ISML-MDE

A practical experience of implementing a model driven environment in a software development organization

Maria Consuelo Franky and Jaime A. Pavlich-Mariscal
Dpto. de Ingeniería de Sistemas, Pontificia Universidad Javeriana, Bogota, Colombia

Maria Catalina Acero and Angee Zambrano
Heinsohn Business Technology, Bogota, Colombia

John C. Olarte
Pontificia Universidad Javeriana, Bogota, Colombia, and

Jorge Camargo and Nicolás Pinzón
Heinsohn Business Technology, Bogota, Colombia

Received 27 April 2016
Revised 30 June 2016
Accepted 8 July 2016

Abstract

Purpose – This purpose of this paper is to present ISML-MDE, a model-driven environment that includes ISML, a platform-independent modeling language for enterprise applications; ISML-GEN, a code generation framework to automatically generate code from models; and LionWizard, a tool to automatically integrate different components into a unified codebase.

Design/methodology/approach – The development comprises five stages: standardizing architecture; refactoring and adapting existing components; automating their integration; developing a modeling language; and creating code generators. After development, model-to-code ratios in ISML-MDE are measured for different applications.

Findings – The average model-to-code ratio is approximately 1:4.6 when using the code generators from arbitrary models. If a model transformation is performed previously to the code generation, this ratio raises to 1:115. The current validation efforts show that ISML properly supports several DSL essential characteristics described by Kahraman and Bilgen (2015).

Research limitations/implications – ISML-MDE was tested on relatively small applications. Further validation of the approach requires measurement of development times and their comparison with previous similar projects, to determine the gains in productivity.

Originality/value – The value of ISML-MDE can be summarized as follows: ISML-MDE has the potential to significantly reduce development times, because of an adequate use of models and transformations. The design of ISML-MDE addresses real-world development requirements,

This article is part of the Project "Desarrollo de nuevos métodos y tecnologías para acelerar la construcción de software: Un enfoque basado en modelos y frameworks de generación avanzados", executed by the SIDRe research group of the Pontificia Universidad Javeriana and Heinsohn Business Technology, co-financed by Colciencias: 1,203-562-37822.

International Journal of Web Information Systems
Vol. 12 No. 4, 2016
pp. 533-556

1744-0084
DOI 10.1108/IJWIS-04-2016-0025

obtained from a tight interaction between the researchers and the software development company. The underlying process has been thoroughly documented and it is believed it can be used as a reference for future developments of MDE tools under similar conditions.

Keywords Enterprise applications, Code generation, Legacy components integration, Model-driven engineering, Xtext

Paper type Research paper

1. Introduction

Enterprise applications are systems that "are designed to integrate computer systems that run all phases of an enterprise's operations to facilitate cooperation and coordination of work across the enterprise" (Gartner-Inc, 2015). Because of their size and heterogeneity, enterprise applications are inherently complex (Fowler, 2002), which introduces several challenges to the development of these kind of applications.

During the past years, frameworks have been developed to assist developers to create complex systems (Oracle, 2015a; Microsoft, 2015a). These frameworks provide several components to simplify tasks commonly performed by enterprise applications and provide a uniform architecture to deploy and interoperate heterogeneous applications. Even though these frameworks provide considerable power to developers, they have important drawbacks. One of the most important ones is that they are still significantly complex. Frameworks, such as Java EE Oracle (2015a) have thousands of classes, several different configuration file formats and deployment platforms, which are provided by different vendors, require learning additional application programming interfaces (APIs) and procedures to properly run enterprise systems over them. Overall, the learning curve for these frameworks can be very steep.

Integrated development environments (IDE) address the above problem to a certain degree. IDEs facilitate development by organizing the code and automating some tedious tasks (Eclipse-Foundation, 2015a; Microsoft, 2015b; Oracle, 2015d; JetBrains, 2015). However, programmers still have to learn an important portion of the APIs of the frameworks and related information to adequately develop enterprise applications.

One of the main reasons for the above complexity is the lack of adequate abstraction mechanisms. Developers have to address a lot of these frameworks' details to properly develop and deploy enterprise applications. A potential solution to this problem is model-driven engineering (MDE) (Kent, 2002). MDE abstracts software-related problems and solutions into models, and uses automatic code generation tools to create part of or all of the code that implements those solutions.

This paper presents the practical experience of developing ISML-MDE, a model-driven environment that comprises three main components: ISML, a modeling language for enterprise applications; ISML-GEN, a code generation framework; and LionWizard, a tool to automatically integrate different code components – including components with crosscutting code – into a unified codebase. The name ISML is the acronym for Information Systems Modeling Language. This name reflects the vision of this language, which is to model most elements of an information system, both from the problem domain and the solution domain. As of the writing of this paper, ISML supports the specification of the solution domain. Ongoing work is to extend ISML to model the problem domain.

Although there are modeling environments similar to ISML-MDE (Baresi *et al.*, 2006; Groenewegen *et al.*, 2015; Kroiss *et al.*, 2009; OMG, 2015; Souer *et al.*, 2008), our approach

differs in that it is designed to facilitate adoption for enterprise software development organizations. To achieve this goal, ISML-MDE was developed in tight collaboration with a large-scale software development company. Design decisions, such as syntax, semantics, design patterns and code generation strategies were driven by the specific necessities of these kind of organizations.

Although the process of developing ISML-MDE is based on requirements elicited from a specific software development organization, we believe that these requirements are common to many similar organizations; thus, ISML-MDE has the potential to benefit a wider audience. Moreover, ISML and the essential parts of ISML-GEN are open source and available in a repository (Pavlich-Mariscal, 2015).

The remainder of this paper describes ISML-MDE and the practical issues of developing it, particularly the strategies used to address specific modeling requirements and achieve an adequate integration with existing assets in a software development organization. Section 2 describes related work; Section 3 summarizes the context of the problem, the solution (ISML-MDE) and its rationale; Section 4 details the process to create ISML-MDE, including the design decisions behind the tool, the architecture, its components (ISML, ISML-GEN and LionWizard) and the integration with existing frameworks or libraries; Section 5 describes ISML, the proposed modeling language; Section 6 describes ISML-GEN, the code generation framework; Section 7 describes three case studies used to test ISML-MDE and the potential benefits in terms of automation and abstraction; and Section 8 concludes and describes future work.

2. Related work

From the modeling point of view, there are several approaches to specify web and enterprise applications. The Interaction Flow Modeling Language (IFML) is an Object Management Group (OMG) standard based on WebML (WebRatio, 2015). IFML focuses on the specification of "content, user interaction and control behavior of the front-end of software applications" (OMG, 2015). As such, IFML specifies presentation concerns of an application, relying on other languages, such as SoaML (OMG, 2012) to integrate with services. As it is based on WebML, information is conveyed through visual means (graph-based diagrams).

A similar language is the one provided by Obeo's Cinematic Designer (Obeo, 2015d), which focuses on the visual aspects of a web application. This language is complemented by other languages developed by Obeo: the Entity Designer (Obeo, 2015c) and the SOA Designer (Obeo, 2015b), which model entities and services, respectively.

The UML-Based Web Engineering Approach (UWE) (Kroiss *et al.*, 2009) is a UML profile to specify content, navigation, business processes and presentation of a web-based application. Being a UML profile, it can be easily specified through case tools that support UML. Mubin *et al.* (Mubin and Jantan, 2014) propose a similar approach – a UML profile to specify a user interaction model from a requirements specification. The user interaction model captures a conceptual model, a navigational model and a user interface model. In their essential form, none of them provide support for services.

All of the above languages are visual, while ISML is a textual language. Although there is no general consensus on the advantages of visual versus textual languages (Mazanec and Macek, 2012), in our experience, the textual focus of ISML provides two main advantages over visual languages: it facilitates the specification of more details

about an application model and integrates easily with version control tools. However, it is harder to specify associations between model entities.

W2000 (Baresi *et al.*, 2006) is a notation to specify complex web applications. It provides facilities to specify the standard components of a Model-View-Controller (MVC) architecture (Gamma *et al.*, 1995): an Information Model for the Model part in MVC, a Navigation Model for the Controller part, and a Presentation model for the View part. W2000 also includes a model to specify services. Although W2000 is able to represent the same elements as our approach, ISML combines all three elements of MVC, plus services, into a single, integrated model.

The approach that is closest to ISML is WebDSL (Groenewegen *et al.*, 2015), a textual language to specify web applications. ISML differs in various elements. First, ISML provides the Controller as a modular unit to group-related actions that can be performed over the system. Second, ISML provides a simpler mechanism to specify events triggered from the user interface called the arrow "−>" operator. Third, WebDSL has some platform-specific elements (e.g. the "ajax" keyword) that may limit the possible target platforms. In contrast, ISML aims to be as independent as possible from the implementation platform. Finally, concerns, such as access control, are managed at the code generation level, so that designers can be oblivious to these concerns.

From the point of view of code generation, there are several frameworks to generate enterprise and web applications. AndroMDA (AndroMDA, 2014) is a code generation framework that uses the Velocity (Apache-Foundation, 2010) template engine to transform simple stereotyped UML models into J2EE or NET implementations. AndroMDA provides code generation components called "cartdriges" to modularize different code generators. Similarly, Taylor (Gilbert, 2008) is a code generation framework that takes as input a simple stereotyped UML model and generates a Java EE application. In contrast, ISML-GEN is based on the Xtext (Eysholdt and Behrens, 2010) code generation framework, which provides better integration with development tools, such as Eclipse (Eclipse-Foundation, 2015a). ISML-GEN also provides a wrapper for the Xtext code generation APIs, which makes it much simpler to develop code generators.

Zathuracode (Gomez, 2015) is another code generator for Java EE that takes as input a database schema and generates an application with basic CRUD (Create, Retrieve, Update, Delete) operations over that schema. A related approach is Equanda (Auwera, 2013) that takes as input an XML file containing the main application entities and generates a Java EE code to access those entities. These tools only support a simple model as input that only denotes application entities. In contrast, ISML and ISML-GEN also support services, navigation and presentation concerns.

The Acceleo JavaEE Generators (Obeo, 2015e) are a set of code generators that take as input a model specified in the three domain specific modelers specified above (Entity Designer, SOA designer and Cinematic Designer) and use the Acceleo (Obeo, 2015a) template engine to generate code in Java EE. While Acceleo and ISML-GEN's underlying Xtext framework are very similar in capabilities and ease of use, the layer that we provide over Xtext has the potential to facilitate development and organization of code generation templates.

Overall, the main characteristics that differentiate ISML-MDE with most related approaches are: the focus on a textual modeling language, support for MVC constructs and services, platform independence, simpler model querying for code generation and facilities to integrate legacy code components in the ISML models.

3. Overview of the problem and the solution

ISML-MDE was developed as a joint project between the Pontificia Universidad Javeriana (PUJ, 2015), a Colombian university, and Heinsohn Business Technology (HBT) (Heinsohn, 2015), a Colombian software development organization. ISML-MDE is the result of a process that spans more than just the design of the tool itself, as many of the problems at HBT were not only related to abstraction and automation but also to standardization and large-scale reuse.

This section describes the context of the problem, design decisions, rationale and the architecture of ISML-MDE.

3.1 Context of the problem

HBT is a software development organization that focuses on large-scale enterprise applications for governmental and commercial companies. Their main development platform is Java EE (Oracle, 2015a).

As of the beginning of this project, there was little automation in the development tasks. Most of the development was performed using Eclipse with Java EE plugins (Eclipse-Foundation, 2015a). Over the years, HBT had developed several components in that platform to address different concerns such as access control, batch processing, persistence and presentation, among others (Franky and Pavlich-Mariscal, 2014). Although these components were frequently required by other projects, their reuse was difficult, because many manual tasks were required to properly integrate them into the codebase. Moreover, each software project at HBT had a different architecture, which made it harder to integrate the existing components. Overall, it was necessary to accelerate the process to integrate components into the codebase of new projects. This goal could be achieved by standardizing the architecture and automating the integration tasks.

Another important issue at HBT was the complexity of the frameworks used in development. For instance, the Java EE 7 (Oracle, 2015a) API has more than 2,000 classes. Several vendors provide their own APIs to complement the Java EE API. In addition to the APIs, developers often have to learn the syntax of configuration files and deployment procedures to test the applications. Overall, the amount of information that a developer must learn to properly use these frameworks is very high and the learning curve is steep. Therefore, developing enterprise applications with these frameworks is far from trivial.

One solution to address the complexity of these frameworks is the use of tools such as IDEs (Gartner-Inc, 2015). These tools assist programmers in creating a codebase, cross-referencing different parts of the code, providing a degree of automatic code generation and facilitating application deployment, among others (Eclipse-Foundation, 2015a; Microsoft, 2015b; Oracle, 2015d; JetBrains, 2015). HBT developers used Eclipse (Eclipse-Foundation, 2015a) as the main IDE to develop enterprise applications. Although IDEs significantly simplified the development tasks at HBT, programmers still had to learn an important portion of the APIs of the frameworks and related information to adequately develop enterprise applications.

One of the main causes of the above complexity is the lack of adequate abstraction mechanisms. As a consequence, developers must learn too many details about the inner workings of enterprise frameworks to properly use them – information that is not directly related to the domain of the solution. Another consequence is that developers

need to perform many repetitive tasks to implement most of the functionality in enterprise applications.

A proper abstraction can hide irrelevant details in the development of enterprise applications, facilitate the automation of repetitive tasks and let developers focus on the essential elements of the solution. In that regard, MDE (Kent, 2002) is a potential solution to this problem. MDE uses models as the central artifact for software evolution, and code generators are used to transform these models into the code that implements them. Models are, by definition, an abstraction of something that exists in the real world or something that will be created in the real world in the future such as a software solution (Kühne, 2005). In other words, models can be used as a way to hide irrelevant details in enterprise applications, while code generators can be used to automatically realize those abstract models into more detailed enterprise software, reducing the need of developers to perform repetitive tasks.

3.2 Requirements

The main purpose for our project is to create an MDE-based tool to address the above problems. Overall, the main requirements underlying ISML-MDE are the following:

- *Abstraction*: ISML-MDE must abstract a significant amount of details from enterprise applications. Engineers must be oblivious to platform-specific details and focus only on a high-level description of the solution;
- *Flexibility*: The degree of abstraction provided by ISML-MDE must not hinder flexibility; in other words, it should be able to specify most of the application's functionality;
- *Automation*: ISML-MDE must provide a mechanism to automatically translate models into working implementations. These mechanisms should be modular and configurable;
- *Simple version control*: ISML-MDE must seamlessly integrate with existing version control tools to facilitate teamwork at the modeling level;
- *Facilitate learning*: ISML-MDE must provide mechanisms to use only parts of its functionality and facilitate learning by gradually enabling more advanced features; and
- *Simple integration*: ISML-MDE must provide mechanisms to seamlessly integrate with existing assets at a software development organization.

3.3 The architecture and design decisions of ISML-MDE

Figure 1 depicts the architecture of ISML-MDE. There are three main components: ISML, the language to specify enterprise applications; ISML-GEN, the code generation framework; and LionWizard, the tool that automatically integrates project archetypes and software components to create the codebase of new projects.

The components interact in the following way: designers use ISML to model the enterprise application. In parallel, programmers may use LionWizard to choose the archetypes and components that will be part of the new application and to automatically integrate them into the new codebase. This codebase has the basic infrastructure of the application. Designers then use ISML-GEN to automatically generate the code of the application over this infrastructure. They can select different generators according to

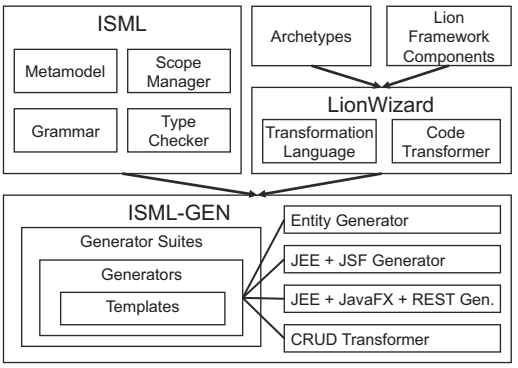

Figure 1. ISML-MDE architecture

their needs, for instance, to create either Java EE with Java Server Faces (JSF) user interfaces (Oracle, 2015c) or Java EE with JavaFX user interfaces (Oracle, 2015b).

In addition, ISML-MDE is based on the following design decisions:

- *Textual language*: ISML is a textual language. The reason for this decision is threefold. First, it facilitates the integration with existing version control tools, which work with text files (Git-Project, 2015; Apache-Foundation, 2015a). Second, in our experience, it provides a degree of flexibility to specify details that are not easily achieved in graph-based diagrams. Third, its textual syntax is similar to common programming languages, such as Java or C# (but much simpler than those languages), which facilitates the adoption in organizations that use those languages;
- *Model-View-Controller specification*: ISML provides constructs to specify the three main elements of a MVC architecture. In addition, ISML can also specify services used by an application. There are two reasons to adopt MVC. First, it is the architectural pattern used by the most widespread enterprise application frameworks (Microsoft, 2015a; Oracle, 2015a); thus, it makes it easier to transition from models to code. Second, since MVC is a commonly known pattern across enterprise application developers, learning ISML becomes easier for them;
- *Partial modeling*: ISML can specify most of a system functionality. To facilitate learning, ISML is designed in such a way that designers can start using a small subset of the language and gradually introduce new features as they learn the basic elements. In other words, designers do not need to learn the entire language to start creating applications. This should facilitate the adoption of the language;
- *Modular code generation*: ISML-GEN includes several code generation modules that automate the implementation of specific parts of the application. Each of them can be enabled/disabled to transition models into different parts of the implementation and can have multiple configuration options; and
- *Automatic integration with existing components*: LionWizard provides an automatic way to compose existing components into the codebase of the application. In addition, ISML provides a series of model libraries whose interfaces match the components that are integrated at the code level. When ISML-GEN generates the implementation of a model, the generated code gets automatically integrated with the implementation of the existing components at the model level.

4. ISML-MDE development process

Based on the information described in Section 3, this section details each phase of the development of ISML-MDE and describes their results with respect to the architecture of Figure 1.

Figure 2 is an overview of the process to create ISML-MDE. Although some phases are not tightly related to MDE, we believe that they are important because they explain the context in which ISML-MDE was created and make it easier to understand the design decisions behind this tool.

The first phase is to standardize the architecture of software projects, to establish a common way to design enterprise applications in the organization. The second phase is to refactor and adapt the existing components to the standard architecture, to effectively realize the architecture in the development process of the organization.

The third, fourth and fifth phases develop the core components of ISML-MDE. The third phase is the development of a tool called LionWizard (Franky and Pavlich-Mariscal, 2014), to automate the integration of existing components and accelerate the creation the codebase in new software projects.

The fourth phase is the creation of ISML as the modeling language to specify enterprise applications. ISML is intended to provide the required abstraction mechanism to address the complexity of enterprise application frameworks. The fifth phase is the creation of code generators to translate an ISML model into different Java EE implementations. These code generators support the abstraction provided by ISML, making developers oblivious to most implementation details.

The following sections explain each phase and their results with respect to the architecture of Figure 1.

4.1 Standardize architecture

As of the start of this process, HBT had no standard architecture for its projects. As a consequence, software components were difficult to reuse and the architectural know-how was not easily transferred from old projects to new projects.

The first phase of the process of Figure 2 addresses this problem by choosing and establishing a standard architecture for new projects, as shown in Figure 3. The design is based on a common multi-tier architecture (Schuldt, 2009) with several layers that modularize different concerns of an enterprise application: presentation, page flow (application layer), services, domain entities and data. Since the HBT uses Java EE

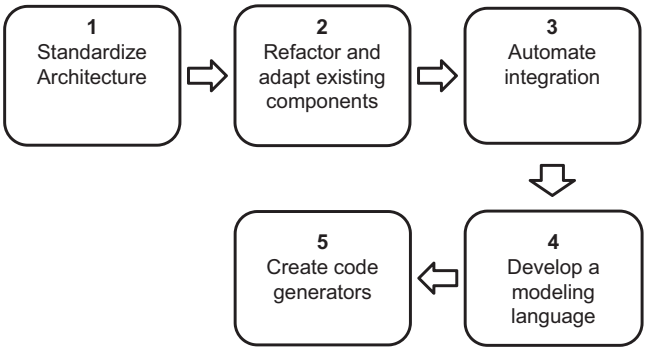

Figure 2.
The process to create ISML-MDE

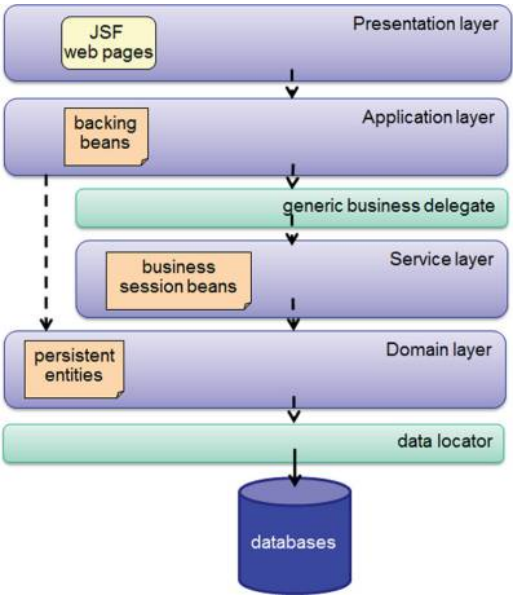

Figure 3.
Standardized architecture at HBT

(Oracle, 2015a) as the main development framework, the figure indicates the elements of that framework associated to each layer. However, the overall architecture is sufficiently generic to be used with different frameworks such as. NET (Microsoft, 2015a).

4.2 Refactor and adapt existing components

Based on the above architecture, developers refactored the existing components at HBT to facilitate their integration in future projects that use this architecture (second phase of Figure 2). These components, code-named "Lion Framework" (Figure 1), have been developed by HBT during the last years to address different concerns of enterprise applications: file processing, auditing, access control, transparent persistence, etc. To improve their evolution and facilitate reuse for future projects, designers aligned each component to the architecture (each component addresses different architectural layers) and organized them into a Maven (Apache-Foundation, 2015b) repository. In addition, the work in this phase yielded some software archetypes (Apache-Foundation, 2015c), based on the standard architecture, which could be used to create the skeletons of new projects.

4.3 Automating integration

The third phase in the process of Figure 2 is to automate the integration of the refactored components to accelerate the creation of the codebase of new projects.

The use of Maven archetypes and repositories facilitates the creation of the codebase of new projects. However, these technologies are insufficient to ensure a seamless integration of all the components. The way Lion components are designed required some manual tasks to properly integrate them: modify parts of the code, specify configuration files, etc. These tasks often differ depending on the elements of the target application platform such as applications servers, databases, etc. As a consequence, the

integration of components used to be a tedious and cumbersome task that often took weeks to complete (Franky and Pavlich-Mariscal, 2014).

To address the above issues, this phase created LionWizard (Franky and Pavlich-Mariscal, 2014), a code transformation tool that automatically creates the skeleton of a new project as a Maven archetype and seamlessly integrates it with the chosen Lion components. LionWizard has two main functionalities: a code transformer that automatically manipulates component files with the specific data to integrate them with other components and an XML-based language to specify the transformation processes for new components. This language ensures the adaptability of LionWizard to facilitate the integration of future components.

4.4 Develop a modeling language

LionWizard is a code transformation tool that significantly improves the creation of the initial codebase of an enterprise application project. However, as its focus is only on the initial stages of development, LionWizard is insufficient to address the complexity of enterprise application development.

As explained in Section 3.1, it is necessary to provide software engineers with abstractions to hide the complexity of enterprise applications and automation mechanisms to translate those abstractions into working software implementations. To address the former, this phase focused on developing the ISML, a language to specify enterprise applications. The name of this language reflects a long-term vision that is currently in progress. Nevertheless, in its current version, ISML focuses on the solution domain (software design). Future work is to expand the language to address the problem domain (requirements specification). Section 5 describes ISML in more detail.

4.5 Create code generators

The last step in the process is the creation of ISML-GEN, a suite of code generators that includes the Entity generator, to create persistent entity code; the JEE+JSF generator, to create a Java EE application with user interfaces and business logic using the JSF framework (Oracle, 2015c); and the JEE+JavaFX+representational state transfer (REST) generator, which creates Java EE applications with JavaFX (Oracle, 2015b) user interfaces and REST services to realize the business logic. Section 6 describes ISML-GEN in detail.

5. Information systems modeling language (ISML)

Phase 4 of the process described in Section 4 is the creation of ISML. This section describes ISML in detail.

From the point of view of the Model Driven Architecture (MDA) (OMG, 2009) standard, ISML is a language to create Platform Independent Models (PIM). In other words, ISML provides constructs to specify the essential elements of a software solution without detailing the target platform. ISML is sufficiently general to model enterprise applications in various domains, while being sufficiently specific to easily translate the models into working implementations. As ISML is semantically close to an enterprise application, it is simpler to directly generate code from ISML models rather than performing an intermediate transformation from PIM to PSM.

ISML-MDE, the whole environment, is a software tool based on the Eclipse Modeling Framework (Foundation, 2013), a set of Eclipse (Eclipse-Foundation, 2015a) plugins and libraries to develop model-driven tools based on the MDA standard. ISML-MDE

facilitates to create ISML models, modify them in syntax-aware editors and perform automatic scope management and type checking. The following sections describe the main components of ISML and the environment to create these models.

5.1 ISML syntax

In its current version, ISML is strongly related to the MVC pattern (Gamma *et al.*, 1995). The MVC decomposes an application design into three categories: Model, which manages the information of the problem domain; View, which is in charge of visualizing the Model information; and Controller, which manages the user interaction and overall control flow of the system.

Figure 4 shows the main elements of ISML. The language is able to specify all of the elements in an MVC-based system: *entities* that represent the domain information of the application; *pages* that display the information of entities; and *controllers* that perform activities requested from the pages and define the page flow. In addition, ISML supports the specification of *services*, which are interfaces to objects that perform specific business processes and can either be fully specified in an ISML model or be reused from predefined services that already exist in the target platform.

To better illustrate these concepts, the remainder of this section presents a simplified example of a company management application. In particular, the example focuses on the information about a company and its employees.

Algorithm 1 is an example of an ISML entity that represents a company. An entity can only contain attributes. Some of them may be part of an association, such as the **employees** attribute, which is one end of an association whose opposite is the attribute **employer** of the **Person** entity. Constraints over attributes are specified with the keyword "must be". In this example, the company's tax ID must have a size between 1 and 20:

Algorithm 1: Entity Example in ISML

```
entity Company {
   String name
   String taxID must be Size(1,20)
   Date creationDate
   Array<Person> employees opposite employer
}
```

Algorithm 2 is an example of a page to edit companies. A page can receive many parameters. In this case, the parameter is an instance of the **Company** entity. A page can have several widgets or visual components, for example, forms, text boxes, calendars (date pickers), buttons, etc.

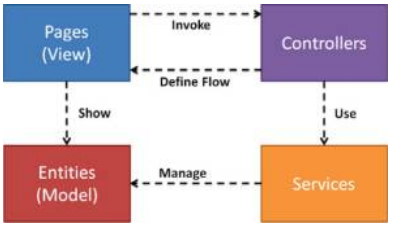

Figure 4.
Main elements in ISML

Algorithm 2: Page Example in ISML

```
page EditCompany(Company company) controlledBy CompanyManager {
  Form {
    Text("Name", company.name, 25, 1)
    Text("Tax ID", company.taxID, 25, 1)
    Calendar("Creation Date", company.creationDate, null, "dd/MM/yyyy", true,
"inline")
    Button("Save", true) -> saveCompany(company)
    Button("Cancel", false) -> listAll()
  }
}
```

Pages are managed by controllers. In this example, the controller **CompanyManager**, shown in Algorithm 3, is the one that processes all of the events coming from the **EditCompany** page. **CompanyManager** provides two actions: **listAll** that retrieves all instances of the company from a database and **saveCompany** that stores a company in a database:

Algorithm 3: Controller Example in ISML

```
controller CompanyManager {
  has Persistence<Company> persistence
  default listAll() {
    show CompanyList(persistence.findAll())
  }
  saveCompany(Company company) {
    persistence.save(company)
    -> listAll()
}
}
```

Another important element in ISML is the arrow operator "−>", which denotes that the execution of a program continues through the action pointed by an arrow. When the user clicks the **Save** button of the **EditCompany** page (see Algorithm 2), the execution continues through the **saveCompany** action (see Algorithm 3). Similarly, at the end of the **saveCompany** action of **CompanyManager**, the arrow operator indicates that the execution continues through the **listAll** action.

To specify the page flow, the keyword **show** denotes the page that should be displayed after an action is finished. In this example, the **listAll** action shows the **CompanyList** page (the latter is not explained in this paper for space reasons).

Controllers are meant to specify the page flow of an application. To execute business-specific processes, controllers delegate to services, which encapsulate processes that do not have a direct relation with page flow and user interaction. The controller of Algorithm 3 uses the keyword **has** to indicate that it will delegate some functionality into the **Persistence** service. Algorithm 4 details this service, which manages entity storage in databases:

Algorithm 4: Service Example in ISML

```
service Persistence <T> {
  native Void save(T obj)
  native T load(T obj)
```

```
    native Void delete(T obj)
    native Array<T> findAll()
}
```

One important feature in ISML is the partial modeling facilities, i.e. to give designers the ability to decide which parts of the application will be specified at the model level or at the code level. In ISML, this is realized through the keyword **native**. This keyword is analogous to the **native** keyword in Java (Oracle, 2014). In ISML, a **native** method means that the body of that method must not be specified at the model level but at the code level. This keyword can also be used to denote actions in controllers whose body will be specified at the code level.

One of the advantages of **native** is that it facilitates learning of ISML. At early adoption stages, engineers do not know all of the language features, so they can declare most actions and methods as **native** and implement them at the code level, in a programming language that they might know better. As engineers learn more features of ISML, they can start modeling more actions and methods with this language and relying progressively less on the target programming language.

Another advantage of **native** is that it can be used to provide model libraries that abstract existing components at the code level. In this example, the **Persistence** service has all of its methods declared as **native**. In addition, there is a component with the same name at the code level that implements those same methods. Using a specific configuration in the code generators, the generation of the **Persistence** code can be disabled and the final implementation can use the *Persistence* component that already exists at the code level.

5.2 ISML metamodel

The metamodel of ISML contains 71 classes. For space reasons, this paper only focuses on some specific parts to illustrate the main language features. Figure 5 shows the classes in the ISML metamodel that represent the main language components. **TypeSpecification** is the superclass of every element in ISML associated to a type, such as entities, views, controllers, services, etc.

One important consequence of this design decision is that primitive data types (e.g. Integer, Float, String, etc.) and primitive widgets (e.g. text boxes, buttons, etc.) are instances of **primitive** and **widget**, respectively. This means that the users of ISML-MDE do not need to modify the metamodel to add new primitives to the language and only need to know the essentials of ISML to perform these changes. For instance Algorithm 5 and 6 show some primitives specified by ISML itself:

Algorithm 5: ISML Primitives Specified Using ISML

```
primitive Any
primitive Null
primitive Void
primitive String
primitive Integer
primitive Float
primitive Boolean
```

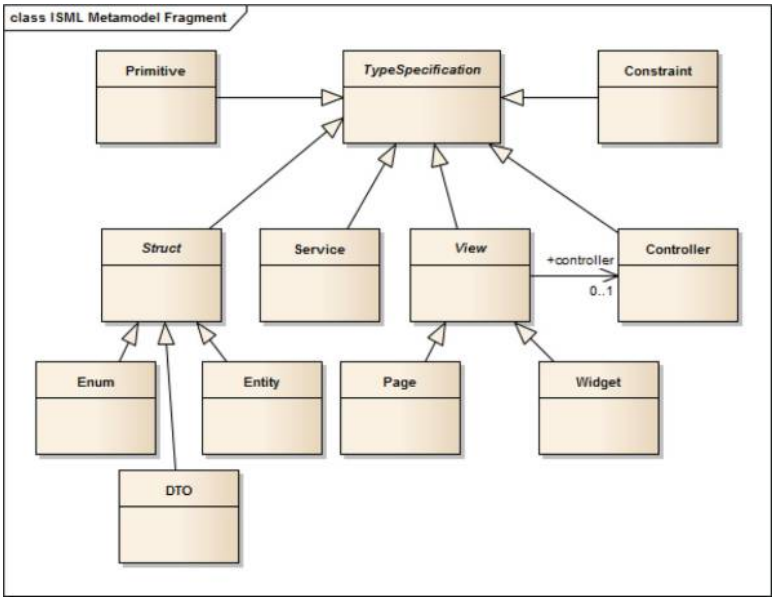

Figure 5.
SML main elements metamodel

Algorithm 6: ISML Widget Primitives Specified Using ISML

```
widget Label(Any text)
widget Image(String imageURL)
widget Link(String label, String url)
widget Form(Block body)
widget Text(String label, Any value, Integer columns, Integer rows)
widget OutputText(String label, Any value)
widget Password(String label, Any value, Integer length)
```

This feature is particularly useful in software development organizations, as they can customize ISML to their specific needs without knowing how to modify the metamodel.

In addition, the ISML metamodel denotes some more specific types such as constraints that apply to entity attributes, enumerations and data transfer objects. Overall, most of the remaining classes in the ISML metamodel focus on specifying instances of any **TypeSpecification** and their contents.

5.3 Scope manager and type checker

The ISML-MDE environment also provides a scope manager and type checker for ISML. The scope manager finds and verifies the correctness of all the cross references between elements in the model and provides content assistance when writing the model. For instance, Figure 6 shows the content assist that suggests possible controller actions that could be executed when a "Cancel" button is pressed.

The type checker verifies that the typed elements in the language (e.g. entities, widgets, primitives, etc.) are correctly used in the model and ensures that the code generation that follows will yield code without type errors. Some important features of the type system in ISML are action and method overloading, generic types and inheritance.

5.4 Model transformation facilities

ISML-GEN also includes CRUD-Transformer, a module to transform ISML models to add CRUD functionality. CRUD-Transformer takes as input an entity in ISML and creates all of the controllers and pages required to perform basic CRUD operations.

For instance, Figure 7 shows three entities:

(1) **Address**;

(2) **Company**; and

(3) **Person**.

CRUD-Transformer generates all of the views and controllers. The figure highlights the **Company** entity and all of the elements generated to provide CRUD functionality for **Company** entities.

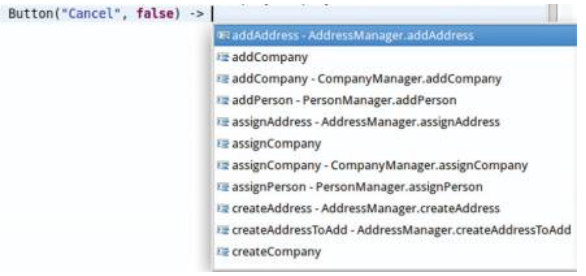

Figure 6.
Scope-based content assist

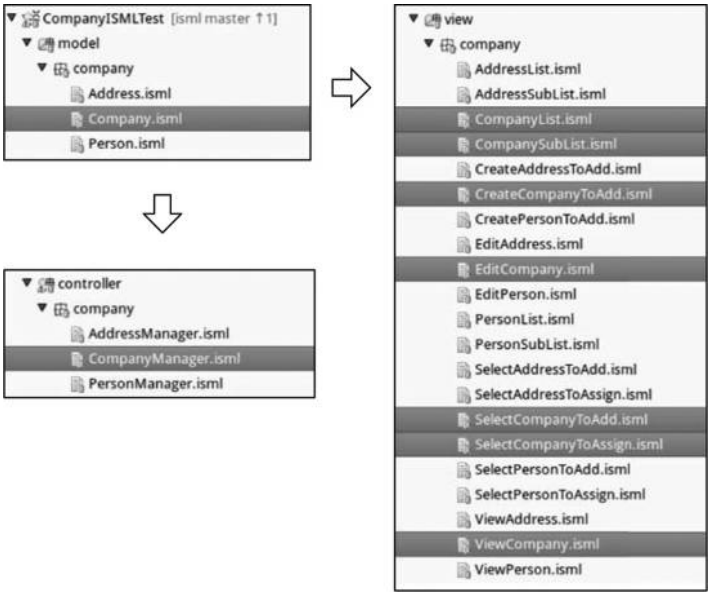

Figure 7.
CRUD-Transformer example

6. Code generation with ISML-GEN

ISML-GEN is a suite of code generators that creates the implementation for ISML models. Currently, there are three code generator modules that map an ISML model into different Java EE implementations, described as follows.

6.1 Entity generator

The Entity Generator processes all of the entities in an ISML model. For each entity, it outputs a java class with the corresponding attributes, getters, setters, comparison methods, unique identifiers and persistence annotations. The generated application uses the Java Persistence Framework (JPA) to persist these classes in a database. In addition, the Entity Generator creates a configuration file (**persistence.xml**) that is required by JPA.

6.2 JEE + JSF generator

One of the functionalities of the JEE+JSF Generator is to process all the pages and controllers and generate the corresponding user interface and page flow code. For each page in ISML, it generates a web page specification using JSF (Oracle, 2015c). For each controller, it generates a Java class called Backing Bean, which manages user events and page flow. In addition, the JSF Generator creates a configuration file (*faces-config.xml*) to facilitate the page flow specification. This generator also creates the business logic of the application through the use of services and integrates them with the entities created by the Entity Generator.

6.3 JEE + JavaFX + REST generator

Similar to the previous Generator, one of the functionalities of the JEE + JavaFX + REST Generator is to generate the user interface and page flow code. For each page in ISML, it generates a web page specification using JavaFX (Oracle, 2015b). For each controller, it generates a JavaFX service that manages user events and page flow. In addition, this generator also creates the business logic of the application, mapping each service in the model to a REST service in the code, and integrating the application with the entities created by the Entity Generator.

6.4 Code generation facilities

ISML-GEN is based on Xtext Eysholdt and Behrens (2010) and is written in the Xtend language (Eclipse-Foundation, 2015b). Although Xtext provides several facilities for model-to-text transformations, it does not have a simple way to query models. To address this issue, ISML-GEN includes a simple set of classes to retrieve elements from an ISML model. These classes combine generic types and reflection to declaratively specify which elements to retrieve from a model.

Figure 8 describes the main code generation classes. Subclasses of **SimpleGenerator** specify the location where files will be generated for a specific model element and the template that will be used. The class **GeneratorSuite** groups all the code generators.

The most important class is **SimpleTemplate**. Code generation templates are defined as subclasses of **SimpleTemplate**. The type parameter **T** passed to **SimpleTemplate** determines the elements to retrieve from the model when generating code with a template. Subclasses of **SimpleTemplate** must implement the **template** method that performs the model-to-text transformation.

For instance, Algorithm 7 is a portion of the class **EntityTemplate** that is used to generate one Java class per entity in an ISML model. **EntityTemplate** inherits from **SimpleTemplate<Entity>**. ISML-GEN reads this **Entity** parameter through reflection and acts accordingly. In particular, ISML-GEN executes the **template** method of **EntityTemplate** once per each **Entity** in the ISML model:

Algorithm 7: Entity Template

```
class EntityTemplate extends SimpleTemplate<Entity> {
  def template(Entity e) ' ' '
    public class ≪e.name ≫ {
      …
    }
  ' ' '
}
```

Similarly, Algorithm 8 is a portion of another class called **PersistenceXMLTemplate**. This class generates a configuration file that requires information from every **Entity** in an ISML model. In this case, **PersistenceXMLTemplate** inherits from **SimpleTemplate<List<Entity>>**. ISML-GEN uses reflection to read the **List<Entity>** parameter and executes the **template** method of **PersistenceXMLTemplate** only once for the entire model, and uses all the entities of the model as input:

Algorithm 8: PersistenceXML Template

```
class PersistenceXMLTemplate extends SimpleTemplate<List<Entity≫ {
  def template(List<Entity> e) ' ' '
    <persistence>
      …
    </persistence>
  ' ' '
}
```

Even though the ISML-GEN API does not provide more elaborate ways to retrieve elements from a model. In our experience, these two retrieval alternatives were sufficient for all the model-to-text transformations in ISML-GEN.

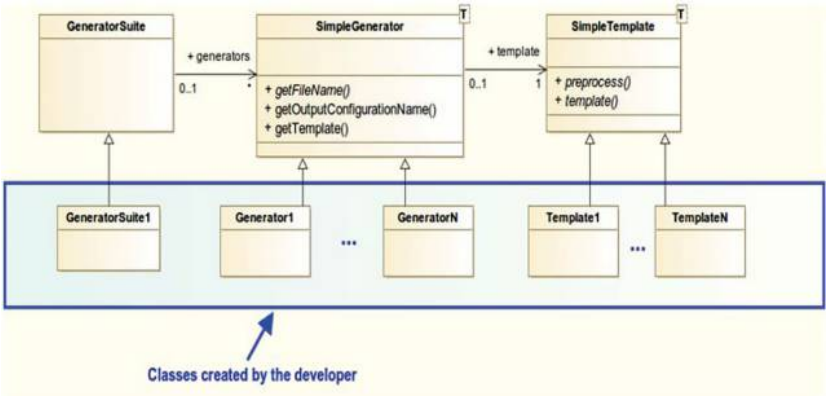

Figure 8.
Code generation classes

7. Validation

A full validation of ISML-MDE is expected to take years, as the company (HBT) fully adopts the approach and uses it in several projects that could yield the quantitative and qualitative data required to measure the effectiveness of the environment. As of the writing of this paper, ISML-GEN validation has advanced in two aspects: a comparison between the model size and the generated code, and an analysis of the DSL characteristics, according to Kahraman and Bilgen (2015).

According to Wu *et al.* (2010), model complexity can be used to understand the amount of effort required to use a modeling language. Also, they point out the importance of comparing models and code (in some cases, using lines of code) to determine the advantages of using the modeling language over implementing the solution in the target language.

To compare models and code size, ISML-GEN was used in three case studies: a Sports Team Application (STA), a Company Application (CA) and a University Application (UA). The STA is a very simple application that manages basic information about sports teams. The CA is a more complex application that manages information about companies and their employees. The UA is the largest and comprises information about teachers, assistants, students, courses, documents published in courses and forums.

All the applications were developed from scratch. Entity and JEE+JSF Generators were used to create the implementation code. To measure the impact of ISML-GEN in the development of these applications, models and code were compared in terms of size. As models in ISML are textual and their syntax is similar to the target implementation language (Java), lines of code were used to perform the comparison. Because of that reason and the use of LOC in DSL measurement frameworks (Wu *et al.*, 2010), we believe that the use of LOC is adequate to perform this comparison. Models were measured at two milestones in the design process. The first one is right after the entities were specified. The second one is right after CRUD-Transformer (see Section 5.4) was used to generate CRUD pages and controllers. None of the projects required modifying the generated code, so the code size was measured right after the code generation. Even though these models contain an important amount of CRUD operations, they provide an adequate insight about the abstraction and automation facilities of ISML, given that CRUD operations are very common in enterprise applications.

Table I summarizes the results. In all three cases, the generated code is more than four times the size of the model after using the CRUD-Transformer. The code size in both the STA and the CA is over 130 times larger than the model before executing the CRUD-Transformer. This ratio is smaller in the UA, because the model for this application before running the CRUD-Transformer includes several elements that are not entities; thus, the model does not grow from those elements after executing the CRUD-Transformer. To be more specific, these elements are the localization data, i.e. the specification of the strings that are shown in different parts of the application, based on its language settings, which are not present in the other two applications.

The above information suggests that ISML-GEN and CRUD-Transformer provide a significant degree of automation in the development of enterprise applications. Moreover, ISML provides a degree of abstraction to make engineers oblivious about most implementation details, but the abstraction is not excessive, as most of the implementation code can be directly derived from ISML models.

Project	Model size (LOC) before using CRUD-Transformer	Model size (LOC) after using CRUD-Transformer	Code size (LOC)	Code-to-model ratio (%) before using CRUD-Transformer	Code-to-model ratio (%) after using CRUD-Transformer
Sports Team application	6	182	908	15,133	499
Company application	24	696	3187	13,279	458
University application	144	2078	8794	6,107	423

Table I.
Size comparison between models and code

In addition to the above measurements, the current validation efforts include an analysis of the main DSL characteristics, according to Kahraman and Bilgen (2015):

- *Functional suitability*: ISML includes the essential concepts of an enterprise application based on the MVC architecture. All of the elements of the language were validated by the architects and developers at HBT and should provide a degree of assurance that they properly represent elements of the problem domain;
- *Usability*: A full-fledged usability validation is underway. However, as the design of ISML hides unnecessary details from the implementation and has a syntax and tooling very similar to what developers are accustomed to use, we believe that ISML should provide a reasonable degree of usability;
- *Reliability*: ISML is a fully typed language, where each expression is checked for correctness against the typing system. Before performing the type checking, ISML automatically links elements in different text files, according to the scoping rules, to ensure that all of them are properly connected;
- *Maintainability*: Primitives in ISML are ISML models themselves. In other words, to evolve the language, one only has to modify an ISML file, without having to recompile the environment. The ISML language is also modular, as it provides packages to organize information and the main modular units associated with the MVC architecture;
- *Productivity*: Although a full-scale validation for productivity is underway, the comparison between model and code size above suggests that productivity should increase by using ISML;
- *Extensibility*: The same arguments used for maintainability apply for extensibility. Modules and primitives can be directly added to extend the functionality of ISML;
- *Compatibility*: Section 3.3 details the rationale behind the design of ISML-MDE and shows that it is directly aimed at making ISML-MDE compatible with a software development workflow;
- *Expressiveness*: Compared to a general purpose language, ISML is less expressive, as it imposes a specific architecture (MVC) to model an enterprise application. However, as ISML is semantically close to the target languages and frameworks, we believe it has a degree of expressiveness superior to more abstract DSLs;
- *Reusability*: The same arguments used for maintainability apply for reusability. Modular units in ISML facilitate its reuse in other models written in the same language. Regarding reuse from other languages and tools, as ISML-MDE is based on the Eclipse Modeling Framework (EMF), it is fully compatible with any other EMF tools. For instance, engineers could use other code generation libraries – e.g. Acceleo (Obeo, 2015a) – to generate code instead of our framework. ISML models can also be processed by code transformation tools – e.g. ATL (Jouault *et al.*, 2008) – or any other supported by EMF; and

- *Integrability*: Currently, ISML does not support syntactic integration with other languages. However, as it is based on EMF, it should be relatively straightforward to integrate with other EMF-based languages.

Overall, even though the validation of the language could take years, the current validation efforts suggest that ISML could be a useful tool to facilitate the development of enterprise applications.

8. Conclusions and future work

This paper presented the practical experience of the creation of ISML-MDE, a model-driven environment to abstract and automate the development of enterprise applications. ISML-MDE is driven by specific design principles that may assist the transition from traditional software programming to MDE in organizations that develop enterprise applications. In particular, the level of abstraction in ISML is sufficiently high to hide implementation details, but sufficiently low to specify the majority of an enterprise application. The partial modeling facilities provided by the *native* keyword in ISML assists engineers to gradually learn the language and facilitate its adoption. The modular code generation and transformation automates most of the application implementation and also contributes to the obliviousness of the target platform details.

The high code-to-model ratio suggests that ISML-MDE can effectively reduce development times of enterprise applications. Our previous work with LionWizard (Franky and Pavlich-Mariscal, 2014) also provided important insights about reduced development times. In addition, the analysis of DSL characteristics suggests that ISML has the potential to simplify the development of enterprise applications.

ISML currently focuses on the design of enterprise applications. However, the vision of ISML is much broader. Future work comprises four lines of work: the first line of work is to expand the language to address requirement specifications and provide model transformers to convert those specifications into a design model. The second line of work is to create new code generators to address different platforms such as ASP with C#, PHP, etc. The third line is to use ISML to create platform-independent models for adaptive systems (Bocanegra *et al.*, 2015). This work may require creating two additional languages: one for specifying adaptive requirements and another for specifying adaptive design, which can then be translated into a regular design model, such as the one specified by ISML.

The fourth line of work is to fully deploy ISML-MDE to use it as the main tool in future projects in the organization. To further validate ISML-MDE, we plan to precisely measure development times and compare them with previous similar projects to determine the precise gains in modeling productivity. In addition, we plan to do a qualitative assessment based on existing frameworks (Kahraman and Bilgen, 2015). The effective deployment of ISML-MDE requires a cultural change in the organization. The main expected change is that the designers-to-programmers ratio will change. To create enterprise applications with ISML-MDE, more designers will be required and less programmers. To maintain and develop ISML-MDE, the organization will require a few specialized programmers and designers, both to evolve the language and the code generators.

References

AndroMDA (2014), "AndroMDA model driven architecture framework - AndroMDA – homepage", available at: www.andromda.org/ (accessed 27 March 2015).

Apache-Foundation (2010), "Apache velocity site - the apache velocity project", available at: http://velocity.apache.org/ (accessed 27 March 2015).

Apache-Foundation (2015a), "Apache subversion", available at: https://subversion.apache.org/ (accessed 12 May 2015).

Apache-Foundation (2015b), "Maven", available at: http://maven.apache.org/ (accessed 24 April 2013).

Apache-Foundation (2015c), "Maven - introduction to archetypes", available at: http://maven.apache.org/guides/introduction/introduction-to-archetypes.html (accessed 24 April 2013).

Auwera, J. (2013), "Equanda", available at: www.equanda.org/ (accessed 27 April 2015).

Baresi, L., Colazzo, S., Mainetti, L. and Morasca, S. (2006), "W2000: a modelling notation for complex web applications", in Mendes, E. and Mosley, N. (Eds), *Web Engineering*, Springer Berlin Heidelberg, pp. 335-364 (accessed 27 April 2015).

Bocanegra, J., Pavlich-Mariscal, J. and Carrillo-Ramos, A. (2015), "MiDAS: a model-driven approach for adaptive software", *Proceedings of the 11th International Conference on Web Information Systems and Technologies*, *Lisbon*, pp. 281-286.

Eclipse-Foundation (2015a), "Eclipse - the Eclipse foundation open source community website", available at: https://eclipse.org/ (accessed 20 April 2015).

Eclipse-Foundation (2015b), "Xtend - modernized java", available at: https://eclipse.org/xtend/ (accessed 27 April 2015).

Eysholdt, M. and Behrens, H. (2010), "Xtext: implement your language faster than the quick and dirty way", *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, ACM, pp. 307-309 (accessed 13 March 2014).

Foundation, E. (2013), "Eclipse modeling framework (EMF)", available at: www.eclipse.org/modeling/emf/ (accessed 11 August 2010).

Fowler, M. (2002), *Patterns of Enterprise Application Architecture*, Addison-Wesley Longman Publishing, Boston, MA.

Franky, M.C. and Pavlich-Mariscal, J.A. (2014), *A Method to Achieve Automation in the Development of Web-Based Software Projects*, Lisbon, pp. 83-88, (accessed 29 July 2014).

Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1995), *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA.

Gartner-Inc (2015), "Gartner IT glossary", available at: www.gartner.com/it-glossary/ (accessed 20 April 2015).

Gilbert, J. (2008), "Taylor model driven architecture on rails", available at: http://taylor.sourceforge.net/index.php/Overview (accessed 4 May 2012).

Git-Project (2015), "Git", available at: http://git-scm.com/ (accessed 12 May 2015).

Gomez, D. (2015), "Zathuracode ≫ Generador de código para JavaEE", available at: http://zathuracode.org/ (accessed 27 April 2015).

Groenewegen, D., Visser, E. and Van Chastelet, E. (2015), "WebDSL", available at: http://webdsl.org/home (accessed 27 April 2015).

Heinsohn (2015), "Heinsohn business technology", available at: www.heinsohn.com.co/ingles/ (accessed 28 April 2015).

JetBrains (2015), "IntelliJ IDEA", available at: www.jetbrains.com/idea/

Jouault, F., Allilaire, F., Bézivin, J. and Kurtev, I. (2008), "ATL: a model transformation tool", *Science of Computer Programming*, Vol. 72 Nos 1/2, pp. 31-39.

Kahraman, G. and Bilgen, S. (2015), "A framework for qualitative assessment of domain-specific languages", *Software & Systems Modeling*, Vol. 14 No. 4, pp. 1505-1526.

Kent, S. (2002), "Model driven engineering", *Lecture Notes in Computer Science*, Springer Berlin/Heidelberg, Vol. 2335, pp. 286-298, (accessed 27 March 2012).

Kroiss, C., Koch, N. and Knapp, A. (2009), "UWE4jsf: a model-driven generation approach for web applications", in Gaedke, M., Grossniklaus, M. and Díaz, O. (Eds), *Web Engineering, Lecture Notes in Computer Science*, Springer Berlin Heidelberg, No. 5648, pp. 493-496 (accessed 27 April 2015).

Kühne, T. (2005), "What is a model", *Dagstuhl Seminar Proceedings: Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Language Engineering for Model-Driven Software Development*, Vol. 04101, pp. 200-210.

Mazanec, M. and Macek, O. (2012), "On general-purpose textual modeling languages" (accessed 27 April 2015).

Microsoft (2015a), "NET downloads, developer resources & case studies | microsoft.NET framework", available at: www.microsoft.com/net (accessed 20 April 2015).

Microsoft (2015b), "Visual studio - Microsoft developer tools", available at: www.visualstudio.com/ (accessed 20 April 2015).

Mubin, S. and Jantan, A. (2014), "A UML 2.0 profile web design framework for modeling complex web application", *2014 International Conference on Information Technology and Multimedia (ICIMU)*, pp. 324-329.

Obeo (2015a), "Acceleo", available at: www.eclipse.org/acceleo/, (accessed 25 November 2011).

Obeo (2015b), "Module: Acceleo JavaEE generators", available at: http://marketplace.obeonetwork.com/module/javaee-generators (accessed 27 April 2015).

Obeo (2015c), "Module: cinematic designer", available at: http://marketplace.obeonetwork.com/module/cinematic (accessed 27 April 2015).

Obeo (2015d), "Module: entity designer", available at: http://marketplace.obeonetwork.com/module/entity (accessed 27 April 2015).

Obeo (2015e), "Module: SOA designer", available at: http://marketplace.obeonetwork.com/module/soa (accessed 27 April 2015).

OMG (2009), "Model driven architecture (MDA)", available at: www.omg.org/mda/ (accessed 1 March 2010).

OMG (2012), "SoaML", available at: www.omg.org/spec/SoaML/ (accessed 27 April 2015).

OMG (2015), "IFML: the interaction flow modeling language | the OMG standard for front-end design", available at: www.ifml.org/ (accessed 27 April 2015).

Oracle (2014), "Java SE 7 java native interface-related APIs and developer guides", available at: http://docs.oracle.com/javase/7/docs/technotes/guides/jni/ (accessed 13 May 2015).

Oracle (2015a), "Java platform, enterprise edition (Java EE)", available at: www.oracle.com/technetwork/java/javaee/overview/index.html (accessed 20 April 2015).

Oracle (2015b), "JavaFX developer home", available at: www.oracle.com/technetwork/java/javase/overview/javafx-overview-2158620.html (accessed 6 May 2015).

Oracle (2015c), "JavaServer faces technology", available at: www.oracle.com/technetwork/java/javaee/javaserverfaces-139869.html (accessed 6 May 2015).

Oracle (2015d), "NetBeans IDE", available at: https://netbeans.org/ (accessed 20 April 2015).

Pavlich-Mariscal, J.A. (2015), "ISML-MDE repository", available at: https://github.com/jpavlich/ISML-MDE (accessed 27 April 2015).

PUJ (2015), "Pontificia universidad javeriana", available at: www.javeriana.edu.co/ (accessed 28 April 2015).

Schuldt, H. (2009), "Multi-tier architecture", in LIU, L. and ÖZSU, M.T. (Eds), *Encyclopedia of Database Systems*, Springer, pp. 1862-1865 (accessed 6 May 2015).

Souer, J., Luinenburg, L., Versendaal, J., van de Weerd, I. and Brinkkemper, S. (2008), "Engineering a design method for web content management implementations", *Proceedings of the 10th International Conference on Information Integration and Web-based Applications & Services, iiWAS '08*, ACM, *New York, NY*, pp. 351-358, (accessed 27 May 2015).

WebRatio (2015), "WebML", available at: www.webml.org/webml/page1.do (accessed 27 April 2015).

Wu, Y., Hernandez, F., Ortega, F., Clarke, P.J. and France, R. (2010), "Measuring the effort for creating and using domain-specific models", *Proceedings of the 10th Workshop on Domain-Specific Modeling*, ACM, New York, NY, p. 14.

Corresponding author
Maria Consuelo Franky is the corresponding author and can be contacted at: lfranky@javeriana.edu.co