An Integration of Semi formal and Formal Specifications: From Use Cases to RSL Signatures

Ana Funes, Aristides Dasso

Software Engineering Group
Universidad Nacional de San Luis, Ejército de los Andes 950,
D5700HHW San Luis, Argentina
{afunes, arisdas}@unsl.edu.ar

Abstract. At early stages of software system development, system requirements often are expressed in natural language. There are a number of techniques to extract useful information from these documents to construct a more precise –and formal– document that expresses the system requirements. Some of these techniques consist in identifying system use cases during requirement analysis work. Particularly, event-based techniques identify –from the elicited documents– the external events that a system must respond to and then related them to use cases and actors. These event lists are simpler than use cases –and are a first step in building them.
Although use cases have been proven to be a useful tool for requirement specification and facilitate the interaction with end users, they lack formality, giving place to misinterpretations and misunderstandings. Having this in mind, we propose a technique that integrates the understandability of graphical notations provided by use case notation with the unambiguity of formal specifications, by supplementing identified use cases –initially as a list of external events– with an initial formal specification consisting of function signatures and sorts in the RAISE Specification Language (RSL). Taking as input the identified external events associated with each system use case, which are expressed in natural language, we process them using a natural language tool that produces as output a structured format from which, by applying a set of rules, we translate them into RSL function signatures.

Keywords: Use cases, Natural Language Processing, Formal Specification, RAISE, RSL, Function Signatures.

1 Introduction

Documents used during requirement elicitation –at the initial stages of software system development– are usually a very informal specification, normally written in a natural language. They include various types of documents coming from different sources including project documentation, business process documentation, and stakeholder interviews.

During this step of software development, informality is present most of the time even when using a semiformal or a formal development process for the rest of the

software development. One of the main reasons for this is that the initial requirements are captured from users not conversant with software engineering practices.

An important stage, following requirement gathering (or elicitation), is the task of going from these informal functional requirements to a more formal or semiformal specification, recording, in various forms, the identified requirements. A commonly adopted technique consists in the use of use case diagrams for requirement specification.

Requirement specification is a crucial task since it is the one that will drive the rest of the software development process. Because of that it is essential to devote all the necessary time to it although it is also true that these initial activities in requirement analysis will have to be refined in a spiral model, meaning that the steps of requirement capture and specification will have to be repeated a number of times. In any case obtaining a clear, complete, consistent and unambiguous semiformal or –better still– formal specification is the aim of the requirement analysis phase in software development process.

Use cases are a popular technique for specifying functional requirements, especially in software development. Although they are easy to grasp by different stakeholders, and, in consequence, they make interaction between analysts, client and final users much easier, they use an informal notation, which can lead to future misinterpretations during software development.

Formal specification is recognized as a useful and quite exact method of software engineering, particularly helpful at the early stages of the software system development process, namely requirement specification. However, formal specifications lack the simplicity and necessary understandability for interaction during this stage of development. In [2], Berry presents a balanced analysis of Formal Methods application.

In this work we present a technique to complement the simplicity and ease of understanding of use cases with the precision and unambiguity given by formal specifications.

It is not easy to transform an informal specification into a formal –or even a semiformal– one, however much of a serious interest is in achieving this transformation as automatically as possible.

There are a number of techniques to extract useful information from these initial, informal, documents to build a more precise –and formal– document that expresses the system requirements. Generally these techniques are kept informal. Our aim is to propose a technique that, in a more automatic and semiformal fashion, makes the passage from a list of external events associated with the identified use cases to a formal specification. External events associated with use cases are expressed in natural language while the produced formal specification is written in a formal specification language.

Since this passage –natural language to formal specification language– can be seen as a translation, we use well-known tools (see [8], [10], [11]) built with the objective of achieving automatic translations between natural languages. Using the component of the tool that analyzes the natural language with the aim of getting semantic from it, we obtain a semantic structure that is used to construct an initial formal specification in a particular formal language.

As we said before, our technique uses two different approaches to obtain the initial system specification. Firstly, we propose employing a non-rigorous approach to use cases, where uses cases are constructed from the system external events identified during elicitation. These events are given in the form of a list expressed in natural language. Secondly, we propose to complement these use cases with a formal specification which can be obtained from the external events. To achieve this, in first place we make an analysis of the natural language employed in the external events associated with each use case. This is done using a natural language analytical tool – Freeling [8]. A mapping –using a set of transformation rules– is then applied to the output resulting from the previous analysis, rendering them into an initial formal specification.

Let us note that it is a difficult task to produce a formal specification, even an abstract one, when there is only an informal specification. This is normally the situation confronted by most software engineers at the start of system development. In our proposal we obtain a first formal specification which can be later refined by the developer. The proposed technique intends to be a way to formalize use cases by enclosing them with the corresponding formal specification given in the form of signatures and types –as sorts– written in the language of the RAISE Method –the RAISE Specification Language (RSL) [18].

RAISE stands for Rigorous Approach to Industrial Software Engineering. It comprises a formal notation, techniques, recommendations and tools for the development of software systems. RAISE provides, beside the method, the RAISE Specification Language (RSL). The RAISE method proposes –using a stepwise refinement technique to develop software– to go from an initial, very abstract specification to a more concrete one which can be translated in some programming language [19].

RSL is a wide-spectrum formal language given that it supports several specification styles, namely abstract, axiomatic as well as concrete and operational styles. It may be used through out the complete life cycle from the initial stage of domain and requirements analysis to a level at which specifications may be translated into executable code. RSL provides a rich, mathematically based notation in which specifications may be formulated and reasoned about.

The rest of this work is organized as follows. Section 2 discusses related work. In Section 3 we give a brief description of use cases, the event-based approach used for identifying use cases and an example to illustrate the concepts. Section 4 describes the natural language processing applied and the used tools. In section 5 we recall some concepts from RSL necessary for the development of this work. Section 6 explains the transformation from natural language to RSL and section 7 its application to a study case. Finally, section 8 gives the conclusions and future work.

2 Related Work

In the literature a number of techniques to go from natural language to a more formal o semiformal description can be found. However not all of them are used in the context of system development. The use of natural language in extracting valuable

information to achieve a given goal –which is after all a valuable objective in system analysis– can be found in different areas, e.g. engineering design [20].

In [4] the authors show a way to produce from a formal specification a document in natural language. The translation system is implemented "using the Grammatical Framework, a grammar formalism based on Martin-Löf's type theory". While this is extremely important in producing up-to-date documentation, is not pertinent to our goal, although it goes to show that natural language and formal methods do have to be considered as a unit when documents are needed.

Jastram et al [14] are concerned with tracing requirements from a natural language specification into a formal language. They employ WRSPM [7] as a foundation requirement model for the Event-B [5] that is their choice of formal model. A heuristic approach it is used to go from a natural language specification to a formal one.

Gunter et al [7] define a model, called WRSPM. This is a reference model for requirements and specifications that can be used with different languages for the description of the WRSPM artifacts. There is no provision for translating the languages used to the WRSPM formalization.

Sampaio do Prado Leite et al [13] are also concerned with requirement tracing, in their case they employed a broader view of scenarios enhancing it with the Language Extended Lexicon [12] that is a rigorous method that uses natural language, but with a reduced vocabulary oriented to the Universe of Discourse closely related to the requirement domain. It is certainly a reasonable approach to natural language usage.

QuARS (Quality Analyzer of Requirements Specification) [9] is a tool to analyze a natural language requirement and see that it complies with a set of rules established in a quality model for software requirements. "The Quality Model we defined for the natural language software requirements is aimed at providing a way to perform a quantitative (i.e. that allows the collection of metrics), corrective (i.e. that could be helpful in the detection and correction of the defects) and repeatable (i.e. that provides the same output against the same input in every domains) evaluation." [6].

3 Use Cases

Use cases are a powerful technique for the elicitation and documentation of functional requirements. Once the requirements have been elicited, use cases can be used in the initial phase of software development to specify the intended behavior of part or the whole system.

The term *use case* was introduced by Ivar Jacobson in 1986 [24]. A use case is a description of a set of interactions between actors and a system, i.e. it is a general way of using some part of the functionality of a system.

An *actor* is a role played by a user. Actors are external entities and they interact directly with the system.

A use case comes mainly in two flavors –either as graphical modeling element such as in the Unified Modeling Language (UML) [15] or in textual form.

In UML, the relationships between actors and system are represented in a use case diagram as it is shown in Figure 1. Use cases are represented by ellipses containing

the use case name inside while an actor is represented by a "stick man" icon with the name of the actor usually above or below the icon.

The UML specification suggests several different formats for specifying the behavior of a use case: "The behavior of a use case can be described by a specification that is some kind of Behavior (through its ownedBehavior relationship), such as interactions, activities, and state machines, or by pre-conditions and post-conditions as well as by natural language text where appropriate." [15] page 606.

Moreover, in a number of instances use cases are instantiated in natural language form to be converted later into a diagram. This use of natural language creates a gap when trying to go from this informal specification to a more formal abstract specification.

Although OMG gives a standard for the diagrams [15], there are no standards for the natural language or textual form, although several proposals can be found, e.g.: [3], [1], [21]. Even if these proposal are based in the use of templates that are quite structured, we can find parts where text is used without any particular or strictly mandatory format.

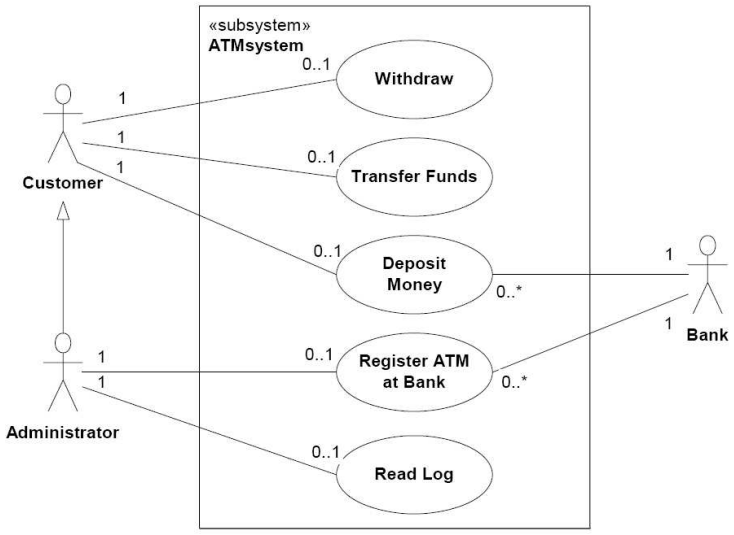

Figure 1. Use case diagram for an ATM system[1]

There are a fair number of tutorials for writing use cases, most of them recommending the use of particular templates either in plain text or in table format. In some instances there are recommendations to start writing a use case using up from narrative scenarios [16]; in other cases [17] after identifying classes of users, outlining the use case suite –a list of actors and associated use case names to finally writing the use case descriptions is recommended.

When identifying use case, we can adopt basically two different approaches. One approach is *actor-based* and the other is *event-based*. The actor-based approach

[1] Figure taken from the OMG Unified Modeling Language™ (OMG UML), Superstructure Version 2.4.1 OMG.

consists in (a) Identifying the actors related to the system or organization by looking at which users will use the system and which other systems must interact with it; (b) For each actor found in (a), identifying the processes they initiate or participate in by looking at how the actor communicate/interact with/use the system.

On the other hand, the event-based approach consists in (a) Identifying the external events that a system must respond to and (b) Relating the events to actors and use cases. In this work, we adopt this second approach. In fact, we see a use case as a sequence of actions that provide a given functionality to a system actor. A use case describes a way in which a real world actor interacts with the system. Each of these interactions corresponds to an external event to which the system must provide an answer. We can then establish a correspondence between the system use cases and the system external events.

An example of a preliminary stage of a use case. To illustrate the point we have identified some external events in a credit card system. A credit card system is something that is familiar to many people but it can have several different characteristics up to and including Automatic Teller Machine (ATM) cards.

The system considered here implies not only a credit and buying card but also some ATM operations (depositing and withdrawing money), as well as operations with bank accounts.

Figure 2 shows a simplified and very limited version of some possible use cases expressed as a list of events for a credit card system.

1. A customer withdraws money from a bank account.
2. A customer withdraws money from a card account.
3. A customer asks a credit from a card account.
4. A customer deposits money on a bank account.
5. A customer pays a balance of a card account.
6. A customer changes PIN.

Figure 2. A reduced list of events for a credit card system.

4 Natural Language Processing

In this section we show how the list of events associated with the system use cases can be processed using a natural language processing tool to produce a more structured text from which it can be easily extracted the information necessary to construct the corresponding RSL function signatures.

To process the input natural language we use FreeLing [8], which is an open suite of language analyzers developed at the Centre de Technologies i Aplicacions del Llenguatge i la Parla (TALP) at the Universitat Politècnica de Catalunya (UPC).

FreeLing uses the EAGLES [11] tags for its morphological analyses and also the WordNet [10] dictionaries. FreeLing can be incorporated to an application or it can be used from a simple one-line executable.

FreeLing's natural language analyzer can produce several different kinds of outputs given as input a file in natural language format. Particularly, it can output a file with the text morphologically analyzed. This file has a simple text format and the input text is tokenized, sentence split and morphologically analyzed one token in each line and the sentences separated with one blank line.
The general format is:

word lemma1 tag1 prob1 lemma2 tag2 prob2 ...

or if sense tagging has been activated, the format is:

word lemma1 tag1 prob1 sense11:...:sense1N lemma2 tag2 prob2 sense21:...:sense2N ...

We have processed the list of external events given in Figure 2 using FreeLing's analyzer with the following command line:

```
analyzer -f config\en.cfg --outf morfo <list.txt >list.mrf
```

This command line tells the FreeLing's analyzer: to use the standard English configuration file (`en.cfg`), that the input file is `list.txt` and the output file – with the input text morphologically analyzed (`morfo` option)– is `list.mrf`.
In the English configuration file, the options are: to tag the output, split the sentences and where most of the options for morphology analysis are set –suffix analysis, detection for multi-words, number, punctuation, dates, do a dictionary search, and probability assignment.

We give as input to FreeLing a text file with a list of events and Freeling returns a text file with the text analyzed. For example, when we give to FreeLing the list shown in Figure 2, it returns the line `bank bank NN 0.997148 bank VBP 0.00285171`, for the word *bank* when treating the first item in the event list, given a probability of 0,99 to it being a noun rather anything else.

To carry out the translation process, we propose to use not only the word morphology probability given by FreeLing in choosing the more proper term but also an ad hoc dictionary and a thesaurus.

5 Types and Signatures in RSL

The signature of a function is the general information about the function, its name, parameters, types and other miscellaneous information. A signature in RSL can be a value signature, i.e. a name with a type expression associated. This is the kind of signature that we intend to obtain from the system use cases –a value signature where the associated type is a function type.

In the transformation proposed here, the initial RSL specification obtained will consists of a set of definitions of sorts and value signatures, where each value is associated with an event in the list. Each of these RSL value signatures corresponds to the signature of a RSL function, which is a mapping from values of one type to values of another type. A type is a collection of logically related values. RSL provides ready built-in types and there are also abstracts types, which can be referred to as sorts.

Sorts are types defined by the user which have no operators defined except equality (=) and inequality (≠); they are used in abstract axiomatic specifications and can be later refined in more concrete types.

In summary, the proposed technique allows to obtain, from each event in the list associated with a system use case, a function signature whose domain and range types are defined by Cartesian products of RSL sorts. Let us note that the resulting RSL specification is abstract and it needs to be further refined.

6 How to go from the Use Cases to Function Signatures

The preliminary list of external events for identifying use cases should be constructed with simplicity in mind. Each event in the list should be a simple, active voice sentence. Complex, compound sentences should be avoided.

A simple sentence is one where there is a subject and a predicate. The predicate has a verb and an object. Figure 3 shows the grammar for this lexical structure.

<*sentence*> ::= <*subject*> <*predicate*>
<*subject*>::= <*article*> <*noun*> | <*noun*>
<*predicate*> ::= <*verb*> <*object*>
<*object*> ::= <*obj_direct*> | < *obj_indirect*>
<*obj_direct*> ::= <*article*> <*noun*> | <*noun*>
<*obj_indirect*> ::= <*preposition*> <*obj_direct*>

Figure 3. Grammar for an external event.

Together with this grammar an ad hoc dictionary with the nouns, verbs, articles and prepositions should be constructed. Also a thesaurus with the necessary synonyms should be created.

The example of the list given in Figure 2 has been constructed paying close attention to these requirements as we can see in Table 1. Compound nouns are accepted.

Table 1. The events in the list given in Figure 2 with the grammar structure.

subject		*predicate*						
			object					
			obj_direct		*obj_indirect*			
						obj_direct		
article	*noun*	*verb*	*article*	*noun*	*preposition*	*article*	*noun*	
A	customer	withdraws		money	from	a	bank	account.
A	customer	withdraws		money	from	a	card	account.
A	customer	asks	a	credit	from	a	card	account.
A	customer	deposits		money	on	a	bank	account.
A	customer	pays	a	balance	of	a	card	account.
A	customer	changes		PIN.				

The technique give us for each use case the corresponding function signature – domain and range only– whose types correspond to products of sorts, as follows:

1. For each entry in the list there is a function, whose name is obtained from the sentence's verb followed by the noun following the verb in the direct object plus the noun that is in the indirect object. RSL accept overloading of identifiers.
2. Nouns in events correspond to types (RSL sorts) or external entities to the system (use case actors). Figure 4 shows a list with the nouns found after the analysis of the event list given in the example. Most of them are used to define the sorts used in the function signatures to be built, except "customer" that corresponds to the use case actor "Customer". Compound names are used as type names with underscores to separate words (e.g. nouns 3 and 4 in Figure 4).

1. customer
2. money
3. bank account
4. card account
5. credit
6. balance
7. PIN

Figure 4. Nouns from the events analyzed

3. The function domain is built from the nouns found (except the subject noun that corresponds to an actor). The range is formed with the noun that follows the verb in sentences that have no indirect object or from the noun that follows the preposition in sentences that have an indirect object. Figure 5 shows the grammar for the function signatures.

<*function_signature*> ::= <*function_name*>:<*function_domain*> → <*function_range*>,
<*function_name*> ::= *word*<*Vx*>_*word*<*Nx after verb*>_*word*<*function_range*>
<*function_domain*> ::= *word*<*Nx*> | <*domain*>
<*domain*> ::= ∅ | × *word*<*Nx*> <*domain*>
<*function_range*> ::= *word*<*last Nx after preposition*> | *word*<*last Nx after verb*>

Figure 5. Grammar (rules) for the RSL function signatures

In the rules given in Figure 5, we use the words tagged with *Vx* for the verbs, where *x* stands for any of the tags used for the attributes of verbs in FreeLing; and we use the words tagged with *Nx* –where *x* stands for any of the tags used for the attributes of nouns in FreeLing.

7 Application of the Proposed Transformation

In this section we show the RSL specification –derived from the list of external events already presented in Section 2– after the application of the transformation proposed.

```
scheme EVENTS =
class
    type
        money,
        bank_account,
        card_account,
        credit,
        balance,
        PIN
    value
    /* Event 1: A customer withdraws money from a bank
    account. */
    withdraw_Money_Bank_Account: Money × Bank_Account →
    Bank_Account,

    /* Event 2: A customer withdraws money from a card
    account. */
    withdraw_Money_Card_Account: Money× Card_Account →
    Card_Account,

    /* Event 3: A customer asks a credit from a card
    account.*/
    ask_Credit_Card_Account: Card_Account × Credit →
    Card_Account,

    /* Event 4: A customer deposits money on a bank
    account.*/
    deposit_Money_Bank_Account: Money× Bank_Account →
    Bank_Account,

    /* Event 5: A customer pays a balance of a card
```

```
    account.*/
    pay_Balance_Card_Account: Balance × Card_Account →
    Card_Account,
    /* Event 6: A customer changes PIN. */
    change_PIN: PIN → PIN
end
```

From event 1 in the list we have obtained the following signature:

```
withdraw_Money_Bank_Account: Money × Bank_Account →
                             Bank_Account,
```

As we can see, the function name was obtained from the sentence's verb plus the noun following the verb (direct object) and the noun that is in the indirect object. The function domain and range were obtained from the sentence's nouns. The function result is made up from the noun in the indirect object –that is a compound name in this case– that follows the preposition *from*. Compound names are joined using underscore.

Following a standard for type's names their first letter is capitalized and the first noun of a function name is not.

The other entries in the list are processed using the same rules.

Constructing the signatures from the list is also a means of exploring the requirements in more detail. Note that there are simple checks that one can apply to the signatures to look for such issues:

- Are the parameters sufficient for computing the result?
- Is every new value or changed component represented as part of the result?
- Are the parameters independent?

For example, applying the last question to function number 5 leads to the following:

The amount paid is somehow tied to the balance of the account and this is surely part of an account statement.

If the amount paid is precisely the outstanding balance then the latter is presumably part of the balance statement, and not necessary a separate parameter.

If the amount paid is arbitrary (the customer chooses) then the amount is a necessary extra parameter, but perhaps the external event would be expressed in a slightly different way, e.g. 'A customer pays a part of the balance of a card account' or 'A customer pays a given amount of the balance of a card account'.

If the amount paid is at least a minimum established in the statement then the amount is also necessary, and the statement must be a parameter since it will have the amount to be paid and the account number.

Another step in this analysis is to decide if the functions obtained are total or partial. This leads to the identification of new functions needed to express preconditions of partial functions. This raises new questions like whether the extraction's limit is exceeded, whether the customer card account exists, where the authentication for the expiration date on the card comes from for event number 1 for example. And this leads to the need for more parameters to be able to compute the preconditions.

The example shown here is simple as is the grammar employed. What happens if the wording use in the list is changed and ambiguities appear?

Figure 6 shows the list used in the previous example but now with some minor changes in items 1, 5 and 6.

1. A customer withdraws **funds** from a bank account.
2. A customer withdraws money from a card account.
3. A customer asks a credit from a card account.
4. A customer deposits money on a bank account.
5. A customer pays an **outstanding** balance of a card account.
6. A customer changes **her** PIN.

Figure 6. The modified version of the list.

In item 1 we have changed *money* for *funds*. Although this is a minor change since *money* and *funds* are synonyms and the change could be solved using a dictionary for the case under treatment as was said above, *funds* have more meanings than money and these appeared reflected in FreeLing's output.

Table 2. The modified list with the grammar structure.

subject		*predicate*								
			object							
						obj_indirect				
				obj_direct				*obj_direct*		
article	*noun*	*verb*	*?*	*article*	*?*	*noun*	*preposition*	*article*	*noun*	
A	customer	withdraws				**funds**	from	a	bank	account.
A	customer	withdraws				money	from	a	card	account.
A	customer	asks		a		credit	from	a	card	account.
A	customer	deposits				money	on	a	bank	account.
A	customer	pays		an	**outstanding**	balance	of	a	card	account.
A	customer	changes	**her**			PIN.				

While the word *money* in the second item of the list has the following result from FreeLing:

```
money money NN 1
```

which does not leave any doubt of its meaning the word *funds* used in item one in the list produced the following result:

```
funds fund NNS 0.499372 fund VBZ 0.00125549 funds NNS
0.499372
```

Here we can see the three different morphological categories output by FreeLing, where two of them having the same probability of being nouns.

Another problem that could create ambiguities in the generation of the signatures is the use of both words in generating the sorts, since we would finish having two different sorts for something that clearly should be only one when seeing from the point of view of the system. This ambiguity can be solved by using a thesaurus.

We have included two words in items 5 and 6 that are not in the grammar given above. Of course this can be detected by a tool that analyzes the grammar as an invalid sentence, or as a valid one by extending the grammar to accept adjectives and pronouns in those places.

In Table 2 the grammar structure for the modified list is shown.

The input to FreeLing is again a text file with the list of Figure 6. As an example, the output for line 6 is:

```
A 1 Z 0.139784 a DT 0.184178 a IN 0.288008 a JJR
0.00975934 a NN 0.378271
customer customer NN 1
changes change NNS 0.959574 change VBZ 0.0404255
her her PP$ 1
PIN pin NP 1
. . Fp 1
```

8 Conclusions and Future Work

In this work we have presented a supplementary integration between semi-formal and formal notations in the form of an approach that semi-automatically generates from external events –associated with the system use cases and given in plain natural language– a more rigorous specification – a set of abstract functions and sorts in RSL.

It is well-known that use cases are a suited communication means for both developers and clients but they lack the formalism that a formal notation provides. The proposed integration overcomes the respective problems inherent to each of these notations.

The obtained specification not only provides the precision and rigour absent in use cases but it also help in giving the first steps in defining and specifying formally a system. Note that stemming from the questions presented at the end of section 6 further refinement of the formal specification is possible as well as necessary.

Due to the inherent complexity and richness of natural languages, going from external events to signatures is more heuristic than automatic since, for instance, determining function's domain and co-domain would come out from a semantic analysis of the sentence verb more than from the sentence syntactic construction. However, for a developer –which is fully involved in the problematic of the system being analyzed– determining functions domain and co-domain should not imply serious problems, and an automated tool based on the proposed transformation can be of help.

Regarding future work, several research lines can be approached having in mind that the proposed technique is a point to start. Based on the ideas given here, a first step is the development of new rules (grammar) to include more sophisticated natural

language constructions and their incorporation to a tool to make the whole process more automatic. The tool should be integrated with a natural language analyzer.

Furthermore, part of the work to do in future implies to validate the results by means of techniques such as recall and precision. We also plan to evaluate or develop ontologies to give semantics to the more common words, used in specific domains. These ontologies can be consulted by the tool to improve the function signatures deduction process.

Finally, an important work to tackle is the generation of the bodies of the RSL functions from the specifications of the respective use cases. On the one hand, a similar treatment could be applied to the more structured text-only use cases descriptions constructed using templates. These are much more complete and organized since they can be considered to be using a constraint or limited language due to the separation –especially when using table templates– of the different parts of the use case and the recommendations about the use of the language. Alternatively, the use of interaction diagrams for giving the specification of use cases raises the possibility of analyzing a transformation to RSL function bodies.

An even more ambitious work is the integration of the technique presented here with a previous work done in this area [23] –an automatic generation of RSL specifications from UML class diagrams– in order to round up a methodology for requirement engineering process.

References

[1] Cockburn A.: Basic Use Case Template, http://alistair.cockburn.us/index.php/Resources_for_writing_use_cases#Use_Case_Templates. Retrieved May 2012.

[2] Berry, D. M.: "Formal Methods: The very idea, some thoughts about why they work when they work". Electronic Notes in Theoretical Computer Science 25 (1999). 1999 Published by Elsevier Science B. V.

[3] CRaG Systems: Use Case Tutorial http://www.cragsystems.co.uk/SFRWUC/index.htm. Retrieved May 2012.

[4] Burke, D. A. and Johannisson, K.: "Translating Formal Software Specifications to Natural Language A Grammar-Based Approach". Logical Aspects of Computational Linguistics. Lecture Notes in Computer Science, 2005, Volume 3492/2005, 47-82, DOI: 0.1007/11422532_4.

[5] Event-B: http://www.event-b.org/. Retrieved May 2012.

[6] Fabbrini, F., Fusani, M. , Gnesi, S. , Lami, G.: "An Automatic Quality Evaluation for Natural Language Requirements". Proceedings of the Seventh International Workshop on RE: Foundation for Software Quality (REFSQ'2001).

[7] Gunter, C. A.; Gunter, E. L.; Jackson, M.; Zave, P.: "A Reference Model for Requirements and Specifications". May/ June 2000 IEEE SOFTWARE.

[8] FreeLing: http://nlp.lsi.upc.edu/freeling/. Retrieved May 2012.

[9] QuARS: http://quars.isti.cnr.it/QuARSreferences.html . Retrieved May 2012.

[10] Wordnet: http://wordnet.princeton.edu/wordnet/. Retrieved May 2012.

[11] EAGLES: http://www.ilc.cnr.it/EAGLES96/home.html. Retrieved May 2012.
[12] Sampaio do Prado Leite, J. C., Franco, A. P. M.: "A Strategy for conceptual Model Acquisition". Proceedings of IEEE International Symposium on Requirements Engineering, 1993.
[13] Sampaio do Prado Leite, J. C., Rossi, G., Balaguer, F., Maiorana, V., Kaplan, G., Hadad, G. and Oliveros, A.: "Enhancing a Requirements Baseline with Scenarios". Requirements Eng (1997) 2:184-198. 1997 Springer-Verlag London Limited.
[14] Jastram, M., Hallerstede, S., Leuschel, M., Russo, A. G.: "An Approach of Requirements Tracing in Formal Refinement". VSTTE'10, Verified Software: Theories, Tools and Experiments. 16th-19th August 2010. Edinburgh, Scotland.
[15] OMG: The Unified Modeling Language™ (OMG UML), SuperstructureVersion 2.4.1 OMG Document Number: formal/2011-08-06 Standard document URL: http://www.omg.org/spec/UML/2.4/. Retrived May 2012.
[16] primaryview.org: Scenarios, Sequence Diagrams, and Narrative Text. February 2003. http://www.primaryview.org/UML/Scenarios.html. Retrived May 2012.
[17] ReadySET Pro: Use Case Tutorial. http://readysetpro.com/whitepapers/usecasetut-all.html. Retrieved January 2012.
[18] The RAISE Language Group: The RAISE Specification Language, Prentice Hall, 1992.
[19] The RAISE Method Group: The RAISE Development Method, Prentice Hall, 1995.
[20] Honda, T., Yang, M. C., Dong, A., Ji, H.: "A Comparison of Formal Methods for Evaluating the Language of Preferenece in Engineering Design". Proceedings of the 2010 ASME IDETC. International Design Engineering Technical Conferences & Information in Engineering Conference. August 15 – 18, 2010, Montreal, Canada.
[21] Fowler, M.: UML Destilled, Third edition, Addison-Wesley, 2004.
[22] Pons, C. and Baum, G.: Formal Foundations of Object-Oriented Modeling Notations. In Proceedings of 3rd IEEE International Conference on Formal Engineering Methods (ICFEM'00), September 04 - 07, 2000, York, England.
[23] Funes, A. and George, C.: Chapter 8: "Formalizing UML class diagrams" in "UML and the Unified Process", ISBN 1931777446, Idea Group Publishing; April 1, 2003.
[24] Jacobson, I.: Object-oriented software engineering : a use case driven approach. New York; Wokingham, Eng.;Reading, Mass.: ACM Press ;Addison-Wesley Pub., 1992.