On the Use of Metaprogramming and Domain Specific Languages: An Experience Report in the Logistics Domain

Pedro Henrique Teixeira Costa, Edna Dias Canedo and Rodrigo Bonifácio de Almeida
pedro.costa@aluno.unb.br,ednacanedo@unb.br,rbonifacio@unb.br
Computer Science Department, University of Brasília - (UnB)
Brasília, DF

ABSTRACT

In this paper we present the main design and architectural decisions to build an enterprise system that deals with the distribution of equipment to the Brazilian Army. The requirements of the system are far from trivial, and we have to conciliate the need to extract business rules from the source code (increasing software flexibility) with the requirements of not exposing the use of declarative languages to use a rule-based engine and simplifying the tests of the application. Considering these goals, we present in this paper a seamless integration of meta-programming and domain-specific languages. The use of meta-programming allowed us to isolate and abstract all definitions necessary to externalize the business rules that ensure the correct distribution of the equipment through the Brazilian Army unities. The use of a domain specific language simplified the process of writing automatic test cases related to our domain. Although our architecture targets the specific needs of the Brazilian Army, we believe that logistic systems from other institutions might also benefit from our technical decisions.

CCS CONCEPTS

• **Applied computing** → **Business rules**; **Military**; • **Computing methodologies** → *Knowledge representation and reasoning*; • **Software and its engineering** → *Source code generation*; *Specialized application languages*; n-tier architectures; Agile software development;

KEYWORDS

Software Architecture, Generative Programming and Domain Specific Languages, Military Logistics

ACM Reference Format:
Pedro Henrique Teixeira Costa, Edna Dias Canedo and Rodrigo Bonifácio de Almeida. 2018. On the Use of Metaprogramming and Domain Specific Languages: An Experience Report in the Logistics Domain. In *Proceedings of 12th Brazilian Symposium on Software Components, Architectures, and Reuse (SBCARS 2018).* ACM, New York, NY, USA, Article 4, 9 pages. https://doi.org/10.475/123_4

SBCARS 2018, September 2018, São Carlos, São Paulo, Brazil

ACM ISBN 123-4567-24-567/08/06.
https://doi.org/10.475/123_4

1 INTRODUCTION

One of the logistics' phases to distribute equipments to an army force contemplates the *distribution planning* of materials throughout military units. This phase is known as *material endowment*, and involves several steps, including the specification of relevant materials, often independent of vendor; the definition of rules that link military materials and equipments to organizational units (e.g., departments, regiments, and military posts); the execution of the rules for deriving the materials originally intended for each *abstract military unit*; and the designation of the materials that must be actually distributed to the *concrete military organizations*, which may suffer variations due to the inclusions and suppressions of positions for a particular organizational unit of an army force. In the context of this paper, the reader might consider the scenario of the Brazilian Army.

Decision support systems are essential in this domain, given the importance and complexity of the processes related to the allocation of military employment materials. In particular, the rules used to predict military materials and equipments are not trivial, and implementing these rules directly in the source code makes the systems difficult to understand and maintain. This problem mostly occurs because the number of conditions that need to be expressed is proportional to the complexity of the organizational structure and the number of materials. In the case of an entire army force, this number is noteworthy. In addition, a "*hard-coding*" solution does not allow domain experts to trivially specify and test new derivation rules. For these reasons, it is preferably to implement these rules declaratively, using languages, tools, or libraries that support some sort of inference mechanism.

Nevertheless, introducing either a logic language (e.g., Prolog or Datalog) or a specific library (such as Drools [**?**]) in a complex enterprise system may lead to some type of *architectural conflict*, particularly in environments whose whole ecosystem system is well established and based on a set of constraints related to both language and libraries usage. Furthermore, these solutions require a reasonable learning curve, which can hamper software maintenance activities.

This article describes the main design and architectural decisions adopted to implement the mechanisms that assist decision makers to plan the distribution of military materials and equipments across the organizational structure of the Brazilian Army. These decisions aim to both (a) abstract the adoption of a business rule engine using **metaprogramming** and (b) allow domain experts to simulate the definition and testing of rules using a **domain specific language**. Accordingly, this paper presents the following contributions:

- A description of the use of generative programming techniques to raise the abstraction level related to the adoption

of a logic programming language in the specification of rules for the distribution of materials and equipments through organizational units.
- A report of an empirical evaluation of our design and architectural decisions, reinforcing that they fulfill not only the needs of the domain specialists, but also existing technical constraints.

Although we discuss the distribution of materials and equipments for the military domain (which is far from trivial), we believe that the architectural decisions discussed here can be exploited to other logistics and endorsements scenarios in which it is also desirable to transparently introduce the support for rule based engines in enterprise systems.

We organized this article as follows. Section 2 presents the concepts related to the domain of material logistics. Section 3 presents the concepts related to business rules and rule based engines and Section 4 presents some background information related to *generative programming*—techniques that we use to support our design and implementation. Section 5 presents the design and architectural decisions we take to develop the mechanism responsible for the distribution of materials. In Section 6 we present the results of an empirical study that validate our approach. Finally, Section 8 presents some final remarks.

2 LOGISTICS SYSTEMS

Logistics is the method used to implement military strategies and tactics as a method to gain competitive advantage [**?**]. Logistics is defined as the art of moving armies, as well as accommodating and supplying the military [**?**]. Strategy and tactics provide the framework for conducting military operations, while logistics provides the means. In the context of this paper, we are more interested in the supplying aspects of logistics. The logistics supplying function refers to the set of activities that deals with the provisioning of the material of different classes (including vehicles, armaments, munition, uniform, and fuel) necessary to the military unities (including a rich organizational structure and a huge number of soldiers).

In order to facilitate the administration and control of materials and equipments, the Brazilian Army uses a catalog system that is similar to one defined by the North Atlantic Treaty Organization (OTAN) [**?**], which classifies items into groups and classes. Instead, to support the provisioning of materials and equipments, the Brazilian Army catalog uses a classification that considers the *type*, *class*, and *family* of the materials, ignoring some details (such as specific brands and suppliers). More detailed information is considered by other areas of the logistic system. Several rules govern the identification of materials and equipments, to avoid duplication and simplify the process of decision making. A logistic system is the integrated set of organizations, personnel, principles, and technical standards intended to provide the adequate flow for supplying materials [**?**]. This is a complex system, and thus must be organized in different modules.

In this paper we present the main design and architectural decisions related to one of the modules (SISDOT) of the *Integrated Logistics System* (SIGELOG) of the Brazilian Army. SISDOT deals with the identification of the necessary materials and equipments (hereafter MEMs) for the organizational units of the Brazilian Army. Each organizational unity is a combination of a set of *generic organizations of specific types*, which describes the expected number of military unities of an organization (including armies, divisions, brigades, and so on). The goal of SISDOT is to estimate the set of MEMs that should be assigned to a generic organization, producing one of its main products (named QDM). Based on a set of QDMs, SISDOT then derives another estimate of MEMs that should be assigned to a concrete organizational unity (a QDMP).

Before generating a QDM, it is necessary to catalog all relevant MEMs to a given class of materials and then specify *high-level rules* (HLRs) that relate MEMs to the generic organizational unities, considering different elements of the structure—from the type of the unity and sub-unities to the qualifications of a soldier. That is, it is possible to elaborate a HLR that states that a military ambulance would be assigned to all *drivers within medical units*. Differently, it is also possible to state that an automatic rifle will be assigned to all *sergeant that has been qualified as a shooter*. There is some degree of flexibility for writing these rules, that might be from two different kinds: rules that consider the entire structure of the Brazilian Army and rules that refer to specific elements of the organizational structure. The former details rules that are more specific for a set of military qualifications. The second allows the generation of rules that might be reused through different unities.

Deriving QDMs and QDMPs is not trivial and its complexity is due to several factors, including the rich organizational structure of the Brazilian Army, the huge number of existing functions and qualifications that must be considered, and the reasonable number of MEMs. The process might also vary depending on the class of the materials. Moreover, it is possible to specify the *high-level rules* in different ways, which in the end might interact to each other leading to undesired results (e.g., the same material being assigned several times to a given soldier).

Due to number of logical decisions, using an imperative language to implement the process of QDM generation is not an interesting choice. However, the SIGELOG architecture constraints the development to use the Java Enterprise Edition platform—so we had little space to introduce a more suitable language (such as Prolog) to generate QDMs. Other constraints guided the architectural decisions of SISDOT, including:

- Maintenance: the set of languages and libraries are limited to those already used in the SIGELOG development.
- Performance: considering a total of 1000 *high-level rules*, the system must be able to generate a QDM in less than 5 seconds.
- Testing: the technical decisions must not compromise the testability of the system, being expected a high degree of automated tests.

To deal with the first constraint, we decided to use a metaprogramming approach that derives *low-level rules* for Drools (a rule based engine for Java) *on the fly*, from a set of *high-level rules* represented as domain objects that must exist in a database. In this way, it is not necessary to understand the language used to specify rules in Drools. Surely, although this approach increases flexibility and allows us to externalize the *low-level rules*, it might introduce some undesired effect on the performance of the system: we generate all low-level rules every time a QDM is built. We are

using a cache mechanism and performance tests (as we discuss in Section 6) shows that our architecture fulfills the second constraint. For the third constraint, we implemented a DSL that simplifies the specification and execution of test cases.

In the next sections (Section 3 and Section 5) we introduce some technical background that might help the reader to understand the design and architectural decisions we take while implementing SISDOT.

3 RULE BASED SYSTEMS

The process for deriving QDMs consists of applying a set of business rules that relates MEMs to the organizational structure of the Brazilian Army. In this paper, we named this set of business rules as *high-level rules*. A business rule is a compact, atomic, and well-formed statement about an aspect of the business that can be expressed in terms that may be directly related to the business and its employees using a simple and unambiguous language that might be accessible to all stakeholders [**?**]. This type of structure is very important for organizations, as they need to deal more and more with complex scenarios. These scenarios consist of a large number of individual decisions, working together to provide a complex assessment of the organization as a whole [**?**].

Rule Based Systems (RBS), also known as production systems or expert systems, are the simplest form of artificial intelligence that help to design solutions that reason about and take actions from business rules (a feature that is interesting in the context of SISDOT), using rules such as knowledge representation [**?**].Instead of representing knowledge in a declarative and static way as a set of things that are true, RBS represents knowledge in terms of a set of rules that tells what to do or what to accomplish in different situations [**?**]. RBS consist of two basic elements: a *knowledge base* to store knowledge and an *inference engine* that implements algorithms to manipulate the knowledge represented in the knowledge base [**? ? ?**]. The knowledge base stores facts and rules [**?**]. a fact is a usually static statement about properties, relations, or propositions [**?**] and should be something relevant to the initial state of the system [**?**]. A rule is a conditional statement that links certain conditions to actions or results [**?**], being able to express policies, preferences, and constraints. Rules can be used to express deductive knowledge, such as logical relationships, and thus support inference, verification, or evaluation tasks [**?**].

A rule "if-then" takes a form like *"if x is A then take a set of actions"*. The conditional part is known as antecedent, premise, condition, or left-hand-side (LHS). The other part is known as consequent, conclusion, action, or right-hand-side (RHS) [**? ?**]. Rule-based systems works as follows [**?**]: it starts with a knowledge base, which contains all appropriate knowledge coded using "if-then" rules, and a working memory, which contain all data, assertions, and information initially known.

The system examines all rule conditions (LHS) and determines a subset of rules whose conditions are satisfied, based on the data available at the working memory. From this information set, some of the rules are triggered. The choice of rules is based on a conflict resolution strategy. When the rule is triggered, all actions specified in its `then` clause (RHS) are executed. These actions can modify the working memory, the rule base itself, or take whatever action that the system programmer chooses to include. This *trigger rule loop* continues until a termination criterion is met. This end criterion can be given by the fact that there are no more rules whose conditions are satisfied or a rule is triggered whose action specifies that the program should be finalized.

Drools is an example of a business logic integration platform (BLiP) written in the Java programming language and that fits some requirements of SISDOT (in particular, its integration with Java and the Wildfly application server). We use Drools as the *rule-based system* in the QDM generation. However, to abstract the use of a RBS, we generate the Drools rules using a meta-programming approach, a specific technique for *generative programming*.

4 GENERATIVE PROGRAMMING

Generative Programming (GP) is related to the design and implementation of software components that can be combined to generate specialized and highly optimized systems, fulfilling specific requirements [**?**]. With this approach, instead of developing solutions from scratch, domain-specific solutions are generated from reusable software components [**?**]. GP is applied on the manufacturing of software products from components in a system in an automated way (and within economic constraints), i.e., using an approach that is similar to that found in industries that have been producing mechanical, electronic, and other goods for decades [**?**]. The main idea in GP is similar to the one that led people to switch from Assembly languages to high-level languages.

In our context, we use GP to deal with two different aspects of SISDOT. First, we raise the level of abstraction by automatically generating low-level Drools rules from an existing set of *high-level rules* (previously defined by the domain experts). This is a type of *meta-programming* based on template engines—we use a template to generate a set of facts and rules (which are the fundamental mechanisms of logic programming) from domain objects. We also raise the level of abstraction to assist developers in the specification and execution of test cases, using a domain specific language and a set of customized libraries and tools.

4.1 Metaprogramming and Template Engines

Metaprogramming concerns to the implementation of programs that generate other programs. In the context of SISDOT, we use a metaprogramming technique to generate programs that are interpreted by Drools. The main goal is to abstract the use of a specific language for implementing the *low-level rules* used to generate QDMs. There are many different approaches for metaprogramming, including preprocessing, source to source transformations, and template engines. A template engine is a generic tool for generating textual output from templates and data files. They can be used in the development of software that requires the automatic generation of code according to specific purposes [**?**].

The need for dynamically generated web pages has led to the development of numerous template engines in an effort to make web application development easier, improve flexibility, reduce maintenance costs, and enable parallel encoding and development in HTML language. In addition to being widely used in dynamic web page generation, templates are used in other tasks, being one of the main techniques to generate source code from high level

specifications. More recently, languages such as Clojure, Ruby, Scala and Haskell adopted the use of template and *quasiquote* mechanisms as a strategy for code reuse and program generation(some of them considering type systems). A template is a text document that combines placeholders and linguistic formulas used to describe something in a specific domain [**?**].

In this work we use the Freemarker template engine, which is an open-source software designed to generate text from templates [**?**]. Freemarker was chosen for several reasons, such as: it follows a general purpose model; it is faster than other template engines; it supports shared variables and model loaders; and its templates can be loaded or reloaded at runtime without the need to (re)compile the application [**? ?**].

4.2 Domain Specific Languages

A language is a set of valid sentences, serving as a mechanism for expressing intentions [**?**]. Creating a new language is a time-consuming task, requires experience, and is usually performed by engineers specialized in languages [**?**]. To implement a language, one must construct an application that reads sentences and reacts appropriately to the phrases and input symbols it discovers. The need for new languages for many growing domains is increasing, as well as the emergence of more sophisticated tools that allow software engineers to define a new language with reasonable effort. As a result, a growing number of DSLs are being developed to increase developer productivity within specific domains [**?**].

Domain Specific Languages (DSL) are specification languages or programming languages with high level of abstraction, simple and concise [**?**], focused on specific domains, and are designed to facilitate the construction of applications, usually declaratively, with limited expressiveness lines of code, which solve these specific problems [**?**]. A DSL is a computer language of limited expressiveness, through appropriate notations and abstractions, focused and generally restricted to a specific problem domain [**? ?**]. DSLs have the potential to reduce the complexity of software development by increasing the level of abstraction for a domain. According to the application domain, different notations (textual, graphical, tabular) are used [**?**].

In this work we used Xtext [**?**] (a language workbench) to implement a DSL that help developers to specify and execute test cases related to the process of QDM generation. The main rationale to building this DSL was the significant costs related to specifying *high-level rules* (particularly during acceptance tests).

5 ARCHITECTURAL DECISIONS

SISDOT is based on the client-server architectural style (according to a typical Java enterprise application), where the components interact to each other by requesting services from other components [**?**]. SISDOT is also organized into logical groups of components (known as tiers)—a strategy often used in the client-server architectural style. The multi-tier architectural pattern can be classified as a *component and connector* pattern, depending on the criterion used to define the tiers [**?**]. Indeed, there are several criteria, such as component type, sharing of the execution environment, and so on [**?**].

The multi-tier pattern also induces some topological constraints that govern how the components might interact with other components. Specifically, connectors might only exist between components of the same tier or that reside in an adjacent tier [**?**]. In addition, according to Bass et al., the organization of a system using tiers supports software maintenance, facilitates the definition of security policies, and might increase the performance, availability, and modifiability of the systems [**?**].

5.1 Runtime Architectural Description

In order to further detail and illustrate the SISDOT architecture, and the main architectural decisions to address the constraints introduced in Section 2, here we present a scenario that refers to the generation of QDMs using a runtime architectural description (see Figure 1).

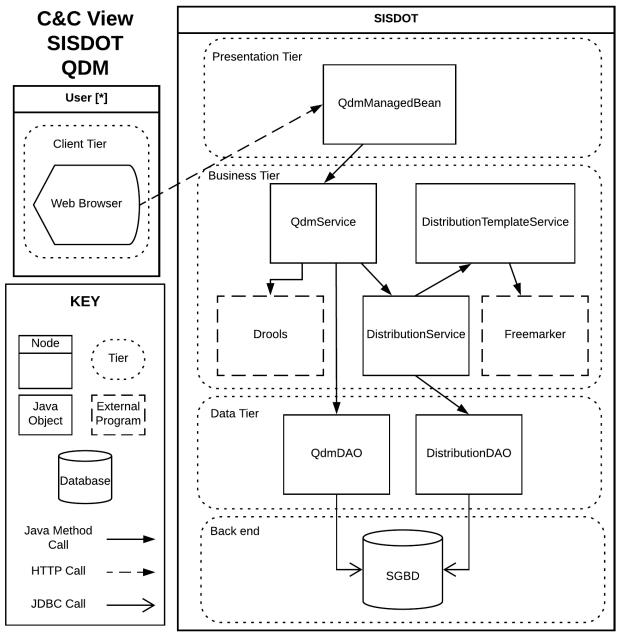

Figure 1: Runtime view of the architecture

Considering the user interaction, a previously authenticated specialist on the distribution of equipments throughout the Brazilian Army, using her web browser, requests the web page that allows the generation of QDMs (one of the main products of SISDOT). After selecting the *generic organizational unity* for which a QDM would be generated, the system sends a request to the server. The client tier is responsible for these actions. The request is then received by the SISDOT presentation tier, more specifically by a *managed bean* that works as a *front controller* (in this case, a JSF controller).

This controller validates the request data, before sending a new request to the business tier. In the business tier, the service that implements the business rules related to the QDM domain object is invoked to start the QDM generation process, which includes the transformation of *high-level rules* into the *low-level rules* specified

using the Drools Rule Language (DRL), the execution of these rules to instantiate a business object that represents a QDM, and finally saving this object in the persistence layer (a relational database).

The process of generating a QDM is started by retrieving a list of previously registered *high-level rules*. An HLR, regardless of its type (either based on the full structure of the organization or based on its components), is related to one or more *military materials / equipments* (MEMs) and defines the respective amount that should be assigned to a military unit (which ranges from an organization, a center, a department, a brigade, or even a military function or qualification of a soldier).

A low-level rule has one or more distribution rules. Each rule has a type, a value, and an associated description. For example, a rule for a department type has the value of the department identifier and the description of the department name. For a better use of the database, avoiding the definition of numerous columns that would inevitably be null for several rule types, we decided to persist the distribution rules of an HLR using JSON (JavaScript Object Notation), which is converted back to an object when it is retrieved. The set of these rules defines exactly who should receive the MEMs specified by the HLR.

One of the factors that motivated the use of a rule based engine was the similarity between *high-level rules* and the "if-then" rules of these mechanisms. An HLR has a set of conditions that fit the conditional part of a rule ("if") and has a set of actions ("then") that, in this case, trigger the distribution of MEMs throughout the expected military unities.

The list of retrieved *high-level rules* contains only Java objects. In order to be able to use the rule engine in a transparent way for the end users (and also for developers in future maintenance scenarios), we decided to use a meta-programming approach, translating these objects into a set of rules (i.e., a program in logic programming) that might be used by a rule-based engine. As mentioned before, we decided to use Drools to support a rule-based engine in the context of SISDOT (Figure 1), mostly because of its integration capabilities with both JEE systems and the Wildfly application server. Accordingly, the *high-level rules* are translated into DRL rules. To this end, we use a template engine (Freemaker) to implement this particular transformation (Figure 1), using a template that contains the required *markups* for transforming HLR into *low-level* Drools rules. That is, for each HLR, the template engine populates the template, resulting in a DRL rule consistent with the converted HLR. A simplified version of the template used for DRL generation can be seen in Listing 1.

A QDM is generated for a generic organizational unity (hereafter QC, from *Quadro de Cargos* in Portuguese), selected by the user on the client tier. Therefore, the user-specified QC must be retrieved from the Brazilian Army's enterprise database through a specific Data Access Object (DAO) of the data tier [**?**]. From the list of DRL rules, created based on existing *high-level rules*, the rule engine is instantiated and these rules are compiled, so that they can be triggered by Drools. The recovered QC is inserted into the working memory as a fact, an information that is considered true. The rules are only executed when their conditions are satisfied, based on the specified QC organizational structure. When a rule is valid, that is, it has been activated and the *Left Hand Side* (LHS) conditions are valid, the QDM is populated with the MEMs specified in the *Right Hand Side* (RHS) of the rules. Therefore, we generate a complete QDM object after verifying all valid rules (previously registered in the database) for the selected QC.

```
1  <#if rule.omType == "BOTH">
2  <#assign operational="(operational==OPERATIONAL || operacional==
       ↪ NOT_OPERATIONAL)">
3  <#else>
4  <#assign operational="operational==${rule.omType}">
5  </#if>
6  rule "Chamador ${rule.codigo}"
7  when
8  $qc: QuadroDeCargosVO(
9  ${operational}
10 <#if map['NATURE']??> , ${map['NATURE']} </#if>
11 <#if map['SUB_NATURE']??>
12 <#if map['NATURE']??>||<#else>,</#if> ${map['SUB_NATURE']}
13 </#if>
14 <#if map['VALUE']?? > , ${map['VALUE']} </#if>
15 )
16 $roles: List( size > 0 ) from accumulate (
17 $dep:DepartmentVO(<#if map['DEPARTMENT']??>${map['DEPARTMENT']}</#
       ↪ if>) from $qc.deps
18 and
19 $role: RoleVO(
20 function == "ROLE"
21 <#if map['ROLE_QC']?? > , ${map['ROLE_QC']} </#if>
22 <#if map['ROLE']?? > , ${map['ROLE']} </#if>
23 <#if map['SUB_CIRCLE']?? > , ${map['SUB_CIRCLE']} </#if>
24 <#if map['RANK']?? > , ${map['RANK']} </#if>
25 <#if map['ARMS']?? > , ${map['ARMS']} </#if>
26 ) from $dep.roles <#if !map['QUALIFICATION']?? && !map['
       ↪ OBSERVATION']?? >;</#if>
27 <#if map['QUALIFICATION']?? >
28 and QualificationVO(code ${map['QUALIFICATION']}) from $role.
       ↪ qualifications
29 <#if !map['OBSERVATION']?? >;</#if>
30 </#if>
31 <#if map['OBSERVATION']?? >
32 and ObservationVO(code ${map['OBSERVATION']}) from $role.
       ↪ observations;
33 </#if>
34 collectList( $role )
35 )
36 then
37 for(int i=0; i < $roles.size(); i++){
38 RoleVO c = (RoleVO) $roles.get(i);
39 helper.add(${rule.id?string.computer}L, c.getFractionId(), c.
       ↪ getRoleId());
40 }
41 end
```

Listing 1: *Freemarker template for generating DRL*

After the QDM generation, it is saved on the database and we let the domain expert know of the success of the operation using a simple user interface message. The domain expert can then perform the necessary operations on the generated QDM, including a workflow involving its edition, homologation, and validation.

5.2 A DSL for testing the generation of QDMs

As explained before, to generate a QDM, a set of *high-level rules* must be previously declared. This is a time-consuming task, particularly when using the interface of the system. In order to facilitate the definition of the *high-level rules* used in the automated test scripts, we decided to create a DSL (Domain Specific Language), enabling us to specify *high-level rules* in a clear, objective, and declarative way.

We implemented our DSL using Xtext, which generates plugins that allow editing code in both Eclipse and IntelliJ, plus an editor that can be embedded in a web application. This way, a developer writing test scripts use our DSL to take advantages of the functionality provided by these plugins.To implement a DSL with Xtext it is necessary to declare a grammar similar to ANTLR [**?**], and can be seen in Listing 2. From the specified grammar Xtext will generate

the lexer, the parser, the AST (Abstract Syntax Tree) model, the construction of the AST to represent the parsed program, and the editor with all the IDE features. Xtext comes with good and smart default implementations for all these aspects. However, every single aspect can be customized by the language designer. [?].

```
grammar br.unb.cic.sisdot.chamador.ChamadorDsl
with org.eclipse.xtext.common.Terminals

generate chamadorDsl "http://www.unb.br/cic/sisdot/ChamadorDsl"

Model:
rules+=Rule*;

Rule:
'rule' code=INT '{'
('type' '=' type=OmType ',')?
'materials' '=' '[' items+=Item
(',' items+=Item)*
'],'
'conditions' '=' '[' conditions+=Condition
(',' conditions+=Condition)*
']'
'}';

Item:
'(' codot=STRING (':' name=STRING)? ':' amount=INT ')';

Condition:
'(' ('cond' '=' cond=ConditionType ',')?
'type' '=' type=RuleType ','
'values' '=' '[' values+=STRING (',' values+=STRING)* ']'
')';

enum OmType:
BOTH | OPERATIONAL | NOT_OPERATIONAL;

enum ConditionType:
AFF | NEG;

enum RuleType:
NATURE | SUB_NATURE | VALUE | DEPARTMENT | ARMS | ROLE_QC |
    ↪ SUB_CIRCLE | RANK | ROLE | QUALIFICATION | OBSERVATION ;
```

Listing 2: *Xtext grammar defining the DSL structure*

Listing 3 presents a simple example of a HLR declaration using our DSL. The goal of this rule is to distribute the materials (defined in the list of materials construct) for *operational* OMs (see the type construct) and militaries *not working in the specified list of departments* with a set of qualifications and roles. While the definition of this rule using our DSL requires 12 lines of code, the corresponding definition of the same rule using a Java test case requires more than 50 lines of imperative code.

```
rule 1 {
 type = OPERATIONAL,
 materials = [
   ("XXX" : "Rifle" : 1),
   ("XXY" : "Rifle Carrying Case" : 2)
 ],
 conditions = [
  (cond=NEG, type=DEPARTMENT, values=[...]),
  (type=QUALIFICATION, values=[...]),
  (type=ROLE, values=[...])
 ]
}
```

Listing 3: *Example of a HLR declaration using our DSL*

A test activity consists first in the specification of the required *high-level rules* in a file. Next, the *behavior under test* is detailed as features using the Cucumber [?] framework (we present an example of a feature in Figure 2). After that, the developer details the implementation steps of the features by (a) specifying which *high-level rules* (previously declared using our DSL) should be considered in the test execution, (b) specifying which QC the QDM should be generated, and (c) specifying the expected results in terms of the properties of the expected QDM.

The structure created supports several operations, with the intention of increasing the productivity of the tester, such as: database connectivity, auxiliary methods for executing queries in the database, and rules in the rule engine. To facilitate the execution even in continuous integration environments, we use a maven configuration file that declares the Xtext plugin, responsible for generating the necessary code from the DSL, and a plugin for running unit tests (such as surefire).

Altogether, our DSL simplifies the process of specifying *high-level rules*. Its primary goal was to assist in test activities, though we presented some examples of *high-level rules* specified using our DSL to domain experts, and they also considered this domain specific language relevant to simulate the specification of rules in order to better understand the effect of each HLR for building QDMs. Now we are working on a custom user interface for SISDOT to enable domain expert to use this specific feature.

A DRL gerada para o chamador da Listing 1 pode ser vista na Listing 4. Na parte RHS da regra, hÃą uma iteraÃğÃčo sobre todos os cargos selecionados. Para cada cargo uma classe que auxilia o preenchimento do QDM Ãĺ chamada para que os itens declarados no chamador, passado como parÃćmetro, sejam disponibilizados para o cargo. Essa classe auxiliar recupera o chamador passado como parÃćmetro a partir da lista de chamadores declarados na DSL, que foram convertidos em objetos Java. AlÃĺm de auxiliar no preenchimento do QDM, esta classe popula alguns objetos que facilitam a verificaÃğÃčo da consistÃłncia do QDM, por exemplo.

```
rule "Chamador 1"
when
  $qc: QuadroDeCargosVO(operational == OPERATIONAL)
  $roles: List( size > 0 ) from accumulate (
    $dep: DepartmentVO(depId not in (14086, 1130)) from $qc.deps
    and
    $role: RoleVO(
      function == "ROLE"
      , roleId in (2, 12, 15, 21)
    ) from $dep.roles
    and QualificationVO(code in ("782", "756")) from $role.
        ↪ qualifications;
    collectList( $role )
  )
then
  for(int i=0; i < $roles.size(); i++){
    RoleVO c = (RoleVO) $roles.get(i);
    helper.add(1L, c.getFractionId(), c.getRoleId());
}
end
```

Listing 4: *DRL generated*

We are also working on the automatic generation of test cases from our DSL. This effort involves a custom Maven plugin that currently generates simple test cases, freeing the tester to create only complex tests related to the QDM generation. These test cases are generated from a combination of conditions, for example: one test case only for operational OMs and another for non-operational OMs; a test case for each OM nature, combining with the operational type, such as: infantry operational OMs and non-operational infantry OMs. These combinations are generated for simple cases

so that they are easy to verify without the need to build a Java solution for generating QDMs.

```
<plugin>
  <groupId>xxx</groupId>
  <artifactId>xxx</artifactId>
  <version>${project.version}</version>
  <executions>
    <execution>
      <id>execute</id>
      <configuration>
        <seed>10</seed>
        <listaQcsString>702311,762314,575311</listaQcsString>
      </configuration>
      <goals>
        <goal>generate</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

Listing 5: *Maven plugin for test generation*

```
Feature: QDM 757311
  As a user
  I want to generate a QDM from HLRs

  Scenario: Test 1
    Given QC with code 757311
    And HLR: 1, 2
    When I generate a QDM
    And the results are consistent
    Then positions must be populated:
    | idFrac   | idPos   | codot      | amount     |
    | 6765748  | 10403   | 1051000025 | 1          |
    | 6765748  | 10403   | 1020100017 | 1          |
    | 6765748  | 10403   | 1020100013 | 1          |
    | 6765748  | 10403   | 1051000006 | 1          |
    | 6765748  | 21328   | 1051000025 | 1          |
    | 6765750  | 10122   | 1051000025 | 1          |
    | 6765793  | 13276   | 1051000025 | 6          |
    | 6765796  | 24655   | 1051000025 | 1          |
    | 6765796  | 24429   | 1051000025 | 2          |
    | 6765798  | 24429   | 1051000025 | 3          |
    | 6765798  | 11433   | 1051000025 | 9          |
    And the final result must be:
    | codot      | amount     |
    | 1020100013 | 74         |
    | 1020100017 | 74         |
    | 1051000006 | 74         |
    | 1051000025 | 74         |

  Scenario: Test 2
    Given QC with code 757311
    And HLR: 1, 2, 3
    When I generate a QDM
    And the results are inconsistent
```

Figure 2: *Cucumber feature*

For each of the possible combinations, we generate a HLR that meets the conditions and a cucumber feature that exercises the HLR and compares the expected results declared in the feature. Each HLR is considered during the generation of the QDM to the QCs specified in the maven plugin. That is, to use this architecture characteristic (here we consider testability as an architecture concern), we "feed" the maven plugin with a list of QC codes for which the QDMs should be generated, besides other optional information, which have default values, such as: connection string for the database, seed to be used in the random selection of values within a combination (e.g., considering "20" natures of organizational units, randomly take 5 for generating the set of combinations), and the output directory where the code should be exported. A basic example of the plugin declaration, omitting some details like `groupId` and `artifactId`, can be seen in Listing 5.

After running the plugin, with the generated code, the tests are executed in a similar way to the test definitions detailed before. The tests run along with the other unit tests declared, whether DSL-based or not.

6 CASE STUDY

For the validation of the proposed architecture we conducted a case study, which consists of the evaluation of the main architectural decisions of SISDOT, which relate to the generation of QDMs. The case study aims to answer the following research questions:

- **RQ.1**: Does our proposed approach, based on meta-programming, generate correct QDMs?
- **RQ.2**: Does the use of a DSL reduce the effort necessary to implement the test cases that target QDM generation?
- **RQ.3**: Does our proposed approach, based on meta-programming, generate QDMs within a satisfactory time-frame?

RQ.1 deals with the most important concern: correctness regarding the QDM generation. It is intended to demonstrate that the solution not only automatically generates the rules, eliminating the work that the developer would have to implement the solution in Java, but also that the proposed approach leads to the expected QDMs when considering a well defined set of *high-level rules*. **RQ.2** is related to the productivity of the tester and demonstrates how much code the developer have to write when using our DSL, compared to the effort to specify *high-level rules* using the Java language in unit test cases. Although there is no formal functional requirement specifying the acceptable time for generating a QDM, it has been determined that a time greater than 5 seconds, when using 1000 *high-level rules*, would be unacceptable. Thus, **RQ.3** serves to ensure that the QDMs are generated in a satisfactory time.

6.1 Data Collection and Analysis

We answer our first question qualitatively. Although this might look like disappointing (under the perspective of empirical methods), we have strong evidences collected from the domain experts that our approach has produced correct QDMs. We believe that this correctness is due to several factors, including the involvement of the domain experts that helped us to full understand the requirements and the expertise of our development team. Nevertheless, the decision of not writing the low-level rules at the source code level, but instead using a program generation approach that has reduced significantly the effort needed to *hand write* those rules, has also contributed to achieve this confidence about correctness.

In a meeting with the domain experts, one of the specialists said: "I got surprised with the correctness of the solution. In the previous version of the system, we had to have several interactions until we got confident about the correct operation of the *high-level rules*". The first time that we generate QDMs, the process was validated. It is important to note that "not all are flowers", and we broke the initial implementation after the introduction of a new, at first not so related feature that, in the end, cut across several classes of the system. We fixed the related issues recently.

To answer the other two research questions, we collected the following metrics: lines of code (LOC) and execution time. The choice of these two metrics is related to the size of the code, which may indicate higher productivity; and to the fulfillment of the functional requirement that determines the maximum execution time for the generation of QDMs.

Table 1: Comparing the number of lines of code need to specify test cases using our DSL and Java

HLR Code	Lines of Code (DSL)	Lines of Code (Java)
1	10	26
2	10	25
3	10	32
4	11	52
5	15	41
6	9	16
7	18	21
8	12	25
15	7	45
58	7	58
59	17	34

We use as a benchmark 66 *high-level rules* declared in its database. These *high-level rules* correspond to real rules used by the Brazilian Army to perform some tests related to the generation of QDMs. We converted each HLR written using JUnit to their correspondent one specified using our DSL. Table 1 presents a sample of the collected data. The first column contains the HLR code, the second column contains the number of lines of code to represent the HLR using our DSL, and the third column contains the number of lines to represent the same rule in the Java language.

Regarding the runtime metrics, we collected the data using an Intel(R) CoreTM i7-4790 processor, with 16GB of RAM, running Ubuntu 16.04.4 LTS 64-bit operating system, with 4.15.0-24-generic kernel. We used 16 QCs, chosen to represent the various OM types, natures, and subnatures. For each QC, we generated QDMs using random sets of *high-level rules* with size: 20, 40, 50, 60, 100, 200, 400, 700, 1000, 1500, 2000, 3000, 5000. Since there were only 66 real *high-level rules* already declared, larger quantities contain repeated callers (which in the end will duplicate the data within a given QDM). The selection of callers was performed at random, but with the same seed, so that the same sets were used in the generation of QDMs for different QCs. Table 2 shows a sample of the results obtained with the execution. The first column contains the QC code, and the second column contains the time in milliseconds for the generation of the QDMs using the indicated amount of *high-level rules* on the respective column header.

We exported the collected data to the CSV format, so that we could perform a data analysis using the R environment [**?**]. In the data analysis for **RQ.2**, which compares the number of lines of code between the two distinct approaches for representing *high-level rules*, we created two additional columns: *difference*, which measures the difference in the number of lines of code necessary to specify *high-level rules* using Java and our DSL; and *percent of reduction*, which indicates how much less code is necessary to specify a HLR using our DSL, when compared to the direct specification in Java code.

Table 2: Time (ms) for generation of QDMs

QC	60	100	400	700	1000	2000
206410	184	275	944	1829	2279	4707
231303	197	282	975	1731	2541	4987
500311	201	291	948	1804	2289	4950
618322	175	264	922	1640	2412	4990
727310	173	302	937	1652	2466	4992
1450311	182	269	890	1649	2527	4785
2203310	197	263	909	1619	2344	4976
5406194	176	253	884	1598	2393	4610
7020003	212	241	922	1610	2361	4470
9131000	171	252	888	1609	2345	4471

Table 3: Analysis of DSL and Java comparison data

	Min.	Median	Mean	SD	Max.
DSL	7.00	9.00	9.67	2.81	18.00
Java	9.00	17.50	21.76	14.68	96.00
Difference	2.00	7.00	12.09	13.94	83.00
%	14.29	43.75	45.23	19.84	87.93

Table 3 presents some descriptive statistics of the collected data, containing the following columns: minimum value, median, mean, standard deviation, and maximum value. There is a strong correlation (0.982) between the lines of code in Java and the *difference* measurements. This correlation can be seen in Figure 3. It is possible to identify that as the size of the Java code grows, the difference for the representation in DSL also increases.

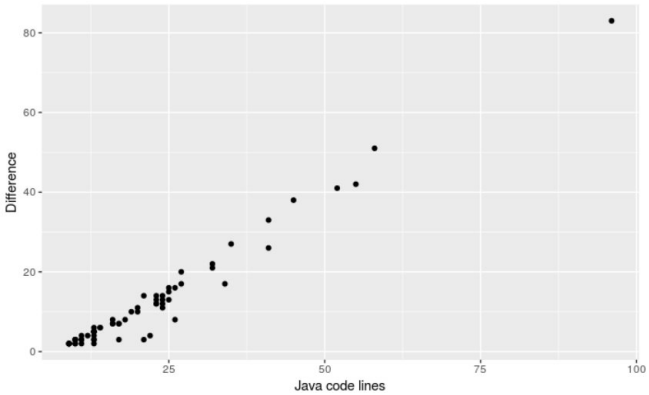

Figure 3: *Correlation between Java and Difference*

Figure 4 shows the time necessary to generate QDMs for 5 different QCs. For each of them, 6 QMS were generated, with different amounts of *high-level rules*, 60, 100, 400, 700, 1000 and 2000.

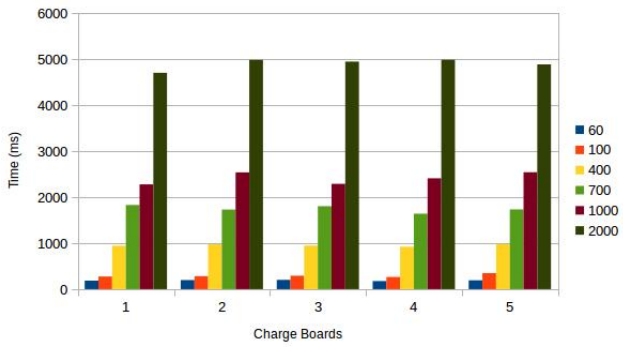

Figure 4: *QDM Generation Time (ms)*

6.2 Discussion

When analyzing the data related to RQ.2, it was possible to observe that the representation of a HLR using our DSL is on average 45% smaller than the representation of the same rule in Java. This indicates a possible productivity gain for the developer when writing test cases, since her will have to write a smaller amount of code. In addition, using our DSL, we specify *high-level rules* using a declarative approach, which is close to the vocabulary of the problem domain. With respect to the third research question, the limit imposed by the functional requirement of 5 seconds was not exceeded, even when considering 2000 *high-level rules*, which is twice the number of *high-level rules* expected for the system in production.

6.3 Threats to validity

We only consider 66 real *high-level rules* in our study, which served as a basis for measuring the generation time of QDMs. SISDOT is expected to have around 1000 *high-level rules* in the production environment. To mitigate the threat related to the small number of available *high-level rules* for testing, we consider their repetition when carrying out the performance test of the application. In this way, we expect that the system would present a behavior similar to the production environment (with respect to performance). The same situation occurs with the comparison of lines of code between the declarations of *high-level rules* in DSL and Java. As the number of callers is relatively small, it may not be possible to generalize the results we found.

7 RELATED WORKS

Nesta seÃğÃčo serÃčo mostrados alguns dos trabalhos relacionados encontrados na literatura, embora nÃčo tenhamos encontrado nenhum com alto grau de similaridade.

8 FINAL REMARKS

In this paper we detailed the main design and architectural decisions we use to implement the mechanisms necessary to support the distribution of materials through the Braziliar Army unities. These decisions included the use of a metaprogamming approach to derive high level business rules in rules specific to a rule based system (Drools) and the definition of a DSL that facilitates the declaration of *high-level rules* to build test scripts.

We carried out an empirical study to validate the proposed architecture, based on the evaluation of one of the main scenarios of our logistic system (SISDOT), related to the generation of QDMs. From the results obtained, it was possible to observe a reduction in the average number of lines of code required for the declaration of a *high-level rules* using our DSL, in comparison with the representation of the same HLR in Java. Another result of the case study shows that our approach fulfills a non functional requirement that constraints the expected time to generate QDMs. This limit was not exceeded even when using twice the number of *high-level rules* expected for the system in production environment.

Although the architecture discussed here considers the specific needs of the Brazilian Army, we believe that logistic systems from other institutions might benefit from our technical decisions as well.