Automatic Source Code Generation for Web-based Process-oriented Information Systems

Jean Pierre Alfonso Hoyos and Felipe Restrepo-Calle
Department of Systems and Industrial Engineering, Universidad Nacional de Colombia, Bogotá, Colombia

Keywords: Fast Prototyping, Requirements, Natural Language Specification, Compilers, Software Construction Automation, Restricted Natural Language, Web Applications, BPMN.

Abstract: Software development life cycle (SDLC) activities include: requirements analysis, design, development, testing and maintenance. The first two steps have wide impact in the project success, and errors in these stages can have large impact in the project duration and budget. To mitigate these problems, strategies like fast prototyping using natural language to specify software requirements have been proposed. These approaches can make the SDLC faster. In this context, this paper presents an approach to automatically generate a web application prototype running business processes using a restricted natural language specification. A comprehensive case study is presented to validate the proposal and demonstrate its applicability.

1 INTRODUCTION

Requirements elicitation tasks can have errors due to incompleteness or ambiguities in the requirements (Walia and Carver, 2009). The designer's lack of knowledge about the customer business operation and limited communication with the stakeholders can also introduce errors in software (Walia and Carver, 2009; Fairley, 2009). Errors in this early phase can have large impact in the project duration and budget (Walia and Carver, 2009).

To mitigate some of these problems, various strategies like fast prototyping and agile software development methodologies have been proposed (Augustine and Martin, 2005). However, these agile methodologies are more suitable for small teams and small-scale value-oriented software projects, and left behind big scale or mission critical software (Githens, 2006; Fowler and Highsmith, 2001). Moreover, fast prototyping approaches attempt to make SDLC faster, less sensible to human errors and less sensible to natural language inherent ambiguities and specification incompleteness. Therefore, they can reduce production costs, time to market delivery and prototyping costs (Harris, 2012). Some examples of this fast prototyping tools are: Bizzagi[1], which uses graphical interactions and specification languages to create an information system; Justinmind[2] that creates interactive UI prototypes of mobile applications and web pages; OpenXava[3] that creates a web application form a set of java classes. In addition, other approaches use a specification written in natural language (restricted or unrestricted) to automate the construction of models from texts (Ibrahim and Ahmad, 2010; Bhatia et al., 2013). These approaches permit a fast validation of the software and faster feedback from stakeholders (Abbott, 1983; Nishida et al., 1991).

In this context, this paper presents an approach to generate a web application prototype running business processes automatically using a restricted natural language specification. This new language is pretended to be used by a designer in a live meeting with the stakeholders, showing the results of the prototype as a mean to validate the requirements of the software in a fast and interactive way. To achieve this goal, a restricted natural language is proposed. It is based on two different transformations between well-known design models (E-R diagrams and BPMN models) and natural language (English). In addition, the expressiveness of the proposed language permits to describe tasks to generate functional prototypes.

The main contributions of this paper are:

- A specification language for software requirements: *restricted natural language*.
- A fast prototyping method for web applications running business processes using a *restricted nat-*

[1]https://www.bizagi.com/
[2]https://www.justinmind.com/
[3]http://openxava.org

Hoyos, J. and Restrepo-Calle, F.
Automatic Source Code Generation for Web-based Process-oriented Information Systems.
DOI: 10.5220/0006333901030113
In *Proceedings of the 12th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2017)*, pages 103-113
ISBN: 978-989-758-250-9

ural language specification.

This paper is structured as follows. The second section describes the related works. The third section presents our proposal. The fourth section shows the implementation details. Later, the fifth section explores the results of a comprehensive case study. Finally, the sixth section summarizes some concluding remarks and suggests the future works.

2 RELATED WORKS

Many work has been done on the software model extraction from texts. Efforts have been focused on: Design attributes mining from texts written in natural language, the use of templates to write requirements, and specification languages inspired by natural languages.

In the first case, the reviewed proposals are aimed to extracting design information from textual specifications written in unrestricted natural language. (Abbott, 1983) and (Saeki et al., 1989) originally proposed manual procedures back in the 80's. Words in the description matching certain parts of speech and phrase structures were used to build a model. This procedure latter was extended and automatized using natural language processing tools. Some general steps of this method include: Stemming, POS tagging, Sentence splitting, Resolve references and anaphoras, use ontologies to determine implicit relations, use wordnet to find synonyms and not explicit stated relations, rule based design information extraction, and finally a production of a model (or a set of models). Some works in this field include: (Bellegarda and Monz, 2015; Cambria and White, 2014; Chioac, 2012; Overmyer et al., 2001). In this way, (Desai et al., 2016) recently proposed a transformation framework between natural language to domain specific languages (DSLs) using machine learning techniques to generate a set of possible programs.

This approach facilitates to any person the description of the desired functionality of software. Even with a few writing errors, information still can be used (Ibrahim and Ahmad, 2010). In the downside, it is necessary some predefined knowledge (Ontologies, Wordnet) in order to properly extract relations between terms (Popescu et al., 2008). Sometimes a training data set (Bellegarda and Monz, 2015) and/or a set of ontologies (Zhou, X; Zhou, 2008) might be needed to achieve acceptable results. Moreover, current NLP tools can be imprecise leading to induce misinterpretations of the specification (Bellegarda and Monz, 2015). In addition, this method is the slowest of the three approaches due to its high processing requirements (Bellegarda and Monz, 2015).

Secondly, many have tried to restrict the way requirements are written using templates. In this way, key design information can be easily extracted from requirements lists to generate a model or a software prototype (Smith et al., 2003; Konrad and Cheng, 2005; Zapata, 2006; Videira et al., 2006; Ilić, 2007; Zeaaraoui et al., 2013; Dahhane et al., 2015).

Although using templates to write the specification of the software is more difficult, when done, is as easy to understand by the stakeholders as in the first approach (Selway et al., 2015). The use of templates facilitates processing tasks, making this approach faster than the first one. However, its main disadvantage is that designers must be very careful writing requirements or some information may be lost. It is necessary to define each template in order to extend the proposed solution, and requirements not matching any template will be discarded leaving information outside of the final model (Granacki and Parker, 1987).

Lastly, restricting the grammar of a natural language such as English, can lead to overcome the ambiguity and imprecision of such languages. For instance, Attempto (Schwitter, 1996) is an executable specification language in which the English grammar is restricted. In this way, writing requirements is very similar to work with a programming language, which avoids ambiguities and facilitates the processing effort (Bryant and Lee, 2002). Nonetheless, having to learn an artificial language is an inconvenient for the designers and the stakeholders (Schwitter, 1996). This approach is slower generating code than the second but faster than the first one.

This paper proposes the use of templates within the context of an specification language, this is, a hybrid between the second and the third approaches explained before.

3 FAST PROTOTYPING OF WEB APPLICATIONS RUNNING BUSINESS PROCESSES

This work proposes a fast prototyping method for web applications running business processes using a restricted natural language specification. To do so, we use the model-view pattern, implementing a set of business processes operating over a set of domain classes, using only the restricted specification.

Fig. 1 presents a general schema of the proposed workflow. Firstly, during a live meeting with stake-

holders, a designer writes down a software specification using the restricted English grammar, which is proposed in this paper (restricted natural language). Although the specification in restricted natural language could be understood by a non-technical user, the target user of the proposed method is a designer with technical background. This is needed because the relations and concepts expressed with the restricted language are not easy to understand to a non-technical stakeholder. Secondly, our code generation engine produces the source code of the web application. Finally, the stakeholders are able to see and validate the generated application prototype. If a modification is required then the specification is changed, the code is generated and validated again in an iterative way. In this way, it is possible to obtain direct feedback from the stakeholders and achieve fast validation of the software.

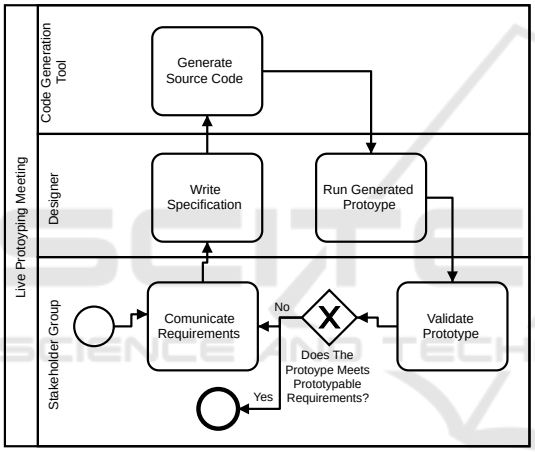

Figure 1: General workflow proposed for fast prototyping of web applications.

3.1 Restricted Natural Language

To work with a restricted natural language to generate web application prototypes without human intervention, we focus only on the English constructs of interest for this task. To do so, we first consider two different transformations from commonly used software design models, i.e. E-R diagrams and BPMN models, to a restricted natural language. Next, we build a language capable of defining the operations and combine the previous two specifications.

BPMN is a graphical specification language designed to show the flow of activities, decisions and events that occur in an organization in order to generate some form of business value.

Let b be a well constructed BPMN model, and let $f(b)$ a function that maps b to source code (s_1); now let t_1 be a restricted natural language specification such that $g(t_1) = b$, where g is a function transforming the input text t_1 into a BPMN model (Equations 1 and 2). The proposed functions can be seen as a generalization of several works such as (Friedrich et al., 2011; Steen et al., 2010).

$$f(b) = s_1 \tag{1}$$

$$g(t_1) = b \tag{2}$$

Moreover, E-R diagrams are graphical specifications that model the relationships between information entities within a system and the information contents of each one of these entities.

Similarly, let e be a well constructed E-R model, $h(e)$ a function that generates source code (s_2); and let t_2 be a restricted natural language specification such that $j(t_2) = e$, where j is a function transforming t_2 into an E-R model (Equations 3 and 4). These functions can be seen as a generalization of works like (Geetha and Mala, 2013; Meziane and Vadera, 2004; Geetha and Anandha Mala, 2014).

$$h(e) = s_2 \tag{3}$$

$$j(t_2) = e \tag{4}$$

In this way, evaluation of functions $f(g(t_1))$ and $h(j(t_2))$ will return source code from the restricted natural language specification. Note that t_1 and t_2 could also be unrestricted specifications, and the previous reasoning still applies. However, for the purposes of this paper, we will restrict the natural language constructs to avoid ambiguities and different context-dependent interpretations.

Suppose that a pair of functions $g^{-1}(b)$ and $j^{-1}(e)$ exists ($e \in E - R$, $b \in BPMN$), meaning that there is a projection of the design models into natural language (Equations 5 and 6). The construction of these functions can be viewed as a "design" process starting from the model space to the natural language space.

$$g^{-1}(b) = t_1 \tag{5}$$

$$j^{-1}(e) = t_2 \tag{6}$$

These projections are not unique, but some subset of natural language can be selected in a way that is not ambiguous and can be used to transform both models to a single common representation: restricted natural language.

Having both models in the same space, it is possible to combine them in a way that exceeds the code generation capabilities of both models separately. Moreover, in this case we will introduce a set of task definition non terminals and integrate those with the previously defined specifications. In addition, with no ambiguity restrictions, a traditional

```
terminal FID :
    '^'?('A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'_'|'0'..'9')*;
Identifier : parts+=FID (parts+=FID)*;
DefinitionItem: arity=('a'|'an'|'many') specName=Identifier
    'named' intanceName=Identifier 'with'
    defaultName=Identifier 'as' 'default';
Definition: childname=Identifier 'is' 'an' specName=Identifier
    ('which' 'exist' 'in' contextName=Identifier)?
    'and' 'needs' ':' definitions+=DefinitionItem
    (',' definitions+=DefinitionItem)* '.';
```

Figure 2: Grammar for the restricted natural language representing E-R diagrams.

parser can be used to process the natural language specification.

It is worth mentioning that unrestricted natural language is not used because the process described in section 2 will induce errors or generate more than one system (Desai et al., 2016), making hard to validate every possible system.

3.1.1 From E-R Diagrams to Restricted Natural Language ($j^{-1}(e)$)

To achieve a textual representation of an E-R model, we propose to restrict the natural language constructs as follows.

To define an entity an entity name (*childname*) is required. Also a set of relations and a set of properties (*DefinitionItem*) are defined. Note that the *entity name* can be in singular and plural forms, thus needing a mechanism to identify them both as the same noun. Fig. 2 shows the excerpt of the proposed grammar for the domain classes. It can be used to generate a class diagram including properties, and relationships.

Furthermore, notice that some defaults should be configured including both data types and their respective memory size. For instance, properties having a VARCHAR data type are assumed to be maximum of 255 in length.

3.1.2 From BPMN to Restricted Natural Language ($g^{-1}(b)$)

In order allow our restricted natural language to represent business processes, we propose to use the following BPMN textual representations. Note that these representations include the most common BPMN constructs. Figure 3 shows the mappings from a BPMN element to BNF rule written in Xtext[4] format.

Note that for the gateway and frontier event elements the rule is spitted in two parts: one defined within task list rule and another defined in a separated paragraph. This allows the specification to take more than a paragraph. Finally note that returning to a previously defined task is done with a special rule "go back to".

```
//------Process and Start Event------
Tasklist: taskList+=Asignedtask ((',' taskList+=Asignedtask)*
'and,' taskList+=Asignedtask)? '.';
Process: 'the' name=Identifier 'process' 'starts' ',' 'then'
':' taskList=Tasklist;
//------End Event------
EndEvent: 'the' 'process' name='ends';
//------Tasks------
SimpleTask: name=Identifier;
GotoJump:'go' 'back' 'to' returnTo=Identifier;
Task:task=(SimpleTask|AfterEvent|WaitSignalEvent|
WaitMessageEvent|GotoJump|EndEvent|SubprocessendEvent|
SubprocessCall|JumpToAsk|QuestionRedirect|EventRedirect|
ParallelRedirect);
//------Exclusive Gateways------
QuestionRedirect: 'ask' 'if' redirectTo=Identifier;
QuestionControl : 'if' 'the' 'answer' 'to'
questionName=Identifier 'is' answer=FID 'then' ':'
taskList=Tasklist ;
//------Parallel Gateways------
ParallelRedirect: 'do' 'at' 'the' 'same' 'time'
'the' redirectTo=Identifier 'tasks';
ParallelControl: 'for' redirectName=Identifier 'tasks'
('also')? 'do':' taskList=Tasklist;
//------Event Gateways------
EventRedirect: 'check' 'the' redirectTo=Identifier 'event';
EventControl : 'if' 'the' 'event' eventName=Identifier 'is'
eventCatched=Identifier ('signal'|'message'|'timer')
'then' ':' taskList=Tasklist ;
//------Subprocess------
SubprocessCall:'the' subprocessName=Identifier 'is' 'made';
Subprocess: 'the' name=Identifier 'subprocess' 'starts' ','
'then' ':' taskList=Tasklist;
//------Events------
WaitMessageEvent:'wait' 'for' 'the' messageName=Identifier
'message';
AfterEvent:'wait' duration=INT
unit=('week'|'weeks'|'hour'|'hours'|'minutes'|
'minute'|'days'|'day');
WaitSignalEvent:'wait' 'for' 'the' signalName=Identifier
'signal';
//------Lanes------
Asignedtask: task=Task ('(by' asignee=Identifier ')')?;
//------Frontier events------
FronterEvent: 'if' 'the' name=Identifier
('signal'|'message'|'timer') 'arrives' 'while' 'doing'
task=Identifier ('stop' 'it'|'wait' 'to' 'complete')
'and' 'then' ':' taskList=Tasklist;
```

Figure 3: Element to rule mapping for BPMN.

3.1.3 Task Definition Language

Another part of the grammar involves the definition of each task in each process to generate the views and controllers for the desired system. This definition must involve some sort of operations in some of the defined domain classes. This operation of course must be one of the four CRUD (Create, Read, Update or Delete) operations. In addition, two extra operations ("single selection" and "multiple selection") are added. This operations solve problems like showing a single instance (one must be selected beforehand) and selecting a list of instances to operate over them.

Furthermore, operations like creation and edition over multiple instances of the domain classes may be useful. The grammar created to fulfill these requirements is shown in Figure 4.

Previous textual representations (E-R and BPMN models) can be combined using this grammar fragment. Here we use the domain classes information to

[4]http://www.eclipse.org/Xtext/

```
TaskRedefinition:taskName=Identifier 'is' 'a' 'task' 'where'
':' partslist=ViewPartsList;
ViewPartsList: list+=AbstractOperation+;
AbstractOperation: operation=(Creation|MultipleView|
SingleView|FieldView);
FieldRestrictor : ',' 'only' fields+=Identifier (','
fields+=Identifier)* 'are' 'shown';
Operations :'edition'|'deletion'|'multiple' 'selection'|
'single' 'selection';
OperationAdition: ',' 'with' permisions+=(Operations)
((','|'and') permisions+=(Operations))* 'capabilities';
Creation: '-' arity=('multiple'|'a') name = Identifier
('are'|'is') 'created' (restriction=FieldRestrictor)? '.';
MultipleView: '-' 'all' 'the' child = Identifier
('in' parent = Identifier)? 'are' 'shown'
(restriction= FieldRestrictor)?
(operations=OperationAdition)? '.';
SingleView : '-' 'a' name = Identifier 'is' 'shown'
(operations=OperationAdition)? '.';
FieldView : '-' 'the' fields+=Identifier ((','|'and')
fields+=Identifier)* 'in' parent = Identifier 'are' 'shown'
(operations=OperationAdition)? '.';
```

Figure 4: Grammar for task definition.

specify the data contained in the generated views and actions to be performed in the controllers.

Overall, this restricted natural language permits to specify not only structural information but also functional requirements of the software. Thus, it allows the designer to build a textual specification of the required software in conjunction with the stakeholders.

3.2 Code Generation Approach

After the software specification is written in restricted natural language, the source code of the target web application is generated as depicted in Fig. 5.

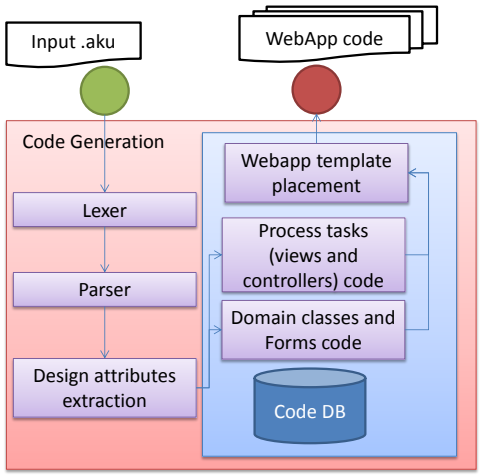

Figure 5: Code generation process.

The input of the code generation process is a file with the textual specification of the software (Input.aku). The first steps consist of traditional lexical and syntactic analyses performed to the specification. Next, the phase of design information extraction is performed where the names of the parts of the system are extracted. Then, this information is used to generate source code for the web application.

The source code generation is aided using a database containing the source code of previously defined specifications. This specifications can include definitions of the process helper views and tasks. Also can include code to generate the domain classes, input fields, HTML for the views and the final template where the generated code is placed.

Thanks to the restricted natural language, the lexer, parser, and payload extraction phases can be performed using a traditional top-down parser, avoiding the inherent ambiguity of natural languages.

After this processing, the code for the domain classes and forms is created. To match expected relationships, we check for the following cases: (1) an attribute has a known data type; (2) an attribute references another class in the specification; (3) a class has an attribute of the form *many*; and (4) class has an attribute of the form *many* and the another class has the first class referenced back in other attribute of the form *many*.

In the first case (1), we use the code database to generate the desired field; in the second case (2), we assign the data type of the attribute to a foreign key referencing the class. In third case (3), we understand this as an one-to-many relationship, thus, we alter the referenced class inserting a foreign key attribute referencing the class that has *a set of* the other class. Finally, for the fourth case (4), this is interpreted as a many-to-many relationship, therefore, we insert an additional table referencing the two involved classes and the respective foreign keys in each class.

Next, the code for the process is generated. For each task a template in the database is fetched and resolved, if the controller generated needs a view the file is generated also. If the task is a *SimpleTask* then the following process is performed:

1. Check for the definition in other paragraph of the specification
2. If the definition exist then resolve the controller and view code.
3. If its not defined use a default controller and view.

This *SimpleTask* definition methods use the *"TaskRedefinition"* non terminal to generate the code for its view and controller. It uses code fragments defined in the database to generate only the needed code to query, update, create or delete the domain classes instances.

In addition, we decided that a process control method is needed. This method will take the current task and determine the next based on the workflow defined in the *TaskList* non-terminals across the paragraphs in the specification. This method will help to resolve the gateways using the input given by the user in the generated view or the events registered. It also will help to determine the behavior of the tasks sequence related to cached frontier events.

The code for the exclusive gateways will be a method where a snippet of HTML is rendered asking the defined question. Then, after a POST request to a controller, use the previously defined controller method do determine the task to be done next.

Finally, these pieces of code are placed in a template prepared to execute the web application as desired in the specification.

4 IMPLEMENTATION

To implement the proposed solution we should make first a separation between the process to get the web application and the application itself. Both of them use different languages and tools to execute its required tasks. For the process we use a set of java based tools and for the result we use a set of python based ones.

For the process of generating the web application we used the *Xtext*[5] toolkit including the *Xtend* language, adding a database of specifications and code templates using SqlLite and a micro ORM framework called OrmLite[6]. Code templates are created using the Freemarker[7] engine.

According to the process illustrated in Fig. 6, which is the specialization of the general code generation process shown in Fig. 5. We use Xtext to generate the lexical analyzer, the parser, and the design information extractor. We implement the previously shown grammars. Also, Xtext creates a set of classes containing the required design information from each rule.

Prior to generate code for domain classes the references in its arguments are resolved first determining the case they belong as described previously in this paper. Notice that it is necessary to take into account the respective transformations between the plural and singular forms of the nouns (*entity names*). This can be done having a predefined set of rules and a dictionary for irregular nouns.

[5]http://www.eclipse.org/Xtext/

[6]http://ormlite.com/sqlite_java_android_orm.shtml

[7]http://freemarker.org/

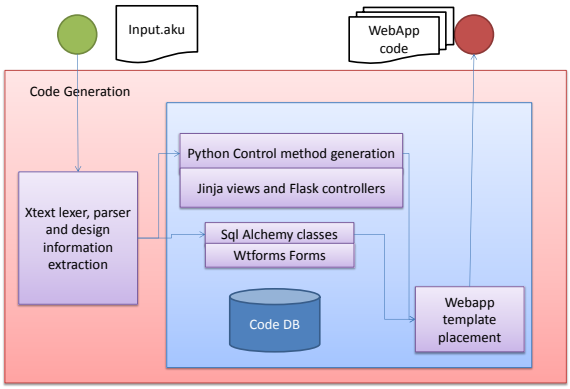

Figure 6: Implementation work flow.

When the references are determined, a set of previously written code templates is used to transform the design information into the target code. For the *entities* a *SQLalchemy*[8] (an *ORM* system) class, and also to class in a *WTForms*[9] (a web forms processor). Note that more than one code fragment can be generated in one template.

After this code is ready, the views and controller methods for each task are created. Again using templates (but this time several of them), we generate code for each of the non-terminals that are option of the non-terminal "Task".

For most of these elements a simple template replacement formula is enough. For instance, the template for "go back to" tasks is shown in Figure 7.

```
@app.route(
'/${r"${procesname}"}/<pid>/redirect/
${data.returnTo.asMethodName()}', methods=['GET']
)
def ${r"${procesname}"}_redirect_$
{data.returnTo.asMethodName()}(pid):
    return
  redirect(${r"${procesname}"}_resolve_next_task(
          request.url_rule,pid,None
    )
  )
```

Figure 7: "Go back to" code template example.

The process of generating a view changes if there is a redefinition for a "SimpleTask" non-terminal. Then the process uses the information given in the "TaskRedefinition" non-temrinal. It uses the information within its parse tree to determine the code result in both controller and view. To do this we use templates to generate partial results of the source code of the method and then merging them in a helper template.

[8]http://www.sqlalchemy.org/

[9]https://wtforms.readthedocs.io/

Moreover, there must be templates used to generate edition controllers with or without selections and only to show previously created elements. Following the same idea we use templates to create the HTML template that the web application needs as UI.

Also we iterate over the "Process" non-terminal instances again to create the helper method. This task is done with recursive calls of the "Tasklist" non-terminal within the process and gateway definitions. It generates a single method which takes as an input the URL of the current task and returns the next task.

After all the source code is generated, using the same template system, the resultant strings are placed directly into a web application template assuming that the needed imports and dependencies are resolved.

The template system used for the result web application was *Jinja2*[10]. Moreover, a small database in SQLite3[11] is used to deploy and test the result of the proposed method.

In summary, the tool chain used for the implementation of our proposal is comprised of: Programming languages: *Python 2.7, Xtend*; Parser generator: *Xtext*; Template systems: *Jinja 2, Freemarker*; Web framework: *Flask*; *ORM* systems: *SqlAlchemy, OrmLite*; Web forms processor: *WTforms*; *HTML*, *CSS*, and *JS* framework for developing responsive applications on the web: *Bootstrap*[12], and the *SB Admin 2 Bootstrap*[13] admin theme; Database: *SQLite3*.

5 RESULTS AND DISCUSSIONS

In order to validate and demonstrate the applicability of the proposal, this section includes the presentation of the results of a case study, and the comparison of this work to related works. The case study is titled: *Odoo Sales Clone*.

5.1 Case Study: *Odoo Clone*

Odoo[14] is an open ERP and CRM system. For the purpose of this work, some modules of this system are developed using our restricted language. The selected modules are: sales and projects. The selected version of Odoo is 9.0.

The sales module has 5 processes. Each one of them consists in a couple of tasks. There are three creation processes including: Client, Opportunity, and Product; and two additional processes to list opportunities and products, and select and edit them.

The project module consists in a series of projects, each one with a set of associated tasks. Also there are processes for the selection, presentation, and creation of tasks and projects.

The resultant software requirements can be specified using the restricted natural language as follows:

Language is an Domain Class and needs: a String named Name.

Contact Type is an Domain Class and needs: a String named Type.

Contact is an Domain Class and needs: a Contact Type named Type, a String named Name, a String named Address, a String named Address2, a String named City, a String named State, a Number named Zip Code, a String named Country, a Website named Homepage, a String named Work Desk, a Phone named Personal Phone, a Phone named Mobile Phone, a Phone named Fax, a Email named Personal Email, a String named Title and a Language named Language.

User is an Domain Class and needs: a String named Name and an Email named Username.

Sales Team is an Domain Class and needs: a String named Name, a Number named Code and a User named Team Leader.

Activity is an Domain Class and needs: a String named Message Type, a Number named Number Of Days and a Sales Team named Assigned Sales Team.

Opportunity State is an Domain Class and needs: a String named Name.

Opportunity Phase is an Domain Class and needs: a String named Name.

Opportunity is an Domain Class and needs: a String named Description, a Number named Income, a Number named Probability, a Contact named Client, an Activity named Next Activity, an Email named Contact Email, a Phone named Contact Phone, a Date named Provided Closing, a User named Seller, a Sales Team named Responsible Sales Team, a Number named Rating, an Opportunity Phase named Phase and a Opportunity State named State.

Product Type is an Domain Class and needs: a String named Name.

Product is an Domain Class and needs: an Image named Photo, a String named Name, a Product Type named Type, a String named Bar Code, a Number named Price and a Number named Cost.

Project State is a Domain Class and needs: a String named Name.

Project is a Domain Class and needs: a String named Name, a String named Task Label, a Contact named Client and a Project State named State.

Task is a Domain Class and needs: a String named Name, a String named Description, a Date named Limit Date, a User named Assigned and a Project named Project.

the Client Creation process starts, then: Create Contact and, the process ends.

Create Contact is a task where: -a Contact is created.

the Opportunity Creation process starts, then: Create Opportunity and, the process ends.

Create Opportunity is a task where: -a Opportunity are created.

the Opportunity List process starts, then: Show Opportunities, Show Opportunity and, the process ends.

Show Opportunities is a task where: -all the Opportunity are shown, with single selection capabilities.

[10] http://jinja.pocoo.org/docs/dev/

[11] https://www.sqlite.org/

[12] getbootstrap.com/

[13] https://startbootstrap.com/template-overviews/sb-admin-2/

[14] https://www.odoo.com/

Show Opportunity is a task where: -a Opportunity is shown, with edition capabilities.

the Product List process starts, then: Show Product List, Product Detail and, the process ends.

Show Product List is a task where: -all the Products are shown, only Name, Price are shown, with single selection capabilities.

Product Detail is a task where: -a Product is shown.

the Product Creation process starts, then: Create Product and, the process ends.

Create Product is a task where: -a Product is created.

the Project Creation process starts, then: Create Project and, the process ends.

Create Project is a task where: -a Project is created.

the Project List process starts, then: List Projects, Show Selected Project and, the process ends.

List Projects is a task where: -all the Projects are shown, with single selection capabilities.

Show Selected Project is a task where: -a Project is shown. -all the Tasks in Project are shown.

the Task Creation process starts, then: Create Task and, the process ends.

Create Task is a task where: -a Task is created.

the Task List process starts, then: List Tasks, Show Task and, the process ends.

List Tasks is a task where: -all the Tasks are shown, with single selection capabilities.

Show Task is a task where: -a Task is shown, with edition capabilities.

Resulting in a set of views linked as shown in Fig. 10. These views contain the creation processes and the two selection and posterior detail processes.

As an example of the generated code, the controller for the "Show Selected Project" task is shown in Fig. 8. This controller renders the view shown in Fig. 9, which is also a template written to be compatible with the Jinja2 template system. In this view each of the fields of the "project" domain class and all the "tasks" associated with the selected "project" are shown.

```
@app.route('/projectList/<pid>/showSelectedProject',
methods=['GET','POST'])
def projectList_showSelectedProject(pid):
    viewparams={}
    data = get_instance_data(pid)
    formdata = request.form.copy()
    for file in request.files:
        formdata[file] = request.files[file].filename
    request.form = formdata
    project = session.query(Project).filter(
       Project.id==data.get("selected_Project",0)
    ).first()
    viewparams['project']=project
    task_list = session.query(Task).filter(
        Task.project_id==data.get('selected_Project',0)
    ).all()
    viewparams['task_list']=task_list
```

Figure 8: Example of a generated controller.

```
<div class="panel-heading">Show Selected Project</div>
<div class="panel-body">
{% for field in project.__table__.columns._data.keys()%}
<div class="form-group">
        <b>{{field}}</b><br /> {{project[field]}}
</div>
{% endfor %}
<div class="col-lg-12">
<div class="table-responsive">
<table class="table">
<tr>
{% for prop in task_list[0].__mapper__.columns.keys() %}
<th>{{prop}}</th>
{% endfor %}
</tr>
{% for ent in task_list %}
<tr>
{% for prop in ent.__mapper__.columns.keys() %}
<td>{{ent[prop]}}</td> {% endfor %}
</tr>
{% endfor %}
</table></div></div></div>
```

Figure 9: Example of a generated view.

Some of the limitations of the generated prototype include: the card-like layout of Odoo is lost in the transcription, the same happens with its state based card list; the images of the products are not shown but they are uploaded; the project labels and their autocomplete field are omitted; and the labels of the form fields are not configurable without editing the source code.

On the other hand, the fast processing of the specification and its prototype generation allows the stakeholders to see within seconds changes made to the project. The textual specification permits to the designer a rapid specification and its corresponding validation after learning the restricted language constructs. Results can be seen in short time after a live meeting starts with the stakeholders, and changes to specifications at this stage are painless and economic. Furthermore, some tools like auto-completion, highlight syntax, and IDE integration can be used to make this process even easier.

5.2 Comparative to Similar Works

Table 1 presents a comparative of related works, including this proposal. The Table includes the main differences with respect to the starting point of the proposal, i.e., templates, (un)restricted natural language, or specification language. It also includes the final result of each work, e.g., executable program, UML models, E-R diagrams, etc.

One can see that most related works manage some form of natural language as starting point for the process. Most of the works end up with a model con-

Figure 10: Odoo clone result views.

structed in some specification language like BPMN, UML (some subset of the possible diagrams), or E-R. Further translations are limited for external tools like Rational Rose or Visual Paradigm. Some of the works also need a manual intervention of the user to select the design attributes. In this work, we obtain a working prototype closer to a final product without intervention from the user or external software.

Only a few proposals obtain executable programs as an output. Among them, the work in (Abbott, 1983) involve a manual process, and the proposal in (Schwitter, 1996) requires to work with previously defined code fragments in order to obtain a command-line based program. Contrarily, in the proposed approach, the result is an executable web application prototype.

6 CONCLUSIONS

This paper presented a fast prototyping method for web applications running business processes using a restricted natural language specification. Two different transformations were proposed between design models (E-R diagrams and BPMN models) and natural language (English) to propose a new specification language (restricted natural language). The proposed transformations based on previous specification languages maintain the expressiveness of the original

Table 1: Comparison to related works.

Work	Start point	Result
(Schwitter, 1996)	Specification language	Executable
(Deeptimahanti and Sanyal, 2011)	Unrestricted natural language	UML diagrams
(Friedrich et al., 2011)	Unrestricted natural language	BPMN diagram
(Liu et al., 2004)	Templates	UML class and sequence diagrams
(Abbott, 1983)	Unrestricted natural language	ADA Executable (manual process)
(Geetha and Mala, 2013) (Geetha and Anandha Mala, 2014)	Unrestricted natural language	E-R diagram
(Popescu et al., 2008)	Templates	UML class diagram
(Chioac, 2012)	Unrestricted natural language	OSMs
(Overmyer et al., 2001)	Unrestricted natural language	UML class diagram
(Desai et al., 2016)	Unrestricted natural language	DSLs
This Work	Restricted natural language + Templates	Executable Web application prototype

languages in the new representation. The use of this new language permits to overcome limitations in traditional specification languages, improving the code generation capabilities considerably.

This work opens up interesting paths for the automatic fast prototyping of web applications. However, there is more work to be done in the future. This includes: to propose a sub-grammar/method for automatic gateway resolution, and to extend the proposed restricted natural language by including information present in other design models besides E-R and BPMN. Other limitations of the present work can be addressed as well, such as: altering tasks execution based on previous tasks; visualizations of data; analytics over the performed processes; geo-referencing fields; fields with special visualizations; and special restrictions in relationships between domain classes.

In exchange for these limitations, a fast prototyping scheme is obtained where results can be seen, executed, and altered in very short time allowing all of this to occur during a live meeting with the stakeholders. The changes made to the specification can be seen instantly thanks to code generation capabilities and IDE integrations. The final product of this prototyping scheme is a source code ready to be part of the final product. This reduces the problems associated with the requirements elicitation and design stages.

REFERENCES

Abbott, R. J. (1983). Program design by informal English descriptions. *Communications of the ACM*, 26(11):882–894.

Augustine, S. and Martin, R. C. (2005). *Managing agile projects*. Robert C. Martin series. Prentice Hall Professional Technical Reference, Upper Saddle River, NJ.

Bellegarda, J. R. and Monz, C. (2015). State of the art in statistical methods for language and speech processing. *Computer Speech & Language*, 35:163–184.

Bhatia, J., Sharma, R., Biswas, K. K., and Ghaisas, S. (2013). Using Grammatical knowledge patterns for structuring requirements specifications. *2013 3rd International Workshop on Requirements Patterns, RePa 2013 - Proceedings*, pages 31–34.

Bryant, B. R. and Lee, B. S. (2002). Two-level grammar as an object-oriented requirements specification language. *Proceedings of the Annual Hawaii International Conference on System Sciences*, 2002-Janua(c):3627–3636.

Cambria, E. and White, B. (2014). Jumping NLP Curves: A Review of Natural Language Processing Research [Review Article]. *IEEE Computational Intelligence Magazine*, 9(2):48–57.

Chioac, E. V. (2012). Using machine learning to enhance automated requirements model transformation. *Proceedings - International Conference on Software Engineering*, pages 1487–1490.

Dahhane, W., Zeaaraoui, A., Ettifouri, E. H., and Bouchentouf, T. (2015). An automated object-based approach to transforming requirements to class diagrams. *2014 2nd World Conference on Complex Systems, WCCS 2014*, pages 158–163.

Deeptimahanti, D. K. and Sanyal, R. (2011). Semi-automatic generation of UML models from natural language requirements. *Proceedings of the 4th India Software Engineering Conference on - ISEC '11*, pages 165–174.

Desai, A., Gulwani, S., Hingorani, V., Jain, N., Karkare, A., Marron, M., R, S., and Roy, S. (2016). Program synthesis using natural language. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 345–356. ACM.

Fairley, R. E. (2009). *Managing and leading software projects*. IEEE Computer Society ; Wiley, Los Alamitos, CA : Hoboken, N.J.

Fowler, M. and Highsmith, J. (2001). The agile manifesto. *Software Development*, 9:28–35.

Friedrich, F., Mendling, J., and Puhlmann, F. (2011). Process Model Generation from Natural Language Text. *Lecture Notes in Computer Science*, 6741:482–496.

Geetha, S. and Anandha Mala, G. S. (2014). Automatic database construction from natural language requirements specification text. *ARPN Journal of Engineering and Applied Sciences*, 9(8):1260–1266.

Geetha, S. and Mala, G. (2013). Extraction of key attributes from natural language requirements specification text. In *IET Chennai Fourth International Conference on Sustainable Energy and Intelligent Systems*

(SEISCON 2013), pages 374–379. Institution of Engineering and Technology.

Githens, G. (2006). Managing Agile Projects by Sanjiv Augustine. *Journal of Product Innovation Management*, 23(5):469–470.

Granacki, J. J. and Parker, a. C. (1987). PHRAN-SPAN: a natural language interface for system specifications. *24th ACM/IEEE conference proceedings on Design automation conference - DAC '87*, pages 416–422.

Harris, I. G. (2012). Extracting design information from natural language specifications. *Proceedings of the 49th Annual Design Automation Conference on - DAC '12*, page 1256.

Ibrahim, M. and Ahmad, R. (2010). Class diagram extraction from textual requirements using natural language processing (NLP) techniques. *2nd International Conference on Computer Research and Development, ICCRD 2010*, pages 200–204.

Ilić, D. (2007). Deriving formal specifications from informal requirements. *Proceedings - International Computer Software and Applications Conference*, 1(Compsac):145–152.

Konrad, S. and Cheng, B. H. C. (2005). Facilitating the construction of specification pattern-based properties. *13th IEEE International Conference on Requirements Engineering RE05*, (August):329–338.

Liu, D., Subramaniam, K., Eberlein, A., and Far, B. (2004). Natural language requirements analysis and class model generation using UCDA. *Innovations in Applied Artificial Intelligence*, pages 295–304.

Meziane, F. and Vadera, S. (2004). Obtaining E-R diagrams semi-automatically from natural language specifications. pages 638–642.

Nishida, F., Takamatsu, S., Fujita, Y., and Tani, T. (1991). Semi-automatic program construction from specifications using library modules. *IEEE Transactions on Software Engineering*, 17(9):853–871.

Overmyer, S., Benoit, L., and Owen, R. (2001). Conceptual modeling through linguistic analysis using LIDA. *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001*, pages 401–410.

Popescu, D., Rugaber, S., Medvidovic, N., and Berry, D. M. (2008). Reducing ambiguities in requirements specifications via automatically created object-oriented models. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 5320 LNCS:103–124.

Saeki, M., Horai, H., and Enomoto, H. (1989). Software Development Process from Natural Language Specification. *Proceedings of the 11th International Conference on Software Engineering*, pages 64—-73.

Schwitter, R. (1996). Attempto-from specifications in controlled natural language towards executable specifications. *Arxiv preprint cmp-lg/9603004*.

Selway, M., Grossmann, G., Mayer, W., and Stumptner, M. (2015). Formalising natural language specifications using a cognitive linguistic/configuration based approach. *Information Systems*, 54:191–208.

Smith, R., Avrunin, G., and Clarke, L. (2003). From natural language requirements to rigorous property specifications. *Workshop on Software Engineering for Embedded Systems (SEES 2003) From Requirements to Implementation*, pages 40–46.

Steen, B., Pires, L. F., and Iacob, M.-e. (2010). Automatic generation of optimal business processes from business rules. pages 117–126.

Videira, C., Ferreira, D., and Da Silva, A. R. (2006). A linguistic patterns approach for requirements specification. *Proceedings - 32nd Euromicro Conference on Software Engineering and Advanced Applications, SEAA*, 2004:302–309.

Walia, G. S. and Carver, J. C. (2009). A systematic literature review to identify and classify software requirement errors.

Zapata, C. M. (2006). UN Lencep : Obtención Automática de Diagramas UML a partir de un Lenguaje Controlado. *Memorias del VII Encuentro Nacional de Computación ENC'06*, pages 254–259.

Zeaaraoui, A., Bougroun, Z., Belkasmi, M. G., and Bouchentouf, T. (2013). User stories template for object-oriented applications. *2013 3rd International Conference on Innovative Computing Technology, INTECH 2013*, pages 407–410.

Zhou, X; Zhou, N. (2008). Auto-generation of Class Diagram from Free-text Functional Specifications and Domain Ontology. (2):1–20.