <DataType>;* ***ref*** *target : <DataType> }*

Equivalence: the linked elements contain similar information, but not exactly the same. However, the translation semantics may be the same as in equality links, i.e., one target element receives the value of a source element. We add a description attribute to provide additional information about the equivalence, and a similarity measure.

class *Equivalence* ***extends*** *WLink {****ref*** *source:<DataType>;*
ref *target: <DataType>;* ***attr*** *description : String;* ***attr*** *similarity : Integer}*

Disjointness: two elements cannot be present at the same time because they have incompatible data. The link type also contains a description.

class *Disjoint* ***extends*** *WLink {****ref*** *source:<DataType>;*
ref *target : <DataType> ;* ***attr*** *description : String }*

Generality: one model element is more general than the other.

class *Inherit* ***extends*** *WLink {* ***ref*** *parent : <DataType>;* ***ref*** *child : <DataType>}*

Non equivalence: it is not always possible to translate all the information produced by one tool into another tool. Some elements from the tool metamodels do not have any semantic relationship, or are not relevant for a given translation and do not need to be generated. The element may be simply ignored. However, it is important for the application developer to be aware of what is not translated. For example the *reproducibility* element contains the frequency of reproduction of a given issue. This element does not exist in Bugzilla.

class *Unique* ***extends*** *WLink {* ***ref*** *element : <DataType> }*

3.3.2 Mapping Expressions

Mappings expressions are mappings that involve a set of source elements and a set of target elements. The definition of high level mapping expressions capable of capturing different kinds of semantic heterogeneities is a main contribution of this paper. The correspondence metamodel encapsulates mapping expressions in metamodel elements.

Many-to-one: many-to-one expressions links set of elements of the source model with a single target element. For example the elements *os* and *os_version* from Mantis contains the operating system and the operating system version. In Bugzilla, this information is available in one single attribute *op_sys*.

class *ManyToOne* ***extends*** *WLink {*
ref *source [*]: <DataType>;* ***ref*** *target: <DataType>}*

Split or one-to-many: the opposite of many-to-one expressions, i.e., split expressions link more than one target element with a single source element.

class *OneToMany* ***extends*** *WLink {*
ref *source: <DataType>;* ***ref*** *[*] target: <DataType>}*

Many-to-many: links a set of elements of source models with a set of elements of target models, for instance the reorganization of the elements of *LongDesc* into the elements of *Note*.

class *ManyToMany* ***extends*** *WLink {*
ref *source [*]: <DataType>;* ***ref*** *[*] target: <DataType>}*

New values in the target: generates values in the target model that do not have a correspondence in the source model. The values are automatically generated (e.g., to automatic generate IDs elements) or take a predefined value from user input.

The class *AutoSetValue* is extended into *AutomaticGenInt* and *ManualInput.* The class *AutomaticGenInt* reads the element that is referred by the *target* reference and generates a random number for it. The class *ManualInput* sets the *target* reference attribute with the value of *sourceValue*.

class *AutoSetValue* ***extends*** *Equal {}*
class *AutomaticGenInt* ***extends*** *AutoSetValue {}*
class *ManualInput* ***extends*** *AutoSetValue {* **attr** *sourceValue : <DataType>}*

3.3.3 Data Value Expressions

Data value expressions differ from mapping expressions because they also evaluate the model elements, not only the metamodel elements. Data value expressions modify the source model values to make them compatible with the target model.

The class *DataExpression* refers to a set of value equivalences. The *source* element is evaluated, and if it is equal to one *sourceValue* from the set of equivalences, it sets the target element with the corresponding *targetValue.* The equivalences may be of any data type.

class *DataExpression* ***extends*** *WLink {* ***ref*** *equiv [*] : Equivalence}*
class *Equivalence* ***extends*** *WLinkEnd {*
ref *sourceValue : <DataType>;* ***ref*** *targetValue : <DataType> }*

We illustrate data value expressions with the *resolution* element. The resolution contains the correction status of a bug (e.g., if it was fixed or not). In Mantis, this element may have the values: *OPEN, FIXED, REOPENED*. The possible values in Bugzilla are: *NEW, FIXED, INVALID, WONTFIX.*

4 Interpreting Tool Heterogeneity

In the previous section, we explained how to define different metamodel extensions to capture semantic heterogeneities. The next step is to create a correspondence model conforming to these extensions and to derive the model into executable transformations. These transformations translate one tool model into another.

In this section, we first introduce a match operation that creates a correspondence model. Then, we present a generic pattern used to automatically produce a model transformation, which is responsible to translate one tool model into another.

4.1 Match Operation

The match operation creates a correspondence model conforming to an extended correspondence metamodel. The match operation is divided into an automatic and a manual phase. The automatic phase executes a set of matching algorithms to search for similar concepts in the tool metamodels. First, we generate a correspondence model containing the cross product of all metamodel elements of each tool. Then, a set of matching algorithms is executed sequentially to calculate a similarity value for every pair of elements. We use existing string comparison algorithms [7], e.g., Levenshtein distance, edit distance and QGrams, and an adaptation of the similarity flooding algorithm [24]. We match only classes, attributes and references. The result is filtered to obtain only the best similarity values, based on a similarity threshold. The correspondence model contains only equality mappings.

The manual phase refines the correspondence metamodel by deleting wrong equality matchings and by adding the complex mapping expressions and data value expressions. This operation is done with the help of a user interface.

4.2 Generic Transformation Pattern

The definition of the generic transformation pattern relies on three facts. First, the core correspondence metamodel is formed by links, link endpoints and extensions of these elements. Second, declarative transformations languages have similar structure. Third, we use declarative transformation patterns that specify only what to transform, and not how to transform. The transformation pattern contains the execution semantics of the correspondence model, because it transforms the different types of links into executable mapping expressions in some transformation language.

We use higher-order transformations (HOT) to specify the generic pattern. A HOT takes as input a correspondence model conforming to an extension of the correspondence metamodel and transforms it into a transformation model.

Definition 4.1 (Higher-order transformation). A higher-order transformation is a transformation $T_{HOT} : T_{IN} \rightarrow T_{OUT}$, such that the input and/or the output models are transformation models. Higher-order transformations either take a transformation model as input, either produce a transformation model as output, or both.

We create a simple syntax for a transformation metamodel to define the generic pattern (as illustrated in Figure 4). The keywords are in bold font. The transformation has a set of declarative rules. The *input* element matches the input correspondence metamodels. The *output* element creates a new element in the output model. The output element has *bindings* to assign the source values to the target elements. The correspondence metamodel has one extension of *WLink* (as shown below) to denote source and target elements. The pattern can also be used with different metamodel extensions.

class *WLinkST* ***extends*** *WLink {* ***ref*** *source : WLinkEnd;* ***ref*** *target : WLinkEnd }*

The pattern contains four rules (see Figure 4). The rule *newRule* creates transformation rules. The rule *newOutput* creates the output elements. Both are based on the value of the *target* reference of a given link. The rule *newInput* creates the input element, and it is based in the value of the *source* reference. This rule may have a filtering condition depending on the link type. The rule *newExpression* creates different mapping expressions. The mapping expressions are created as bindings to the output elements.

```
Module TransfGen (C: $\omega_C$)
inputModel:  C: $\omega_C$ /* a correspondence model conforming to a metamodel $\omega_C$*/
outputModel: T: $\omega_T$ /* a transformation model conforming to $\omega_T$ */
rule newRule
  input  WLinkST (parent isA WModel)  /*classifiers (classes, references, attributes)*/
  output Rule
        input ← source
        output ← target
rule newInput
  input WLinkEnd (link.source = self)
  output InputElement
       element ← getElement (element.ref)
       condition ← /*depends on the WLinkST and WLinkEnd types*/
rule newOutput
  input WLinkEnd (link.target = self)
  output OutputElement
       element ← getElement (element.ref)
       bindings ← link.child /*get the sibling WLinkEnd*/
rule newExpression
  input WLinkST (parent isA WLinkST)
   output Binding
       source ← MapExp (getElement (source.element.ref) ) /*mapping expressions here,*/
       target  ←  getElement (target.element.ref)              /*according to the WLinkST type*/
```

Fig. 4. Higher-order transformation pattern

This pattern is the basis to define a new model management operation called *TransfGen.* This way it is possible to separate the tool interoperability process into distinct operations. The correspondence model is created by a *Match*. The correspondence model is translated into a transformation model using *TransfGen*. The translation between models are encapsulated in automatically generated transformations, which are themselves specific data transformation operations.

5 Experimental Validation

In this section we validate our approach with experiments using the bug tracking tools from the motivating example, Mantis and Bugzilla. The experiments are conducted using our model management platform, which is composed by different plugins to manipulate models. The two plugins used are the AMW (ATLAS Model Weaver) plugin [9] and the ATL (ATLAS Transformation Language) plugin [16]. AMW is responsible for managing the metamodel extensions and for the manual match. ATL is used to implement all the model transformations of the process. ATL has a textual concrete syntax and an engine to execute the transformations. Both plugins are open source and are available as Eclipse subprojects [2, 3].

We first show the creation of a correspondence model based on the correspondence metamodel extensions from Section 3. Then we demonstrate how we use the generic transformation pattern to interpret the correspondence model and to automatically produce model transformations. We end with a discussion about our results.

5.1 Correspondence Model

The metamodels of both tools are stored in different data sources. The tool metamodels are originally in SQL-DDL. They are translated into Ecore [12], which is the metametamodel used by our plugins (AMW and ATL). The semantics of Ecore is very close to KM3. This allows us to write metamodels using KM3 textual syntax. One part of the metamodels is illustrated in the graphical concrete syntax in the motivating example. The Bugzilla metamodel has 146 elements. The Mantis metamodel has 62 elements.

We implement the metamodel extensions defined in Section 3. We show below an excerpt of the correspondence metamodel. It specifies a data value expression used to translate enumeration values. It compares the value of given *source* element with the set of *sourceValue*, and sets the *target* element with the corresponding *targetValue*.

class *EnumerationEquiv* ***extends*** *DataExpression {****ref*** *equiv [*] : EnumEqual};*
class *EnumEqual* ***extends*** *Equivalence {*
    ref *sourceValue: String;* ***ref*** *targetValue: String };*

We create the correspondence model using ATL transformations to execute the sequence of matching algorithms, which refine the initial input (the cross-product of elements) and generate a correspondence model. Our AMW plugin is used to generate the interoperability metamodel based on a set of extensions and to refine the correspondence model during the manual phase.

An excerpt of the correspondence model is shown in Figure 5. We use a human readable syntax to represent information models, similar to HUTN [29].

```
EnumerationEquiv = {
    source.ref = Left.priority.id;
    target.ref = Right.priority.id;
    equivalence = { source = "NONE";  target = "pt_null"};
    equivalence = { source = "low";  target = "pt_P1"};
};
Left = {
    name ="Mantis";
    ref = "c:\Tool_interoperability\Mantis.ecore";
    priority {  id = "EAttribute_priority";   }
};
Right = {
   name ="Bugzilla";
   ref = "c:\Tool_interoperability\Bugzilla.ecore";
    priority {   id = "EAttribute_priority";   }
}
```

Fig. 5. A correspondence model

The model contains the equivalencies between the *priority* values. Note that both tool models have a priority property and both have the same ID “EAttribute_priority”. This does not cause problems because it is relative to the containing model.

The complete correspondence model has 312 elements. This difference on the number of elements is due to the structure of the correspondence metamodel, because for every couple of referred elements there is at least one element indicating the link type, plus the source and target elements. In addition, the source and target elements refer to an element that contains their identifiers (in the *Left* and *Right* elements).

5.2 Interpreting the Correspondence Model

The execution semantics of the correspondence model is specified in a transformation that takes the correspondence model as input and produces a transformation model as output. The transformation (485 lines) is implemented based on the generic transformation pattern. The ATL transformation rules are divided in three parts: the *from* block filters the appropriated model elements by their type; the *to* block contains the declarative code; the *do* block contains imperative code. We show in Figure 6 the rules that interpret the metamodel extension to translate the enumeration values. The AMW identifier denotes the correspondence metamodel. The ATL identifier denotes the transformation metamodel.

```
rule EnumDataTranslation {
  from amw : AMW!EnumerationEquiv
  to atl : ATL!Binding (
        propertyName <- MOF!EClassifier.getInstanceById(amw.target.element.ref).name
    )
  do { atl.value <- thisModule.CreateEnum(amw, amw.enumEqual);}
}
rule CreateEnum(amw: AMW!EnumerationEquiv, attrEnum: Sequence (AMW!EnumEqual)){
  to ifExp : ATL!IfExp (
          thenExpression <- targetEnum,
          condition <- operation
        ),
        operation : ATL!OperatorCallExp (
          operationName <- '=',
          arguments <- sourceEnum
        ),
        endExp : ATL!StringExp (),
        sourceEnum: ATL!StringExp (
          stringSymbol <- attrEnum->first().sourceValue.toString()
        ),
        targetEnum : ATL!StringExp (
          stringSymbol <- attrEnum->first().targetValue.toString()
        )
  do { operation.source <- amw, amw.source->collect(e | e.element.ref),true);
          if ( attrEnum->size() = 1 ) {
                ifExp.elseExpression <- endExp;
          } else {
                ifExp.elseExpression <- thisModule.CreateIfEnum(amw,
                                      attrEnum->subSequence(2,attrEnum->size()));
          }
  }
}
```

Fig. 6. Higher-order transformation

The rule *EnumDataTranslation* matches the element *EnumerationEquiv* from the correspondence model. It produces a *Binding* element conforming to the ATL metamodel. A binding has a *propertyName* that corresponds to the target model element. The target element is obtained by *getInstanceById* function. The property *value* calls the rule *CreateIfEnum.* It receives the set of enumerations as parameters and produces a model with a set of nested *IfExp* (conditional expressions).

The *IfExp* contains a *condition* expression, which is formed by an equality operator (*OperatorCallExp*). This operation compares the source value of the enumerations and sets the correct target value specified at *thenExpression*. The *StringExp* elements return the *sourceValue* and *targetValue* (an empty *String* if there is no equivalence). The complete transformation produces a transformation model with a set of rules. This model is extracted into a text representation that is executed in the ATL engine.

5.3 Discussion

The metamodel extensions enable producing a domain specific (tool interoperability) correspondence metamodel. Among the different metamodel extensions that are created, the most used are the concatenation of elements (e.g., *os* concatenated with *os_version*), data type conversions (e.g., *Integer* to *String,* references to attributes, etc.) and conversion of enumerations values.

One interesting observation is that the values of the enumerations from Mantis are not described in the metamodel, only in a Php file. Since the tool metamodels cannot be modified (otherwise the services provided might not work properly), the enumerations are added in one metamodel extension. This is a very specific extension, which is probably not useful outside the bug-tracking example, but it is still necessary to be able to create the output transformation.

The correspondence model has composite elements that conform to a combination of metamodel extensions. For instance we combine the conversion of "references to attributes" extension with the "concatenation" extension. This way, it is possible to create more complex output transformation models with the same set of extensions.

The metamodel extensions ease the task of repeatedly creating complex mapping and data value expressions between tool metamodels. The adaptive user interface is used together with semi-automatic matching algorithms (see the survey at [32]). The extensibility of the correspondence metamodel enables leaving human intervention essentially on the matching phase, because all the necessary information to produce transformations is available in the correspondence model. This is different from traditional approaches that have an extra step of mapping discovery [19, 26]. However, it is still possible that a correspondence metamodel covers most semantic interoperability cases, but not all. Complex expressions that are not often used can be coded manually in the final generated transformation.

The declarative structure of the correspondence metamodel allows a clear separation of the input model (the correspondence model) from the output model (a transformation model). Thus, it is relatively straightforward to modify only the output model and produce different transformation models. This also enables generating different expressions in the output transformation. For instance, the translation of enumeration values may be implemented as nested ifs (our final choice), or using *case*-like statements. This opens the possibility of optimizations of the output transformations (however, this is not the focus in this work). On the negative side, transformation languages may have complicated metamodels, in particular for querying and navigation expressions (e.g., OCL, XPath).

Another important result is that we are capable to use most of the metamodel extensions also in the importing process from SQL-DDL to an Ecore metamodel. The process is the following: we create a SQL-DDL metamodel conforming to Ecore (to support standard CREATE TABLE statements). The textual SQL-DDL is translated into a model conforming to the SQL-DDL metamodel. We then use most part of the correspondence metamodel extensions (excluding for instance the extensions concerning enumerations) to create a correspondence metamodel with AMW, and then a correspondence model to link the SQL-DDL model with a KM3 metamodel The translation from KM3 to Ecore is straightforward. The SQL-DDL model has 48 elements. The KM3 metamodel has 47. The correspondence model has 132. The output transformation has 83 lines. This transformation translates the SQL-DDL model into a KM3 model. In this case, extensions to generate default values are constantly used, because KM3 models have attributes such as *lower*, *upper* (for cardinality), *isAbstract*, that are not present in the SQL-DDL definition.

To summarize, our experiments demonstrate that the use of MDE enables to improve two data integration phases (matching and transformation production) to solve tool interoperability problems in a practical and efficient manner. We are able to define different extensions of the core correspondence metamodel to cope with distinct kinds of semantic heterogeneity. We create a correspondence model using some matching algorithms and a user interface. We implement the transformation pattern that automatically generates a transformation to transform the tool models.

6 Related Work

There has been extensive work on data integration that can be applied to tool interoperability. The usual approach is to identify the relationships between elements and to save these relationships in some kind of mapping. The most common mappings are 1-to-1 correspondence [1, 27, 26, 24]. These correspondences are not adapted to represent complex mappings semantics.

The use of model-based correspondences was introduced in [30]. The correspondence model is used to merge models. However, it has only equality and similarity link types. More expressive representations have been proposed to bridge between different ontologies [28, 22, 11]. These approaches have mappings as first class entities. The set of valid mapping constructs involve complex axioms, such as equivalence and generalization. The main limitation is that the fixed set of mapping constructs cannot be extended in a straightforward way as in our approach.

In our solution, we present a correspondence metamodel that is capable of capturing virtually all of the representations above, because the metamodel is extensible. This means we may specify a domain specific mapping with only 1-to-1 relationships until complex structures as in ontology-based approaches.

InfoQuilt [31] provides ways to represent mapping expressions through a library of mapping functions. However, the library can be used with no restriction, i.e., they are not separated by application domain. The functions are not part of the mapping definition, but expressions written in terms of the mapping language. The work in [18] presents a classification of the semantic and syntactic differences between schemas. This work proposes a *semPro* predicate to formalize the semantic proximity between elements. It is a formal work that focuses in the semantic heterogeneities, and not a complete integration platform as in our solution. It is a basis for our classification of tool heterogeneities, but we separate the heterogeneity types based on their complexity.

Our approach is complementary to existing matching algorithms, as we provide an efficient way to represent mapping expressions. For example the iMAP prototype [8] could be used to create a set of complex mapping expressions in our solution. iMAP implements different complex searchers. Every searcher could be associated with a correspondence metamodel extension.

The work in [14] proposes the alignment of ontologies based on the computation of similarities of 1-to-1 and 1-to-m mappings. The similarities are computed taking into account ontological structures. However, the similarities denote only equivalence mappings. The 1-to-m similarities could be used as input to algorithms that generate correspondence models with complex kinds of mappings.

The mappings are used to produce transformations. Clio [26] is one of the first solutions to provide a semi-automatic mechanism to produce transformations based on a set of correspondences. Our proposal has a similar architecture. However, Clio focuses on the generation of nested structures and on foreign key dependencies. There is no support for different kinds of complex mapping expressions. The work in [19] proposes an algorithm to generate XQuery. The algorithm uses 1-to-1 correspondences between a set of input XML schemas.

We differ from both approaches because we factor out part of the generation problem into a generic pattern. We leave the complexity of creating expressions to the matching phase, as in [8]. This means for instance that we do not implement a *chase* procedure to identify possible joins as in Clio. The generic pattern is independent of the structure of the input models (e.g., nested format), though still dependent of the core correspondence metamodel.

Model management solutions [5, 24, 4, 20] propose the creation of operations that encapsulate the most frequently executed metadata tasks. The work in [25] implements a model management platform using a logic mapping language. The logic language is translated into XSLT using an ad-hoc implementation. Our approach presents a model management solution focusing on the creation of element level constructs. The correspondence model as a whole acts as a high level specification for data integration operations.

To the best of our knowledge, none of the existing solutions consider the transformations and correspondences as models at the same time as in our approach. The model management operations may be applied to transformations as well. This enables using the declarative pattern to generate transformations from a correspondence model, and to encapsulate this pattern into a *TransfGen* operator.

7 Conclusions

In this paper, we have presented a practical and flexible approach that improves data integration techniques applied to tool interoperability problems. We based our solution on MDE principles to capture the semantic heterogeneities and to produce operational mappings between these tools.

Considering two tools in a set of tools dealing with the same problem domain (bug tracking in our case), the main problem is to deal with different kinds of semantic heterogeneities, in particular, complex heterogeneities that involve mapping expressions. After having provided a classification of semantic heterogeneities between tools, we have shown how this classification may be translated in various types of links defined in a correspondence metamodel. Furthermore, the correspondence metamodel may be seen as an extension of a core metamodel that provides basic support for link management. The main original aspect of our approach is to offer maximum extensibility to capture the semantic of complex mapping and data value expressions.

We have shown that metamodel extensions allow expressing the different kinds of semantic heterogeneities with a dedicated vocabulary and in a declarative way. Every domain specific metamodel prevents from developing a generic language (and not well adapted) without the capability to explicitly express the semantic heterogeneities.

The correspondence models conforming to these metamodel were used to produce transformations. We have shown that the correspondence model can be interpreted following a generic and declarative pattern. The semantic of this pattern is the basis for a novel model management operation called *TransfGen*. Based on this pattern, we were capable to develop higher-order transformations that automatically produced output transformation models. The transformations were generated automatically because we leave all the human intervention to the matching phase.

Finally, considering all entities as models (tools, correspondence and transformations) enabled to manipulate all of them using the same set of principles. The main principle is to define different types of domain models and to apply transformations between them. This was particularly useful when specifying the semantic heterogeneities and when translating a correspondence model into executable transformation models.

We validated our approach within our model management platform using AMW and ATL plugins. We developed a domain specific metamodel to solve a set of tool interoperability problems. We created metamodel extensions for mapping expressions, data value expressions, and for elements that do not have equivalencies. We applied our solution in bug tracking tools using a real world setting.

As future work, we plan to extend the correspondence metamodel for different tool interoperability scenarios. We envisage verifying if our techniques adapt well to create *ModelGen* [4] operations. We also plan to study how to adapt existing matching algorithms to automatically create complex mappings.

References

1. Abiteboul S, Cluet S, Milo T. Correspondence and Translation for Heterogeneous Data. In proc. of ICDT 1997, pp 351-363
2. AMW: The ATLAS Model Weaver. Ref. site: http://www.eclipse.org/gmt/amw, 06/2006
3. ATL: ATLAS Transformation Language. Ref. site: http://www.eclipse.org/gmt/atl, 06/2006
4. Atzeni P, Cappellari P, Bernstein P A. Model independent schema and data translation. In proc. of EDBT 2006, pp 368-385

5. Bernstein P A. Applying Model Management to Classical Meta Data Problems. In proc. of the 1st CIDR 2003, pp 209-220
6. Bugzilla Bug Tracking Tool. Reference site: http://www.bugzilla.org, 06/ 2006
7. Cohen W, Ravikumar P, Fienberg S E. A Comparison of String Distance Metrics for Name-Matching Tasks. In proc. of IIWeb 2003, pp 73-78
8. Dhamanka R, Lee Y, Doan A, Halevy A, Domingos P. iMAP: Discovering Complex Semantic Matches between Database Schemas.In proc. of SIGMOD 2004
9. Didonet Del Fabro M, Bézivin J, Jouault F, Valduriez P. Applying Generic Model Management to Data Mapping. In proc. of BDA 2005, Saint-Malo, France, pp 343-355
10. Doan A, Halevy A. Semantic Integration Research in the Database Community: A Brief Survey. AI Magazine, Special Issue on Semantic Integration, Spring 2005, pp 83-94
11. Ehrig M, Haase P, Hefke M, Stojanovic N. Similarity for Ontologies - A Comprehensive Framework. In proc. of ECIS 2005
12. EMF. Eclipse Modelling Framework. Reference site: http://www.eclipse.org/emf, 06/2006
13. Euzenat J. An API for Ontology Alignment. In proc. of ISWC 2004, pp 698-712
14. Euzenat J, Valtchev P. Similarity-based ontology alignment in OWL-Lite. In proc. of ECAI2004, pp 333-337, Valencia, Spain, August 2004
15. Flanakin M. Web Log. Comments and complaints on software and technology in general. Comparison: Web-based Tracker. 08/08/2005. http://geekswithblogs.net/flanakin/articles/CompareWebTrackers.aspx
16. Jouault F, Kurtev I. Transforming Models with ATL. In proc. of the Model Transformations in Practice Workshop at MoDELS 2005, Montego Bay, Jamaica, pp 128-138
17. Jouault F, Bézivin J. KM3: a DSL for Metamodel Specification. In proc. of 8th FMOODS, LNCS 4037, Bologna, Italy, 2006, pp 171-185
18. Kashyap V, Sheth A P. Semantic and Schematic Similarities Between Database Objects: A Context-Based Approach. VLDB J. 5(4): 276-304, 1996
19. Kedad Z, Xue X. Mapping discovery for XML data integration. In proc. of CoopIS 2005, Agia Napa, Cyprus, November 2005, pp 166-182
20. Kensche D, Quix C, Chatti M A, Jarke M. GeRoMe: A Generic Role Based Metamodel for Model Management. OTM Conferences (2) 2005, pp 1206-1224
21. Lenzerini M. Data Integration: A Theoretical Perspective. In PODS 2002. pp 233-246
22. Maedche A, Motik B, Silva N, Volz R. Mafra - a mapping framework for distributed ontologies. In proc. of EKAW 2002, pp 235-250
23. Mantis Bug Tracking System. Reference site: http://www.mantisbt.org/, 06/2006
24. Melnik, S. Generic Model Management: Concepts and Algorithms, Ph.D. Dissertation, University of Leipzig, Springer LNCS 2967, 2004
25. Melnik S, Bernstein P A, Halevy A, Rahm E. Supporting Executable Mappings in Model Management. In proc. of SIGMOD 2005, Maryland, US, pp 167-178
26. Miller R J, Hernandez M A, Haas L M, Yan L-L, Ho C T H, Fagin R, Popa L. The Clio Project: Managing Heterogeneity. In SIGMOD Record 30, 1, 2001, pp 78–83
27. Milo T, Zohar S. Using Schema Matching to Simplify Heterogeneous Data Translation. In proc. of VLDB 1998, pp 122-133
28. Mitra P, Wiederhold G, Kersten M. A graph-oriented model for articulation of ontology interdependencies. LNCS, 1777:86+, 2000
29. OMG (Object Management Group). Human Usable Textual Notation (HUTN) Specification, Final Adopted Specification. (ptc-02-12-01)
30. Pottinger R A, Bernstein P A. Merging Models Based on Given Correspondences. In proc. of VLDB 2003. Berlin, Germany, pp 862-873
31. Sheth A P, Thacker S, Patel S. Complex relationships and knowledge discovery support in the InfoQuilt system. VLDB Journal. 12(1): 2-27, 2003
32. Shvaiko P, Euzenat J. A Survey of Schema-Based Matching Approaches. Journal of Data Semantics IV: 146-171 (2005)