Generation of Platform-Specific Model Transformation Plugins for EJB 3.0

András Balogh
Budapest University of Technology and Economics
Department of Measurement and Information Systems
H-1117 Budapest, Magyar tudósok körútja 2.
abalogh@mit.bme.hu

Gergely Varró
Budapest University of Technology and Economics
Dept. of Computer Science and Information Theory
H-1117 Budapest, Magyar tudósok körútja 2.
gervarro@cs.bme.hu

Dániel Varró
András Pataricza
Budapest University of Technology and Economics
Department of Measurement and Information Systems
H-1117 Budapest, Magyar tudósok körútja 2.
{varro,pataric}@mit.bme.hu

ABSTRACT

The current paper presents a technique for generating stand-alone model transformation plugins for the EJB 3.0 platform from platform-independent specifications of transformations given by a combination of graph transformation and abstract state machine rules (as used within the VIATRA2 framework). As a result, the design of transformations can be separated from the execution of transformations. This also enables to run platform-independent VIATRA2 transformations on very large models stored in underlying relational databases or to integrate such transformations into existing business applications.

Keywords: model transformation, platform-specific transformers, EJB 3.0, graph transformation, abstract state machines.

1. INTRODUCTION

Nowadays, the immense role of model transformation (MT) concepts and tools is unquestionable for the success of the Model Driven Architecture (MDA) [10]. Model transformations approaches should enable a cost and time efficient design of (i) manipulations within a single modeling language (or domain) (ii) mappings and synchronization between different modeling languages and/or source code (iii) semantic translations into various mathematical domains to carry out formal model analysis.

As model transformation is becoming an engineering discipline (*transware*), conceptual and tool support is necessitated for the entire life-cycle, i.e. the specification, design, execution, validation and maintenance of transformations. However, different phases of transformation design frequently set up conflicting requirements, and it is difficult to find the best compromise. For instance, the main driver in the execution phase is performance, therefore, a *compiled MT approach* (where a transformation is compiled directly into native source code) is advantageous. On the other hand, *interpreted MT approaches* (where transformations are available as models) have a clear advantage during the validation (e.g. by interactive simulation) or the maintenance phase due to their flexibility.

In the paper, we argue that advanced model transformation tools should support both interpreted and compiled approaches to separate the *design* (validation, maintenance) of a transformation from its *execution*. Interpreter-based *platform-independent transformers (PIT)* [3,16] ease the testing, debugging and validation of model transformations within a single transformation framework without relying on a highly optimized target transformation technology. *Platform-specific transformers (PST)* are compiled (in a complex model transformation and/or code generation step) from an already validated transformation specification into various underlying tools or platform-dependent transformation technologies (e.g. Java, XSLT, etc.) and integrated into off-the-shelf CASE tools as stand-alone plugins.

This current paper summarizes our experiments in generating platform-specific transformer plugins within the VIATRA2 model transformation framework.PIT model transformations are captured in VIATRA2 by a high-level, rule and pattern based paradigm offered by the combination of *graph transformation (GT)* [5] and *abstract state machines (ASM)* [4]. For the first target technology of stand-alone transformer plugins, we have chosen the evolving Enterprise Java Beans 3.0 (shortly, EJB3) standard [12] with tool support provided by JBoss 4.0.1. Our techniques will be demonstrated on the standard model transformation benchmark of the object-relational mapping (see e.g. [13]), which generates a relational database model from UML class diagrams. Furthermore, we compare the performance of EJB3-based transformation plugins to pure Java solutions.

2. OVERVIEW OF THE APPROACH

An overview of generating EJB3-specific transformer stand-alone plugins from VIATRA2 PITs is provided in Fig. 1. The general flow of developing and executing *platform-independent transformations (PIT)* in VIATRA2 is the following.

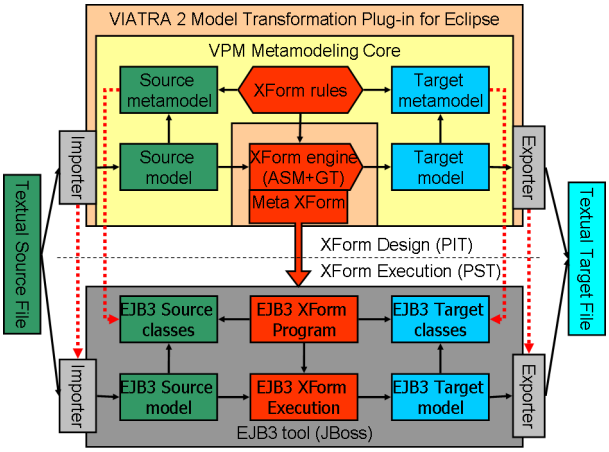

Figure 1: Overview of the plugin generation approach

1. **Metamodel design.** Metamodels of the source and target modeling languages (or domains) are designed and stored according to the VPM approach [15].

2. **Develop importers.** VIATRA2 accepts arbitrary textual source files by offering a flexible way to write model importers.

3. **Model import.** These importers build up an internal VPM representation of the source model which corresponds to its metamodel.

4. **Transformation design.** Model transformations between source and target metamodels are captured by a seamless integration of GT and ASM rules [14].

5. **Transformation execution.** These transformations are executed on the source model by a transformation engine to derive the target model.

6. **Model export.** Finally, the target model can be post-processed by special model transformations generating code and code formatters to be printed in the required output format.

The main goal of the paper is to show how stand-alone platform-specific transformers can be generated for EJB3 from platform-independent VIATRA2 models and transformations. We propose a solution for the following tasks.

- **From PIM to PSM models.** *EJB3 entity bean classes* will be generated from the source and target metamodels including the reference objects representing the mapping between them. The persistent storage of EJB3 entity beans will then be handled by the EJB3 application server.

- **From PIT to PST transformations.** *EJB3 session beans* will be generated from the PIT specification (i.e. from ASM and GT rules) in the form of fully functional EJB business methods. Transaction handling will be provided by the JBoss execution environment to prevent complex transformations from introducing conflicts when manipulating the model in parallel.

- **Reuse of model importers and exporters.** Our experience shows that the development of a model importer also requires significant work[1]. As a consequence, the reuse of VIATRA2 importers (exporters) implemented for building up (extracting) an internal (VPM) model representation from (to) a textual source (target) file in the EJB3-specific transformation plugin is also an important goal. This will be achieved by providing an alternative, EJB3-specific implementation of the model manipulation interface of the VIATRA2 metamodeling core, that will redirect and translate VIATRA2 -specific calls to EJB3-specific calls.

In the rest of the paper, we first provide a brief overview on the platform-independent modeling and transformation techniques of VIATRA2 (in Sec. 3). Then, the main part of the paper (in Sec. 4) discusses how to generate EJB-specific model transformation plugins from VIATRA2 (PIT-level) transformation specifications. A comparison with related approaches is given in Sec. 5, finally Sec. 6 concludes the paper.

3. MODELS AND TRANSFORMATIONS IN VIATRA2

We first briefly introduce the main notions of models and metamodels (as used in VIATRA2), and then how these models can be manipulated by using graph transformation (GT) and abstract state machine (ASM) rules.

3.1 Metamodels and models

In order to present the concepts of models, metamodels and transformations, a standard object relational mapping (see e.g. [13]) will be used throughout this paper as a running example, which generates a relational database schema from a UML class diagram.

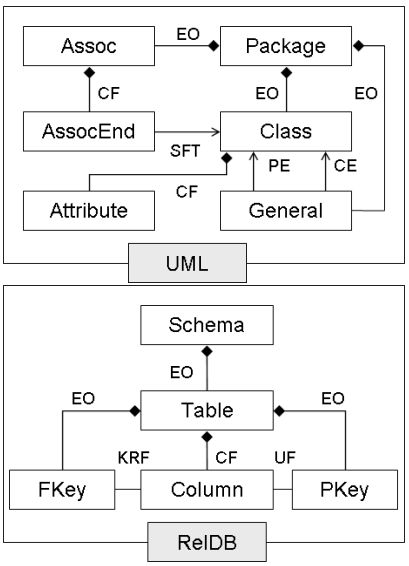

Figure 2: An extended metamodel for the object relational mapping

[1]VIATRA2 has importers for several industrial UML and BPM modeling tools.

A *metamodel* describes the abstract syntax of a modeling language. The metamodels of UML class diagrams and relational database schemas (following the CWM standard [9]) are depicted in Fig. 2 with minor simplifications. For the purpose of the current paper, we group elements of traditional MOF metamodels into three main categories: (i) *classes* (e.g. Schema, Table), (ii) attributes and *many-to-one associations* (like EO – elementOwnership, CF – classifier feature), and (iii) *many-to-many associations* (such as KRF – key relation feature) between two classes. For the paper, we assume that associations are navigable along both directions to make the discussion simpler.

Models are abstract representations of specific systems, and formally, they are instances of the metamodel. Model-level model elements in MOF are called *objects, slots* and *links*, which are instances of classes, attributes and associations, respectively.

3.1.0.1 Overview of VPM metamodeling..

VIATRA2 uses the non-standard VPM metamodeling approach [15] for uniformly representing models and metamodels. VPM is very succinct but more expressive than MOF; in fact, models taken from different technological spaces (and thus using different approaches for language design including, e.g. MOF, EMF, XML Schemas) can be integrated smoothly into the model space.

VPM uses a graph representation for storing models and metamodels, thus both classes and objects appear as *VPM entities* (nodes), while attributes and associations (as well as links and slots on the model-level) are represented as *VPM relations* (edges). Naturally, multiplicities, aggregation, etc. can be associated to VPM relations (used on the meta-level).

Inheritance may be defined between two entities (classes) as well as relations (associations), which denotes that the child model element can be used wherever the parent model element is expected.

The main essential difference between VPM and MOF metamodeling approach is the handling of *instantiation*. VPM uses a more flexible representation of metalevels than MOF by introducing *explicit instance-of relations*. Model-level constructs of MOF such as objects, links and slots are handled in VPM as ordinary entities and relations with an explicit instance-of relation between the metamodel and model constructs.

3.2 Graph transformation

Graph transformation [5] provides a pattern and rule based manipulation of graph models. Each rule application transforms a graph by replacing a part of it by another graph.

A *graph transformation rule* contains a left–hand side graph LHS, a right–hand side graph RHS, and negative application condition graph NAC. The LHS and the NAC graphs are together called the precondition PRE of the rule.

In the paper, we use the graphical representation initially introduced in [6] where the union of these graphs is presented. Elements to be deleted are marked by the del keyword, elements to be created are labelled by the add, while elements in the NAC graph are denoted by the neg keyword.

The *application* of a GT rule to an*host model* M replaces a matching of the LHS in M by an image of the RHS. This is performed by (i) finding a matching of LHS in M (by graph pattern matching), (ii) checking the negative application conditions NAC (which prohibit the presence of certain objects and links) (iii) removing a part of the model M that can be mapped to LHS but not to RHS yielding the context model, and (iv) gluing the context model with an image of the RHS by adding new objects and links (that can be mapped to the RHS but not to the LHS) obtaining the *derived model* M′. The *pattern matching phase* consists of Steps (i) and (ii), while the *manipulation phase* is constituted by steps (iii) and (iv).

Example 1. In Fig. 3, a graph transformation rule classR is depicted which carries out the transformation of UML classes to database tables.

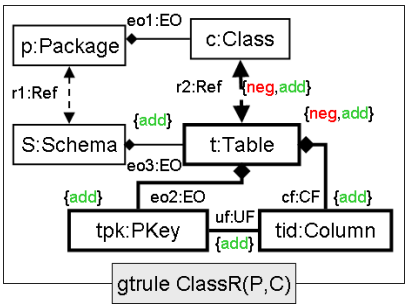

Figure 3: A sample graph transformation rule classR

As precondition for the rule application, the LHS prescribes that there exists a class c located in such a package p, which is already transformed into a corresponding schema s, while NAC prescribes that this class c is not allowed to be related to any table t. The rule manipulation part prescribes that a new table t is generated with a column tid and primary key constraint tpk as result of the rule application. Note also that package p and class c is passed to the GT rule as parameters to allow the same GT rule to be executed in different modes.

The entire object-relational mapping formalized as graph transformation rules can be found in [17].

3.3 Abstract state machines

While the elementary steps of complex model transformation is captured by GT rules, these GT rules are assembled into a complex transformation program by *abstract state machine* rules [4]. In order to semantically integrate the two paradigms, GT rules are treated as traditional ASM rules (called by the apply construct), and graph patterns (used in the LHS and NAC graphs) can be used as existentially quantified Boolean formulae in ASM conditions (find construct). More information on the semantic integration between graph transformation and abstract state machines can be found in [14].

ASMs provide complex model transformations with all the necessary control structures including the sequencing operator (seq), rule calls to other ASM rules (call), variable declarations and updates (let and update constructs) and if-then-else structures, non-deterministic selected (random) and executed rules (choose), iterative execution (applying a rule as long as possible), and the deterministic parallel rule application at all possible matchings (locations) satisfying a condition (forall).

Example 2. An extract from the ASM program driving the GT rules of the object-relational mappings is listed in Fig. 4 using a simplified VIATRA2 notation.

```
rule obj2rel() =
iterate seq {
  choose P with packageR.pre(P) do
  seq {
    apply packageR(P);
    forall C with classR.pre(P,C) do
    seq {
      apply classR(P,C);
      /* other rules applied here */
    }
  }
}
```

Figure 4: ASM program of the object-relational mapping

1. The transformation obj2rel first iterates over a sequence of steps. When the application any rules within the sequence fails, the execution of the iterate construct terminates.
2. The first step in this sequence non-deterministically selects a package P which is not yet processed and transforms it to a corresponding schema (packageR). The selection criteria for binding variable is prescribed by the precondition (LHS and NAC) of the GT rule.
3. Then classR is applied to all classes C within that package P as defined by (the precondition of) the rule classR (see Fig. 3).
4. Then the transformation proceeds with the handling of attributes, associations, generalizations, etc.

4. GENERATION OF EJB3-SPECIFIC MODEL TRANSFORMER PLUGINS

Now we discuss how to generate platform-specific transformation plugins from PIT transformations for the selected EJB3 platform.

4.1 From VIATRA2 models to EJB3 models

4.1.1 From platform-independent to platform-specific model representations

As discussed earlier, VIATRA2 uses a simple, generic representation for metamodels and models. While this formalism is suitable for the simultaneous, design-time representation of metamodels and models taken from multiple domains, it is not optimal from a performance perspective for the representation of a single (domain-specific) metamodel and its instance models.

Although VIATRA2 supports the definition of multiple meta levels, we restrict the PIT generation to two neighboring meta levels (e.g. the model-level and the meta-level). This is suitable for most practical MDA transformations such as PIM-to-PSM mappings, model analysis or code generation. We assume that the metamodels are fixed, and model manipulations are only carried out on model level.

This enables the generation of a static class structure in the target implementation platform (i.e. classes, methods, and attributes) from the meta-level elements to structures of the target implementation platform . The model-level elements will be instances (objects) of these classes as in any object-oriented language. This results in a more user-friendly structure tailored to the concepts of the specific metamodel, which allows also the easy integration of the generated class structure to existing user programs (e.g. by postprocessing the results of the transformation).

4.1.2 Enterprise Java Beans 3.0.

Enterprise Java Beans (EJB) is one of the most fundamental parts — the main business data layer and business functionality — of the Java 2 Enterprise Edition (J2EE) platform, which defines a layered architecture for scalable, distributed application development including several Java standards and APIs. After deployment, EJB beans are executed by an application server (also called container), which is responsible for efficiently providing many high-level services (such as transactions, security, persistence, etc.).

The main types of EJBs used in the current paper are the following.

- *Entity beans* are application-level, persistent data objects which are kept synchronized with an underlying relational database by means of an object-relational mapping. Entity beans can be in relationship with other entity beans referring to each other by direct references (many-to-one or one-to-one relationships) or typed collections (many-to-many, or one-to-many relationships). Relationships are also mapped automatically to relational database tables while providing a high-level, object-oriented API form manipulating models.
- *Session beans* implement the business logic of the application. There are two types of session beans: stateful and stateless. *Stateless session beans* can be considered as simple collections of business methods. Stateless session beans are typically pooled (i.e. the pre-instantiated objects are redistributed to client calls) by the application server to enhance performance. *Stateful session beans* contain an internal state that is maintained between method calls. This can be used for storing session-specific data on the server side.

Selecting an underlying technology as EJB 3.0 which has not yet reached the standard-level seems to be a risky choice at first sight. As a compensation, (i) the underlying EJB3 application server offers a robust solution for persisting models, handling transformations as transactions, and clustering for increasing performance, (ii) there is already a powerful open-source implementation as provided by JBoss 4.0.1 for EJB3, and (iii) the Java interfaces of EJB3 beans are very close to that of pure Java models (generated with the popular Eclipse Modeling Framework (EMF) [1] or similar). In fact, in our experiments (see Sec. 4.4), we compare the same transformation algorithm using an underlying EJB3 framework and a pure Java solution.

Finally, the reason for selecting EJB3 as the underlying platform instead of the currently industry-leading EJB 2.0 is merely the convenience as offered by (i) the introduction of annotations and templates in Java 1.5 and (ii) getting rid of the hard-to-maintain interfaces (local, remote, home, etc.) of EJB beans.

4.1.3 Mapping metamodels and transformations to EJBs.

As described earlier, the basic constructs of the metamodels has to be transformed to constructs of the target platform. In the case of EJB 3.0, these constructs are entity

beans and their relations. The mapping between metamodel elements and EJBs is the following:

- Metamodel classes (formally, VPM entities) are mapped to entity bean classes. The inheritance relations between entities are maintained.
- Metamodel associations and attributes (formally, VPM relations) with *at most one* multiplicities are mapped to scalar-type Java attributes in the entity bean that corresponds to the source of the relation. If the target of the function is a primitive type (for example, datatypes.String) the Java attribute will be the corresponding Java primitive type (java.lang.String). If the target of the function is a metamodel element, the mapping will create a one-to-one EJB relationship and the type of the attribute will be the target EJB class.
- Metamodel associations (formally, VPM relations) with *arbitrary (*)* multiplicities are mapped to one-to-many EJB relationships that are represented by a *Collection* attribute. For optimization purposes we also create an inverse, many-to-one relationship from the target of the VPM relation to the source.

Example 3. Below we show an extract of an EJB3 bean generated from the metamodels of Fig. 2.

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public class Package extends UMLElement
                  implements Serializable{
  // Attributes
  private String name;
  private SchemaElement schema;
  private Collection<Class> ownedElements;
  // Accessors
  @OneToOne
  public SchemaElement getSchema() {
    return schema;
  }
  @OneToMany
  public Collection<Class> getOwnedElements() {
    return ownedElements;
  }}
```

Example 3 illustrates the mapping with the Table class that corresponds to the CWM Table metaclass in the target metamodel of our transformation. The schema attribute is the inverse relationship of the elementOwnership relation, the foreignKeys collection is the representation of the keyRelationFeature relation. Name represents the simple name function of the table.

4.2 Reusing model importers

Model importers are used to import models from various file formats of modeling tools into the VIATRA2 framework. The implementation of an importer for a given format requires significant work, hence the reusability of importers in PST programs was a strong requirement.

The VIATRA2 importers operate on VPM models that contain entities and relations, which is a platform-independent but low-level model representation in contrast to the EJB-based solution introduced in Sec. 4.1. This semantic gap between the two representations has to be bridged, thus we implemented a special wrapper that converts the low-level model space calls to EJB object creations and method calls.

The VIATRA2 model space provides a single Java interface for importing modules to query and update the content of the model space. The operations of this interface manipulate basic VPM constructs (for example, method newEntity() creates an entity, setName() renames a model element, and newInstanceOf() creates a new type-instance relationship between two model elements).

The model space wrapper offers the same interface for the input module as the original implementation in the VIATRA2 framework, and contains a special facility to map the method calls to EJB operations. However, the native transformation plugin contains only two meta-levels, a static meta-level and a dynamic model-level, which implies the following considerations.

- The model space wrapper generates read-only entity wrapper objects if the importer queries the model space for the meta-level elements.
- If a new model-level element is created, the wrapper generates a dynamic proxy object. This object stores the VPM level properties of the created model element as long as the type of the element is undefined. If the model importer creates an explicit instanceOf relation between the model element and a metamodel element, the type of the given element can be determined, and a corresponding entity bean can be generated. The cached properties are propagated to the entity bean.

As a result, the model space wrapper facility hides the implementation platform from the model importer; therefore it eases the porting of importers to various platforms.

Note that we neglected the treatment of *model exporters* in this section due to the fact that they are typically implemented using the transformation based code generation features of VIATRA2 . Therefore, their reuse in PSTs will be covered essentially in Sec. 4.3.

4.3 From PIT to PST transformations

Now we generate EJB3-specific transformations from PIT descriptions using the metamodel-specific model manipulation API derived in Sec. 4.1. Due to space considerations, we primarily focus on the handling of GT rules, which is the most critical and complex part of generating EJB-specific transformer plugins since ASM programs already provide a close correspondance with object-oriented Java technology. More specifically, we show

1. how to handle input and output rule parameters;
2. how to perform graph pattern matching differently in case of parallel (forall) and single (choose) rule applications;
3. how to check the success or failure of pattern matching
4. how to manipulate models in a persistent and transactional way.

Our concepts will be demonstrated using our ongoing example by processing ClassR of Fig. 3 and the ASM program of Fig. 4.

4.3.1 Binding input and output rule parameters.

While typically input parameters of a rule are mapped to input parameters of a method, and the output parameter of a rule is mapped to the return value, this solution

is unfeasible, since there can be several output parameter, moreover, an inout parameter can be considered as both input and output. Therefore, a Java Map is used for passing rule parameters in each directions.

```
public Map classR_forall(Map in) { // Input parameters
    HashMap result = new HashMap(); // Output parameters
    ...
    Package p = (Package) in.get("p");
    result.put("p",p);
}
```

4.3.2 Graph pattern matching (in Forall and Choose mode).

It is well-known that the most critical step for the performance of graph transformation is the graph pattern matching phase. For this purpose, the generation of *search plans* is a frequently used and efficient strategy. Informally, a search plan defines the order of traversal for the nodes of the instance model to check whether the pattern can be matched. Complex model-specific optimization steps can be carried out for generating efficient search plan as reported in [18]. Here, we assume that a search plan is available for the precondition of the GT rule.

Furthermore, we need to distinguish whether a graph transformation rule is called from the context of a choose or a forall ASM rule, which prescribes different behavior. As a result, for each GT rule, several methods may be generated that quantifying different rule parameters universally (with forall) or existentially (by default, or using the choose construct).

Now, we investigate the execution of classR in forall mode (as prescribed in Fig. 4). Let the ordering of pattern nodes for the derived search plan be p, s, c.

This search plan first selects an initial package p from which the pattern matching should start. In case of calling classR in ASM program of Fig. 4, this node is passed as a parameter, which can be determined by a compile-time analysis. Then all other pattern nodes and edges are traversed one by one according to the search plan.

- When an edge with at-most-one multiplicity (one-to-one or many-to-one in EJB3 terminology) is traversed, a single object is navigated, which is retrieved directly by reference.
- In case of arbitrary multiplicity in the traversed direction (one-to-many or many-to-many), an iterator is generated to investigate all possible continuations.

Example 4. The principle of generating while or if-then constructs according to multiplicities is demonstrated below when applying classR (in forall mode).

```
public Map classR_PRE_forall(Map in) {
  HashMap result = new HashMap();
  // Matching Package node
  Package p = (Package) in.get("p");
  if (p != null) {
    result.put("p",p); // Copy to output params
    // Matching Schema node
    Schema s = p.getTargetRef();
    if (s != null) {
      // Iterating over all Class nodes
      Iterator iter_c = p.getOwnedElement().iterator();
      while (iter_c.hasNext()) {
            Class c = (Class) iter_c.next();
            result.put("c", c);
            // Check negative conditions ->
            // Apply model manipulations ->
      } } } }
```

When an ASM machine prescribes that a GT rule should be applied on at most one matching (using the choose construct), further optimization is possible, namely, one can immediately terminate the execution of loops (and if-then constructs) as soon as the rule finds the first matching. For this purpose, we introduce a new Boolean variable success and inject an additional condition (!success) into all guards of loops and if-then-else constructs. Finally, a RuleFailedException exception is thrown if the pattern matching failed.

Example 5. For instance, the implementation of a NAC (see classR_NAC_choose below), it is sufficient to find a single matching of the NAC graph to refute a matching of the LHS.

```
public Map classR_NAC_choose(Map in) {
  HashMap result = new HashMap();
  boolean success = false;
  Class c = (Class) in.get("c");
  result.put("c",c);
  Table t = c.getTargetRef();
  if(t != null && !success) { // Additional condition
    success = true;
  }
  if (success)
    return result;
  else throw new RuleFailedException();
}
```

4.3.3 Checking the success of pattern matching.

VIATRA2 patterns may contain calls to other (positive or negative) patterns, which means that as soon as the pattern matching of the local pattern is successfully completed, it should continue with matching additional positive and/or refuting all negative patterns. The call environment needs to be set up by passing the appropriate common parameters between such patterns (e.g. class c in case of the LHS and NAC patterns of rule classR). The fact that the matching of a pattern failed is denoted by catching a RuleFailedException.

```
// Check negative conditions (in classR)
HashMap nacIn = (new HashMap()).put("c",c);
try {
  classR_NAC_choose(nacIn);
  // Process next potential matching
}
catch (RuleFailedException e) {
    // Apply model manipulations ->
}
```

In addition to checking positive and negative conditions, the result of this success attribute is frequently used later to drive the iterate and if-then-else ASM constructs constructs. For instance, the iterate construct in the ASM program of Fig. 4 can be implemented as follows.

```
HashMap in1 = new HashMap();
try {
  while (true) {
    packageR_choose(in1);
    // Other rules inside seq{}
  }
}
catch (RuleFailedException e) {...}
```

4.3.4 Model manipulations.

Model elements are manipulated according to the EJB3-specific Java API. For instance, in case of creating a new entity bean, the traditional new construct is used first, then the fields of the entity bean are set, finally, the database manager is called to persist the new object.

```
// Model manipulation for ClassR
t = new Table();
tpk = new PrimaryKey();
tid = new Column();
t.setSchema(s);
c.setReference(t);
tid.setTable(t);
tpk.setTable(t);
tpk.setColumns((new HashSet()).add(tid));
// Persisting objects
manager.persist(t);
manager.persist(tpk);
manager.persist(tid);
```

If a method implementing a transformation terminates successfully, then model changes are committed automatically due to the default transaction handling facilities of EJB3. Furthermore, if inconsistencies (i.e. exceptions) arise during EJB3 method calls, the transaction manager initiates rollback to restore a consistent model state. This automatically prevents inconsistencies introduced by applying a GT rule in forall mode in case of conflicting rule applications.

4.4 Performance evaluation

To evaluate the performance of EJB3-based transformation plugins, we carried out experiments based on the object-relational performance benchmark selected from [17]. Our main goal was to assess the overhead caused by an application server and the underlying relational database required to run EJB3 applications. Therefore, we executed the same algorithm (generated according to the guidelines of the paper) once as Enterprise Java Beans and then as pure Java objects (without application server and database)[2]

The results of our experiments are shown in Table 1 where the average time of finding a single matching (match) and then executing a rule (update) on this match is depicted for three different rules of the transformation.

	Class	Pure Java Obj		MySQL - EJB3	
	#	match msec	update msec	match msec	update msec
classR	10	0,10	0,06	0,14	1,69
	100	0,03	0,02	0,14	1,84
	250	0,02	0,02	0,14	1,77
	1000	-	-	0,15	3,66
assocR	10	0,27	0,12	0,15	1,65
	100	0,04	0,03	0,15	1,81
	250	0,27	0,02	0,14	1,91
	1000	-	-	0,15	2,16
assocEndR	10	0,15	0,07	0,67	1,95
	100	0,03	0,02	0,51	1,90
	250	0,02	0,02	0,71	2,37
	1000	-	-	0,62	2,69

Table 1: Performance evaluation of EJB3 plugins

[2]For our experiments, we used an Intel Pentium-M 1600 MHz processor with 2GB RAM and 60 GB HDD machine running on Windows XP SP1, Java SDK 1.5, MySQL 4.1 database, and JBoss 4.0.1 application server.

In order to fix a a complete, deterministic, but parametric test set, the structure of the initial model and the transformation sequence was fixed up to a numerical parameters, i.e. the number of Classes in the initial instance model (denoted by N). The initial model has a single Package that contains N classes. An Association and two AssociationEnds are added to the model for each pair of Classes, thus initially, we have $N(N-1)/2$ Associations and $N(N-1)$ AssociationEnds connected appropriately. This way, parameter $N = 1000$ denotes a model with around 1.5 million elements, which is already a very large model.

Our experiments show a trade-off between performance and model size as stated in the following observations:

- *Performance.* On a given model size, the pure Java solution runs one order of magnitude faster during the pattern matching phase and two orders of magnitude faster in the manipulation phase. Most probably, in the pattern matching phase this is caused by the overhead of the database connection, while in the manipulation phase this is a joint effect of database connection and transaction handling. In addition to that there is significant amount of initial overhead when connecting the database for the first time.

- *Model size.* On the other hand, the EJB3 plugin was able to handle very large models (>1 million elements) since the models are persisted in a relational database, while pure Java programs ran out of memory.

In addition, it is worth pointing out that a significant amount of time was required initially with database and application server overhead when executing the first rule of the model transformation. On the other hand, as soon as the database and the application server is already running, we did not experience significant overhead.

5. RELATED WORK

While there is already a large set of model transformation tools available in the literature using graph rewriting, below we focus on providing a brief comparison with the most popular compiled approaches that show conceptual similarities with our work.[3]

Fujaba [8] compiles visual specifications of transformations [6] into executable Java code based on an optimization technique using search graphs with a breadth-first traversal strategy penalizing many-to-many multiplicity constraints. Our approach is different from Fujaba in the use of EJB3 beans instead of pure Java classes and the model-sensitive generation of search plans (see [18] for details).

PROGRES [11] supports both interpreted and compiled execution (generating C code) of programmed graph transformation systems. It uses a sophisticated cost model for defining a priori costs of basic operations (like the enumeration of nodes of a type and navigation along edges) for generating search plans. Our solution was, in fact, influenced by PROGRES in the use of a cost function. However, our model-specific cost function provides a better estimation for the complexity of matching graph transformation rules.

The pattern matching engine of compiled GReAT [19] (generating C++ code) uses a breadth-first traversal strategy starting from a set of nodes that are initially matched.

[3]A more detailed comparison on the difference in the applied graph transformation strategies can be found in [18].

Such a C++ solution typically provides an efficient solution for compiled transformations when integrated into embedded systems. In contrast, our transformation plugins primarily target web-based target platforms and achieve high performance as being deployed on application servers.

OPTIMIX [2] is a tool for generating algorithms in C or Java which construct and tranform directed relational graphs with a special focus on tasks in program compilation and optimization. OPTIMIX supports edge addition rewrite systems (EARS) and exhaustive graph rewrite systems (XGRS) using an input language equivalent to a subset of Datalog.

The idea of separating transformation design from transformation execution also appears in the MOLA environment [7] by providing an Eclipse EMF-based execution environment. While EMF-based models and EJB3 models show conceptual similarities concerning their structure, EJB3 provides additional support for important dynamic features such as e.g. transaction handling.

6. CONCLUSIONS

We presented a technique for generating stand-alone model transformation plugins for the EJB 3.0 platform from platform-independent specifications of transformations given by a combination of graph transformation and abstract state machine rules as used within the VIATRA2 framework. As a result, the design of transformations can be separated from the execution of transformations.

We assessed the performance of our EJB3-based plugin-generator on a graph transformation benchmark selected from [17] by comparing it to pure Java-based solutions. Our experiments showed that while pure Java-based solutions are (unsurprisingly) faster for models of the same size, very large models can only be handled by EJB3-based plugins as pure Java solutions run out of memory. This way, EJB3 transformer plugins can be integrated into existing business (J2EE) applications manipulating industrial size models.

In the future, we intend to extend our approach to better exploit the use of the underlying EJB 3.0 technology by using (i) message-driven beans for long running transformations, (ii) stateful session beans for user-guided model transformations, and (iii) the query language of EJB-QL to further optimize pattern matching on models.

7. REFERENCES

[1] Eclipse Modeling Framework. `http://www.eclipse.org/emf`.

[2] U. Assmann. *In [5]*, chap. OPTIMIX: A Tool for Rewriting and Optimizing Programs, pp. 307–318. World Scientific, 1999.

[3] J. Bézivin, N. Farcet, J.-M. Jézéquel, B. Langlois, and D. Pollet. Reflective model driven engineering. In P. Stevens, J. Whittle, and G. Booch (eds.), *Proc. UML 2003: 6th International Conference on the Unified Modeling Language*, vol. 2863 of *LNCS*, pp. 175–189. Springer, San Francisco, CA, USA, 2003.

[4] E. Börger and R. Stärk. *Abstract State Machines. A method for High-Level System Design and Analysis*. Springer-Verlag, 2003.

[5] H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg (eds.). *Handbook on Graph Grammars and Computing by Graph Transformation*, vol. 2: Applications, Languages and Tools. World Scientific, 1999.

[6] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story diagrams: A new graph transformation language based on UML and Java. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg (eds.), *Proc. Theory and Application to Graph Transformations (TAGT'98)*, vol. 1764 of *LNCS*. Springer, 2000.

[7] A. Kalnins, J. Barzdins, and E. Celms. Model transformation language MOLA. In *Proceedings of MDAFA 2004 (Model-Driven Architecture: Foundations and Applications 2004)*, pp. 14–28. Linköping, Sweden, 2004.

[8] U. Nickel, J. Niere, and A. Zündorf. Tool demonstration: The FUJABA environment. In *The 22nd International Conference on Software Engineering (ICSE)*. ACM Press, Limerick, Ireland, 2000.

[9] Object Management Group. *CWM: Common Warehouse Metamodel*. `http://www.omg.org`.

[10] Object Management Group. *Model Driven Architecture — A Technical Perspective*, 2001. `http://www.omg.org`.

[11] A. Schürr, A. J. Winter, and A. Zündorf. *In [5]*, chap. The PROGRES Approach: Language and Environment, pp. 487–550. World Scientific, 1999.

[12] Sun Microsystems. *Enterprise Java Beans 3.0*.

[13] J. D. Ullman, J. Widom, and H. Garcia-Molina. *Database Systems: The Complete Book*. Prentice Hall, 2001.

[14] D. Varró. *Automated Model Transformations for the Analysis of IT Systems*. Ph.D. thesis, Budapest University of Technology and Economics, Department of Measurement and Information Systems, 2004.

[15] D. Varró and A. Pataricza. VPM: A visual, precise and multilevel metamodeling framework for describing mathematical domains and UML. *Journal of Software and Systems Modeling*, vol. 2(3):pp. 187–210, 2003.

[16] D. Varró and A. Pataricza. Generic and meta-transformations for model transformation engineering. In T. Baar, A. Strohmeier, A. Moreira, and S. Mellor (eds.), *Proc. UML 2004: 7th International Conference on the Unified Modeling Language*, vol. 3273 of *LNCS*, pp. 290–304. Springer, Lisbon, Portugal, 2004.

[17] G. Varró, A. Schürr, and D. Varró. Benchmarking for graph transformation. Tech. rep., Budapest University of Technology and Economics, 2005. `http://www.cs.bme.hu/~gervarro/publication/TUB-TR-05-EE17.pdf`.

[18] G. Varró, D. Varró, and K. Friedl. Adaptive graph pattern matching for model transformations using model-sensitive search plans. In G. Karsai and G. Taentzer (eds.), *GraMot 2005, International Workshop on Graph and Model Transformations*, ENTCS. In press.

[19] A. Vizhanyo, A. Agrawal, and F. Shi. Towards generation of efficient transformations. In G. Karsai and E. Visser (eds.), *Proc. of 3rd Int. Conf. on Generative Programming and Component Engineering (GPCE 2004)*, vol. 3286 of *LNCS*, pp. 298–316. Springer-Verlag, Vancouver, Canada, 2004.