A DOMAIN-SPECIFIC LANGUAGE TO SPECIFY BEHAVIOR IN A MANAGEMENT GAME SIMULATOR

EDWIN LEONARDO TÉLLEZ SÁNCHEZ

Advisor:

Rubby Casallas Gutiérrez, PhD

Department of Systems and Computing Engineering
Engineering Faculty
Universidad de los Andes
July 2011
Edwin Leonardo Téllez Sánchez: *A Domain-Specific Language to Specify Behavior in a Management Game Simulator, DSML Application Case Study, July 2011*

Submitted in partial fulfilment of the requirements for the degree of Magister in Systems and Computing Engineering at Universidad de los Andes

**ADVISOR:**
Rubby Casallas Gutiérrez, PhD

**CO-ADVISORS:**
Oscar Gonzalez, PhD
Rafael Meneses, Msc

**JURY:**
Silvia Takahashi, PhD
Hugo Arboleda, PhD

**LOCATION:**
Bogotá, Colombia

July 2011
ACKNOWLEDGMENTS

First of all, I give thanks to Rubby Casallas for giving me the opportunity to work with her as my thesis advisor, for being exigent with me, for her support, motivation and help in order to succeed in this project.

I also want to thank Oscar Gonzalez for his reviews during the design, the implementation, and the evaluation phases, and for his constructive suggestions until the end of the project.

I am grateful to Rafael Meneses for his help organizing the project’s scheduling and deliverables, for the periodical reviews, and for his help during the writing of the research paper (which constitutes a big part of this document, and which was submitted to the DSL2011 Conference). Also, I thank him as the MAGES’ project leader, for giving me all the required meetings with the developers and for his collaboration conducting the evaluation workshop.

Also, I want to thank Daniel Díaz, Daniel Santamaría, and Jesús Vargas (MAGES’ developers), for their extensive collaboration during my training in the MAGES system and for their effort and dedication doing the Evaluation Workshop. I thank specially to Daniel Díaz for his availability for doing the 100 meetings during the complex testing phase.

... I also thank my friend John Paul R., for the time spent reading and reviewing my texts in English.

... Thanks to my family and friends for their continuous support.

Finally, I thank God for all, because without Him nothing would be possible.
## CONTENTS

1 **INTRODUCTION** 1  
1.1 Problem Statement 1  
1.2 Objectives 2  
1.3 Solution Strategy 3  
1.4 Previous Work 5  
1.5 Document Organization 5  

2 **RESEARCH CONTEXT** 7  
2.1 Business-Logic Modeling in MDE 7  
2.2 Domain-Specific Languages 9  
2.3 MDE and DSL Relationships 12  

3 **MANAGEMENT GAME SIMULATOR SYSTEM** 15  
3.1 Context 15  
3.2 Case Study Scenario 15  
3.3 Domain Analysis: Generalizing Concepts 18  
3.4 Understanding the Decision-Making Process 19  
3.5 Overview of the Decision Metamodel 20  

4 **MAGES-DSL: BUSINESS DECISION LANGUAGE** 23  
4.1 General Features 23  
4.2 Concrete Syntax 25  
4.2.1 Parameters 25  
4.2.2 Accounting Transactions 27  
4.2.3 Operative Transactions 28  
4.2.4 Type System and Industry Model Navigation 29  
4.2.5 Variables 30  
4.2.6 Execution of Decision Models 30  
4.2.7 Commitment Model 31  
4.2.8 Execution of Commitment Models 32  
4.2.9 Library Model 32  
4.3 IDE (Integrated Development Environment) 33  
4.4 Summary 34  

5 **MAGES-DSL: CODE GENERATION PROCESS** 37  
5.1 Generated Artifacts 37  
5.2 Simulation Process 39  

6 **PROJECT VALIDATION** 41  
6.1 Design Principles Verification and Collected Metrics 41  
6.2 Evaluation Workshop 43  
6.2.1 Workshop Goal 43  
6.2.2 Workshop Setup 44  
6.2.3 Results 44  

7 **FUTURE WORK** 53  
7.1 A Language for Domain Experts 53  

8 **CONCLUSIONS** 57
APPENDIX 59
A MAGES-DSL EVALUATION WORKSHOP 61
B MAGES-DSL: DESCRIPTION OF INPUT PARAMETERS 65
C MAGES-DSL: RENDERING OF INPUT PARAMETERS 67
D MAGES-DSL: SUPPORTED EXPRESSIONS 69
E MAGES-DSL: GRAMMAR 71

BIBLIOGRAPHY 72
LIST OF FIGURES

Figure 1  The payoff of DSL approach. Taken from [10]  11
Figure 2  MDE Fundamental relationships with programming languages. Taken from [5].  12
Figure 3  Sample scenario for the MAGES system  16
Figure 4  Conceptual model for a Business Decision  19
Figure 5  Excerpt of the Industry model  20
Figure 6  Conceptual model for a Business Decision (2)  21
Figure 7  Excerpt of Business Decision Metamodel  22
Figure 8  MAGES-DSL building-blocks  23
Figure 9  Integrating the system’s Industry model in the language  25
Figure 10 MAGES-DSL Type system  29
Figure 11 Decision model execution  31
Figure 12 Commitment model execution  33
Figure 13 MAGES-DSL Editor  33
Figure 14 Editor’s validation system  34
Figure 15 Decision Maker Web interface (automatically generated from MAGES-DSL specifications)  37
Figure 16 Generated artifacts for the Buy Asset example  38
Figure 17 MAGES-DSL Simulator for the Buy Asset decision.  39
Figure 18 Execution of Buy Asset decision instances.  39
Figure 19 Evaluation workshop results  50
Figure 20 Percentage of time per task if MAGES-DSL it is not used.  51
Figure 21 More time-consuming task per element  51
Figure 22 Design of a visual editor inspired in worksheets  54
Figure 23 Other design of the editor  54

LIST OF TABLES

Table 1  Productivity of well-known DSLs. Tables taken from [13]  10
Table 2  Accounting transactions for the Take Local Loan example  17
Table 3  Accounting transactions for the Pay Local Loan example  18
Table 4  Library syntax and definition of formulas  35
Table 5  Business decisions currently in production  42
Table 6  Productivity data  43
LIST OF ALGORITHMS

4.1 Basic structure of the Buy Asset decision example . . . . . . . . . . . . 26
4.2 Parameters of the Buy Asset decision example . . . . . . . . . . . . . . . 27
4.3 Accounting Transactions for the Buy Asset example . . . . . . . . . 27
4.4 Operative Transactions for the Buy Asset example . . . . . . . . . . . . . 28
4.5 Variables used in the Buy Asset example . . . . . . . . . . . . . . . . . . . . . 30
4.6 Expressions allowed for variables . . . . . . . . . . . . . . . . . . . . . . . . . 31
4.7 Pseudo-code for the execution of a Business Commitment . . . . . 31
4.8 Commitment model syntax . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
4.9 Iterable keyword . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
INTRODUCTION

The challenge faced redesigning a major part of the Management Game Simulator system (MAGES) has motivated the development of this thesis work. MAGES is a game-based learning system used at the Universidad de los Andes for training of about 300 students per year in the strategies and procedures of running a successful business.

In a Business Game Simulator, the corporations (i.e. students) compete in the same industry making business decisions to increase market shares and shareholder equity. Thus, the main interaction mechanism that students have in MAGES is the decision-making process. And the Simulator component (MAGES’s core) is a generic interpreter that is able to execute any decision instance created by each corporative group. The simulation process will be described in detail later, however we anticipate that this process depends on a set of business rules (Business Decisions) and a set of global variables (Industry model) representing the state of the game in a given period of time.

The main concern with MAGES has always been to get a near real-world business environment suitable for teaching and self-training. That means that the knowledge embedded in the system (business decisions specifications) must continuously evolve in order to be consistent with the teaching objectives of the course.

In this context, the worst scenario is to implement these business-logic changes directly in the system source code each time a domain expert wants to modify the behavior of the simulation regarding a business rule, or each time a new business rule has to be added to the simulation. But this is a too-expensive and error-prone solution according with the system’s size and complexity (the source code reaches the 800 KLOCs).

In the system construction time, these problems were detected, and as a consequence the system was developed to be very flexible: its architecture is extensible and its development was carried out using model-driven development techniques. Even, it has been re-factored many times to better support the extensibility quality attribute. However, the requirements for better simulation capabilities have increased in size and complexity, and the lack of convenient support and tools in the current implementation has made the system’s maintenance and extension a high-cost process that requires exhaustive testing. These limitations raised the necessity to redesign the extensibility mechanism.

1.1 PROBLEM STATEMENT

The MAGES system requires the design of a new strategy to implement the business logic associated with Business Decisions, as these are the artifacts that
involve the core functionality of the system and the main interaction mechanism for the players.

The intended solution has to deal with the following key issues:

1. **Requirements elicitation is costly.** The knowledge necessary to define the business rules of the simulation (*Business Decisions*) is thinly spread among a series of domain expert stakeholders, each belonging to diverse domains, such as advertising, financing, marketing, manufacturing, handling human resources, etc. For that reason requirements elicitation is a costly process that requires formal specifications and continuous validation by experts.

2. **Functional requirements are not static.** It is not possible to know all the simulation requirements from the beginning, and the number of business decisions is not fixed. For that reason, the system must be capable to incorporate new business decisions easily. Additionally, the system must allow to update and maintain existing business decisions.

3. **Business decisions are fully system-dependent.** The business-logic modeled in the Business Decisions is fully-dependent on the MAGES execution environment. In particular, it is needed to cross-reference the system’s data model, and to access to system’s services.

4. **Testing is costly.** Due to the full system dependency, it is very difficult to test the business-logic modeled in isolation (*i.e.* outside the system).

5. **Development time is too-high.** The capability to produce new releases in shorter times is also a business requirement.

### 1.2 Objectives

The main goal of this project is to (re)design an (re)implement the involved system’s modules in order to allow the implementation of the business-specific system behavior according to the problem statement.

In order to implement the overall solution, the following objectives were defined:

- To design a textual Domain Specific Language (DSL) with a suitable syntax for the specification of business decisions, which must meet the following requirements:

  **High-expressiveness.** This is to support the modeling of the business logic required to specify business decisions.

  **Simplicity.** It is required to provide a simple syntax with a well-defined semantics easy to learn and use.

  **Model-orientation.** The artifacts modeled have to conform to a *business decision metamodel*.

- To implement an integration mechanism at model level between the DSL and the MAGES industry model, in order to model Business Decisions conforming to the industry setup.
1.3 Solution Strategy

Due to the complexity to elicit the functional requirements, the challenge to implement all of them in a clear and extensible way, and the other mentioned requirements, we opted for a Domain Specific Modeling (DSM) approach, described as follows:

A) To use DSM/DSL techniques to provide a solution in the Business Management domain, modeling business decisions in high-level platform-independent specifications, through the use of a custom Domain-Specific Language.

B) To use Model Driven techniques to produce a source code generator, which produces near 100% of the final implementation code from business decisions models.

The way in which each challenge is solved is as follows:

Challenge 1. Through strategy (A) it is raised the abstraction level beyond the system source code [5] by abstracting the business decision concept and its execution. Therefore, although the requirements elicitation process is not altered itself, the communication gap between stakeholders and developers is greatly reduced by using high-level models [6].

Challenge 2. We don’t tried to make static the requirements but to support better the extensibility mechanism. Extensibility was already supported by the system at the implementation level (i.e. Business Simulator and Decision Manager components). The current design done in previous
projects allows the integration of new business decisions at all levels (UI, logic, data). Through the model-based interpretation process (B), all business decisions specifications are automatically converted into source code and entered into the application to be deployed immediately in a production environment. Strategy (A) also allows a logic separation between simulation-specific business logic and the system’s core implementation. Thus, maintainability can be done at model level (in the business domain). Finally, the synchronization between models and the source code is automatic because the latter is generated (B) [12].

**CHALLENGE 3.** The MAGES industry model (the semantic model that models the business environment in the game) can be referenced in each business decision specification through model composition. In this way, the integration between the specifications and the system is achieved at model level.

**CHALLENGE 4.** Although currently it is not possible to carry out testing at model level, the testing effort is greatly reduced by using an editor equipped with syntactic and semantic validations that prevents most common errors and invalid models. In this way, it is achieved higher-quality implementations.

**CHALLENGE 5.** Time to market is greatly shortened because the source code generated represents near 100% of the final implementation (B), the development effort is concentrated in the analysis and specification of the business-specific logic (and less in the platform-specific implementation), and the errors are detected earlier (at modeling time).

The validity of the proposal is justified as follows:

- **The domain is specific:** if the system would be a general purpose application, the functional and expressiveness requirements can only be implemented using a general purpose language. However, the domain is restricted to the business administration field and the modeling concern is restricted to business decisions (it is not intended to model the business-logic of the whole system).

- **The number of functional requirements is large:** The implementation cost of an extensibility mechanism (through the development of a DSL from scratch) is very high due to the upfront domain analysis effort and the cost of the development of the environment and tools [5]. Therefore, the investment is justified only if the DSL is extensively used for the implementation of many requirements in the target domain [5].

- **The application is a traditional transactional system:** The context in which we applied the strategy is a transactional system without exigent quality requirements (security, performance, scalability, reliability, etc). For that reason, the modeling of pure business logic without the need to know low-level implementation details nor critical quality considerations is feasible.
1.4 PREVIOUS WORK

Before discuss previous work, we clarify that the MAGES system has been operating in production for more than one year. Therefore, the starting point for this project is a mature system that has a well-defined architecture, fully-implemented components, and whose global requirements has been widely achieved.

The solution strategy proposed has been highly influenced by previous projects and ideas that belongs to people that worked in previous stages of the MAGES project. Specifically, we acknowledge the following facts:

1. From the start, MAGES was designed as an extensible system. Both the Web interface and the simulation engine are generic enough to allow the implementation of new Business Decisions without impacting existing artifacts (other Business Decisions or MAGES’ components).

2. Code Generation: It was already possible through the use of XML descriptors. The XML schema (XSD) designed was the first approach to obtain a metamodel for Business Decisions. The code generation was achieved by using code templates\(^1\), which were the guide for the implementation of the DSL compiler (although they weren’t model-driven artifacts). The description of the XML mechanism is beyond the scope of this document.

Finally, we remark that our contribution is the development of a DSL to model Business Decisions, and the implementation of the model-driven code generator. The followed premise was to solve the problems without modifying existing MAGES modules (including the Business Simulator).

1.5 DOCUMENT ORGANIZATION

In the following chapters, the project’s development and the achieved results will be described in detail. Chapter 2 presents the research context focusing in the DSL approach. Chapter 3 describes in detail the context of the project: MAGES system and the specific requirements. Chapter 4 introduces MAGES-DSL showing in detail its concrete syntax and its building blocks. The language execution and the code generator are described in Chapter 5. Chapter 6 presents the project evaluation process, some metrics and the results obtained. Chapter 7 presents possible improvements and future directions for the project. Finally, Chapter 8 compiles the overall project results, the advantages obtained, the challenges faced and the trade-offs of the strategy.

\(^1\) For the implementation of code generation was used Velocity (a template-based code-generator engine)
RESEARCH CONTEXT

According with the problem statement, we are interested in strategies for solving the problem of separating the core business logic from the low-level implementation code, in order to achieve the goals of extensibility, maintainability, better stakeholders communication, and time-to-market reduction. The scope is limited to model-driven techniques for the development of applications with a large number of domain-specific functional requirements.

Usually, the targeted applications will be transactional systems (e.g. information systems) with expressiveness requirements for arithmetic expressions, use of libraries for technical computations, and sub-languages for querying data or navigate objects models (which were the basic modeling requirements for the MAGES system, as it will be described in detail in chapter 3).

2.1 BUSINESS-LOGIC MODELING IN MDE

The modeling of business-logic in applications developed using Model-Driven Engineering (MDE), whose purpose is restricted to a specific domain, has been actively studied in recent years due to the importance for the field. However it remains as a state-of-the-art challenge. We will describe shortly some of the general strategies proposed in recent studies.

In the proceedings of the First European Workshop on Behaviour Modelling in Model Driven Architecture (BM-MDA) [1] were presented different papers with different approaches with the aim of providing better support for MDA, with a general vision that fully automatic generation of the code from models is an open problem, and if it works at all, restricted to specific application areas. According with [15], modeling languages that aim to capture PIM level behavior continues being a challenge. The proposals include (1) changes to the semantics and notations of UML behavior modeling techniques in order to achieve executability of models, (2) the introduction of new modeling semantics, and (3) better approaches to providing execution semantics for behavioral models [1].

One paper,[11], proposes to extend or replace the semantics of the behavioral diagrams in the UML, which shows some deficiencies in UML sequence diagrams, and as an alternative they propose a new activity-based notation for the behavior description, which preserves most of the aspects of sequence diagram notation and gives a greater expressiveness at the same time (specially for the data flow description facilities, and the incorporation of loops). The intended goal is to gain readability and manageability of a complete behavior description for a certain system fragment. According with their developed prototype tool, they claim that this approach is not expensive in practice and the gain is visible. However, it is not specified how many sequence diagrams should be used to
specify a system, and what is the semantics of the composition of multiple diagrams. Finally, an evident disadvantage of this draft notation (which is not an UML profile) is that the extension of a UML diagram brings the immediate lack of support by the available tools (as it is shown by the need to develop an own prototype tool) and the addition of compatibility and standardization issues for the interchange of models and artifacts reuse.

Other approach presented in the workshop is [15], in which Riccobene et al. propose to model behavior at platform-independent model (PIM) level through a modeling language based on the Abstract State Machines (ASM) formalism. This UML extension focuses on intra-object interactions by addressing the behavior occurring within structural entities (such as UML class diagrams). Different from UML state charts, it uses operators and constructs with well-defined semantics (Action Semantics). The construction of executable class models is achieved through a weaving function between structural and ASM diagrams. It is needed to identify join-points between the UML and the ASM metamodels (asmM) rather in the manner of aspect weaving, resulting in a combined language they call UML+. The authors also provide a tool set, called ASMETA (that includes a textual DSL for human-readable ASM models and a compiler) that could be used for other projects. However, achieving a weaving with the Class model is the easiest case, as stated in [1], as there is no semantic overlap, and a real test of the approach is needed to observe sensible results, with undue complexity when considering more general Class behavior, including inter-object communication. Another challenge is to see when the behavioral models defined as ASMs can be used in the context of UML models.

Gherbi et al. [7] cite other article1 to explain how the MDE approach must use multiple modeling languages in order to describe a complete application. Thus, the use of different models (using these DSLs) will result in multiple connected models defined in different languages, which have to be combined and made executable using a transformation process.

In fact, this strategy: “solve a problem using multiple DSLs” is implemented in part in MAGES-DSL (the language that we developed), as it models only a segment of the system’s business logic (business decisions). The other business-logic concerns are delegated to other languages/strategies (e.g. the Reporting Component functionality , etc).

Bessam, in [2, 3], acknowledges the increasing interest in the problem:

“Specifying behavior of system architecture is necessary to have a complete software architecture description. And necessity of meta-modeling to supervise behavioral concepts in all domain specific languages has increased”.

In particular, he presents a generalized metamodel of behavioral aspects, which is organized in packages according to four functional perspectives: interface, static behavior, dynamic behavior, and interaction protocols. The proposal is focused in modeling behavior at the metamodel level, which increases the complexity for their usage in real applications, as a resulting trade-off of their wide scope.

Finally, there exist other studies presenting interesting alternatives such as the use of formal methods in the MDE context, which are out of the scope of this document. However, we can comment that, as it is stated in [4], many successful applications of formal methods in industrial contexts used these formal notations and techniques to model at most 10% of the system under development (e.g. critical components), which brings a de facto domain-specific modeling strategy.

In the next section, we will discuss the DSL approach as other strategy to model system behavior, typically through textual specifications.

2.2 Domain-Specific Languages

Domain Specific Languages (DSLs) have proven to be more suitable and useful than General Purpose Languages (GPLs) in a variety of contexts for the implementation of domain-specific functionality. Some of its advantages are the user orientation to ease the expression of functionality, and the opportunity to automate the generation of artifacts for constrained and well-defined application contexts[5].

Fowler in [6], defines a domain-specific language as: “a computer programming language of limited expressiveness focused in a particular domain”².

From the definition, he emphasizes in four characteristics that a DSL must have: (1) computer programming language (i.e. is executable by a computer), (2) language nature, (3) it has a limited expressiveness (a DSL must support only the features needed to cover its domain requirements), and (4) a domain focus (to have a clear focus in order to ensure usefulness). From this point of view, Fowler explains the domain focus characteristic as a direct consequence of limited expressiveness.

To illustrate the diffusion of DSLs, we can mention some well-known examples: in the web development area (CSS, HTML), in the typesetting area (Latex), query languages (SQL and its variants like JPA-QL), grammar specification languages (BNF/EBNF syntax), regular expressions, and system administration (Puppet³). Mernik in [13] also states that fourth generation languages (4GLs) are DSLs for database applications.

Other examples are the set of tools classified as Language Workbenches. According with Fowler in [6], a Language Workbench is a tool that follows the language oriented programming paradigm (a software development style which seeks to develop software around a set of DSLs), whose design is directly targeted to end users. Examples of well-known LWs are Microsoft Software Factories and Meta Programming System⁴.

² Fowler, Martin. "Domain-Specific Languages (Addison-Wesley Signature Series (Fowler))". Addison-Wesley Professional. 2010. p. 31.
³ http://www.puppetlabs.com. It is an interesting and successful DSL targeted to business users (i.e. system administrators), who are technical people but not necessarily programmers. The gained value is high productivity, high quality configurations (avoiding error-prone repetitive hand-made configurations) and high level of reuse (configure once, use since then).
⁴ http://www.jetbrains.com
In Table 1, which is taken from [13], it is offered a list of well-known DSLs. The table shows the results of an experiment whose goal was to measure the productivity of each listed DSL. The right table shows the equivalences between levels and function points. Only to get a quick comparison, the Java language has level 6 while Ms Excel has level 57 (which is about 1/10).

<table>
<thead>
<tr>
<th>DSL</th>
<th>Application Domain</th>
<th>Level</th>
<th>Productivity Average per Staff Month (FP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BNF</td>
<td>Syntax specification</td>
<td>n.a.</td>
<td>1-3</td>
</tr>
<tr>
<td>Excel</td>
<td>Spreadsheets</td>
<td>57 (version 5)</td>
<td>5-10</td>
</tr>
<tr>
<td>HTML</td>
<td>Hypertext web pages</td>
<td>22 (version 3.0)</td>
<td>4-8</td>
</tr>
<tr>
<td>M$\phi$X</td>
<td>Typesetting</td>
<td>n.a.</td>
<td>9-15</td>
</tr>
<tr>
<td>Make</td>
<td>Software building</td>
<td>15</td>
<td>16-23</td>
</tr>
<tr>
<td>MATLAB</td>
<td>Technical computing</td>
<td>n.a.</td>
<td>16-30</td>
</tr>
<tr>
<td>SQL</td>
<td>Database queries</td>
<td>25</td>
<td>24-55</td>
</tr>
<tr>
<td>VHDL</td>
<td>Hardware design</td>
<td>17</td>
<td>&gt; 55</td>
</tr>
</tbody>
</table>

(a) Productivity  
(b) Level measuring

Table 1: Productivity of well-known DSLs. Tables taken from [13]

The advantages of using a DSL vary according with the context. Mernik in [13] presents evidence of many benefits such as the expression of the solution at the same level of abstraction as the problem domain, improvements in productivity, support for extensibility, and better reliability in software systems, etc.

On the other hand, the cost of developing a custom DSL is so high that it is considered one of the main disadvantages of the approach. The cost is due to the domain analysis effort and the development of the environment and tools [5]. Additionally, it is required both domain knowledge and language development expertise, which is not commonly found in people [13].

In our case, the domain analysis was one of the most costly stages because it was needed to discover all the expressiveness requirements before design the language. Furthermore, it was required to fulfill these requirements keeping simplicity to ease the implementation and favoring usability. Concerning implementation, the cost grows specially for external (and/or graphical) DSLs. Therefore the investment must be balanced, for example by using the DSL in many applications in the target domain, or by using automated development environments to reduce implementation costs [5].

The payoff for the DSL approach is shown in Figure 1. According with [10], there is always a starting cost (higher compared to conventional methodologies), but as the infrastructure is applied through time, the overall cost is greatly reduced and remains below the conventional cost. However, the DSL’s life-cycle tend to be very short.

Concerning to implementation, a DSL can be classified as follows[8]:

**EXTERNAL-DSL** Are DSLs that define their own syntax (and their corresponding semantic model), therefore require the development of their own parsers, interpreter or compilers. The advantage is the possibility to define a custom syntax and semantics at the cost of the developing the language infrastructure.

5 However this is changing in the nowadays. For example, there are products like Poseidon for DSLs which allows to create a graphical editor for an arbitrary EMF model and configure it in less than 3 or 4 hours (http://www.gentleware.com/poseidon-for-dsls.html).
2.2 Domain-specific languages

**Internal-dsl** are languages built on top of an existing host programming language, which reduces drastically the overall development effort by allowing them to reuse the host language’s infrastructure (such as IDEs and compilers). There are GPLs that facilitate the development of internal (embedded) DSLs, being a typical example the Ruby programming language (in [8] there are many examples of DSLs implemented in this programming language). Due to the syntax is inherited from the host language (except for some customized adaptations allowed by some GPLs), the DSL is limited to the goodness and drawbacks of that.

In the case of some XML languages, Fowler in [6], states that in many cases they are also DSLs (differentiating them to the use of XML descriptors only for data interchange or simple configuration files). He recognizes some advantages of the use of XML DSLs: (1) they are based in standard and mature technology (XML Schema, XML parser generators, XML editors and validators), (2) they offer flexibility to model complex hierarchical data structures, cross referencing and even (limited) expressions; (3) the persistence mechanism is plain text (serialization) and therefore the documents are directly human-readable, etc.

On the other hand, he classifies these XML languages as external DSLs, because XML is not a programming language, and as such a XML specification is not itself executable. XML is a syntactic structure that has to be processed and interpreted in some way. From this point of view, a DOM parsing is the Tree Construction pattern (a pattern to navigate the language semantic model), and SAX parsing leads to Embedded Translation (or syntax directed interpretation as it is described also in [14]). However, (viewing XML as the carrier syntax) a XML DSL is very similar to an internal DSL, where the host language provides a carrier syntax for the DSL. A commonly discussed drawback of the XML syntax is the that “[...] it introduces far too much syntactic noise” [6] which spools the whole purpose of DSLs (understandability). Finally, he also discuss other strengths of the XML usage, as the existence of Parser Generators, which makes easier the implementation of custom parsing processing; the maturity of the

![Figure 1: The payoff of DSL approach. Taken from [10]](image-url)
technology to handle classic problems such as quoting and escaping to avoid inconsistencies; validation against a XML schema; and error handling and diagnostics. He compares these off-the-shelf benefits against external DSLs, where there is much effort needed to get similar results, depending on the framework/toolkit used.

Another implementation-level strategy related with DSLs are the application libraries. Mernik in [13] states that any GPL in combination with an application library can act as a DSL. Because the API constitutes a proper vocabulary of class, method, and function names that is available through object instantiation and/or method invocation to any GPL program using the library. However, he highlights the fact that the DSLs were developed in the first place because they can offer domain-specificity in better ways[13]. Therefore, for him many application libraries are DSLs that were left in the middle stage of their development process.

Finally, as other important subject for building successful DSLs are the Design Principles (or patterns), as explained in [8]. This topic is covered in chapter 7.

2.3 MDE AND DSL RELATIONSHIPS

The MDE field has many analogies and relationships with the programming languages design field. Estublier et al. present in [5] these fundamental relationships, as it is shown in Figure 2. The language side is shown in yellow, while the MDE side is shown in green. For programming languages, the meta-meta level consist of a grammar definition language (typically BNF / EBNF), while for MDE there is a model definition language, like the Meta Object Facility (MOF). The relation between a meta-meta model and the
metamodel is usually expressed as: the meta-meta model is the language in which the metamodel is written.

At the meta level, a language grammar must be defined to allow the modeling of all possible models in the language. Thus, a grammar is defined as a model of the programming language. The grammar is typically specified using the BNF notation.

Finally, at the model level is where a program is written, which is defined as a model of the system. Where the relation between a program and its grammar is in fact the conforming to relationship.

These conceptual relationships explain for example why it is possible to automatically generate a metamodel from a grammar definition and vice versa, which is one of most worthy strengths provided by the language development framework we used for the implementation (in chapter 4 are given more details).
In this chapter we describe a sample scenario that illustrates how the system works and which are the specific behavior requirements that need to be modeled. After that, it will be described how it was gathered a domain model and specifically the Decision metamodel, which will be the base artifact used to implement the DSL.

### 3.1 Context

MAGES is a Web-based system used for about two years at the Universidad de los Andes for training near 300 students per semester in the strategies and procedures of running a successful business. The construction of the system has involved many participants from diverse areas, both from the Management side and from the Engineering side, due to the complexity of its business requirements and its particular quality attributes.

Regarding implementation, a big part of MAGES was developed using model-driven engineering by modeling the whole industry environment and generating code from it\(^1\). Along the life of the project (about five years), many other software development strategies have been proposed and implemented in order to achieve the business goals with the highest quality possible, having special focus in the system’s maintainability and evolution through the time\(^2\).

### 3.2 Case Study Scenario

In a business simulator, students constitute corporative groups and make business decisions; the system in turn must simulate the execution of these decisions and reflect their results as changes on their financial states and stocks. Business

---

\(^1\) The MAGES system is composed by 4 modules: the Industry Manager, the Business Simulator, the Decision Manager (the target of this research), and the Reporter component. The Industry manager code was generated by using a Model Transformation Chain (which do consecutive model-to-model transformation steps and a final model-to-text step to produce the JEE code). Note that currently only one of the modules uses MDE. For that reason, we try to involve the Industry model asset into the Decision Manager (in this thesis) and the Business Simulator (in subsequent projects) in order to achieve better support for maintainability and extensibility.

\(^2\) For example before the implementation of our project, the Decision Manager module used a custom extensibility mechanism based in XML documents, which allowed to model Business Decisions in order to generate code from them. For that, it was designed a custom XML schema (XSD) to model a generic Business Decision. This XSD was an starting point to obtain a metamodel for Business Decisions. As it was stated in the Introduction, the XML language was not model oriented and for that reason the integration with the MAGES Industry environment and the other requirements described in the Problem Statement section were not fully solved.
decisions are related to very diverse domains, such as advertising, financing, marketing, manufacturing, handling human resources, etc. allowing a player develop a complete business strategy during the playing time (usually an academic semester).

Because MAGES is a simulation system for a generic industry (i.e. any business vertical), it must be configured for a specific business. MAGES’ current configured scenario is the automobile industry. That means that all corporations compete in this market having to deal with car production lines, engineers hiring, raw materials purchasing, distribution costs, and so on.

Figure 3 shows a sample scenario for the MAGES system: a player manages a corporative group (Abc Corporation), which is composed by three business units, each one located at different regions (Region A, B and C), and each region is configured to have a different currency (as would be expected). Once the market is defined, and the corporative groups are formed, the move can start.

The system is a discrete time simulator with a predefined number of periods. Each period is divided into two main parts. First is the setup stage, where each company records the set of actions to execute in order to develop its strategies and plans. The second stage refers to the simulation step, at the end of the period, where the system executes all the companies’ actions as far as possible, provided they are valid moves.

In MAGES, every company action is defined as a business decision. In Figure 1 are showed some decision examples (in red color). For instance, the decision #1 has the following specification (not showed in the figure):

**Decision #1**: “take a local loan of $100 million (in the Region B’s currency) with maturity of 2 years and a monthly interest of 1% and payments at the end of the period”.

Figure 3: Sample scenario for the MAGES system
In this case, the simulator needs to know what a loan means and what are its consequences for the company. The decision corresponds to a financing action on the Distribution unit. According with the system specification, finance decisions have to be reflected in the company’s accounting states at the end of the period. The accounting transactions responsible to do the financial states’ changes are described in Table 2, where Total = 100 million (loan present value).

<table>
<thead>
<tr>
<th>Account</th>
<th>Operation</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cash</td>
<td>Debit</td>
<td>Total</td>
</tr>
<tr>
<td>Financial Liabilities</td>
<td>Credit</td>
<td>Total</td>
</tr>
</tbody>
</table>

Table 2: Accounting transactions for the Take Local Loan example

So far, we can get a general description of a decision: (1) all decisions are numbered (i.e. are ordered), and (2) they have a name and some parameters. What a company does for playing is to fill the requested parameters in order to make the wished decision.

Continuing with the example, we have not mentioned yet how the debt will be paid each period and who is responsible for doing it. To solve this requirement, the game provides other type of decisions: business commitments. While a decision can be made by any player at any time, the commitments are system-executed transactions derived from other decisions (as in this case) or because a change occurs in the market context.

For the decision #1, we can express the loan characteristics as follows:

- The loan has to be paid in 24 months
- The interests are caused at the end of the period
- The interest rate is: \( i = 1\% \)
- the interest is: \( \text{Interest} = \text{Total} \cdot i \)
- Monthly payment is: \( \text{Fee} = \text{Total}/24 \)

Thus, the commitment will consist of the monthly automatic transactions needed to amortize the company’s debt. The transactions are shown in Table 3.

At this point, it is important to remember the accounting concept of double-entry, which ensures that the next invariant must be met, in order to ensure accounting consistency:

\[
\sum_i \text{Credit}_i = \sum_i \text{Debit}_i
\]
### Table 3: Accounting transactions for the Pay Local Loan example

<table>
<thead>
<tr>
<th>Account</th>
<th>Operation</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expenses</td>
<td>Debit</td>
<td>Interest</td>
</tr>
<tr>
<td>Passive</td>
<td>Credit</td>
<td>Interest</td>
</tr>
</tbody>
</table>

(a) Transaction 1

<table>
<thead>
<tr>
<th>Account</th>
<th>Operation</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Passive</td>
<td>Debit</td>
<td>Interest</td>
</tr>
<tr>
<td>Financial liabilities</td>
<td>Debit</td>
<td>Fee</td>
</tr>
<tr>
<td>Banks</td>
<td>Credit</td>
<td>Fee + Interest</td>
</tr>
</tbody>
</table>

(b) Transaction 2

In words, the equation states that in every accounting transaction the sum of the credits must be equal to the sum of the debits. For the example’s transactions, this can be easily verified.

Additional to the accounting states, each company has also a set of operative states. An operative state is a data set that keeps track of the company’s possessions: physical inventory (i.e. assets, raw materials, products, etc), human resources contracts, non-tangible assets, etc. The way to effectuate operative state changes is through the use of operative transactions.

For the loan example, both the decision (Take Local Loan) and the commitment (Pay Local Loan Amortization) have to define operative transactions. While the decision cause the loan item creation (a physical debt record), the commitment does the updates every period.

In fact, concerning other decisions and commitments examples in Figure 3, all of these modify operative states.

Finally, note that there are many more factors that can affect any decision: market variables (taxes, interest rates, etc), industry variables (transportation costs, distances between regions, etc), unexpected events (fire, breaks, natural catastrophes), etc., that are out of the scope of this description.

In summary, the listed transactions illustrated by the sample scenario are the type of business logic that game’s administrators need to specify in a formal and fluent way to setup the decisions for players.

### 3.3 Domain Analysis: Generalizing Concepts

One of the first tasks that we undertook in the DSM process was aimed at identifying the main concepts and features of a business decision regardless of the category to which it belonged (i.e. marketing, production, etc). Our intention was to specify the behavior of any business decision using the same elements.

We identified the following common key concepts: (1) Input Parameters, which characterize the input information needed to make a decision; (2) Accounting
and Operative Transactions, which define the specific effects that a decision has on the accounts in the company’s financial state, or the effects on the items of the company’s operative state; and (3) Variables and Formulas which correspond to the operations and computations required to obtain each transaction’s data.

In Figure 4 is presented the conceptual model defined so far for a Business Decision.

![Conceptual model for a Business Decision](image)

Figure 4: Conceptual model for a Business Decision

3.4 UNDERSTANDING THE DECISION-MAKING PROCESS

It was not sufficient to identify the common elements that characterize a business decision. It was also necessary to understand that some of these elements requires information from the system’s running environment. This can be seen clearly in the above examples. For example, to support the making decision process of raw material purchases, in the Parameters section of this decision, the system must offer to the students the information of the possible regions where the resources can be bought, the type of resources that can be purchased (all resources to produce cars must correspond to the automotive industry and not to any other type of industry), and the types of transportation that can be used for transferring the purchased resources, e.g. aerial, fluvial or terrestrial. In the Variables section, at simulation time, this decision also needs to query industry information, particularly to calculate the total transportation cost of purchased resources. Thus, the system needs to know the transport costs per unit and by transportation type (the cost of transporting resources by air is more expensive than by ship).

We define the Industry Model as the set environment entities and variables conforming the state of the game. This information model is created, updated, and queried within the system in almost every possible operation, specially when a decision is instanced, and when it is carried out of the simulation process. An excerpt of the Industry model is shown in Figure 5 (which is taken from the MAGES documentation).

This information model is managed by a functional component called Industry Manager (IM). The goal of the IM is to allow course administrators and instructors to configure parameters of the system and the industry. On the one hand, administrators have to set up, among other concepts, corporate groups, student roles, and generic business rules. On the other hand, instructors have to
configure the values of the industry such as currencies and exchange rates; market parameters such as market segments and product characteristics by segment; personnel wages, associated to each specific positions; the required resources (raw materials) in order to build each product; production costs; parameters for investments; definitions of loans, etc. All these elements characterize the business environment with specific details regarding market segments, regions where corporations sell products, manufacturing mechanisms, supply chain management, distribution, publicity, payroll, fixed assets, finances, taxation, etc.

To conclude, MAGES has a Business Simulator component (the simulation engine), which can interpret the definition of any decision in order to execute the instances created by each corporative group. The simulation process is the same for any decision instance: (1) the input parameters are fetched, (2) the required business environment data is collected, (3) the information is processed through the evaluation of the decision business logic, (4) the accounting transactions are executed modifying the involved financial statements, and finally (5) the operative transactions are executed to update the operative states.

3.5 OVERVIEW OF THE DECISION METAMODEL

In order to define the Decision Metamodel we identified that the central concept in the business simulator from the user’s point of view is the Business Decision, because decision-making is the main interaction mechanism the players have in MAGES. Thus, the metamodel’s root is the Decision concept. Additionally, a decision can be logically divided in two sections: parameters and the execution block.

PARAMETERS. A Parameter characterizes the information that the user must input to create the decision instance. Parameters can be numbers (integers or real types), text values or references to the Industry Model.

- For simple types, we have defined the SimpleParameter concept which represents a quantity measure, e.g. money, number of units, etc.
For complex data types the *FilterParameter* and *ComplexParameter* concepts were defined. A filter parameter describes a set of objects queries directly from the Industry Model, e.g. Business Units, Assets, Production Lines, Brands, Regions, etc. Complex parameters are used when a simple query on the model is not enough to obtain the required information, thus it has to be obtained from complex queries with additional processing.

*System Data* is a special kind of parameter automatically obtained from the context where the decision is made, typically it corresponds to the corporative group that makes the decision and the period of time where the decision is executed.

**Execution-block.** This block contains a set of accounting and operative transactions.

- An *Accounting Transaction* consists of a series of financial movements (*Accounting Movements*). These movements define the specific effect a decision has on an account in the financial state of a specific unit. A financial movement has a type of movement to apply (credit, debit), a quantity and an account to be affected, e.g. reduce the amount of money in the *SuppliesCash* account from a production unit.

- An *Operative Transaction* consists of a set of *Operative Movements*. These movements define the specific effect a decision has on items in the operative state of a specific unit. An operative movement has a type of action (create, modify, remove) and specific values (that vary depending on the type of action) to apply in the action to specific parameters, e.g. create a new supply Item to represent the engines that were bought, this item has various attributes and each has to be set using parameters or results of calculations of the decision.

In Figure 6, it is presented the updated conceptual model defined so far.

![Conceptual model for a Business Decision](image)

**Figure 6: Conceptual model for a Business Decision (2)**

Additionally, to support the required expressiveness for the transactions were defined the *Formula and Expression* concepts. A *Formula* allows the definition of
reusable processing for any decision or commitment. There are three different types of formulas: (1) arithmetic formulas, (2) search formulas, and (3) service formulas. An Arithmetic Formula basically corresponds to a function defined as an arithmetic expression, (e.g. total cost = multiply the amount of supplies to buy by the cost of the supply). A Search Formula is the definition of a query about the Industry Model (e.g. find the specific cost of an engine supply in a specific region). A Service Formula allows the simulator to invoke external services and algorithms that cannot be defined using simple expressions (e.g. the distance between two regions having as parameters their latitude and longitude).

Finally, the resulting metamodel is shown in Figure 7. This metamodel covers the concepts required for the complete specification of business decisions.

Figure 7: Excerpt of Business Decision Metamodel

In conclusion, we have presented how we found and defined the basic concepts of a Business Decision and the way that they are related to constitute the Decision Metamodel. This metamodel is the main asset for the development of the domain-specific language, because it corresponds to its semantic model.
In this chapter, we describe in depth the Domain Specific Language built as a result of the analysis and design stages. We show its general features, its main building blocks, and its expressiveness power provided by its concrete syntax.

### 4.1 General Features

MAGES-DSL is a textual strongly-typed procedural language. It is statically typed to make easier ensuring data-type matching and safety, and as a side effect bringing better performance at run-time (because is not required to do type computations at run-time as in the case of dynamic typing). Concerning execution, the language is procedural: the Decision’s transactions are executed sequentially because the Simulator has to execute the decisions in a given strict order due to the fact that all transactions have side-effects on the Industry model.

![Diagram of MAGES-DSL building-blocks](Figure 8: MAGES-DSL building-blocks)

To allow the specification of any new Decision or Commitment in the system, the language provides different modeling elements. In Figure 8 are shown the
language building blocks. Each outer box is a kind of model. While Decision, Commitment, and Library models are written by the user, the Industry Model is an internal resource available to all models, representing the set of entities, attributes and relationships that conforms the industry in the game.

**Decisions** A Decision model in MAGES-DSL can be viewed as a set of resources consisting of (1) a Decision file, (2) the required library files, and (3) the Industry model file. The Decision file consists of a single Decision or Library. A Decision model is defined mainly by (1) a unique identifier, (2) a set of input parameters (system data, simple parameters, filter parameters and complex filters) and (3) the run block (which contains all the inner working variables and the transactions). Note that parameters and variables are very different concepts. The former are specifically designed to be the inputs sorted by the user, while the variables block is used as a worksheet to compute the values required for the transactions.

**Commitments** A Commitment model has the same dependencies as a Decision model (i.e. libraries and the Industry model). In the Game, a Commitment is a decision instancer. To create decision instances, a commitment uses an iterator (For-Each statement), which allows the execution of accounting and operative transactions on each company’s business unit.

**Libraries** A Library model starts with the library identifier, followed by the set of formulas. There are a variety of formula types, according to what we mentioned earlier. These are: (1) arithmetic formulas, (2) search formulas (read-only queries on the Industry Model) and (3) service formulas (calls to external Java services).

**Expressions** To support the required expressiveness to model financial transactions and Industry Model object updates, the language has a built-in expression language. This embedded language provides support for arithmetic expressions (arithmetic operators: +, −, ∗, /, ( ), number literals, conditional-if, and formula calls). Also the language provides a basic string manipulation support (concatenation operator “&”, constants, and string literals). An additional support is given to call non-numeric formulas. The object navigation operator it will be discussed later (dot notation).

**Name-Spaces** MAGES-DSL is name-space aware, the global name-space is composed by all the Decisions and the set of shared Libraries. Any element in the name-space can be referenced in other models through its fully-qualified name (except if it is not in the current scope). A Decision element can refer to formulas from different libraries, but it can never refer to other Decisions (each Decision is fully independent of each other). Thus, all libraries in the same name-space are imported automatically in all Decisions. To call formulas, the only requirement is the fully qualified name (built as [module name].[formula name]). Behind this modularity are at least two advantages: user-level model reuse and better understandability. Finally, to simplify the current implementation a library can not import other libraries.

In Figure 9 is shown a class diagram that illustrates how a Decision interacts with the Industry model to represent the accounting and operative consequences
for a company. Note that each hierarchy is conform to its own metamodel. The relationships between a Decision model and the Industry model through model composition, define the integration mechanism between the system and the language.

![Diagram of Industry Model and Decision Model](image)

**Figure 9:** Integrating the system’s Industry model in the language

### 4.2 Concrete Syntax

The concrete syntax of MAGES-DSL is very similar to Java-like languages (except for some higher level features). For example, it supports the one-line and multi-line comments (`//Comment` and `/*Comment*/` respectively). And to declare any variable or parameter, the general syntax rule is: to write the type first and then its name (e.g. `CorporativeGroup myGroup`). Except for the main block, single sentence ends with semicolon (“;”), even when it is ended by a curled bracket (“};”).

Curled brackets normally define a closed scope for all items between them (except for parameters which are visible to the whole Decision).

Basic data types are in lower case, e.g. `string`, `real`, `int`, while object types use the camel-case convention, e.g. `CorporativeGroup`, `Unit`, etc. Lastly, String literals are between quotes (“double” or ’single’).

Next, we describe the meaning of the main Decision’s keywords and its usage through a real example as it is deployed in the MAGES production environment. Figure 4.1 shows the skeleton code for the “Buy Fixed Asset” Decision example.

#### 4.2.1 Parameters

In the listing is shown all the Decision’s blocks, the input parameters are shown between lines 6 and 19, while the `run` block starts at line 21. Listing 4.2 shows
the definition of each kind of parameters. *SystemData* are the parameters taken from the context at run-time, the mandatory parameters here are the group and the period. In this Decision there are two *SimpleParameters*, see lines 6 and 7, which represents the quantity of assets to buy and the corresponding serial number to identify the asset in the inventory states. The *FilterParameters* defined are the asset, the asset type and the asset’s target business unit, see lines 10 to 24. Each *FilterParameter* is defined as a selection list, whose data set is taken from a *SearchFormula* call (see lines 12, 17, and 22). Note that filters have to be displayed in a Web interface, so it has to have a plain-text representation. This representation is built collecting the name’s attribute of the list’s elements (see line 18).
4.2.2 Accounting Transactions

In listing 4.3 we detail the accounting transactions (which go inside the transactions block). Here we have two transactions, each one having two movements. First transaction, see lines 1 to 6, affects the unit’s accounts. The first movement increases the account balance (that corresponds to the asset) in a totalValue value. Note that the start of each movement is an informative label that shows the account nature (e.g. assets, expenses, etc.), as is shown in lines 3, 4, 9, and 10.

Prog. 4.2 Parameters of the Buy Asset decision example

```java
systemData{
    CorporativeGroup group;
    Period period;
}

simpleParameters{
    int quantity;
    string serialNumber;
}

filterParameters{
    filter assetType{
        type=AssetType
        dataset=getBuildingAsset()
        display=name
    };
    filter unit{
        type=Unit
        dataset=getCorporativeUnitsByGroup(group)
        display=name
    };
    filter asset{
        type=Asset
        dataset=getAssetsByAssetType(assetType,unit)
        display=name
    };
}
```

Prog. 4.3 Accounting Transactions for the Buy Asset example

```java
accounting asset_payment {
    unit: unit;
    assets: buyAccountName += totalValue;
    expenses: cashBuyAccountName -= totalValue;
}

accounting terrain_payment {
    unit: unit;
    assets: terrainBuyAccountName += terrainTotalValue;
    expenses: terrainCashBuyAccountName += terrainTotalValue;
}
```
4.2.3 Operative Transactions

Next, we show how an operative transaction is written. Listing 4.4 shows the operative transactions. In this case, what we want to say is the bought asset has to be added to the company’s inventory, where depending on the user selection, the asset has particular values for its attributes. In text, it is written as an add operation, which receives the item type as argument. In this case, we are specifying the new inventory element is an item of type asset (ItemAsset). In the system there exists about 33 kind of items. The add operation is an object instancer, whose body is a collection of attribute-value assignments, it is the way we implemented this Industry model update operation. This operation shows the tight integration between the Decision with the Industry model, because each item’s attribute is taken from the Industry model. The first assignment, see line 4, sets the item’s buy period to periodNumber, which is a variable computed before into the variables block.

Prog. 4.4 Operative Transactions for the Buy Asset example

```plaintext
operative asset_record {
    unit: unit;
    add(ItemAsset){
        buyPeriod = periodNumber;
        acquisitionCost = totalValue;
        lifetime = lifetime;
        saveValuePercentage = saveValuePercentage;
        historyCost = totalValue;
        actualCost = totalValue;
        serialNumber = serialNumber;
        quantity = quantity;
        unitCost = unitCost;
        state = state;
    }
}

operative terrain_record {
    unit: unit;
    add(ItemAsset){
        asset = terrainAsset;
        buyPeriod = periodNumber;
        acquisitionCost = terrainTotalValue;
        lifetime = terrainLifetime;
        saveValuePercentage = terrainSaveValuePercentage;
        historyCost = terrainTotalValue;
        actualCost = terrainTotalValue;
        serialNumber = terrainSerial;
        quantity = quantity;
        unitCost = terrainUnitCost;
        state = state;
    }
}
```
4.2.4 Type System and Industry Model Navigation

A critical requirement for the language was the capacity to reference Industry-Model business-entities as valid types for any variable, in order to access the real system objects at run-time. To achieve Industry-model referencing, we defined a type system composed by two data types: (1) Single Type and (2) Array Type, as it is shown in Figure 10.

A Single Type can be a simple data type (string, real, int, and boolean), or an Object Type (which refers to Industry entities, such as CorporateGroup, Unit, Period, etc). On the other hand, the Array Type corresponds to standard arrays that have as its base type a Single Type (therefore it is not possible to define an array of arrays). Although all these types are supported by the language, their use is restricted to very specific constructs (for example the arrays are only allowed to define filter parameters, as formula return types and into the For-Each operator).

The type system is integrated in the language parser and compiler to check type matching between elements in definitions, expressions, and assignments.

On the other hand, to allow navigation on the model already imported, the language defines the access formula expression as part of the expression language. An access formula implements the standard OCL (OMG’s Object Constraint Language) dot operator. Through this operator is possible to access attributes from a given variable whose type is an Object Type (e.g. “businessUnit.region”). It is important to notice that MAGES-DSL does not define operators to handle collections (except into Filter and Complex parameters) and therefore this operator is restricted to show only single-valued associations and attributes.
4.2.5 Variables

Due to the variables block is the user’s work area, it is the main place where the full expression language usage is allowed (the other place is into a library). In the "Buy Asset" example we have shown the transactions with all the values already computed. However these values are taken from variables initialized in the variables block. In listing 4.5 we show this block, which illustrates the access formula and the other expression language usages. For example, to get the region of the target business unit we use an access formula, see line 3. In line 4, we use a call to a search formula, and in line 6 we use a simple arithmetic operation. In line 9, we use the constant ItemAsset::State::BOUGHT (using the convention entity_attribute::value, which describes where the constant can be assigned). Note totalValue variable is the variable used in the accounting transaction shown in listing 4.3 in line 3 and 4.

```
Prog. 4.5 Variables used in the Buy Asset example

variables{
    //asset cost
    Region region = unit.region;
    AssetRegion assetRegion = getAssetRegion(asset, region);
    real unitCost = assetRegion.acquisitionCost;
    real totalValue = unitCost * quantity;

    //other values
    string state = ItemAsset::State::BOUGHT;
    int periodNumber = period.number;
    int lifetime = assetRegion.lifetime;

    //accounts
    string buyAccountName = assetType.buyAccount;
    string cashBuyAccountName = assetType.cashBuyAccount;
    //values for the terrain (...)
}
```

To illustrate the allowed expressiveness for variables, in Listing 4.6 is shown the BNF-like grammar for the variables block (in the appendixes there are additional details).

4.2.6 Execution of Decision Models

At simulation time, once all players have made their decisions, the system database will contain the set of decision instances ready to be executed. A decision instance is executed in the order shown in Figure 11.

The results are visible to users through the reports provided by the game.
4.2 concrete syntax

**Prog. 4.6** Expressions allowed for variables

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Variable:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>type=SingleType name=ID ’=’ value=Value</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Value:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>ArithmeticExpression</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>TextExpression</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>FormulaCallSS // SearchForm</td>
<td>ServiceForm</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>AccessFormulaCall</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>If</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

![Diagram](image)

Figure 11: Decision model execution

4.2.7 Commitment Model

As described at the beginning, Commitments are not run by the player but by the system, for that reason they can’t have parameters different to system data. Additionally, a Commitment usually needs to traverse all the company’s business units in order to access the desired items.

In a Commitment model is used the runForEach statement to iterate on a collection of items in order to execute transactions depending on the element processed (i.e. loans, assets, etc.).

The semantics of a commitment can be specified through the pseudo-code listed in 4.7 (where group is the Corporative Group).

**Prog. 4.7** Pseudo-code for the execution of a Business Commitment

```
For u in group.units do
  For i in getItemsFromUnit(u) do
    run.transactions(u, i);
  end
end
```

The listing 4.8 shows the skeleton syntax of a Commitment model.
The iterator invokes a SearchFormula to get the items to be traversed. The formulas usable in this context have to be defined with the `iterable` keyword. The formula definition for the above example is shown in the Listing 4.9:

**Prog. 4.9 Iterable keyword**
```
search allUnits():Unit[] iterable
{
  return 'select u from Unit u';
}
```

### 4.2.8 Execution of Commitment Models

At simulation time, the execution process for a Commitment model is very different from a Decision model. The execution order is shown in Figure .

### 4.2.9 Library Model

A Library is a package of user-defined formulas. Each type of formula with its syntax is described in table 4.
MAGES-DSL is built on top of Xtext (a language development framework) and EMF (Eclipse Modeling Framework). Xtext allows to specify a new language through a EBNF-grammar, from which it is able to generate automatically the language parser, validator, and editor with many configurable features.

MAGES-DSL Editor is composed by an Eclipse-based UI editor and the language parser/serializer. It has support for type safety validation (for variable assignments, variable references and complete expressions), full parsing of the embedded expression language and basic and custom validation (e.g. existence and uniqueness of symbol identifiers). To refer to any Industry entity (Object Type) all models import the Industry Model (an EMF Ecore model file), where the entities are all the EClasses contained into it.

In figure 13, it is shown how the Industry model is linked into a Decision model. Note the auto-completion feature the editor has (see center panel), which results corresponds to the Industry model concepts (see right panel).

Other important editor feature is the validation system we have mentioned. Figure 14, shows an error example, which is the violation of a restriction of the
allowed dependencies between input parameters: `SimpleParameters` can depend on `FilterParameters` but not the opposite. Because, the latter could lead to cyclic dependencies, which is effectively reported by the editor.

![Figure 14: Editor’s validation system](image)

### 4.4 SUMMARY

This chapter has presented some details of the MAGES-DSL language. We have shown its basic definition (hard typed and procedural nature), its basic working (name-space scoping, import mechanism, and type system), its textual syntax (through one Decision example) and its embedded expression language. Finally, we have shown the MAGES-DSL editor and its more important features.
<table>
<thead>
<tr>
<th>Formula Type</th>
<th>Description</th>
</tr>
</thead>
</table>
| **Arithmetic formulas** | **Meaning:** Formula defined as an arithmetic expression (as defined above in the expression types).  
**Example:**  
formula totalValue(real unitCost, real quantity):real  
  = unitCost*quantity;  
**Valid argument and return types:** int, real  

| **Search formulas** | **Meaning:** Read-only queries about the Industry Model. These formulas use external code to specify the current query definition using the JPA-QL language.  
The formula body can have local variables, which are variables used to obtain some attribute from one argument when the argument cannot be directly used into the JPA-QL query. Once a local variable is declared, its use into the query string is mandatory. This also applies for the declared with simple data types (int, real, string, boolean).  
**Example:**  
search getUnits(CorporativeGroup g):string  
  {  
    int id = g.corporativeGroupID;  
    return 'select g from CorporativeGroup g where g.corporativeGroupID=:id';  
  }  
**Valid argument types:** ObjectType, int, real, string, boolean  
**Valid return types:** Array[SingleType], Array[Object]. The array-of-arrays type is not supported by the language.  

| **Service formulas** | **Meaning:** calls to external Java services.  
**Example of call to an EJB Session bean:**  
service getTransportDistance(Region source,  
  Region destiny):real  
  {  
    fromClass="utils.DistanceService"  
    invoke="distanceBetween"  
    option=ejb;  
  }  
**Example of call to a static class:**  
service sin(real a):real {  
    fromClass='java.lang.Math' invoke='sin' option=static;  
}  
**Example of call to a plain Java class using object instantiation:**  
service sin(real a):real {  
    fromClass='own.Math' invoke='sin' option=simple;  
}  
**Valid argument types:** ObjectType, int, real, string, boolean  
**Valid return types:** Array[SingleType], Array[Object]. The array-of-arrays type is not supported by the language.  

Table 4: Library syntax and definition of formulas
MAGES-DSL: CODE GENERATION PROCESS

In general, the execution of a DSL can be in two ways: through code transformation into a general-purpose programming language (through Model Driven Techniques) or through interpretation (i.e. building high-level or low-level interpreters [14]). We have opted for code generation, which allows us to generate all the artifacts to implement both the Web interface and the simulation back-end.

5.1 GENERATED ARTIFACTS

From the input models (MAGES-DSL files), we execute a Model-to-Text transformation to automatically generate the following artifacts: (1) the Decision-Maker Web interface, (2) the libraries code (Data Access Objects and business-logic services) and (3) the report descriptors. Due to the fact that MAGES is fully implemented in the JEE5 (Java Enterprise Edition) platform, these artifacts also correspond to JEE elements: JSP pages, XML files, reports descriptors, Java Web beans, EJB Entity beans and EJB Session beans.

![Image of the Decision Maker Web interface]

Figure 15: Decision Maker Web interface (automatically generated from MAGES-DSL specifications)

The generated Decision-Maker Web interface corresponding to the "Buy Fixed Asset" Decision example is shown in figure 15). It is the component used by
the end-users (i.e. the students) to make decisions (in this case to buy assets). Therefore, this component allows users to instantiate a decision and to persist it into the system. The instances are stored in the MAGES database in order to be loaded and executed later by the MAGES Simulator.

The second group of generated artifacts (libraries code) corresponds to the back-end business logic services. Each MAGES-DSL Library is transformed into a corresponding Java service class and each library formula is transformed into a Java method. Here, the original Java services referenced in the Service formulas are also integrated. The last group of artifacts corresponds to additional descriptors used in the Reporter component (the reporter component is out of the scope of this paper).

(a) EJB code

(b) Web code

Figure 16: Generated artifacts for the Buy Asset example

In Figure 16 are shown all the JEE artifacts generated for the Buy Asset decision (showing also the size in LOCs).
Once the Decisions are modeled, transformed, deployed, and tested, they are released into the production environment. There, the system is configured by the administrators in order to start the game. In each period, each corporative group creates Decision instances according with its business strategy. When the decision-making stage ends, the system starts the simulation process.

In Figure 17 is shown the Business Simulator component, which is an orchestrator of concrete simulators (in this case it is shown the Buy Asset simulator). The Buy-Asset simulator can only execute Buy-Asset instances (loaded from the database). The execution process is shown in Figure 18.

An initial set-up is needed in order to load the values of the input parameters from the Decision instance. The minimum execution block is a transaction. So the evaluation starts with the first transaction found in the Decision. The transaction execution starts by evaluating the first transaction-movement. This movement can depend on the values of a variable or a parameter. The first time, all variables are blank; therefore if it is referenced a variable it has to be resolved. But if the transaction depends on a parameter, the value can be directly fetched.

A variable is computed by evaluating its assigned expression, which can be formed by other variables, parameters, and formula calls. If a variable is found into the expression it must be resolved. This chain ends when a concrete value is found (i.e. a parameter is found). The language does not allow null values nor
circular references. For that reason, a variable must have only one assignment, and must be evaluated only once. If a variable is referenced many times, the first time is resolved and the subsequent times it is returned its current value.

The whole execution ends when all movements in all transactions are executed. The language does not specify a way to handle execution errors (exception handling), but it logs all the simulation activity in order to provide a traceability to system administrators in case of run-time exceptions.
PROJECT VALIDATION

As it is stated in [13], quantitative validation of DSLs is hard and an important open problem (in the general case and even for particular cases). For that reason, our evaluation is largely qualitative. First we present an internal-view evaluation: we verify the compliance of design principles and show some collected metrics. Second, we present an external evaluation, done by the stakeholders (and not by ourselves), with the aim to verify the compliance of the requirements and expectations.

6.1 DESIGN PRINCIPLES VERIFICATION AND COLLECTED METRICS

There are some works which aim at defining principles for the design of DSLs [6, 8, 12]. We are going to follow the ideas in [8] and to discuss how we fulfill the main principles established there:

Representation. This principle involves a concrete and optimal syntax that enables users to write unambiguous DSL concepts and a code structure to arrange the different grammatical sentences. In our case, the syntax was defined following syntactic representations common to the business domain such as accounting transactions, operative transactions, etc.; inside these blocks, we use familiar concepts of this domain such as the vocabulary of accountable movements. The blocks of the language can be distinguishable easily.

Absorption. Our DSL absorbs common practices within the Business Domain. These practices can be materialized in formulas which are defined inside libraries that besides facilitate reuse.

Standardization. Our DSL restricts its grammar and semantics to concrete constructors that the user has to use to specify the decisions.

Abstraction. We use this principle to define the rich (complex) data-types that come from the Industry Model and are available to the user. Also, this principle was used during the definition of complex parameters that can refer to other simpler constructors.

Regarding Expressiveness, MAGES-DSL has been extensively tested by implementing a set of representative test models (i.e. the largest business decisions) using the DSL. Through this process we verified that the required business logic was fully supported by the language, and we ensured that the simulation (behavior) was correct through the execution of automatic regression tests.

This testing work was done in collaboration with the MAGES development team, which configured the test scenario and implemented the regression tests (for it was used the Selenium testing framework which allows the automatic interaction with the Web interface).
Currently, there exist 73 business decisions on diverse domains that will be migrated in the mid-term. Table 5 shows the number of decisions implemented by each business domain. We can confirm that the expressiveness of the language was enough to specify these decisions of diverse domains.

<table>
<thead>
<tr>
<th>Decisions Category</th>
<th>Number of Decisions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed Assets</td>
<td>10</td>
</tr>
<tr>
<td>Commercialization</td>
<td>9</td>
</tr>
<tr>
<td>Finances</td>
<td>21</td>
</tr>
<tr>
<td>Taxes</td>
<td>3</td>
</tr>
<tr>
<td>Investment and Development</td>
<td>6</td>
</tr>
<tr>
<td>Production</td>
<td>24</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>73</strong></td>
</tr>
</tbody>
</table>

Related to Compression, the average number of lines (sentences of the MAGES-DSL language) of a decision is 116. The shortest decision has 25 lines of specification and the longest is about 566. All these files are easy to edit in our tool which provides a series of facilities to perform validations and completions. In general terms, the complexity of each decision resides in the number and complexity of parameters and formulas, specially in the formulas section, because the calculations that have to be done to modify the financial and operative states, are expressed by arithmetic and search formulas which are the language components that involve the higher number of lines of specification.

Regarding Productivity and Quality of our approach, we can say that the use of the DSL and model driven techniques proved to be very successful in terms of the evolution of the requirements and extensibility. We were able to reduce the cost of modifying the simulation of existing decisions and the cost associated with adding new business decisions within the system. Our solution facilitate maintenance and testing tasks, it avoids developers to implement changes to the code each time a domain expert wants to modify the behavior of the simulation regarding a business decision or each time a new decision has to be added to the simulation. Table 6 shows some productivity data about the generated and manual code associated with our DSL.

Many of the business decisions have been modified once in production because the domain expert decides to do so for several different reasons including corrections to the specifications. The process of modifying an already deployed business decision is pretty straightforward: the user changes the decision specification file and executes a process which automatically produces the artifacts ready to be deployed into the system.

The ratio of generated code versus hand-made code can be illustrated with one mid-sized Business Decision (Buy Asset). The number of lines in MAGES-DSL
Table 6: Productivity data

<table>
<thead>
<tr>
<th></th>
<th>KLOCs</th>
<th>Percentage(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generated code</td>
<td>279.2</td>
<td>97.1</td>
</tr>
<tr>
<td>Manual code</td>
<td>8.5</td>
<td>2.9</td>
</tr>
<tr>
<td>Total</td>
<td>287.7</td>
<td>100</td>
</tr>
</tbody>
</table>

is about 113. The generated code corresponds to 4399 LOCs (counting the size of the 9-generated elements). The resulting ratio is 38.9. In this example, one line in MAGES-DSL corresponded to near 39 JEE LOCs (automatically generated).

6.2 EVALUATION WORKSHOP

In order to get feedback from stakeholders about the overall project success, we did a workshop to evaluate the implementation of the MAGES-DSL language. The workshop was organized according with the guidelines presented in [9]. The paper presents a practical methodology and a questionnaire to measure DSL success factors.²

The workshop consisted of: (1) a short training session about MAGES-DSL and its integration with MAGES, (2) a worksheet with 7 exercises, and finally (3) a survey to summarize the experience they had with the tool.

Due to the target population of the DSL is the MAGES development team, the workshop was designed for them (the team consists of 4 persons). However, the workshop was done only by three of them. Here, we clarify that the MAGES-DSL project members does not belong to the MAGES development team.

Two participants have been involved many years in the MAGES project while the other has been involved for 6 months. Therefore the profiles are very different: two are experts in the data model and the implementation details but only one has been responsible of the elicitation and specification of Business Decisions. The last participant has been working in a GUI application for the DSL (in a parallel project) but does not have a strong background in the MAGES data model and its implementation.

Below we present the goal, the requisites, and the results obtained.

6.2.1 Workshop Goal

The goal of the workshop was to evaluate the expressiveness, usability, and learnability of the MAGES-DSL language from the users’ point of view. By expressiveness we refer to the language’s capability to represent all the domain

² In [9], they validated their proposed questionnaire using different criteria, such as experts advice, pair revision, a test of concept, and they used it to evaluate their own DSL in a real environment. In fact, their DSL shared some characteristics with MAGES-DSL: both are textual and were designed for developers.
elements. The usability refers to how easy and fluent is the modeling process. Finally, the learnability refers to the learning curve required to become a master in the language.

6.2.2 Workshop Setup

To develop the workshop, some familiarity was required with the Management Game Simulator, the MAGES-DSL language, and the JPA-QL query language (as part of the DSL’s foreign code). The first two are covered in general terms in the Introduction to MAGES-DSL document given as part of the workshop’s material. Additionally, the workshop required the use of the language editor (for the modeling) and the use of the MAGES development environment (for code generation). The whole environment is available at: http://mages-dsl.virtual.uniandes.edu.co.

The workshop consisted in the specification of 7 exercises in order to answer a survey (this material is available in the website mentioned above). Each exercise consisted in the specification of one decision or commitment using the DSL (to be developed individually without time restrictions). Finally, the survey consisted of 27 questions (which can be found in the Appendix A), including quantitative and qualitative information. The estimated time for the development of the workshop was 11.5 hours (1 h for learning, 1.5 h per exercise).

6.2.3 Results

The results obtained will be shown according with each category evaluated. Some statistics will be shown in terms of the absolute frequencies or even the raw data, because the survey population is very small.

6.2.3.1 Development Costs

The development effort was evaluated by mean of the next questions:

- Time spent developing the whole workshop: 10:30; 09:40; 11:53 (average: 10:41 hours). Which is below the estimation (11.5 h).
- Time distributed in days (#): 3; 10; 5 (average: 6 days).
- Time spent developing the exercises only: 09:45; 06:34; 10:23 (average: 8:54 hours). This time is also below the estimated value (10.5 h).
- The times recorded per exercise are shown in Figure 19
- Can you estimate the percentage of time (0 – 100%) that would be spent on the following tasks if MAGES-DSL was not used for this project (i.e. using the Java language directly)? See Figure 20.
If MAGES-DSL is used, there are tasks that do not have to be performed anymore, but it still takes time to get everything to work. For example, repetitive tasks such as copying files, validating Java code, linking of Java classes, queries’ validation, generating code, compilation, etc. Please indicate how much time this took you in total (10 - 100 %): 10%; N/A; 8%; In average it is required an effort of about 9% for doing repetitive tasks (which in average corresponded to less 1 hour in this workshop). According with the Figure 21, the required effort can correspond to the validation of Search Formulas, which needs JPA-QL code that is not fully automatically validated.

6.2.3.2 Learnability

The questions concerned with learnability (and their answers) were:

– Have you had experience with the Game before this workshop?: 3/3
– (If you have experience) how much time?: 6; 7; 0.5 (average: 4.5 years)
– (If you have experience) which role have you played?: All are developers. One focused in designing Decisions, other in the Decision Manager implementation, and last one in the GUI implementation for Decision specifications. This made the exercise more interesting, because all the participants had different roles.
– How many years have you worked as a software developer? 6; 0.5; 7 (average: 4.5 years)
– Time learning about MAGES-DSL: 0.75; 3; 1.5 (average: 1.75 hours)
– Do you think the use of MAGES-DSL against the Java counterpart increases the quality of the delivered code?
  → The product complies better with the customer’s requirements: Agree; Neutral; Agree; (Partial agreement)
  → Fewer errors occur: Agree; Neutral; Neutral; (It is not possible to conclude anything)
  → The code is more readable: Agree; Strongly Agree; Strongly Agree; (By consensus the code tends to be very readable)

6.2.3.3 Usability

– (For each exercise) specify whether you succeed or not: All exercises were completed except the last one for two developers. The reasons are described in the next question.
– What do you consider complex and unclear to specify in MAGES-DSL?:
  “[...]Lack of examples for external Decisions”;
  “Services and external Decisions”;
  “External Decisions and complex parameters”.

The answers clarified that the Guide given to them did not show any examples about external Decisions, which did not allow to specify the last exercise.
- Did you easily specify the exercises using MAGES-DSL? All answered yes.

- How many lines of MAGES-DSL code does your project have? The number of LOCs per exercise are shown in Figure 19. A quick view over these numbers shows that the workshop required in average about 800 LOCs. These magnitudes are close to the average recorded during the testing. However, these are big numbers that could be reduced through some strategies (compressing the syntax, developing a complete API ready to use, etc).

- Can you estimate the percentage of time (0 – 100%) that would be spent if MAGES-DSL was not used for this project?

- Did the MAGES-DSL user interface help you modeling? How?

  [Through] auto-complete functionality, syntax validation, and type validations.

  Navigating the attributes of variables.

  When the syntax was not clear or I didn’t remember what to write, I pressed Control+Space and the interface showed the available options for auto-completion.

  The most used and worthy feature was the auto-complete function, which corresponded to one of the main objectives outlined in the implementation phase: to allow the navigation of the Industry model.

- Did you use other tools for modeling in this project, next to the MAGES-DSL interface (for example Excel, pen and paper)? Yes; No; No.

- [...] Why? All used only the specification documents in order to resolve the exercises. However, there was need to use the MAGES development environment in order to register the needed external services by hand.

  Only with the html specs. Only the specification of decisions to know which were their parameters and transactions.

- Which of the following tasks was more time-consuming while using MAGES-DSL? See Figure 21.

- Did you have to write extra code (other than custom code referred in the previous question) to implement features you weren’t able to generate with MAGES-DSL? c) Sometimes; e) never; e) never (Code generation is near 100% of the final implementation)

- How often do you interact with the generated code? d) seldom; e) never; d) seldom.

- Are you satisfied with MAGES-DSL?

  - MAGES-DSL reduces the decisions development time dramatically: Agree; Neutral; Agree
  - MAGES-DSL makes modeling easier: Agree; Agree; Agree
  - MAGES-DSL makes implementation easier: Agree; Agree; Agree
Do you consider that reasons exist that make the use of MAGES-DSL not viable?: All answered: Neutral.

Do you consider that MAGES-DSL allows you to (Yes/No)

- Validate models to ensure they are coherent? All answered yes.
- Correct syntactic and semantic errors with helpful automatic advices? All answered yes.

6.2.3.4 Expressiveness

- What kind of decisions, expressions or logic would you like to specify with MAGES-DSL that you consider are not supported in the current version?

  “The ability to make access operations inside a filter or complex parameter”.
  “Allow access operations directly into the transactions”.
  “[There is] no for-each commitments”

After a small discussion, we concluded that the first answer concerns to an expressiveness restriction that must be resolved. The second one, corresponded to a desired improvement in expressiveness. Finally, the last one corresponded to other desired improvement that was not discovered during the elicitation process.

- [...] Why? All answered: Option D: missing the notation to do it.

- What changes or additions to MAGES-DSL do you propose?

  - To automate the update process for new account types or new Industry model. To give support to create more synthesized operations. To provide code templates for quick creation of: empty decisions, empty commitments, and external decisions.

  - To change some names in the account types. I suggest this structure for accounting transactions: the use of the financial statement name as part of the account full name [...]. Direct constant modifications on operative attributes for numbers, as it is for booleans. To allow access operations in transactions.

  - To allow assign null values in variables. Formulas should be able to receive objects as parameters and not only integers or reals. A developer should be able to make an access operation inside a filter parameter and then use the result of such operation as a parameter for the filter itself.

There were #4 proposed automations, some improvements and corrections; and # 3 improvements. However, according with the exercise none of them were required. However, many of them are related to increase productivity by removing restrictions or by adding improvements.
- Do you have any other remarks regarding MAGES-DSL?

“Specifications are still complex, and could be difficult to understand for everyone”

This answer suggested us that is not the same “it is easy to use” to “it is easy to understand for others”. Specifically, we interpret that as: “Please improve the arrangement of the syntax in order to highlight better the semantics.”

- MAGES-DSL restricts my freedom as programmer: Agree; Neutral; Disagree (A conclusion is not possible).

- MAGES-DSL doesn’t have all features I need: Agree; Neutral; Disagree (A conclusion is not possible, however it can be concluded that it will completely different once the features they request are implemented).

- How much external code did you need?

  \[ \rightarrow \text{\# of External transactions: 0; 2; 0 (This answer corresponds to the exercise that was not resolved by two of them)} \]

  \[ \rightarrow \text{\# of Service Formulas: 2; 20; 1 (the excessive usage registered by User 2 reflects that many things really were possible to be specified without the need to use services).} \]

- [Do you agree with …] MAGES-DSL is powerful (i.e. very expressive): Disagree; Neutral; Agree. (The opinions are contrastive).

- Do you consider that MAGES-DSL allows you to (Yes/No)

  \[ \rightarrow \text{Model complete Decisions for the Management Game? 2/3 answered yes.} \]

  \[ \rightarrow \text{Model transactions using convenient syntax? All answered yes.} \]

  \[ \rightarrow \text{Model using Game Industry elements in a clear and non ambiguous way? All answered yes.} \]

  \[ \rightarrow \text{Model the same business-logic in a easier way than the Java counterpart? All answered yes.} \]

6.2.3.5 Reusability

- How many models did you specify from existing ones: 7; 3; 0 (In average, a third part of the workshop was resolved reusing existing models)

- How do you proceed?

  b) I copied an existing model to modify it

  b) I copied an existing model to modify it

  c) Other way: I only reused some formulas or queries in the libraries.

  It suggests that in practice it is easier to reuse partial models (through copy-paste) than through user-defined formulas.
- How many models did you specify from scratch (without referring to existing models)? 0; 4; 7 (In average, a third part of the workshop was resolved without needing other existing models)
Figure 19: Evaluation workshop results

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>BuySalesPoint</td>
<td>63</td>
<td>74</td>
<td>88</td>
<td>75</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>DepreciateStraightLineSalesPoint</td>
<td>42</td>
<td>50</td>
<td>48</td>
<td>47</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>HireDealerEmployee</td>
<td>55</td>
<td>76</td>
<td>78</td>
<td>70</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>FireDealerEmployee</td>
<td>99</td>
<td>125</td>
<td>135</td>
<td>120</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>PayDealerMarketingAndSalesStaffPayroll</td>
<td>63</td>
<td>85</td>
<td>84</td>
<td>77</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>MoveDealerEmployee</td>
<td>127</td>
<td>140</td>
<td>135</td>
<td>134</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>SellProducts</td>
<td>72</td>
<td>119</td>
<td>115</td>
<td>102</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>(user-defined formulas)</td>
<td>179</td>
<td>151</td>
<td>218</td>
<td>183</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>700</td>
<td>820</td>
<td>901</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Productivity (LOC/H)</td>
<td>72</td>
<td>125</td>
<td>87</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Figure 20: Percentage of time per task if MAGES-DSL is not used.

Figure 21: More time-consuming task per element
The ultimate goal of the project in the mid-term is to allow domain experts to specify directly the business rules in a very high-level language. In that scenario, the instructors will be able to model, execute, and test the business decisions in an integrated modeling environment, without the need to communicate with the developers, allowing them to resolve any modeling problem by themselves. In that way, the more challenging problem will be fully resolved: elicitation and compliance of functional requirements and time to market reduction.

Concerning the current implementation, we have found some things to improve. According to the Evaluation Workshop, we identified a set of improvements that should be done on syntax, expressiveness, modeling tools, support for testing, etc. Additionally, a documentation mechanism has been requested.

In the following lines we will present some possible future directions for the project:

- **Testing Support (Language Debugger):** This line of work seeks to provide a more interactive execution similar to scripting languages, where code generation, compilation, and deployment, is not needed in order to run a model. Candidate technologies are: MXF\(^1\) and Xtext Interpreter (which provide execution traces). We have knowledge also about RedView (a framework for dynamic GUI, which can be used to accelerate the modeling process).

- **GUI Editor inspired in Spread Sheets:** Currently, there is a parallel project for building a more user-friendly interface to elicit the decisions. We are taking the ideas from spreadsheet applications to facilitate the edition of formulas and variables. In Figure 22, it is shown a possible design of a visual editor inspired in worksheets. The current implementation, however, is closer to the design presented in Figure 23.

### 7.1 A Language for Domain Experts

We are conscious that a language targeted to non-technical people needs a lot of high-level features supported from the GUI (as in Figure 22) to the back-end services.

We identified the following improvements on the current implementation to achieve that goal:

- Raise the abstraction level to remove low-level elements.

\(^1\) [http://www.eclipse.org/proposals/mxf/](http://www.eclipse.org/proposals/mxf/)
- Remove unnecessary restrictions mainly to reduce the size of the specifications.
- Give better assistance for query building, through a simpler query language or through improved tools.
- Reduce the necessity for external Java code.

Referring to the language back-end, a direct way to achieve high-level elements is to add a new layer in the language stack. In that way, we can design a new syntax centered in management operations, conforming to a higher-level metamodel. Thus, through a model-to-model transformation we can generate a new model conforming to MAGES-DSL.

![Management Game Simulator - Decision Builder](image)

**Figure 23:** Other design of the editor
This strategy seeks the next goals:

1. To achieve the new language syntax reusing the current MAGES-DSL infrastructure, and
2. To simplify the language editor development.

The second goal refers to the fact that the development of a GUI editor is greatly reduced if the syntax and the underlying metamodel are simple.
CONCLUSIONS

We have presented a complete case study of the development of a DSL and its implementation using Model Driven techniques. The language, MAGES-DSL, was designed to model Business Decisions. The process of designing the language has included stakeholders (knowledge experts) from diverse domains (marketing, production, human resources, etc.). It is worth noticing that the most important and difficult step was the conceptualization process which lead us into a generalization on the consequences of decisions as transactions to modify financial and operative states. We have presented an evaluation of our language in terms of some design principles, and in terms of some data we have gathered during the whole development process. This evaluation allows us to confirm the benefits of DSM/DSL approach to substantially gain in productivity and quality.

With the DSM approach we have achieved a separation between the business knowledge and the complexity of the underlying implementation platform. In general terms, the use of the DSL and model driven techniques proved to be very successful in terms of the evolution of the requirements and extensibility. We were able to reduce the cost of modifying the simulation of existing decisions and the cost associated with adding new business decisions within the system. In addition, our solution facilitates maintenance and testing tasks. It avoids developers the need to implement business-logic changes directly in the system source code each time a domain expert wants to modify the behavior of the simulation.
Part I

APPENDIX
MAGES-DSL EVALUATION WORKSHOP

GOAL  The goal of this workshop is to evaluate the expressiveness, usability, and learnability of the MAGES-DSL language from the developer’s point of view. By expressiveness we refer to the language’s capability to represent all the domain elements. The usability refers to the modeling in an easy and fluent way. Finally, the learnability refers to the learning curve required to become a master in the language.

REQUIREMENTS  It is required some familiarity with the Management Game Simulator, the MaGes-DSL language and the JPA-QL query language. The first two are covered in general terms in the INTRODUCTION TO MAGES-DSL document. Additionally, the workshop requires the use of the language editor (for the modeling) and the use of the MaGes development environment (for code generation). The whole environment is available at: http://mages-dsl.virtual.unianes.edu.co

EXERCISE  Please use MaGes-DSL to specify the solution for the exercises described at the end of this document and answer the survey. Please keep track of the time you spend to solve each exercise as it will be asked to fill the survey. Survey

SURVEY

1. Have you had experience with the Game before this workshop? If yes, for how long?
2. If yes, which role have you played?
3. If no, how much time did you spend learning about the Game?
4. How many years have you worked as a software developer?
5. Time learning about MAGES-DSL
6. Please record the time you spent solving each exercise and additionally specify if you succeed with each one.
7. Time spent developing the whole workshop
8. Did you easily specify the decisions using MaGes-DSL?
9. What do you consider complex and unclear to specify in MaGes-DSL?
10. What kind of decisions, expressions or logic would you like to specify with MaGes-DSL that you consider are not supported in the current version? Why?
11. What changes or additions to MaGes-DSL do you propose?
12. Do you have any other remarks regarding MaGes-DSL?
13. How many models did you specify from existing ones? How do you proceed?

14. How many models did you specify from scratch (without referring to existing models)?

15. Did the MaGes-DSL user interface help you modeling? How?

16. Did you use other tools for modeling in this project, next to the MaGes-DSL interface (for example Excel, pen and paper)? (Please describe why)

17. Which of the following tasks was more time-consuming while using MaGes-DSL?

18. Do you agree with the following statements (Strongly Disagree, Disagree, Neutral, Agree, Strongly Agree)?
   a) MaGes-DSL restricts my freedom as programmer
   b) MaGes-DSL doesn’t have all features I need

19. How many lines of MaGes-DSL code does your project have?

20. How much external code did you need?

21. Did you have to write extra code (other than custom code referred in the previous question) to implement features you weren’t able to generate with MaGes-DSL?

22. How often do you interact with the generated code?

23. Do you think the use of MaGes-DSL against the Java counterpart increases the quality of the delivered code? Please answer according to each point (Strongly Disagree, Disagree, Neutral, Agree, Strongly Agree)
   a) The product complies better with the customer’s requirements
   b) Fewer errors occur
   c) The code is more readable

24. Can you estimate the percentage of time (0 – 100%) that would be spent on the following tasks if MaGes-DSL was not used for this project (i.e. using the Java language directly)?

25. If MaGes-DSL is used, there are tasks that do not have to be performed anymore, but it still takes time to get everything to work. For example, repetitive tasks such as copying files, validating Java code, linking of Java classes, queries’ validation, generating code, compilation, etc). Please indicate how much time this took you in total (10 - 100 %).

26. Are you satisfied with MaGes-DSL (Strongly Disagree, Disagree, Neutral, Agree, Strongly Agree)?
   a) MaGes-DSL reduces the decisions development time dramatically
   b) MaGes-DSL makes modeling easier
   c) MaGes-DSL makes implementation easier
d) MaGes-DSL is powerful (i.e. very expressive)

27. Do you consider that reasons exist that make the use of MaGes-DSL not viable?

28. Do you consider that MaGes-DSL allows you to (Yes/No)

   a) Model complete Decisions for the Management Game?
   b) Model transactions using convenient syntax?
   c) Model using Game Industry elements in a clear and non ambiguous way?
   d) Validate models to ensure they are coherent?
   e) Model the same business-logic in a easier way than the Java counter-part?
   f) Correct syntactic and semantic errors with helpful automatic advices?
## MAGES-DSL: Description of Input Parameters

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
</table>
| **System data** | **Meaning:** System Data is a special kind of parameter obtained from the “context” where the decision is taken.  
These context variables are explicitly declared as follows:  
- group (player’s Corporative Group)  
- period (current Period)  
**Syntax for Decisions:**  
```java  
systemData{  
    CorporativeGroup group;  
    Period period;  
}  
```  
**Syntax for Commitments:**  
```java  
systemData{  
    CorporativeGroup group;  
    Period period;  
    Unit unit;  
}  
```  
In this case, unit refers to “for each Corporative Group’s unit” |
| **Simple Parameters** | **Meaning:** Used to hold simple values. It represents a quantity measure, e.g., money, number of units, etc.,  
**Example:**  
```java  
simpleParameters{  
    int quantity;  
}  
```  
**Example for money quantities:**  
```java  
simpleParameters{  
    int moneyAmount  
    currency:getCurrencySymbolFromUnitRegion(unit);  
}  
```  
The Currency symbol is resolved at runtime, because it has to be obtained from the Unit’s region (Unit is just other parameter).  
**Valid data types:** string, real, int |
<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
</table>
| Filter Parameters  | **Meaning:** Used to hold values corresponding to Industry elements. Filter parameter describes a set of objects queries directly from the Industry Model, e.g., Business Units, Assets, Production Lines, Brands, Regions, etc.  
**Example:**
```
filter unit{
  type=Unit
  dataset=getCorporativeUnitsByGroup(group)
  display=name
};
```
**Note:** the dataset has to be set from a search formula call  
**Valid data types:** ObjectType                                                                                                                                                                                                 |
| Complex Filters    | **Meaning:** Complex parameters are used when a Search Formula is not enough to obtain the valid data set. Through the use of Service Formulas, the user can specify a choose list from complex data processing.  
**Example for simple types:**
```
complexFilter lineName{
  type=string
  dataset=getLinesByUnit(plant)
};
```
**Example for object types:**
```
complexFilter unit{
  type=Unit
  dataset=getCorporativeUnitsByGroup2(group)
  display=name
};
```
**Note:** in this case the difference against filter parameters is the use of service formula calls.  
**Valid data types:** ObjectType
At run-time, all input parameters are rendered in the Decision Maker Web interface as shown in the figure.

System data and simple parameters are interpreted in a direct form, while the filters require additional data:

- **Filter parameters:**
  - Visualization: Is rendered as a combo box control (single selection)
  - Type: is the real type of the current parameter value
  - Dataset: is the data list obtained from a formula Search $f$, which returns the game objects.
  - Display: Is a simple attribute for textual representation of the objects (object identifier).
  - Value: is the value selected by the user.
  - Working: at simulation, the current object is obtained through a query that receives the identifier and returns the real object (this service is automatically generated: $f^{-1}$)

- **Complex filters:**
  - Visualization: Is rendered as a combo box control (single selection)
  - Type: is the real type of the current parameter value.
Dataset: is the data list obtained from a service formula \( f \), which returns the game data. This service is usually a complex query involving data from multiple data types (in this case the parameter type has to be Array of simple types). For that reason there is not a Display tag.

Value: is the value selected by the user.

Working: at simulation, the current object cannot be obtained automatically as in filter parameters. Therefore it is usually required to use a variable in the Run block that calls the inverse service to get the object’s reference \((f^{-1})\).
## MAGES-DSL: Supported Expressions

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Arithmetic expressions</strong></td>
<td>- Operations: &quot;+&quot;, &quot;+&quot;, &quot;+&quot;, &quot;/&quot;, &quot;(&quot;</td>
</tr>
<tr>
<td></td>
<td>- Number literals: 123 and 123.1</td>
</tr>
</tbody>
</table>
|                                 | - Formula calls: \( f(x), g(y+1), f(g(x+y)) \)
| **Note:**                       | the allowed operands are variables or constants (and arithmetic formula calls)                                                                                                                                |
| **Text expressions**            | - Concatenation operator: variable1 & variable2 & ...& variable                                                                                |
|                                 | - String literals: "string" or 'string'                                                                                                           |
|                                 | - Text Constants: `ItemAsset_State::BOUGHT` represents a constant that is a valid value for the ItemAsset’s state attribute.                                                                           |
| **Note:**                       | the allowed operands are variables or constants.                                                                                                 |
| **If-statement**                | - If(numeric_condition; variableValue; variableValue)                                                                                             |
|                                 | Ex: if(x=1; y; z). Also: if(x<>1; z; y). Where y, z can be a constant or a variable.                                                             |
| **Formula calls**               | - Search formula calls: getDistance(y, z).                                                                                                      |
|                                 | Where y, z can be a constant or a variable.                                                                                                |
|                                 | - Service formula calls: getPrice(z).                                                                                                           |
|                                 | Where z can be a constant or a variable.                                                                                                       |
| **Note:**                       | To use formula calls in the Variables block, the assignment operator has to be "=" (not "+")                                                                                                               |
| **Access expression (dot operator)** | - Access attributes from a given variable whose type is an ObjectType:                                                                            |
|                                 | Example: unit.region                                                                                                                             |
|                                 | This operator is restricted to show only single-valued associations and attributes.                                                            |
MAGES-DSL: GRAMMAR

See the attached document: APPENDIX-MAGES-DSL-grammar.


