Chapter 1

Multi-level Transformation from Conceptual Models to Databases in ReMoDeL

Ahmad F. Subahi, Anthony J. H. Simons
Dept. Computer Science, University of Sheffield
{a.subahi, a.simons}@dcs.shef.ac.uk

This chapter presents a framework for model transformation, organised around Java agents. Internally, the agents are hierarchically composed to build each translation step, offering a fine-grained control over the transformation. Externally, a linear composition of translation steps is used to create a multi-level, forwards transformation from high-level models to executable code, in which intermediate representations reduce multi-language dependency. The approach is illustrated with a database generation example.

1.1 Introduction

Model transformation plays a significant role in supporting Model-Driven Engineering (MDE) tasks, such as refactoring and refining models (transformation, translation), and in synchronizing and weaving together different views of a system (merging) [1, 2]. Model transformation is defined as a process of converting one model into another using transformation rules acting on models at different levels of abstraction [3, 4]. A goal is to identify modular and reusable transformation components, which generalise some of the transformation tasks, but which can be deployed in different contexts, such that effort is not wastefully duplicated [2]. This is intended to lead to more comprehensible and maintainable transformation modules [5].

The composition of model transformations has emerged as an important research topic in its own right. Tasks include designing a suitable orchestrating mechanism for controlling the execution of the decomposed transformations, in order to produce a single consistent result [6]. This paper reviews different approaches to composing model transformations in section 1.2 contrasting these with our own approach, which embodies two kinds of composition. Firstly, our transformations are executed by collaborating Java agents, each with a focus on a different level of detail in the input models. Secondly, we demonstrate a linear composition of transformation stages, in a two-phase forward model transformation approach. Our general architecture for model transformation is described in section 1.3. To exemplify this architecture, a particular framework for generating databases from conceptual models is presented in 1.4 and is illustrated in section 1.5 with examples from a case study.

1.2 Composition of Model Transformations

The composition task is influenced by the diversity of models and transformations to be composed. For instance, one approach to composition simply chains together several predesigned model transformations, whether they are expressed in one, or several, languages and metamodels, as found in the ATL framework [7]; or whether they are executed by one, or several transformation tools, as in the UniTI unifying architecture [9]. These are heterogeneous approaches, which benefit chiefly by building on existing work, but may impose a high learning curve to master the different languages [7], or may introduce significant overheads in the lifting and grounding of bespoke models into, and from, the common translation framework [9].

In contrast, homogeneous compositional frameworks chain together several transformations within the same representational scheme. Hidaka et al. [2] develop a homogeneous compositional framework for graph-based model transformations, by extending the graph query language UnQL, which operates on graphs encoded in the JSON format. Transformation steps consist of extending, replacing or deleting subgraphs. Larger transformations integrate several transformation steps via intermediate graph representations. Our own approach is similar, to the extent that all models are built in a family of XML languages, developed according to different XML schemas. While the processing tasks are technically homogeneous, the different XML schemas may also express different metamodels, supporting endogeneous and exogeneous model transformations.

Composition is also influenced by what designers perceive to be the basic units of modularity. Typically, whole rules are chosen as the units of modularity, converting source to target elements in a single step. This is one of the perceived benefits of the declarative rule-based approach, in which a transformation may be codified in some universally valid and transparent way. However, when more complex problems are addressed, it has been found that single rules are not ideal units of modularity.

In situations where the transformation task is subtle, based on aspects that may be decomposed in multiple dimensions, the same conceptual transformation must be split over several rules [10], each one triggered by different aspects of the transformation task. This then increases the burden of rule ordering and synchronisation. Instead, it might be preferred to break transformation rules down into atomic CRUD actions that incrementally create, read, update or delete parts of the target model [11]. This is a fine-grained approach to modularity. Bespoke combinations of these atomic actions can be maintained in a single translation module, which helps to maintain the consistency and modifiability of the transformation as a whole, which is not easily accomplished in the whole-rule approach, since this splits a single complex mapping over many rules.

At the coarser end of the scale, it is sometimes desired to adapt complete sets of rules maintained in a module. A technique called module superimposition allows sets of rules to be merged, extended, or adapted [12, 13]. The merger of rules from a base and an extension set is conducted according to union with override, whereby named rules in the extension set replace those rules in the base set that have the same names. The resulting rule set is constructed dynamically, at load time. Changes to the behaviour of the rule set may be subtle, or quite far-reaching, since not only may the extra rules fire under different preconditions, and achieve different transformations, but they may assert different postconditions that affect the ordering of all other rules. Module superimposition has been implemented for the Atlas Transformation Language (ATL) [7] and the QVT Relations language [14].

The benefit of declarative rules is sometimes challenged by the indeterminacy of the rule-ordering. For this reason, our work adopts the direct manipulation approach [6, 3] to model transformation, using imperative Java programs to navigate and construct XML models. While Java could in general support any kind of imperative style, ranging from the clean to the highly obfuscated, we have found it useful to modularise transformations around hierarchies of Java agents, each concentrating on a particular level of detail in the source or target model. Agents delegate to collaborators to handle the next level of detail. Agents also extend and specialise the behaviour of abstract super-agents, which capture common aspects of each transformation. Transformations are modular in the individual methods of each agent, which are designed according to a regular and predictable scheme.

Generally speaking, our approach employs two strategies for composing transformations. Internally, transformation modules are built from agents, whose methods may perform fine-grained graph surgery on either source or target models, or both. This gives sufficient flexibility for complex transformations with in-place modification and rule-ordering issues. Composition and overriding of behaviours occurs naturally via inheritance in the object-oriented model, but with less risk of nondeterminism in the result, compared to module superimposition with rules. Externally, it is possible to compose transformations in a linear fashion, where the target model output by one transformation is the input source model for the next transformation. In general, we expect to perform multi-level translations, with possibly many intermediate representations.

1.3 ReMoDeL Project Architecture

The ReMoDeL project [15] aims to develop a working proof-of-concept for Model-Driven Engineering using simple, available technologies such as Java and XML. The acronym stands for Reusable Model Design Languages, reflecting the project's emphasis on developing families of related models for MDE. The expectation is that many different kinds of model will eventually be needed to achieve complete, automatic generation of software systems from models, including novel, intermediate kinds of model that fold together traditional software models, such as the views supported by UML diagrams.

Since the focus elsewhere in MDE research is more geared towards making the transformation rules explicit, our adoption of Java and XML may seem unusual. The reasons for choosing Java for the transformation technology are both practical and theoretical. They include that the language is well-known and does not impose any additional conceptual burden on developers. By contrast, a moderate to steep learning-curve is faced by users of bespoke rule-based transformation languages [7] or hybrid imperative-declarative languages [8]. There is a ready pool of trained developers who may immediately take on ReMoDeL projects. The other modular and decidable advantages of Java over production rule-based architectures were discussed in section 1.2.

The reasons for choosing XML for the model technology are more involved. Originally, we had expected to work in a uniform object-oriented framework, similar to Kermeta [8], in which the models were graphs, built in the same programming language. In practice, we found that metamodels and their Java code infrastructure underwent frequent evolution, as the real information needs for particular transformations became more apparent. So, the effort invested in developing early metamodel APIs in Java was often wasted. For this reason, we adopted XML as the *lingua franca* for expressing models, since the toolset for reading, writing and manipulating XML models could then remain constant, and changes only impacted the transformation algorithms, whose development had forced the need for change in the first place. Nonetheless, the style of XML chosen to express each model follows a conceptual metamodel, which evolves in step with changes made to models and eventually stabilises.

The ReMoDeL approach is based on a multi-layered, forwards-transformation methodology [6] leading from multiple abstract views, via intermediate representations, to concrete models and generated code. Various XML-based modelling languages are used to express different aspects of information systems, such as the high-level *Work Flow Graphs* (WFG) and *Database and Query* (DBQ) models [16, 17], or the low-level *Functional Programming* (FUN), *Structured Programming Language* (SPL) and *Object-Oriented Programming* (OOP) models [19, 16, 18].

One of the hallmarks of the ReMoDeL approach is the use of intermediate representations. We believe that constructing multi-layered transformations via intermediate models is critical to the success of the whole MDE enterprise. Forcing model transformation via intermediate representations improves the quality of a translation, by making each step explicit and controllable. It is easier to validate a small transformation step than a large one. It also reduces the number of different source-to-target mappings that will eventually be required (c.f. the EU language translation problem, where having a common *interlingua* would reduce the number of language-pairs). More importantly, it actively promotes the development of novel transformation algorithms, by revealing intermediate levels at which new constraints may usefully be brought to bear (c.f. the work of Marr in computer vision [21]). In general, a multi-layered approach provides better support for folding together different abstract views of a system, as translations progress towards the concrete. The translation rules must also fill in the missing design and implementation detail, using boilerplate coding strategies, which apply at later stages in a forwards-transformation architecture. The lowest-level models eventually contain complete, generic implementation detail, ready for automatic code generation on different platforms [15, 17].

The Java transformation architecture developed for ReMoDeL is simple and flexible. Every model manipulation is classified either as a kind of in-place *transformation* (endogeneous transformation), or as a kind of *translation* mapping from one metamodel to another (exogeneous transformation), or as the *generation* of executable code from a model. All ReMoDeL translation frameworks are derived from the three abstract base classes *AbstractTransformer*, *AbstractTranslator* and *AbstractGenerator*. Descendants of these classes are styled as agents, with the responsibility for handling a particular level of detail in a source model. Agents may delegate to further subcontractor agents, which deal with the next finer level of detail. Whereas transformers and generators act on a single model, translators act on a source and target model. Agents are constructed with their supplied models, and a back-reference to the parent agent which spawned them. A translation framework consists of a dynamically constructed tree of agents, which execute according to a common protocol (c.f. the *Command* pattern, [22]). Agents traverse XML models and dispatch internally to different methods according to the type of model-element visited (c.f. the *Visitor* pattern [22]). These private methods are obviously named according to the translation rule they implement.

1.4 Database Generation Framework

The database generation framework reported in this paper is one example of this approach, which transforms a high-level conceptual data schema into executable database scripts. The framework outwardly consists of a linear composition of two transformations (figure 1.1). The first step is a model-to-model translation from a conceptual data model, consisting of records and semantic relationships, into a low-level logical model, consisting of tables, both expressed in the DBQ language. This stage performs data normalisation and is common to all schema translations, being totally independent of any target language. The second step generates code in the particular idioms of any database scripting language, using MySQL for illustration, but we also remark on differences where the target for generation is the Oracle database.

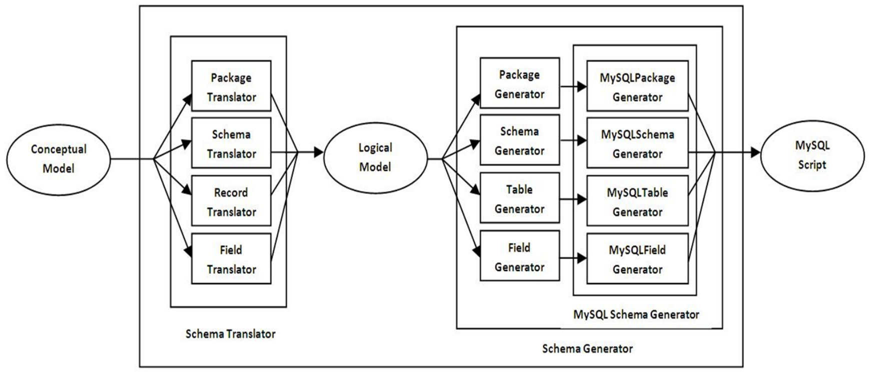

Figure 1.1: Composition of schema translation and database generation

Composition also plays a role internally. The initial translation step is performed by a hierarchical composition of Java agents, each concerned with a different level of detail in the source models (figure 1.2). Each translator class is constructed with *source* and *target* models and obeys a *translate()* protocol, acting on the pair of input and output models, and delegates to sub-translators that follow the same protocol, when the appropriate level of model detail is reached. The compositional hierarchy of the translators here follows the structure of the high-level DBQ concepts, namely, *PackageTranslator*, *SchemaTranslator*, *RecordTranslator*, and *FieldTranslator*. This decomposition improves the maintainability and modularity of the translation code. Translation rules are Java methods that explore the structure of the source model and perform surgery upon the target model (both expressed as graphs, encoded as XML trees). In this framework, the orchestration is controlled by *SchemaTranslator*, which determines a suitable order for handling the high-level DBQ model concepts.

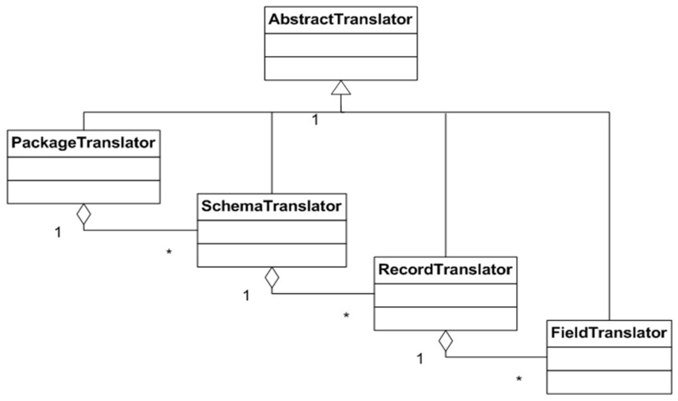

Figure 1.2: Internal composition of agents for the schema translation step

The architecture for the second database generation phase is similar, consisting of a compositional hierarchy of *PackageGenerator*, *SchemaGenerator*, *TableGenerator* and *FieldGenerator* agents. These classes are an abstract layer in a framework that is specialised for the different flavours of SQL required for different database engines. In this example, four specialised generators *MySQLPackageGenerator*, *MySQLSchemaGenerator*, *MySQLTableGenerator* and *MySQLFieldGenerator* are used to produce specific code for the MySQL database engine. These classes are constructed with a single *source* model and follow the protocol *generate()* and output database scripts in the MySQL language. The layer of abstract classes captures what is common in the synthesis of generic SQL, common to all database vendors. A different specialisation layer may be used to generate Oracle SQL. Further specialised layers may be provided for each new target database vendor [17].

1.5 Case Study: Online Ordering System

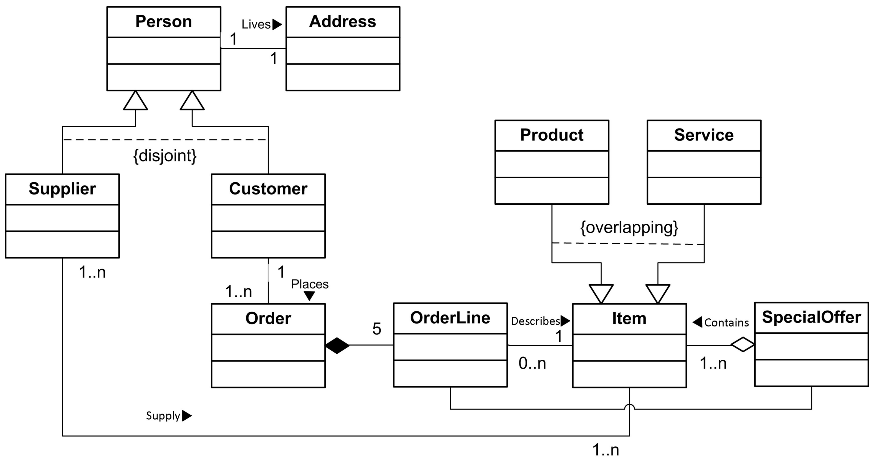

Figure 1.3: Conceptual schema for the *Online Ordering System*

This section presents a case study for model transformation, the *Online Ordering System*. The initial model is a fairly complex conceptual data schema, encoded in DBQ and visualised as the UML class diagram in figure 1.3. This model contains records, associations, generalisations and aggregation relationships, with the illustrated semantic refinements. The first model translation step maps from this to the logical data model illustrated in figure 1.4. This contains selectively normalised tables and fields, some of which are marked as primary keys.

In general terms, the translation rules map records in the source to tables in the tar-get having similar sets of fields. The mapping is not strictly one-to-one, since tables may be merged; and extra linker tables may be created. Semantic relationships in the source are rendered implicitly in the target as linked pairs of primary key and foreign key fields. Specific patterns are handled as follows.

Associations: Where records are in 1:1 association (*Person, Address* in figure 1.3), these are merged to satisfy 3NF; the retained concept acquires the renamed fields of the deleted concept. Where records are in 1:M association (*Customer, Order* in figure 1.3), this is translated into a foreign key (FK) on the many-side (*Order.customerID* in figure 1.4) relating to the primary key (PK) on the one-side (*Customer.personID* in figure 1.4). Where records are in M:M association (*Supplier, Item* in figure 1.3), a linker table is created for the whole association (*Supply* in figure 1.4), storing FKs for each related table (*Supply.dealer, Supply.item*). The linker table is named after the association, which is named in the source model. Where associations have their own fields (c.f. a UML association class), these are also promoted to tables in the target model.

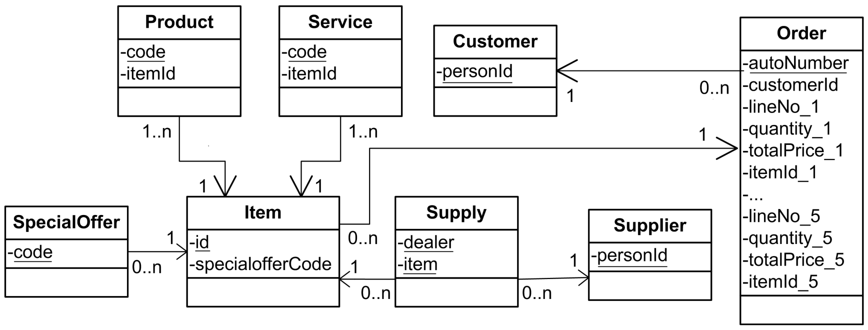

Figure 1.4: Logical schema for the *Online Ordering System*

Generalisations: Where subclasses are disjoint (*Supplier, Customer* in figure 1.3), tables are created only for the concrete subclasses; the fields of the abstract parent (*Person*) are replicated in each table and the parent is deleted. Where subclasses are overlapping (*Product, Service* in figure 1.3), tables are created for all records, preserving 3NF. Note how the rules make an intelligent decision about selective de-normalisation, for increased speed of retrieval (eliminating the need to join tables). So far, we have not yet explored a third "fat superclass" strategy for the overlapping case. The translation does not currently handle multiple inheritance.

Aggregations: A weak aggregation relationship (*SpecialOffer, Item* in figure 1.3) is treated like a 1:M association, inserting a FK in the component part (*Item.specialOfferCode* in figure 1.4) relating to a PK in the aggregate whole (*SpecialOffer.code*). Where a stronger composition relationship is indicated (*Order, OrderLine* in figure 1.3), the repeated fields of the part are indexed and merged with the whole (*Order*, in figure 1.4). This is another optimisation, to speed the retrieval of whole orders. Selective de-normalisation plays a key role in the automated transformation decision to achieve a reasonable balance between query complexity, system performance and disk space [17].

Figure 1.5: A fragment of the generated MySQL DDL, showing tables optimised by flattening generalisation and aggregation; and a trigger procedure enforcing a range constraint on the age of *Customer*

```
CREATE TABLE Customer (
  personId INT(7) NOT NULL,
  personForeName VARCHAR(10),
  personSurName VARCHAR(10),
  personAge INT DEFAULT 1,
  personAddressPostCode VARCHAR(7) UNIQUE,
  personAddressUnitNo INT(5) UNIQUE,
  personAddressStreet VARCHAR(30) UNIQUE,
  personAddressCity VARCHAR(20),
  id INT(7) UNIQUE,
  details VARCHAR(250),
  PRIMARY KEY(personId));

CREATE TABLE Order (
  autoNumber INT NOT NULL AUTO_INCREMENT,
  date Date,
  details VARCHAR(250),
  cusId INT(7) NOT NULL,
  lineNo_1 INT(10),
  quantity_1 INT DEFAULT 1,
  totalPrice_1 DOUBLE,
  itemId_1 INT(12) NOT NULL,
    ...
  PRIMARY KEY(autoNumber),
  FOREIGN KEY(custId) REFERENCES Customer(personId) ON DELETE CASCADE,
  FOREIGN KEY(itemId_1) REFERENCES Item(id) ON DELETE CASCADE);

CREATE TRIGGER customerCheck BEFORE INSERT ON Customer
FOR EACH ROW
  IF (NEW.personAge < 1 OR NEW.personAge > 99) THEN
    SET NEW.personAge = DEFAULT;
  END IF;
```

Key fields: Records may suggest candidate key fields using a tagged value from: *key* = {*total, partial, auto*}. A total field is used as the PK. The set of all partial fields is used as a compound PK. If no key field is marked, an auto PK is generated. PKs may be demoted to dependent unique fields by other rules; for example, when merging 1:1 associations, one PK is retained, the other demoted. Likewise, when flattening a disjoint generalisation, the parent's PK is retained, and the subclasses' keys are demoted. Again, in a composite aggregation, the key for the whole is retained and the keys of the enclosed parts are demoted. If an auto PK is ever demoted, it may be safely deleted. FKs are synthesised from the names of the association end-roles (elided in figure 1.3), or the associated records, or both. FK fields refer to their corresponding PKs in the target model (*Order* and *Item* have FKs in figure 1.4). A size threshold rule controls a further intelligent decision about when to replace a large compound FK by a simple FK, for the sake of optimising table join operations. In the related table, the compound PK is demoted, replaced by an auto PK.

The database generation step translates the logical model (figure 1.4) into database scripts, here into the MySQL data definition language (DDL). Part of the output file defining the *Customer* and *Order* tables is shown in (figure 1.5). Much of the target code follows directly from the logical model. The generic rules applied during generation seek to preserve semantic constraints, for example producing an *ON DELETE CASCADE* statement to maintain referential integrity (*Order*, in figure 1.5).

However, certain aspects must be handled specially by MySQL-specific generators. ReMoDeL DBQ supports fields with range constraints on their values. Whereas these may be translated directly into *CHECK* constraints in some flavours of SQL (such as the Oracle DDL), the target MySQL DDL did not support these. Instead, the generator emits *BEFORE INSERT* trigger procedures (from MySQL version 5) as an alternative way to perform data validation [20]. A range check for a *Customer*'s age is shown in figure 1.5. If validation fails, this field takes on a default value, defined in the DBQ model.

1.6 Implementation Details

Some examples of the DBQ models and transformations upon them are presented below. Initially, there are separate *Record* models for *Person* and *Address*, as illustrated in figure 1.6. However, these are later merged as a single *Table* by the rule that merges records that are in a 1:1 association.

The translation rule which performs this action is one of the methods of the *SchemaTranslator* agent, which is in charge of ordering the various transformations (figure 1.7). The task of merging 1:1 associations comes early in the ordering, before associations are mapped onto PKs and related FKs. The *SchemaTranslator* delegates to a *RecordTranslator* for each *Record* and *Association* found in the source model. The *RecordTranslator* is able to review all the associations in the original source model, to see if any tables need to be merged. The merging action is performed by one of its private translation

Figure 1.6: Examples of DBQ models used in the transformation step. This includes an intermediate *Person* table, which is eventually removed when its fields are flattened with *Customer* and *Supplier*

```
<Record name="Person">
 <Field name="id" type="Natural"  size="7" key="total"/>
 <Field name="foreName" type="String" size="10"/>
 <Field name="surName" type="String" size="10"/>
 <Field name="age" type="Natural" range="{0-120}" default="0"/>
</Record>

<Record name="Address">
 <Field name="postCode" type="String" size="7" key="partial"/>
 <Field name="unitNo" type="Natural" size="5" key="partial"/>
 <Field name="street" type="String" size="30" key="partial"/>
 <Field name="city" type="String" size="20"/>
</Record>

<Association name="Lives">
 <Role name="owner" type="Person" multiple="mandatory"/>
 <Role name="home" type="Address" multiple="mandatory"/>
</Association>

<!-- Person Table after merging 1:1 association -->
<Table name="Person">
   <Field name="id" type="Natural" size="7" key="total"/>
   <Field name="foreName" type="String" size="10"/>
   <Field name="surName" type="String" size="10"/>
   <Field name="age" type="Natural" range="{0-120}" default="0"/>
   <Field name="addressPostCode" type="String" size="7"unique="true"/>
   <Field name="addressUnitNo" type="Natural" size="5" unique="true"/>
   <Field name="addressStreet" type="String" size="30" unique="true"/>
   <Field name="addressCity" type="String" size="20"/>
</Table>
```

Figure 1.7: Fragments of the Java code for *SchemaTranslator* and *RecordTranslator*

```
public class SchemaTranslator extends AbstractTranslator {
 private Element target;

 public SchemaTranslator(Element source, PackageTranslator parent) {
   super(source, parent);
   target = new Element(source.getName());
 }

 public Element translate() throws TreeException {
   target.setValue("normal", "true");
   translateOneToOneAssoc();
   translateOnetoManyAssoc();
   translateFlattenRecord();
   translateManyToManyAssoc();
   return target;
 }
 ...
}

public class RecordTranslator extends AbstractTranslator {
...
 private void translateOneToOneAssoc() throws TreeException {
   Element major = getRoleType(getMajorRole(model));
   Element minor = getRoleType(getMinorRole(model));
   target = new Table(major.getName());
   for (Element field : major.getChildren("Field")) {
     target.addContent(field.clone());
   }
   for (Element field : minor.getChildren("Field")) {
     Element renamed = field.clone();
     renamed.setValue("name",
       mergeName(minor.getName(), renamed.getValue("name"));
     target.addContent(renamed);
   }
   ...
   getParent().addTable(target);
 }
...
}
```

methods (see figure 1.7). This makes a decision about the major and minor roles, then proceeds to create a new *Table* for the major role, merging the fields of the minor role, after renaming them, to avoid potential name clashes. The renaming algorithm uses the lower-cased name of the minor record as a prefix to the capitalised old field names. The method then removes both records from the source model, so that they are not visited again when translating *Record*s to *Tables* at a later stage in the algorithm.

1.7 Discussion and Conclusions

Unlike the declarative approaches of e.g. [2, 11, 14], all transformation rules in ReMoDeL are encoded as imperative methods of the agents in the transformation framework. This avoids problems of rule-ordering and non-deterministic firing [10], by performing an ordered sequence of transformations on XML trees. The private methods of each agent have access to a local portion of the source and target models, and, via their dominating parent agents, to more widely-scattered information. This solves some of the tangling/scattering issues discussed in [10] by providing explicit agent protocols to access non-local information (c.f. the *Law of Demeter*, [24]), where this is required.

Internally, the methods of each agent dispatch on the names of XML elements in the source, mimicking the type-dispatching in the *Visitor* pattern [22]. The imperative approach also supports construction of fine-grained transformations that modify models in-place, where required, with similar benefits to the composed CRUD actions in [11]. Method naming conventions ensure that all translation steps are modular and easily identified for maintenance purposes.

While we share the minimalist goals of SiTra [23], which is also Java-based, we do not construct explicit rule-objects to mimic the pattern-driven approach, but use internal dispatching on nodes, as described above. While SiTra uses pure Java for both its transformations and its models, we use XML for the models, since this allows models to evolve more rapidly, in step with the transformation algorithms, whose information requirements may frequently change, as they are being developed. XML is also the obvious *lingua franca* for input and output.

The two-phase linear composition of translation and generation steps illustrates the benefits of intermediate layers. Here, we can support generation of optimal code in different target DDLs from the intermediate model. Similarly, translation from flow diagrams to object-oriented code might need to go via intermediate structured programs [16]. In other respects, our external composition approach is homogeneous, c.f. [2], acting on uniform families of XML-based models, and does not require lifting and grounding to and from an abstract rule layer, as in UniTI [9].

Bibliography

[1] M. Biehl, Literature Study on Model Transformations, *Technical Report, ISSN 1400-1179*, Royal Institute of Technology, Sweden, 2010.

[2] S. Hidaka, Z. Hu, H. Kato, K. Nakano, Towards a compositional approach to model transformation for software development, *Proceedings 24th ACM Symposium on Applied Computing (SAC 09)*, Honolulu, Hawaii, USA, ACM Press, pp. 468–475, 2009.

[3] T. Mens, Model transformation: a survey of the state-of-the-art, in: S. Gerard, J.-P. Babau, J. Champeau (eds.), *Model Driven Engineering for Distributed Real-Time Embedded Systems*, Wiley, 2010.

[4] P. S. Kaliappan, State of the Art - Model Driven Architecture, *Technical Report*, Brandenburg Technical University, Cottbus, Germany, 2007.

[5] T. Mens, P. van Gorp, A taxonomy of model transformation and its application to graph transformation technology, *Proceedings International Workshop on Graph and Model Transformation*, Tallinn, Estonia, pp. 1–17, 2005.

[6] T. Mens, P. van Gorp, A taxonomy of model transformation, *Electronic Notes in Theoretical Computer Science, 152*, Elsevier, pp. 125–142, 2006.

[7] ATL Home Page, *http://eclipse.org/atl/*

[8] Kermeta Home Page, *http://www.kermeta.org/*

[9] B. Vanhoof, D. Ayed, S. V. Baelen, W. Joosen, W. Berbers, UniTI: a unified transformation infrastructure, in: G. Engels, B. Opdyke, D. C. Schmidt, F. Weil (eds.), *MoDELS 2007, LNCS vol. 4735*, Springer Verlag, pp. 31–45, 2007.

[10] I. Kurtev, K. Van den Berg, F. Joualt, Rule-based modularisation in model transformation languages illustrated with ATL, *Proceedings 21st ACM Symposium on Applied Computing (SAC 06)*, Dijon, France, ACM Press, pp. 1202–1209, 2006.

[11] A. Göknil, N. Y. Topaloglu, G. G. van den Berg, Operation composition in model transformations with complex source patterns, *Technical Report, No. 65119*, Centre for Telematics and Information Technology, University of Twente, Netherlands, 2008.

[12] D. Wagelaar, Composition techniques for rule-based model transformation languages, *Theory and Practice of Model Transformations*, pp. 152–167, 2008.

[13] D. Wagelaar, R. van der Straeten and D. Derrider, Module superimposition: a composition technique for rule-based transformation languages, *Software and Systems Modelling, 9 (3)*, Springer Verlag, pp. 285–309, 2010.

[14] QVT Relations in Alloy, *http://www.alloy.mit.edu/community/node/373/*

[15] ReMoDeL Home Page, *http://staffwww.dcs.shef.ac.uk/people/A.Simons/remodel/*

[16] U. Dojki, ReMoDeL Activity Workflow Models, *MSc Dissertation*, Department of Computer Science, University of Sheffield, 2011.

[17] A. F. Subahi, ReMoDeL Database Generator, *MSc Dissertation*, Department of Computer Science, University of Sheffield, 2010.

[18] A. J. H. Simons, ReMoDeL Object-Oriented Programming Model, Version 0.3, *Technical Report, 26 March*, Department of Computer Science, University of Sheffield, 2010.

[19] A. J. H. Simons, ReMoDeL Functional Programming Model, Version 0.4, *Technical Report, 31 March*, Department of Computer Science, University of Sheffield, 2011.

[20] MySQL Home Page, *http://www.mysql.com/*

[21] D. Marr, *Vision*, W. H. Freeman, 1982.

[22] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented software*, Addison Wesley, 1995.

[23] D. H. Akehurst, B. Bordbar, M. J. Evans, W. G. J. Howells, K. D. McDonald-Maier, SiTra: simple transformations in Java, in: O. Nierstrasz, J. Whittle, D. Harel, G. Reggio (eds.), *MoDELS 2006, LNCS vol. 4199*, Springer Verlag, pp. 351–364, 2006.

[24] K. Lieberherr, I. Holland, Assuring good style for object-oriented programs, *IEEE Software, September*, pp. 38–48, 1989.