Industrial-strength Rule Interoperability using Model Driven Engineering

Marcos Didonet del Fabro, Patrick Albert, Jean Bézivin, Frédéric Jouault

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Industrial-strength Rule Interoperability using Model Driven Engineering

Marcos Didonet Del Fabro – Patrick Albert – Jean Bézivin – Frédéric Jouault

N° 6747

November 2008

Thème COM

ISSN 0249-6399

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche RENNES - BRETAGNE ATLANTIQUE

Industrial-strength Rule Interoperability using Model Driven Engineering

Marcos Didonet Del Fabro[1], Patrick Albert[2], Jean Bézivin[3], Frédéric Jouault[4]

Thème COM – Systèmes Communicants
Projet AtlanMod

Rapport de recherche n° 6747 – November 2008 - 19 pages

Abstract: Model Driven Engineering (MDE) is rapidly maturing and is being deployed in several situations. We report here on an experiment conducted in the context of ILOG, a leader in the development of Business Rule Management Systems (BRMS). BRMSs aim at enabling business users automating their business policies. There is a growing number of BRMS supporting different languages, but also a lack of tools for bridging them. In this paper, we present an approach based on MDE techniques for bridging rule languages; the solution has been fully implemented and tested on different BRMS. The success of the experiment has led to the development of a significant number of model transformations. At the same time, this deployment has shown new problems arising from the management of a high number of artifacts. We discuss the positive assessment of MDE in this field, but also the need to address the complexity generated.

Keywords: Model Driven Engineering, model transformations, business rules, BRMS, interoperability.

[1] ILOG SA – mddfabro@ilog.fr
[2] ILOG SA – palbert@ilog.fr
[3] AtlanMod (INRIA & EMN) – jean.bezivin@inria.fr
[4] AtlanMod (INRIA & EMN) – frederic.jouault@inria.fr

Centre de recherche INRIA Rennes – Bretagne Atlantique
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : +33 2 99 84 71 00 — Télécopie +33 2 99 84 71 71
www.irisa.fr

Interopérabilité entre règles métier utilisant l'Ingénierie Dirigée par les Modèles

Résumé: L'Ingénierie Dirigée par les Modèles (IdM) est utilisée de plus en plus dans plusieurs applications. Nous présentons dans ce rapport une expérience menée dans le cadre d'ILOG, un leader dans le développement des systèmes de gestion de règles métiers (SGRM). Les SGRMs permettent aux utilisateurs d'automatiser leurs politiques métier. Il existe un nombre croissant de SGRMs supportant différents langages, mais l'interopérabilité entre eux est très limitée, voir inexistante. Dans ce rapport, nous présentons une approche basée sur l'IdM pour permettre l'interopérabilité entre langages des règles; notre solution a été implantée et testée sur différents SGRMs. Le succès de l'expérience a conduit au développement d'un nombre important de transformations de modèles. Aux même temps, cette expérience a montré des nouveaux problèmes relatifs à la gestion d'un nombre élevé de modèles et d'autres éléments logiciels. Nous examinons l'utilisation de façon positive de l'IdM dans ce domaine, mais également la nécessité de prendre en compte la complexité qui a été générée.

Mots clés: Ingénierie Dirigée par les Modèles, transformations de modèles, règles métiers, SGRM, interopérabilité

1 Introduction

In 2000, the OMG [23] (Object Management Group) proposed a new model-based vision for the development and maintenance of software systems. The initial idea was to capture platform independent business models with modeling languages such as UML and then to map them on a given platform language (Java, C#, etc.) with the help of automated transformations. Since then, Model Driven Engineering (MDE) has considerably broadened its application area and many new applications currently fall in its scope. The one discussed here is related to language bridging, a central issue in companies like ILOG.

This paper describes an experiment to apply MDE techniques to the field of Business Rule Management Systems (BRMS). The experiment was conducted over one year within an industrial environment (at ILOG Company), with a set of open source MDE tools previously built at INRIA (the AMMA tool suite [22]).

The field of BRMS is characterized by a number of normative, open source, or proprietary systems and languages (ILOG JRules [12], JBoss Drools [15], Fair-Isaac Blaze Advisor [7], etc.), allowing the expression of various solutions to business problems at a high abstraction level, but with heterogeneous sets of capabilities and languages. Going from the initial problem (e.g., UML to Java translation) to a more general problem of DSL (Domain Specific Language) to DSL translation is an important step that we faced in this industrial case.

Rule Interchange Format (RIF) [27] and Production Rules Representation (PRR) [26] are two standard proposals respectively developed at the W3C and at the OMG aiming at providing a level of standardization to the domain. But these necessary initiatives are either not ready (or approved) or only covering a share of the BRMS languages.

The question to answer in the project was thus about which help - if any - MDE could bring to provide interoperability between BRMSs. A first limited experiment had been conducted before to bridge normative business rule languages [1]. Though our project has a much greater size and complexity, the former has provided us with inspiration and reasonable hope for convergence.

Our goal was to take projects of the source BRMS (Drools) as input and to automatically produce ready-to-use projects in the target BRMSs (ILOG JRules). We developed two interoperability tools (i.e., bridges): a reasonably simple one from DRL (Drools Rule Language [15]) to IRL (ILOG Rule Language [12]), and a much more complex bridge that takes IRL rules as input and that generates Business Rules written in the ILOG “Business Action Language” (BAL) [12].

The experiment was much broader that typical one-to-one transformations, because we had a higher number of artifacts that should be produced (e.g., different kinds of project or control files). In particular, two major issues have risen. First, we had to do an inventory of all artifacts and to create all the operations/transformations to produce them. Second, we had to coordinate their execution in a coherent flow. We describe the operations that we created, and how they were composed to handle these issues.

To summarize, the major contributions of this paper are the following. We report on an experiment that included a complex MDE architecture encompassing a large number of input or output artifacts and operations – no less than twenty – for developing bridges between two BRMS. We focus on the definition of the operations and on the coordination of their execution. We apply our solution to different projects, going from standard business rules benchmarks to real-life interoperability project.

This paper is organized as follows. Section 2 rapidly recalls the basic principles of MDE and presents the tools used in the experiment. Section 3 introduces the field of BRMS and presents the general architecture of the two main BRMS used in the experiment. Section 4 presents the

interoperability solutions that have been developed. Section 5 describes the lessons learned. Section 6 concludes.

2 Model Driven Engineering

The primary software artifacts of the MDE approach are models, which are considered as first-class entities. These models may be defined using the OMG standards (e.g., UML), *de facto* standards (e.g., Eclipse Modeling Framework [10]), or may even be other kinds of artifacts like programs (e.g., Java, C#, SQL), XML documents, databases, etc. Every model conforms to a metamodel (or grammar, or schema), which defines its abstract syntax. One of the most common operations applied on models is transformation, which consists in the creation of a set of target models from a set of source models according to specific rules.

Recently, a convergence between MDE and DSL engineering has been observed [22]. There was previously a strong focus on a small number of relatively large general-purpose metamodels (e.g., UML). But, an increasing number of smaller domain-specific metamodels are now being defined and used. Additionally, MDE techniques are also being used to represent existing DSLs. n this new MDE-DSLs combined approach a language is typically captured as a set of coordinated models. There are three main elements in such a set:

- A metamodel represents an **abstract syntax** as a set of domain concepts and relations between them.
- A **concrete syntax** is defined by a model specifying concrete representations for metamodel elements. This notably applies to visual and textual (e.g., context-free, XML) syntaxes. A given language may have several concrete syntaxes.
- The semantics of a language may be encoded in a transformation from its metamodel (i.e., abstract syntax) to another metamodel, which semantics is known (e.g., state machines, Petri nets). Other kinds of definitions may also be used.

This list may be extended as appropriate in order to take into account other aspects of a language. These may in turn be captured as models, transformations, or both.

A modeling platform generally offers languages and tools that support the definition of abstract and concrete syntaxes, as well as transformations. The experiment presented in this paper is based on the AMMA [22] modeling platform that we are developing. The three main tools offered by AMMA are the following;

- **KM3** (Kernel MetaMetaModel) [18] is a simple and expressive metamodel specification language (i.e., a metametamodel). It is used to define abstract syntaxes of DSLs.
- **TCS** (Textual Concrete Syntax) [17] is a bidirectional mapping tool between metamodels and grammars. It is able to perform both text-to-model (injections) and model-to-text (extractions) translations from a single specification. TCS is used to map context-free concrete syntaxes to metamodels.
- **ATL** (AtlanMod Transformation Language) [19] is used to express model-to-model transformations. ATL has a development environment integrated into Eclipse.org [16], which notably includes a textual editor, and a debugger. ATL transformations are notably used to translate from one DSL to another.

There are several other tool suites similar to AMMA (e.g., Epsilon [20] or oAW [24]). The closest one is oAW, which is also available on the Eclipse.org platform. AMMA and oAW share the same model-centric vision and only differ on some implementation choices.

In this work, we use the AMMA platform in order to represent existing business rules DSLs, and to define transformations between them.

3 Business Rule Management Systems

A BRMS [25] is composed of several components supporting the definition, management, and execution of business rules. Business rules are mostly an evolution of production rules [8] – an

established A.I. technique – used to enable business users to automate policies. The rules are written in Domain-specific declarative languages as close as possible to the application domain and terminology. The technical details of the rules and application are hidden by a "business-level" front-end close to natural language.

This enables the policies owners to create and manage the rules by themselves with little to no support from software developers. Being largely independent from IT, the business users can react almost immediately to most policies changes by adding, removing, or changing rules. Well-formalized processes guarantee that new or modified rules are well managed and have little to no risk of breaking the application logic.

Rule authoring components are key elements in BRMS. They provide facilities for rule editing and maintenance. They act directly on the business rules produced by the rule developers.

In this section, we first present business rules. Then, we zoom into the two BRMS studied in our experiment, focusing on the business rules.

3.1 Rules

The rules are written in domain specific languages, which may have different levels of abstraction. In this section we present an overview of business rules.

3.1.1 Production rules

Production rules (or simply rules) are more and more used to help business organizations automating their policies, providing properties such as reliability, consistency, traceability, and scalability. The rules are mostly made of *If-Then* statements (as shown in Figure 1) involving predicates and actions about sets of business objects.

Rules variables are bound to objects of a certain type; when the condition part recognizes that a set of objects satisfies its predicates, the action is triggered.

```
rule applyDiscount {
  when {
    ?c : Customer(status == "Gold");
  }
  then {
    ?c.applyDiscount(20);
 }
}
```

Figure 1. A simple production rule

3.1.2 Business rules

Compared to earlier rules languages such as the seminal OPS5 [8] intended to be used by software developers or knowledge engineers, the business rules approach is an attempt to bring the power of rules programming to business users willing to automate business policies.

The business rules are expressed in a language close to natural language that can be understood and managed by "business analysts". This Business-level language is compiled into a lower-level technical language (such as the one in Figure 1); the (not so) simple 'production rules' semantics remains unchanged.

The "Business Layer" is composed of additional models supporting the definition of the rules (see Figure 2).

- A Business Object Model (BOM) defines the classes representing the objects of the application domain. A BOM is in fact a simplified ontology defining the application classes with their associated attributes, predicates, and actions.
- A Vocabulary model (VOC) defines the words or phrases used to reference the BOM entities.

- A Business Action Language (BAL), close to natural language, which uses the elements of the VOC to build user understandable rules, such as “If the customer status is GOLD, apply a 20% discount”.

The business object model and vocabulary are defined in an early stage of the project, and their projection on an executable layer. The executable layer is composed of the eXecution Object Model (XOM) and of the technical rule language – which is implemented by the IT specialists. Figure 2 illustrates how the BOM is implemented by the XOM (typically a set of Java or C# classes), and how the business language is compiled to the technical rules language. The technical rules are applied to the XOM objects.

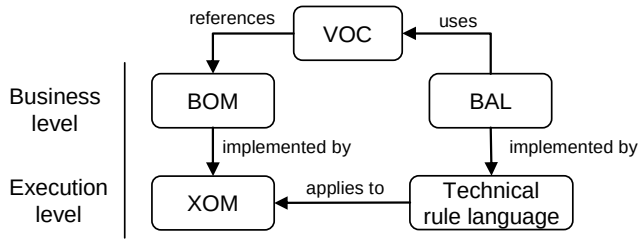

Figure 2. Different layers of production rules

3.2 Drools

Drools is an open-source BRMS part of the JBoss foundation. Its rules language is called DRL, for Drools Rule Language [14]. Though Drools has recently introduced some sort of macros for defining business friendly rules, we have been focusing our work on the technical rules language, as illustrated in Figure 3.

```
rule "approve"
when
   not Rejection()
   $policy : Policy( approved == false )
then
   System.out.println("approve" + $policy.getPrice());
   $policy.setApproved(true);
   System.out.println("Policy approved");
end
```

Figure 3. Example of a rule in DRL

3.3 ILOG JRules

JRules is the product developed and marketed by ILOG. It supports the two different levels of a full-fledged BRMS: the technical level targeted at software developers and the business action language targeted at business users. They are detailed below.

3.3.1 Technical rules

The technical rules of JRules are written in IRL (see an example in Figure 4). The complete specification of IRL may be found at [13]. Consider the rule illustrated in Figure 4. The rule has the same semantics as the DRL rule presented in Figure 3, and the rules’ structures are very similar.

```
rule approve {
  when {
    not Rejection();
    ?policy : Policy(approved == false);
  }
  then {
    out.println("approve"+?policy.getPrice());
    ?policy.setApproved(true);
    out.println("Policy approved");
  }
}
```

Figure 4. Example of a rule in IRL

3.3.2 Business Action Language

The Business Action Language of JRules hides implementation details of IRL, allowing business analysts to concentrate on the business logic. The equivalent rule for the previous examples (Figures 3 and 4) is shown in Figure 5.

```
definitions
  set 'policy' to a policy;
if there is no rejection and 'policy' is not approved
then
  print "approve:" + the price of 'policy';
  make it true that 'policy' is approved;
  print "Policy approved";
```

Figure 5. Example of a rule in BAL

IRL and BAL are independent languages in JRules, i.e., each one has its own schemas, syntax, and editing facilities. JRules provides built-in translation from BAL into IRL, but not in the opposite direction.

3.3.3 Business Object Model

The Business Object Model (BOM) is mapped to the execution objects that support the actual application data. In the simplest cases, the BOM can be automatically extracted from the classes definitions in Java, C# or XML-Schema. For instance, if the execution objects are Java objects, the business model will define one BOM class per Java class. We illustrate in Figure 6 the BOM of the Policy concept used above. The origin property indicates the source used for extraction.

```
property origin "xom:/Input/Policies-xom"
public class Policy {
    public boolean approved;
    public void setApproved(boolean arg1);
    public void setPrice(int price);
    public float getPrice();
    public Policy ();
}
```

Figure 6. BOM for the *Policy* business object

Note that the design of the BOM of JRules has been inspired by the Java syntax, thus the main structure looks like the declaration of a Java class. However, the BOM may provide additional information, such as the origin of the classes, the specification of the domains and annotation properties (not shown here).

3.3.4 Vocabulary

The vocabulary (VOC) defines a text file with a controlled vocabulary that "closes" the rules syntax on a fixed set of allowed words and fragments.

An excerpt of the vocabulary of class Policy is shown in Figure 7. For instance, the verbalization of class *Policy* is *policy*. Then, the attribute *insurancePrice* is mapped to "*insurance price of {this}*". The *{this}* token indicates the calling object.

```
Policy#concept.label = policy
Policy.approved#phrase.action = make it {approved} that {this} is approved
Policy.approved#phrase.navigation= {this} is approved
Policy.price#phrase.action= set the price of {this} to {price}
Policy.price#phrase.navigation= {price} of {this}
```

Figure 7. Vocabulary with the terminology of the *Policy* class

The business rules editor proposes only the valid terms while a business user creates or modifies a rule.

3.3.5 B2X and project files

The "Business To eXecution" model (B2X) describes how the logical elements represented in the BOM are actually mapped to physical data structures (XOM), i.e., it is used when there is not an implicit 1-to-1 mapping towards Java, C# or XML. Thus, the application developer might add new B2X constructs to specify the way the new "BOM elements are implemented with the target programming language.

For instance, consider we need to call an *insert* method to add a new object in the memory, which is a common scenario. However, the BAL does not provide a built-in vocabulary for insert. The solution is to create a new "virtual" method in the BOM called *InsertAction*, which takes a *java.lang.Object* as parameter (see Figure 8). A verbalization is created as well. When the corresponding method is called in a business rule, the engine executes the expression "insert (object)" (the target language is Java).

```
<method>
  <name>InsertAction</name>
  <parameter type = "java.lang.Object"/>
  <body language = "java">
      insert (object);
  </body>
</method>
```

Figure 8. B2X mapping

A JRules application has an additional project file - .ruleproject (RP) – which specifies the project parameters, the folder where the rules, the BOM and the vocabulary are stored and a URL.

4 Rule interoperability

Our solution applies MDE techniques to achieve interoperability across BRMS. As already stated, the central concept in a BRMS is the rule language. For that reason, our main goal is to transform a set of rules of a source BRMS, into an executable set of rules (and associated files) of a target BRMS.

The architecture follows the usual MDE pattern: **inject, transform, extract**, and relies on core MDE practices and technologies (see section 2): Domain Specific Languages (DSLs) [22], metamodeling [22], model transformations [19] and projections across technical spaces (i.e., injections and extractions) [21].

In our scenario, we have two complementary objectives: translating a set of rules in DRL into IRL and translating IRL into BAL. We first produce the IRL rules from the DRL ones, and

then we produce the BAL rules from IRL. In order to have a complete and ready-to-run JRules application, it is also necessary to produce the vocabulary, the BOM and the project files. This has led to the definition of several operations, not only a single model-to-model transformation.

The existence of several interacting artifacts raises two major issues. First, we have to discover (and to develop) all the required operations. Second, we have to connect them (and in a correct order) to be able to define an automatic bridge.

First, we present the model management operations that we have defined. Then, we show how the operations are connected through the execution of chains of transformations.

4.1 Model management operations

The bridge is composed of *twenty four* operations (as shown in Figure 9). The operations are identified by an initial letter depending on the category, plus an integer increment. The labels indicate the kinds of artifacts that are produced, for instance, a BOM model, or a BOM file. The operations are categorized below.

- **External** (**X_i**): any operation implemented prior to the development of the bridge. For instance, the generation of BOM and VOC by JRules from the Java classes.
- **Injection** (**I_i**): injection of the input textual files into a model. For instance, the input files for DRL, BOM, VOC, and optionally IRL.
- **Transformations** (**T_i**): the model transformations; they come in different types.
  - **Refactoring**: endogenous model transformation for DRL and IRL.
  - **Augmentation**: growing endogenous model transformation for BOM, VOC, B2X, and Rule Project.
  - **Translation**: exogenous model transformation for DRL to IRL, IRL to BAL and BAL to BRL.
  - **XML-ification**: a special kind of exogenous model transformation toward XML, for B2X, BRL and Rule Project.
- **Extraction** (**E_N**): toward ad-hoc files for IRL, BOM, VOC and BAL, and toward XML for B2X, BRL and Rule Project.

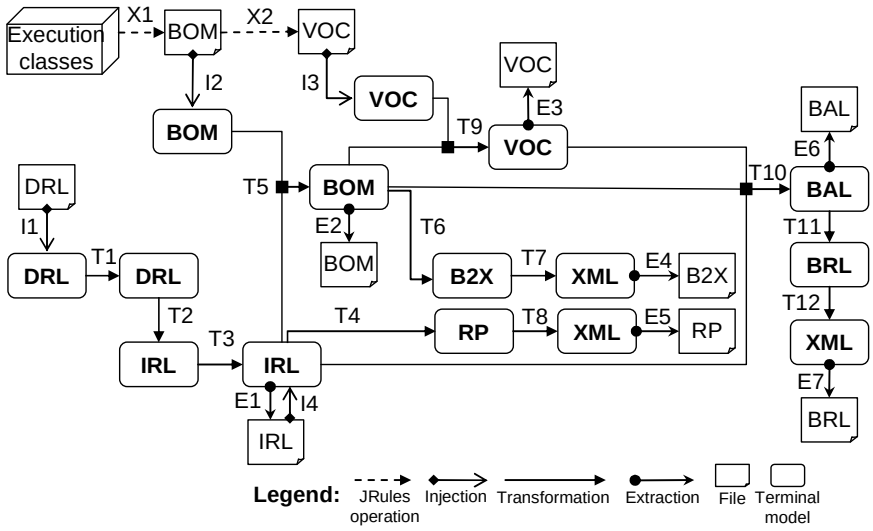

Figure 9. DRL → IRL → BAL complete process

An operation has the following signature:

$$<M_{OUT1} : MM_{OUT1}, \ldots, M_{OUTm} : MM_{OUT}m> = T[TS_{OUT}\text{-}TS_{IN}](<M_{IN1} : MM_{IN1}, \ldots, M_{INn} : MM_{INn}>).$$

T is the operation name; [TS_{OUT}-TS_{IN}] are markers used to indicate the output and input technical spaces. For instance, when injecting a textual file into a model, we mark the operation with [MDE-EBNF]. These markers are optional and are not used when specifying model transformations in the MDE technical space.

<M_{IN1} - M_{INn}> are the set of input models (n >= 1); the input models conform to the input metamodels <MM_{IN1} - MM_{INn}>; the input metamodels may be equal; M_{OUT1} - M_{OUTm} is the set of output models (m >= 1); the output models conform to the output metamodels <MM_{OUT1} – MM_{OUTn}>.

4.1.1 DRL to IRL operations

This bridge produces an ILOG JRules project including rules written in the IRL language from a set of files written in DRL.

The KM3 metamodels are based on the online specification of Drools and JRules (respectively expressed as XSD schemas and EBNF notation). For the sake of maintainability, we chose to create the metamodel elements sticking to the original specifications. Still, in the Drools case, we need to refactor the KM3 to take into account the nested nature of XSD schemas. The bridge has five operations.

drl_1 : DRL = I1[MDE-EBNF] (drl : DRL)	**(1)**
drl_2 : DRL = T1 (drl_1 : DRL)	**(2)**
irl_1 : IRL = T2 (drl_2 : DRL)	**(3)**
irl_2 : IRL = T3 (irl_1 : IRL)	**(4)**
irl : IRL = E1[EBNF-MDE] (irl_2 : IRL)	**(5)**

The first operation **(1)** parses the textual file (conforming to the EBNF grammar of DRL) and it produces as output a model conforming to the DRL metamodel.

However, there are a few DRL expressions that are not natively supported by IRL. For example, DRL conditions might be connected by an *OR* predicate – such as in "`not Rejection() OR Policy(approved == false)`" – which is not possible in IRL. Instead of handling these kinds of expressions in a single transformation, we first run an endogenous refactoring transformation **(2)** that takes a DRL model as input and that produces a refactored DRL model.

Then, we translate the refactored DRL model into an IRL model **(3)**. The IRL model is refactored as well **(4)**, for instance, to rename the variables that use keywords.

The separation in two transformations enables the specification of a relatively simple "DRL to IRL" transformation. These kinds of refactorings are typically formed by expressions that are not frequently used, thus it is possible to skip its execution in order to improve performance. Finally, the IRL models are extracted into the IRL files **(5)**.

4.1.2 IRL to BAL operations

This set of operations produces an ILOG JRules Project including rules written in the BAL syntax. As this bridge is far more complex than the previous one – it includes additional input and output models, such as the BOM, the VOC and the B2X.

We have created five KM3 metamodels, for the BOM, VOC, BAL, B2X and project files (RP), and three TCS schemas, because only the BOM, VOC and BAL have a concrete textual syntax. The BOM and the VOC models are relatively simple. So, despite the lack a formal specification, we have been able to create the metamodels by testing successive alternatives in the JRules editor. The B2X and RP metamodels have already an Ecore specification. The definition of BAL has been quite challenging, because BAL is essentially a composition of small fragments of free text.

The bridge is a composition of twenty (20) operations. They follow the sample principle of the previous one: injection of the input files, different sets of transformations and extraction of the output models.

$bom : BOM = X1\ (java : JAVA)$ **(6)**
$voc : VOC = X2\ (bom : BOM)$ **(7)**
$bom_1 : BOM = I2[MDE\text{-}EBNF]\ (bom : BOM)$ **(8)**
$voc_1 : VOC = I3[MDE\text{-}EBNF]\ (voc : VOC)$ **(9)**
$irl_2 : IRL = I4[MDE\text{-}EBNF]\ (irl : IRL)$ **(10)**

The first step in the bridge is the generation of the VOC and BOM - **(6)**, **(7)**. Though these operations could be defined using model transformations, we rather use the existing facilities provide by the JRules API, because it has been widely used and tested. Then, the generated files, plus the input IRL files are injected into models - **(8)**, **(9)**, **(10).**

Note that **(10)** is optional when the whole DRL→IRL→BAL bridge is executed, avoiding an extra injection.

$bom_1 : BOM = T5\ (bom1 : BOM, irl_2 : IRL)$ **(11)**
$voc_1 : VOC = T9\ (bom_1 : BOM, voc_1 : VOC)$ **(12)**
$bom : BOM = E2[EBNF\text{-}MDE]\ (bom_1 : BOM)$ **(13)**
$voc : VOC = E3[EBNF\text{-}MDE]\ (voc_1 : VOC)$ **(14)**

However, the initial vocabulary is not always complete. For instance, BAL does not natively support *insert* or *retract* actions and *for*, *while* and *if* statements.

We augment the vocabulary and BOM - **(11)**, **(12)** - to support these primitives. For instance, we add a virtual *InsertAction* method, with a verbalization "*insert object X"*. The augmented models are extracted - **(13)**, **(14)** - overriding the initial generated files.

$b2x_1 : B2X = T6\ (bom_1 : BOM)$ **(15)**
$rp_1 : RP = T4\ (irl_2 : IRL)$ **(16)**
$b2x\_xml_1 : XML = T7\ (b2x_1 : B2X\ )$ **(17)**
$xml_2 : XML = E4[XML\text{-}MDE]\ (b2x\_xml_1 : XML\ )$ **(18)**
$rp\_xml_1 : XML = T8\ (rp_1 : RP\ )$ **(19)**
$xml_1 : XML = E5[XML\text{-}MDE]\ (rp\_xml_1 : XML\ )$ **(20)**

A B2X mapping model is produced with all the new virtual methods **(15)**. The project files are produced from the input IRL **(16)**.

However, the project files and the B2X mappings do not have a concrete textual syntax, i.e., they are saved in a specific XML format. We first produce an XML model (XML-ification), conforming to an XML metamodel - **(17)**, **(18)**. Then, the XML model is extracted into and XML file - **(19)**, **(20)**. Another option would be to create a single extraction operation for each model. We choose a two-step approach to be able to reuse the final XML extraction.

$bal_1 : BAL = T10(irl_2 : IRL, bom_1 : BOM, voc_1 : VOC)$ **(21)**
$bal : BAL = E6[EBNF\text{-}MDE]\ (bal_1 : BAL)$ **(22)**
$brl_1 : BRL = T11\ (bal_1 : BAL\ )$ **(23)**
$brl\_xml_1 : XML = T12\ (brl_1 : BRL)$ **(24)**
$xml_3 : XML = E7[XML\text{-}MDE]\ (brl\_xml_1 : XML\ )$ **(25)**

The central operation of the bridge is the transformation of the IRL models into the BAL models **(21)**. The transformation must search the corresponding expressions in the BOM and in

the vocabulary. Once the verbalization is found and transformed into the correct expressions (e.g., arithmetical expressions). Though the development of this transformation has been quite challenging, we do not describe further details of its implementation, since it is out of the scope of this paper. The BAL models are extracted into their textual format **(22)**.

However, the BAL rules are encapsulated into one more XML format, called BRL. Thus, the BAL rules are transformed into BRL **(23)**, which is in turn XML-ified - **(24)**, **(25)**.

4.1.3 Measurements

The development of all the KM3 metamodels, TCS definitions and transformations required considerable work. We provide some quantitative information below. Table 1 presents the size of the KM3 metamodels and TCS models.

Table 1. Size of KM3 and TCS of the main DSLs

	TCS size	**KM3 size**
BOM	260 lines, 21 templates	102 lines, 21 classes
VOC	235 lines, 16 templates	100 lines, 18 classes
DRL	654 lines, 98 templates	520 lines, 102 classes
IRL	1135 lines, 179 templates	946 lines, 185 classes
BAL	986 lines, 191 templates	935 lines, 194 classes

Table 2 lists the size of the two main transformations, one refactoring, two set of helpers, and two augmentation transformations. The size of the IRL2BAL transformation indicates the increase in complexity when dealing with models that are close to natural language.

Table 1. Size of the transformations

	Transformation size
DRL2IRL (T2)	1005 lines, 61 rules
DRLRefactor (T1)	1412 lines, 81 rules
IRL2BAL (T10)	2396 lines, 90 rules
BOM augmentation (T5)	677 lines, 32 rules
VOC augmentation (T9)	778 lines, 31 rules

4.2 Chaining and parameterization

The correct chaining of transformations is a key factor of success of the project, because the bridge must be easy to configure and to run, acting over several input and output models. An operation cannot be fired until all its parameters are loaded. Consequently, the dependency relations shown in the schema of Figure 9 must be respected during the execution.

We use the AM3 [3] tool to create scripts that execute chains of transformations. AM3 provides a set of Ant tasks [4] integrated with the Eclipse environment. Ant is a widely used tool to automate different tasks, such as compilation, file copy, and others. AM3 provides three model management tasks: *load*, *transform* and *save*. The use of AM3 enables a rapid integration with the Eclipse platform, without the need to code an ad-hoc application.

Consider the DRL to IRL bridge. The operations that have been defined take a fixed number of models/files as input and they produce a fixed number of files/models as output. However, a typical BRMS has hundreds or thousands of rules. Thus, the operations must be executed several times, and the files must be created in the correct folders. We illustrate below a simple script (using pseudo-code) that performs the transformation of several DRL files.

```
procedure DRL2IRLBridge() {

Registry r = new Registry();
Transformation t1 = r.newTransformation(“DRLRefactor”);
Transformation t2 = r.newTransformation(“DRL2IRL”);
Transformation t3 = r.newTransformation(“IRLRefactor”);
FileList fList = readfiles(“/Input/”);
For each file in fList {
   Model drl = r.inject(aFile, “DRL”); //I1
   drl = t1.execute(drl);
   Model irl = t2.execute(drl);
   irl = t3.execute(irl);
   r.extract(irl, “/Output/”); //E1
   }
}
```

First, the script creates a model registry. Then, it loads three transformation based on their file names and it reads all the input files. It loops over each file; each file is injected into a model, refactored, transformed into IRL and extracted into a text file.

This script is implemented using a combination of native Ant tasks (from its API) and AM3 tasks. We illustrate below three AM3-specific tasks: *loadModel*, *atl* and *saveModel*. We do not show the whole DRL2IRL script due to space reasons (XML is very verbose).

The first *loadModel* task loads the IRL metamodel specified in the path. The *modelHandler* attribute indicates that the file is read using the EMF API [11]. The metamodel is KM3 (in this case a metametamodel). It assigns a unique name (*IRL*) to the metamodel. The second *loadModel* task does the same for DRL.

```
<am3.loadModel modelHandler="EMF" name="IRL" metamodel="KM3"
   path="/IRL/MM/IRL-KM3.xmi"/>
<am3.loadModel modelHandler="EMF" name="DRL" metamodel="KM3"
   path="/DRL/MM/DRL-KM3.xmi"/>
```

The following task reads the *drl1.drl* text file, and injects it into a model. The task uses EBNF as the base parser, and the TCS injector implemented at *DRL-parser.jar*. It assigns a unique name *in_DRL* to the injected model.

```
<am3.loadModel modelHandler="EMF" name="in_DRL" metamodel="DRL"
   path="/Inputfolder/drl1.drl">
  <injector name="ebnf">
    <param name="name" value="DRL"/>
    <classpath>
      <pathelement location=
       "/DRL/Syntax/DRL-parser.jar"/>
    </classpath>
   </injector>
</am3.loadModel>
```

The task below executes the DRL2IRL transformation. We define the transformation path, the input and output models. The *name* attribute must be the same as the one declared in the transformation header. The *model* attribute uses the unique names that are previously affected to the models.

```
<am3.atl path="/DRL2IRL/DRL2IRL.atl">
  <inModel name="DRL" model="DRL"/>
  <inModel name="IN" model="in_DRL"/>
  <inModel name="IRL" model="IRL"/>
  <outModel name="OUT" model="out_IRL"
     metamodel="IRL"/>
</am3.atl>
```

Finally, the *saveModel* task extracts the transformed model (*out_IRL)* into the specified path. It uses the IRL TCS extractor in order to produce the IRL file.

```
<am3.saveModel model="out_IRL" path="/Output/out.irl">
  <extractor name="ebnf">
    <param name="format" value="IRL.tcs"/>
  </extractor>
</am3.saveModel>
```

The specific settings of each migration application are saved on separate property files (e.g., the path of input and output files). This way, the model management scripts can be easily reused in different migration projects.

5 Lessons learned

This experiment has shown that MDE tools reached a reasonable level of maturity allowing their use in the context of industrial projects. They have good performances, little residual bugs, and are easily available as Eclipse open source projects with fairly good documentation. The major advantage of MDE is the possibility to concentrate on the problem specification and to apply a declarative and modular approach using a small set of principles and tools.

The developed solution has reached a level of complexity in terms of the number and types of operations performed. It may be considered as a significant deployment of an MDE application in a real environment to solve a practical problem. The current implementation mainly supports Drools and JRules, but the concepts may be easily extended to be used in similar applications.

The bridges have been first tested on the two standard business rules benchmarks: *manners* and *waltz* [2] and on an “insurance claim” demonstration [9]. We have run the *manners* benchmark with settings for 16, 32, 64, 128, 256 and 512 objects. Then, the bridges were applied to the migration of more than 100 Drools rules used by a banking application. The rules are executed over the Java objects provided with the examples. In all cases – DRL, IRL and BAL - the execution of the rules produced the same results. The generated rules correctly preserve the operational semantics of their original models.

KM3 enables the definition of the metamodels in a practical way, with no particular limitations. On the syntax side, thought we could eventually reach our objective, we found more difficulties with TCS, because of its context-free approach. Such a limitation has introduced an unwanted level of complexity in our metamodels. We thus produced a BAL metamodel covering a large subset of the language, but not all. We could have used some existing code generation tool to produce the final BAL, because they have fairly good capabilities and they can be parameterized easily. However, using a bidirectional specification enabled the reutilization of the generated code into other MDE applications, for instance, if we intended to implement the transformation in the other direction, i.e., BAL to IRL to DRL.

Declarative model transformations are a concise and practical approach. The ATL transformations are particularly elegant in the DRL to IRL bridge, because the languages have similar semantics. However, the metamodels have specific details that have required considerable development time. The development of a bridge in the opposite direction would be a useful development as well. It would be interesting to verify how much of the transformation code could be reused.

As far as we know, a MDE bridge requiring such a large number of transformations has not been deeply explored before. Current solutions still need to improve their modularization capabilities. For instance, we would like to have transformations and libraries separated by packages, and with easy ways to navigate through the transformation code. We chose to modularize the transformation code using separate library of helpers.

The bridges do not have specific error handling; this is a major concern because they are designed to be used by external integrators. Thus, any time there is an error, we initially got an execution error that was not easy to understand. We improved the transformation by adding messages on points where errors could occur. However, unpredicted errors remain unhandled.

We believe that it is rather rare to have a project that executes a single transformation. Thus, the utilization of a script language coupled with property files has proved very important in the usability of the bridges. It would be unproductive to execute all these operations manually.

The use of Ant as a basis was very useful because it is a well-know tool that can be learned relatively fast. However, an integrated repository, using graphical interfaces integrated within Eclipse would help a lot. As a side consideration, we can observe here the need to get tools to handle complex networks of transformations. This is a field where MDE has still to provide new solutions.

In addition, the scripts and the property files were not models, violating the base principle "everything is a model" of a complete MDE approach. We think the script language can be integrated into a megamodeling platform [6].

So far, we have used a fixed naming convention for the folders. However, it would be valuable to have an extra presentation tool to better organize all the models and files. Another possible solution would be to use some existing workflow engine. However, we are not aware of a product that provides such MDE capabilities.

6 Conclusions

In this paper, we have presented an experiment to provide interoperability between industrial BRMS. The utilization of MDE techniques enabled to successfully develop two bridges amongst rule languages with different degrees of expressiveness.

On top of their expected practical usefulness, implementing these bridges has revealed both the strength and some limits of available MDE tools. We have greatly appreciated the power of declarative programming associated with a scripting platform, but we have also suffered from the main limit relative to the parsing of non context-free grammars.

Our approach has been validated by executing the bridge on a set of well-known benchmarks for business rules, on a demonstrative example and on an industrial application. The bridges produced the expected results.

To the best of our knowledge, this is the first approach implementing a solution for transformations with such a large number of transformations. The discovery, development and chaining of complex transformations has been a challenging task. This project and particularly Figure 9, has been used by Don Batory [5], to illustrate the need for regular and complete frameworks for Model Driven Engineering.

The presence of several transformations, models and metamodels showed that megamodeling is a crucial issue when dealing with large projects. The scripting language is a first step that helped a lot, together with its parameterization. However, we think much more work can still be done on that area, especially in the specification of generic megamodeling platforms.

There are several possibilities for future work, such as the creation of similar bridges amongst different rule languages (e.g., Blaze Advisor). The parsing can be internationalized into different languages, which is a common requirement of industry. Finally, we plan to study how to use MDE techniques to parse context-aware grammars, or even natural language.

The main conclusion of this work is that MDE has reached a level of maturity that allows using it to revisit traditional solutions to complex real-life problems and not only to toy examples. However, care should be taken to control the accidental complexity generated by this solution. New ways to manage important numbers of related modeling artifacts are among the most urgent needs to prepare model driven engineering for moving to full-scale industrial deployment.

7 Acknowledgments

This work has been partially supported by ANR IdM++ project.

8 References

[1] Abouzahra A, Barbero M. Implementing two business rule languages: PRR and IRL. Ref. site: http://www.eclipse.org/m2m/atl/usecases/PRR2IRL/. 03/07

[2] Academic Benchmark performances. Ref. site: http://blogs.ilog.com/brms/2007/10/22/academic-benchmark-performance/, 27-12-2007

[3] Allilaire F, Bézivin J, Brunelière H, Jouault F. Global Model Management. In proc. of eTX Workshop at the ECOOP 2006, Nantes, France

[4] ANT, The Apache Ant Project: http://ant.apache.org/

[5] Batory D, Azanza M, Saraiva J. The Objects and Arrows of Computational Design. Keynote talk at MoDELS 2008

[6] Bézivin J, Jouault F, Valduriez P. On the Need for Megamodels. In proc. of OOPSLA/GPCE: Best Practices for Model-Driven Software Development workshop, 19th OOPSLA, 2004

[7] Blaze Advisor. Ref. site: http://www.fairisaac.com/fic/en/product-service/product-index/blaze-advisor/. April 08

[8] Brownston L, Farrell R, Kant E. Programming Expert Systems in OPS5 Reading, Massachusetts: Addison-Wesley, (1985)

[9] Drools Examples. Ref. site: http://download.jboss.org/drools/release/4.0.4.17825.GA/ drools-4.0.4-examples.zip. 31-03-2008

[10] Eclipse Modeling Project. Ref. site: http://www.eclipse.org/modeling/. April 08

[11] EMF. Eclipse Modeling Framework. Ref. site: http://www.eclipse.org/emf, August 2008

[12] ILOG JRules. Ref. site: http://www.ilog.com/products/jrules/index.cfm. April 08

[13] ILOG Rule Languages. Ref. site: http://www.ilog.com/products/jrules/documentation/jrules67 July 08

[14] JBoss Drools User Guide. Ref. site. http://downloads.jboss.com/drools/docs/ 4.0.4.17825.GA/html_single/index.html. 15-01-2008

[15] JBoss Drools. Ref. site: http://www.jboss.org/drools/. July 08

[16] Jouault F, Allilaire A, Bézivin J, Kurtev I.ATL: a Model Transformation Tool. Science of Computer Programming 72(3, Special Issue on Second issue of experimental software and toolkits – EST):31-39, 2008

[17] Jouault F, Bézivin J, Kurtev I. TCS: a DSL for the Specification of Textual Concrete Syntaxes in Model Engineering. In proc. of GPCE'06, Portland, Oregon, USA, pp 249-254

[18] Jouault F, Bézivin J. KM3: a DSL for Metamodel Specification. In proc. of 8th FMOODS, LNCS 4037, Bologna, Italy, 2006, pp 171-185

[19] 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

[20] Kolovos D, Paige R, Polack F. A Framework for Composing Modular and Interoperable Model Management Tasks, In proc. of Workshop on Model Driven Tool and Process Integration (MDTPI), EC-MDA 2008, Berlin, Germany

[21] Kurtev I, Bézivin J, Aksit M. Technological Spaces: An Initial Appraisal. In proc. of CoopIS, DOA'2002 Federated Conferences, Industrial track, 2002, Irvine, California, USA

[22] Kurtev I, Bézivin J, Jouault F, Valduriez P. Model-based DSL Frameworks. In proc of. Companion of OOPSLA 2006, 22-26/10, Portland, OR, USA, pp 602-616

[23] OMG. Object Management Group. Ref. site: http://www.omg.org. April 08

[24] OpenArchitectureWare. Ref. site: http://www.openarchitectureware.com/. October 2008

[25] Owen J. Business rules management systems. Extract business rules from applications, and business analysts can make changes without IT breaking a sweat. Infoworld, 25-06-2004

[26] Production Rule Representation (PRR), Beta 1, Document Number: dtc/2007-11-04 Ref. site: http://www.omg.org/spec/PRR/1.0/

[27] Rule Interchange Format (Working Group). Ref. site: http://www.w3.org/2005/rules/wiki/RIF_Working_Group. April 08