Computing and Informatics, Vol. 26, 2007, 421–445

A DERIVATION STRATEGY FOR FORMAL SPECIFICATIONS FROM NATURAL LANGUAGE REQUIREMENTS MODELS

María Virginia Mauco, María Carmen Leonardi

Facultad de Ciencias Exactas
Universidad Nacional del Centro de la Pcia. de Buenos Aires
Tandil, Argentina
e-mail: {vmauco, cleonard}@exa.unicen.edu.ar

Manuscript received 3 October 2006; revised 19 February 2007
Communicated by Ulrich Eisencker

Abstract. Formal methods have come into use for the construction of real systems, as they help increase software quality and reliability. However, they are usually accessible only to specialists, thus discouraging stakeholders' participation, crucial in first steps of software development. To address this problem, we present in this paper a strategy to derive an initial formal specification, written in the RAISE Specification Language, from requirements models based on natural language, such as the Language Extended Lexicon, the Scenario Model, and the Business Rules Model, which are closer to the stakeholders' language. We provide a set of heuristics which show how to derive types and functions, and how to structure them in a layered architecture, thus contributing to fruitfully use the large amount of information usually available after requirements modelling stage. In addition, we illustrate the strategy with a concrete case study.

Keywords: Language extended lexicon, scenario model, business rules model, formal specifications, RAISE method

1 INTRODUCTION

Formal methods have come into use for the construction of real systems, as they help increase software quality and reliability, and even though their industrial use is still limited, it has been steadily growing [29]. By using formal methods early in the software development process, ambiguities, incompleteness, inconsistencies, errors, or misunderstandings can be detected, avoiding their discovery during costly testing and debugging phases. However, one of the problems with formal specifications is that they are hard to master and inappropriate as a communication medium, as they are not easily comprehensible to stakeholders, and even to non-formal specification specialists. This is particularly inconvenient during the first stages of software development, when interaction with stakeholders is very important. The information gathered during requirements elicitacion is heavily based on natural language as stakeholders must be able to read and understand the results of this phase [2, 23]. Therefore, a good formal approach should use both informal and formal techniques [3, 29].

A well-known formal method is the RAISE method, which has been used on real developments [6]. RAISE is an acronym for "Rigorous Approach to Industrial Software Engineering" [27]. The RAISE method includes a large number of techniques and strategies for doing formal development and proofs, as well as a formal specification language, the RAISE Specification Language (RSL) [26], and a set of tools [9]. However, it does not include any strategy for the requirements step that addresses the problems mentioned before. As we had some experience in requirements modelling, we proposed the use of natural language requirements models to elicit, model, and communicate requirements in an easy and friendly way, and then map them to an initial RSL specification that will be the basis for software development using RAISE method.

The Language Extended Lexicon (LEL) and the Scenario Model are two well known natural language requirements models, used and accepted by the Requirements Engineering community [15]. In addition, the Business Rules Model allows the explicit specification of the policies of the organization in natural language [14, 18, 25]. These models ease and encourage stakeholders' participation, very important in early stages of software development. However, they have to be reinterpreted by software engineers into a more formal design on the way to a complete implementation. Many works aiming at reducing the gap between the requirements step and the next steps of software development process have been published. Some of them describe, for example, different strategies to obtain object oriented models or formal specifications from requirements specifications [4, 7, 13, 19].

In this paper, we present a technique to derive an initial formal specification in RSL from the LEL, Scenario, and Business Rules Models. This work enhances the strategy presented in [21, 22] by adding the heuristics related to the Business Rules Model [14]. This initial RSL specification may be validated by using a prototype produced by the SML translator [30].

The rest of the paper is organised as follows. In Sections 2 and 3 we briefly introduce the natural language requirements models and RAISE, respectively. The core of the paper is in Section 4, where we describe the strategy to derive the initial specification in RSL. We exemplify our proposal with a real case study: the Milk Production System [20]. Section 5 contains an approach to validate our proposal. Finally, in Section 6 we present some conclusions and outline possible future work.

2 NATURAL LANGUAGE ORIENTED REQUIREMENTS MODELS

The models presented in this section are accepted and used by the Requirements Engineering and Business Modelling communities. They provide an attractive way of communication and agreement between software engineers and stakeholders. As they are written using natural language they can be read and understood by the stakeholders, thus allowing them to participate actively in the requirements definition process.

FIELD

Notion

– Land where cows eat pastures.
– It has an identification.
– It has a precise location in the dairy farm.
– It has a size.
– It has a pasture.
– It has a hectare loading.
– It is divided into a set of plots.
– It has a list of previous plots.

Behavioural Response

– A dairy farmer divides it into a set of plots, separated by electric wires.
– Many different groups of cows can be eating in it simultaneously.

Table 1. Example of an object LEL symbol

2.1 Language Extended Lexicon

The LEL is a structure that allows the representation of significant terms of the Universe of Discourse (UofD) [15]. Its purpose is to help understand the vocabulary and the semantics of the UofD, unifying the language and enhancing stakeholders' participation.

LEL is composed by a set of symbols. Each symbol has a name (and possibly a set of synonyms), a notion, and a behavioural response. The notion describes what the symbol is, while the behavioural response describes how the symbol acts upon the UofD. LEL symbols may be classified, according to their semantic in the UofD, in objects, subjects, verb phrases, and states, and there are some heuristics that suggest what to include in the notion and behavioural response of each kind of symbol. In the description of the symbols two rules must be followed simultaneously: the closure principle that encourages the use of LEL symbols when describing other LEL symbols, and the minimum vocabulary principle which minimizes the use of symbols external to the lexicon. Table 1 shows an object LEL symbol, taken from the case study. Underlined phrases correspond to other LEL symbols.

2.2 Scenario Model

A scenario describes a situation in the UofD [15]. Scenarios also use a natural language description as their basic representation and they are naturally linked to the LEL. Scenarios can be derived from the LEL by applying a set of heuristics. In [15] the scenario construction process is described.

A scenario is composed by a title to identify it, a goal describing its purpose, a context to define geographical and temporal locations and preconditions, actors which are entities actively involved in the scenario (generally persons or organizations), a set of resources that identify passive entities with which actors work, and a set of episodes where each episode represents an action performed by actors using resources. An episode may be explained as a scenario; this enables a scenario to be split into sub-scenarios.

Table 2 contains an example of one scenario taken from the Milk Production System Scenario Model. Underlined words or phrases are symbols defined in the LEL, and phrases written in the episodes using capital letters correspond to the title of other scenarios.

TITLE: Feed a group
GOAL: Register the daily ration given to a group.
CONTEXT: It is done once a day. Pre: Group is not empty.
RESOURCES: Group Date Quantity of corn silage Quantity of Hay Quantity of concentrated food Feeding form
ACTORS: Dairy farmer
EPISODES:
– COMPUTE RATION.
– The dairy farmer records, in the Feeding form, the date and the quantities of corn silage, hay and concentrated food given to each cow in the group.
– COMPUTE PASTURE EATEN.

Table 2. Example of a scenario

2.3 Business Rules Model

Business rules are a key concept in the requirements definition process. A business rule is a statement about the way an enterprise does business [14]. Business rules can be viewed as expressing functional and nonfunctional requirements which are characterized by their strategic importance to the business, and consequently they deserve special consideration. According to the Semantic of Business Vocabulary and Business Rules Specification (SBVR) [25], a rule is an element of guidance that introduces an obligation or a necessity. This definition gives two main categories of rules:

- Structural rules (necessities): Define how the business chooses to organise (i.e. structure) the things it deals with. They supplement definitions.
- Operative rules (obligations): They govern the conduct of business activity; they can be directly violated by people involved in the affairs of the business.

A cow has an earring with the identification number.
Category: *Structural* (Necessity)

A pregnant heifer must be vaccinated against diarrhoea in its seventh month of pregnancy.
Category: *Operative* (Obligation)

Table 3. Examples of business rules

Table 3 shows examples of each type of rule, taken from the Milk Production System business rules model.

The business rules construction process [18] begins with the identification of the sources of information. Organisation documents, such as ISO required documents and organisational models are generally the best sources. If the company does not have any documentation, other techniques such as observation, interviews and meetings should be used to acquire the information. We categorise the sentences that appear in the sources considering their purpose in the organisation, trying to distinguish sentences referring to: limits, responsibilities, and rights. To decide if a sentence is a business rule, we analyse if it is determined by a decision of the organisation or if it is an inherent sentence to the functionality of the UofD (in which case it is not considered a rule).

Business rules are classified and documented following syntax patterns, connecting them with the corresponding LEL symbols. At this point, it is important to identify the LEL symbol/s directly affected by the rule. After documentation, the model is verified against the set of scenarios and the LEL. Finally, the model is validated with stakeholders to detect elicitation errors or organisational conflicts.

3 RAISE

RAISE gives its name to a wide spectrum specification and design language, the RAISE Specification Language (RSL), an associated method, and an available set of tools to help writing, checking, printing, storing, transforming, and reasoning about specifications. Complete descriptions of RSL and the RAISE method can be found in the corresponding books [26] and [27], while the tools are described in [9], and they can be downloaded from UNU-IIST's web site (www.iist.unu.edu).

Usually the first RSL specification is an abstract, applicative and sequential one, which is later developed into a concrete specification, initially still applicative and then imperative, and sometimes concurrent. A specification in RSL is a collection of modules. A module is basically a named collection of declarations; it can be a scheme or an object. Each module should have only one type of interest. However, the kernel module concept is that of a class expression. A basic class expression is a collection of declarations enclosed by the keywords **class** and **end** and it represents a class of models. Objects and schemes are defined using class expressions.

A typical applicative class expression contains **type**, **value**, and some **axiom** definitions. Axioms may be used to constrain the values or to model invariants.

RSL is a typed language. This means each occurrence of an identifier representing a value, variable, or channel must be associated with a unique type. Besides, it must be possible to check whether each occurrence of an identifier is consistent with a collection of typing rules.

A type is a collection of logically related values, and it may be specified by an abstract or a concrete definition. An abstract type, also referred to as a sort, has only a name. It is a type we need but whose definition we have not decided yet. A concrete type can be defined as being equal to some other type or using a type expression formed from other types. In order to provide concrete definitions for types, we need a collection of types to use. RSL has seven built-in types (Bool, Int, Nat, Real, Char, Text, and Unit) with their corresponding operators, and a number of ways of constructing types from other types (type constructors, record types, variant types, union types, and subtypes). Type constructors allow the definition of composite types: products ($\times$), functions ($\rightarrow$ for total functions, $\overset{\sim}{\rightarrow}$ for partial ones), sets (**-set** for finite sets, **-infset** for infinite ones), lists (* for finite lists, $^\omega$ for infinite ones), and maps ($\underset{m}{\rightarrow}$ for finite maps, $\overset{\sim}{\underset{m}{\rightarrow}}$ for infinite ones). Sets, lists, and maps define collections of values of the same type. A set is an unordered collection of distinct values, while a list is a sequence of values, possibly including duplicates. A map is a table-like structure that maps values of one type into values of another type.

The following definitions exemplify some of the concepts defined above.

type
  Cow_id, /∗ abstract type ∗/
  Cow, /∗ abstract type ∗/
  Cows = Cow_id $\underset{m}{\rightarrow}$ Cow /∗ concrete type, map type expression ∗/

Records are very much like those common in programming languages. Each component of a record has an identifier, called a destructor, and a type expression. Optionally a record component can have a reconstructor.

Variant types allow the definition of types with a choice of values, perhaps with different structures. Subtypes are types that contain only some of the values of another type, the ones that satisfy a predicate. For instance, we can define the type Pregnant_cow as the one containing values that satisfy the predicate is_preg_cow.

type
  Pregnant_cow = $\{|$ c : Cow • is_preg_cow(c) $|\}$

Values are constants and functions. Their definition must include at least the signature, that is a name, and types for the result, and for the arguments, in case of a function. A function is a mapping from values of one type to values of another type, and it can be total or partial. It is total when it is defined for every value of the arguments, and it is considered partial when it is not known to be total.

4 THE STRATEGY FOR THE DERIVATION OF THE RSL SPECIFICATION

As an attempt to bridge the gap between stakeholders and the formal methods world, we propose a strategy to derive an initial formal specification in RSL taking as input LEL, scenarios, and business rules. This proposal is an extension of the works described in [20, 21, 22]. The derivation of the specification is structured in four steps which guide the definition of RSL types and functions, and their structuring in modules. The four steps are:

- Derivation of Types (Section 4.1),
- Definition of Modules (Section 4.2),
- Derivation of Functions (Section 4.3),
- Enhancement of the Specification (Section 4.4).

Although these steps are not strictly sequential, the strategy always begins by deriving types and finishes with the application of business rules heuristics. In the following sections we briefly describe each step, providing examples taken from the case study.

4.1 Derivation of Types

This step produces a set of abstract as well as concrete types, which model the relevant terms in the UofD. We perform the derivation of the types in two steps. First we identify the types, and then we decide how to model them. Most of the types derived in the Identification step will be abstract types, and many of them will be replaced by more concrete ones in the Elaboration step. This way of defining types follows one of the key notions of the RAISE method: stepwise development. The replacement of an abstract type by a more concrete one follows the implementation relation. Implementation is very important because if an initial specification meets the requirements and all its developments follow the implementation relation, then they all meet the requirements. Sections 4.1.1 and 4.1.2 present the heuristics to identify the types of the RSL specification and to model them respectively.

HITid	**LEL symbol**	**RSL type**	**RSL specification**
HIT1	Subject/object name is a singular noun		
HIT1.1	Object representing a computable property	Abstract type	Property_name
HIT1.2	Subject/object name is a noun, also a symbol in the LEL, modified by a phrase:		
HIT1.2.1	If it represents a category, state or situation	Subtype expression (not always)	Main_type, /* already defined */ Subtype = {| s: Main_type • is_subtype(s) |}
HIT1.2.2	If it represents a different subject/object	Abstract type	Symbol_name
HIT1.3	Otherwise	Abstract type	Symbol_name
HIT2	State		
HIT2.1	If name refers to a symbol in the LEL	Subtype expression (not always)	Main_type, /* already defined */ Subtype = {| s: Main_type • is_subtype(s) |}
HIT2.2	If name does not refer to a symbol in the LEL	Abstract type	State_name
HIT3	Verb represents an action with data to save	Abstract type	Verb_name
HIT4	Symbol name is a plural noun or symbol is an element of a collection:		
HIT4.1	If instances have an attribute or set of attributes for identification	Map type expression	Sym_id, Sym_name, /* already defined */ Map = Sym_id $\overrightarrow{m}$ Sym_name
HIT4.2	If instances need an ordering	List type expression	Sym_name, /* already defined */ List = Sym_name*
HIT4.3	Otherwise	Set type expression	Sym_name, /* already defined */ Set = Sym_name-**set**

Table 4. Heuristics to identify RSL types

4.1.1 Identification of Types

The main goal of the Identification of Types step is to determine an initial set of types that are necessary to model the different entities present in the analysed UofD. This initial set will be completed, or even modified, during the remaining steps of the specification derivation. For example, during the Definition of Modules step it may be necessary to define a type to reflect the state of the UofD. Also, when defining functions it may be useful to define some new types to be used as result types.

Table 4 summarises the heuristics we propose to define the relevant types. The prefix HIT, used to distinguish each heuristic, means Heuristics for the Identification of Types. The LEL is the source of information as LEL subjects and some objects represent the main components or entities of the analysed UofD. In general, LEL subjects and objects will correspond to types in the RSL specification. In addition, LEL verbs may also give rise to the definition of more types, when they represent an activity which has its own data to save.

For example, by applying the heuristics HIT1.3 and HIT4.1 to the LEL symbol Field from Table 1 we obtain the following specification:

type
  Field_id, /∗ from HIT4.1 ∗/
  Field, /∗ from HIT1.3 ∗/
  Fields = Field_id $\underset{m}{\rightarrow}$ Field /∗ from HIT4.1 ∗/

4.1.2 Elaboration of Types

Having defined a preliminary set of types and in order to remove under-specification, we propose to return to the information contained in the LEL and the Scenario Model. In particular, the analysis of the notion, and sometimes the behavioural response, of each symbol that motivated the definition of an abstract type, can help decide if the type could be developed into a more concrete type. As we have already mentioned, all the developments we suggest satisfy the implementation relation. Table 5 presents some heuristics to assist in this task. The prefix HDT, used to distinguish each heuristic, means Heuristics for the Development of Types. During this step, it might be necessary to introduce some type definitions that do not correspond to any entry in the LEL. They appear, in general, when modelling components of some other type. Symbols without an entry in the LEL may represent an omission or a symbol considered outside the UofD language. When an omission is detected, it is necessary to return to the LEL to add the new definition, and update the Scenario and Business Rules Models to maintain the consistency between their vocabulary and the LEL itself.

For example, the abstract type Field, coming from an object LEL symbol (Table 1) as set by heuristic HIT1.3, may be developed into a short record definition with five components (HDT1.1 and HDT1.2), representing the properties identified from the symbol notion (location, size, pasture, set of plots, and previous plots). Hectare loading is discarded by heuristic HDT1.2 because it is a computable property. The RSL definition for the type Field is then:

type
  Location,
  Size,
  Pasture,
  Plots,

HDTid	Type comes from	RSL type	RSL specification
HDT1	Subject/object symbol:		
HDT1.1	Notion contains one or more properties of the symbol	Short record definition	Sym_name:: prop_1: Prop_type_1 ... prop_n: Prop_type_n /∗ n >0 ∗/
HDT1.2	Notion contains computable property of the symbol	Simple type	Prop_name = Type_expression /∗ Remove property from record ∗/
HDT1.3	Notion represents symbol state or category	Variant type	Categ == cat_1 | ... |cat_n /∗ (n >0) ∗/
HDT1.4	Categories or states share some attributes	Variant type Short record definition Subtypes (if necessary)	Categ == cat_1 | ... |cat_n, Main_type:: common_attr_1: Attr_type_1 ... common_attr_m: Attr_type_m distinguishing_attr: Categ, St_Cat_1 = {| mt : Main_type • has_cat_1(mt) |}, ... St_Cat_n = {| mt: Main_type • has_cat_n(mt) |} /∗ (n, m > 0) ∗/
HDT1.5	Object behavioural response suggests symbol property		/∗ In general, model the property as part of the object∗/
HDT2	Verb describing an action applied to an object		/∗ In general, model the action as part of the object∗/

Table 5. Heuristics to develop identified types

```
Field::
   location: Location
   size: Size
   pasture: Pasture
   plots: Plots
   past_plots: Plots
```

4.2 Definition of Modules

Modules are the means to decompose specifications into reusable units. This decomposition into modules is particularly useful when designing complex systems, as it eases and encourages separate development, one of the principles the RAISE method is based on.

This step helps to organise in modules all the types produced by the Derivation of Types Step in order to obtain a more legible and maintainable specification. These modules would be later completed with the definition of functions, and probably with more type definitions. A summary of the heuristics we propose to define the

HDMid	**Type**	**RSL module**	**RSL definition**
HDM1	For all the types that must be visible to users or used in at least two modules	Two modules: a scheme and a global object which is an instantiation of the scheme	**scheme** GLOBAL_TYPES = **class** **type** Global_type_1, ... Global_type_n **end** /∗ n > 0 ∗/ context: GLOBAL_TYPES **object** GT:GLOBAL_TYPES
HDM2	Models an element of a collection	Scheme	**scheme** COLL_ELEM = **class** **type** Coll_elem **end**
HDM3	Models a collection	Scheme, where the scheme modelling each element in the collection is defined as an embedded object	context: COLL_ELEM **scheme** THE_COLLECTION = **class** **object** CE: COLL_ELEM **type** /∗ if collection specified with a map ∗/ The_Collection = GT.Coll_id $\xrightarrow[m]{}$ CE.Coll_elem /∗ if collection specified with a list ∗/ The_Collection = CE.Coll_elem* /∗ if collection specified with a set ∗/ The_Collection = CE.Coll_elem-**set** **end**
HDM4	For all the types modelling UofD components	One top level module defined as a scheme, with each scheme defining a UofD component instantiated as an embedded object	context: DOM_COMP_1, ..., DOM_COMP_n **scheme** DOM_STATE = **class** **object** DC_1: DOM_COMP_1, ..., DC_n: DOM_COMP_n **type** Dom_state:: dom_comp_1: DC_1.Dom_Comp_1 ... dom_comp_n: DC_n.Dom_Comp_n /∗ n > 0 ∗/ **end**

Table 6. Heuristics to define modules

modules is shown in Table 6. The prefix HDM, used to distinguish each heuristic, means Heuristics for the Definition of Modules.

In defining these heuristics, we closely followed the features RSL modules should have according to the RAISE method. For example, each module should have only one type of interest, defining the appropriate functions to create, modify, and observe values of the type, and the collection of modules should be, as far as possible, hierarchically structured. Therefore, we first identify class expressions to define schemes, and then we assemble these schemes defining objects to express dependencies between them.

For example, as suggested by heuristics HDM2 and HDM3, the following two scheme modules result from the map Fields defined during the Derivation of Types step:

```
scheme FIELD =
    class
        type
            Field::
                location: Location
                size: Size
                pasture: Pasture
                plots: Plots
                past_plots: Plots
    end

context: FIELD      /* to make the scheme FIELD visible */
scheme FIELDS =
    class
        object F: FIELD  /* embedded object for the collection element */
        type
            Fields = GT.Field_id $\underset{m}{\rightarrow}$ F.Field
                  /* GT is a global object where the type Field_id is defined */
    end
```

4.2.1 The Architecture of the Specification

The modules defined by applying the heuristics we proposed can be hierarchically organised to show the system module structure. The root of this hierarchy is the system module, the second level contains the modules that define each UofD component, and the remaining levels correspond to the modules that help to define the upper ones. A diagram showing this hierarchy can be generated automatically by the RAISE tools.

The RAISE method provides many guidelines to hierarchically structure a specification, aiming at encouraging separate development and stepwise development. These guidelines allow to obtain a hierarchy of modules that may be specified using the Layers pattern [5]. As the strategy we propose closely follows all these guidelines, the RSL specification derived can be structured in layers. Considering the Layers Pattern, the global architecture we suggest is composed of three layers: specific layer, general layer, and middleware layer [21]. Figure 1 is an example of a module diagram with the layers identified. A layer is a set of RSL modules that share the same degree of generality. Lower layers are general to several domain specifications, while higher ones are more specific to a concrete domain. The specific layer contains application-specific modules not shared by other parts. The general layer includes modules that are not specific to a single application and then they can be reused for many different

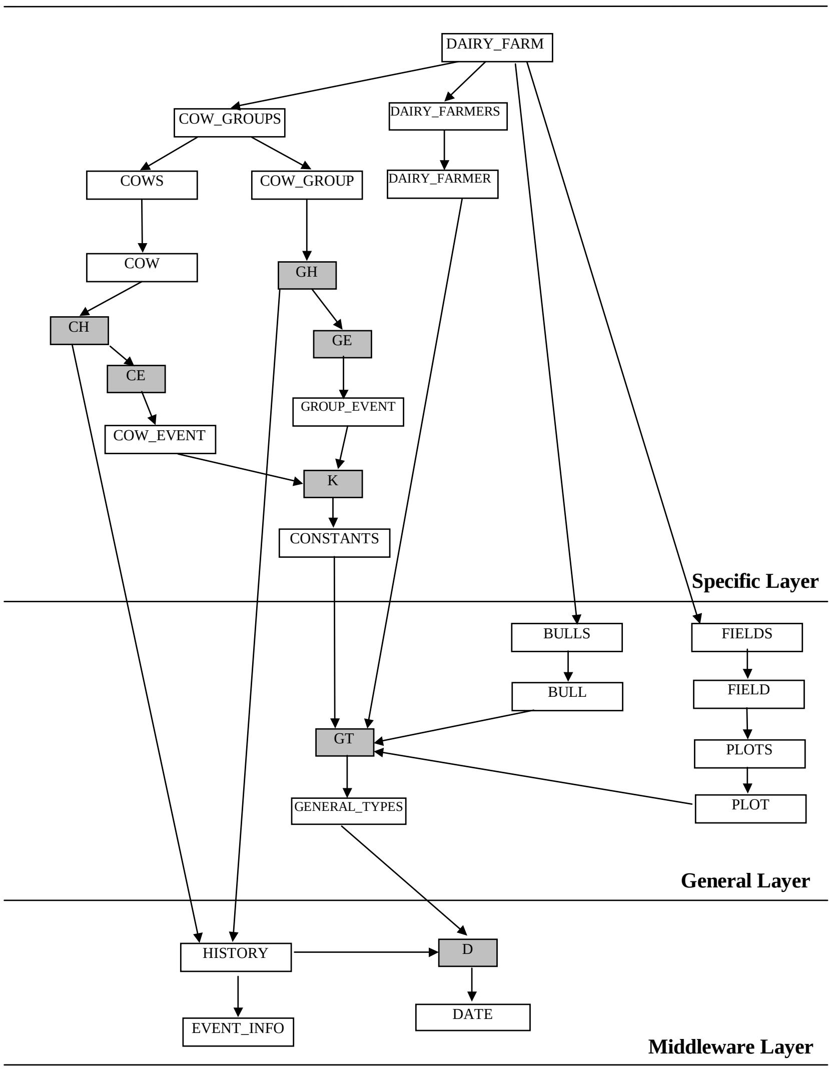


Fig. 1. Architecture of the milk production system RSL specification

applications within the same domain or business. The middleware layer has modules that are so general that can be used in any domain. Examples of middleware layer modules are standard specifications such as bags, stacks, queues, etc. at different levels of abstraction. A specific module, which is located in the specific layer, can use modules of the general layer or the middleware layer. Modules located in the general layer can use modules in the middleware layer. This way of defining use relationships between layers is similar to the one proposed in [5], although more flexible.

RSL provides control over visibility and hence dependency, allowing a layer to be partially opaque. This means that some of its modules are only visible to the next higher layer, while others are visible to all higher layers [20].

In general, the development of a specification into another one has no impact on the layered architecture. Development in RAISE typically involves replacing more abstract modules with more concrete ones, and sometimes it also involves introducing new child modules. A child appears when the development of a module introduces a new component or concept worthy of its own module. A developed module will be in the same layer as its more abstract counterpart, while the new child may be in the same layer or in a lower one.

Our proposal of using the Layers Pattern to structure the hierarchy of modules of a specification in RSL assumes all the modules have the same specification style, as for example applicative sequential as in our case study. When developing the modules into a different style, such as imperative sequential, the architecture could be respected as long as the implementation relationship holds between the modules of the different layers of both specifications.

4.3 Derivation of Functions

In this section we present a set of heuristics which help to identify and to model the functions of the RSL formal specification. Scenarios are the main source of information when defining functions, as they are natural language descriptions of the functionality in the UofD. In addition to functions that are specific to the considered UofD, we also provide definitions for the appropriate functions to create, modify, and observe the type of interest of each module defined in the Definition of Modules step. Functions are usually identified at the top level as scenarios help generate them there. Functions at one level in the hierarchy of modules frequently have counterparts at lower levels, but with different parameters. For each function in the top level module we model the necessary functions in lower level modules, in order to simplify the legibility and maintainability of the specification. See [21] for details.

We perform the derivation of functions in two steps: Definition of top level functions and Definition of low level functions, which are detailed in the next two sections.

4.3.1 Definition of Top Level Functions

Top level functions represent the main functionality in the UofD and they are defined in the system module.

Behavioural responses of LEL subjects include the main functionality in the UofD, and each of them is usually described with more details in a scenario. Therefore, in general, each scenario will motivate the definition of a top level function. We classify each scenario as modifying or observing depending on whether it produces a change in the UofD or not. We model the first ones with generator functions and the second ones with observer functions.

Table 7 summarises the heuristics we propose to specify top level functions (HTF stands for Heuristics for Top Level Functions). After determining which functions to define in the top level module (HTF1), the next steps are the formulation of their signatures (HTF2, HTF3, HTF4) and the definition of their bodies. The definition of the signature of a function involves determining its arguments and result type as well as classifying it as partial or total.

HTFid	**Scenario Model**	**RSL**
HTF1	Scenario describes a LEL subject behavioural response	Top level function
HTF2	Scenario type	Function type
HTF2.1	Modifying	Generator (not always)
HTF2.2	Observing	Observer
HTF3	Scenario components	Function signature
HTF3.1	Modifying scenario	
HTF3.1.1	Resources to modify and resources with data to modify	Arguments
HTF3.1.2	Actors	Probably arguments
HTF3.1.3	Resources and/or actors modified	Result
HTF3.2	Observing scenario	
HTF3.2.1	Resources to access information	Arguments
HTF3.2.2	Actors	Probably arguments
HTF3.2.3	Information returned	Result
HTF4	Context	Total or partial function

Table 7. Heuristics to model top level functions

For example, the scenario shown in Table 2 is a modifying one, motivating the definition of the generator function feed_group (HTF2.1). The resources of the scenario suggest that the group, the date, the quantities of concentrated food, hay and corn silage, and the feeding form should be the arguments of the function (HTF3.1.1). The actor dairy farmer should not be an argument because its only responsibility is the execution of the action described by the scenario (HTF3.1.2). The resource Feeding form is the place where the change performed by the scenario is stored, so it will represent the function result (HTF3.1.3). The context of the scenario Feed a group establishes that a group should be fed once a day and only if the group is not empty. The function feed_group must then be defined as partial (HTF4). An informal definition for this function would be

feed_group: group $\times$ date $\times$ quantity of corn silage $\times$ quantity of hay $\times$
quantity of concentrated food $\times$ feeding form $\overset{\sim}{\rightarrow}$ feeding form

We use this kind of informal definition to determine the types to include in the signature of the function. These arguments and result are replaced by the corresponding types previously defined during the Derivation of Types step (Section 4.1).

For a generator function, the result type is always the record type representing the system state. This record type is also included as an argument type because it contains the definition of all the domain components. The rest of the argument types correspond to the types used to define the data required to identify the components to modify as well as the information with which to modify them. Then, when the type comes from a subject or an object whose collection was defined as a map, the type of the corresponding identification argument will be only the set of attributes defined as the map domain. For the function informally defined above, the signature is the following:

value
feed_group: GT.Group_type $\times$ D.Date $\times$ GT.Quantity $\times$ GT.Quantity
$\times$ GT.Concentrate $\times$ Dairy_farm $\overset{\sim}{\to}$ Dairy_farm

We use Group_type as argument because the LEL symbol Group is an argument whose collection was represented with the map Groups. Feeding form is apparently not included as argument. But, when deriving the types we decided to model all the events applied to groups of cows as part of the type Group [20].

The function feed_group is a partial one, and its precondition will be formulated as a call to a function defined in the top level module. Some preconditions arise implicitly when we need to check values input by the user. For example, to record the birth of a calf to a cow, we would check that the cow's identifier refers to an existing cow. This might not appear explicitly in the scenario.

The hierarchy of modules affects the specification of functions. Most of the top level functions will call functions in the second level, which in turn will call functions in the levels below, thus motivating the definition of more functions in lower level modules.

In general, the body of each top level function will contain a call to one or more functions defined in modules in the second level. In the case of a generator function, the body will contain at least one call to a chg_component function which in turn will call to the function or functions that perform the modification of the corresponding component(s). Thus, each chg_component function will have as its first argument a call to a second level function in charge of doing the change, and as its second argument the system state. To determine the appropriate second level function to call, we analyse the informal definition we have previously proposed. This definition shows which components are modified. From these components, the types obtained during the derivation of types, and the hierarchy of modules, we can identify the arguments and result type of the second level function, and the second level module in which it should be defined. For example, the complete definition for the function feed_group is

value
can_feed_group: GT.Group_type $\times$ D.Date $\times$ Dairy_farm $\to$ **Bool**
can_feed_group(gt, d, df) $\equiv$ ...,

feed_group : GT.Group_type $\times$ D.Date $\times$ GT.Quantity $\times$ GT.Quantity
$\times$ GT.Concentrate $\times$ Dairy_farm $\overset{\sim}{\to}$ Dairy_farm

```
feed_group(gt, d, corn, hay, conc, df) ≡
   chg_groups(CGS.feed_group(gt, d, corn, hay, conc,
                              groups(df), cows(df)), df)
pre can_feed_group(gt, d, df)
```

The definition of observer functions is quite similar. Details can be found in [20].

4.3.2 Definition of Lower Level Functions

Top level functions and their preconditions are modelled in terms of functions in the second level modules. For each function that is called in the body or the precondition of a top level function, and whose name has an object name as prefix, we analyse the object name to determine in which second level module we should define the function. From the call in the top level module and the informal definition obtained from the corresponding scenario, we can also find out the signature of the second level function, i.e. function arguments and result type and its classification as partial or total function.

For example, the function CGS.feed_group should be defined in the module COW_GROUPS of which CGS is an instance. From the call in the top level module we can also find out the signature of the second level function.

The body of a second level function usually contains a call to one or more functions in a lower level module, the one that manipulates each individual value in the map range. Following the heuristics proposed in [20], the complete formal definition for the function feed_group in COW_GROUPS is

```
value
   feed_group : GT.Group_type × D.Date × GT.Quantity
      × GT.Quantity × GT.Concentrate × Cow_groups
      × CS.Cows ~→ Cow_groups
   feed_group(gt, d, corn, hay, conc, cgs, cs) ≡
   let
      ration = GT.mk_Ration(0.0, corn, hay, conc),
      /∗ Ration is a record with 4 components ∗/
      new_r = GT.chg_pasture(compute_pasture_eaten(gt, ration, cgs, cs), ration)
      /∗ chg_pasture takes a ration and modifies its pasture producing
      a new ration ∗/
   in
      cgs † [gt ↦ CG.feed_group(new_r, d, cgs(gt))]
   end
   pre can_feed_group(gt, d, cgs, cs)
```

In the specification above, CG.feed_group is the lower level function that generates a new Cow_group value which feed_group uses to update the Cow_groups map.

The episodes in a scenario are a good source of information when trying to define the bodies of functions like CG.feed_group. However, it is not easy to provide general guidelines for defining the bodies of such functions because they will depend on the details of the types and module structure chosen.

4.4 Enhancement of the RSL Specification

In this section we describe how to include business rules in the RSL specification, modelling the necessities and obligations of the organization in terms of RSL elements.

As we described in Section 2.3, business rules involve LEL symbols, some of them representing the receiver of the rule and the others giving information about the business rule itself (structure or behaviour information). Thus, it is necessary to identify to which RSL elements these symbols were mapped to in order to decide where to attach the rule. This attachment is also greatly influenced by the hierarchy of modules (Section 4.2). In general, business rules define invariants, pre and post conditions for functions, and they also may define new type elements or functions.

Table 8 summarises the heuristics we propose to map business rules to RSL elements. The following are some examples of business rules taken from the Business Rules Model of the case study and their mapping to the RSL specification.

Rule	**LEL + RSL involved terms**	**Modification of RSL elements**
Structural	LEL objetc/subject modelled as RSL type/type component	– Define invariant for type component – Define new type component
Operative	Subject/Object modelled as type or Verbal phrase modelled as type/function	– Define pre and/or post-condition for function – Define new type component – Define new function

Table 8. Business rules heuristics

Rule: It is necessary that the <u>dairy farmer</u> has an <u>official authorization</u> to operate.

Category: Structural

Involved LEL Symbols: Dairy farmer, Official authorization

This structural rule modifies the definition of the type Dairy_farmer, adding an axiom to verify the existence of an official authorization.

```
type     /∗ in module DAIRY_FARMER.rsl ∗/
   Dairy_farmer::
                        salary: GT.Salary
                        employees: GT.Employee-set
                        off_authorization: GT.Off_auth
axiom
∀ df: Dairy_farmer • df_authorization(df) = true
```

Rule: If a dairy farmer detects a dairy cow has any disease it is obligatory to dry the cow for discard.

Category: Operative

Involved LEL Symbols: Dairy farmer, Dairy cow, Dry the cow for discard

According to the hierarchy of modules, this rule is attached to the module Cows.

```
...
scheme COWS =    /* in module COWS.rsl */
class
...
value
...
dry_cow: GT.Cow_id × D.Date × Cows ~→ Cows
dry_cow(ci, d, dc, cs) as cs
post  cs † [ ci ↦ C.set_dry(d, dc, cs(ci)) ]
pre has_disease(cs(ci))
...
end
```

4.5 Discussing the Strategy

When analysing developments of RSL specifications of different domains, we found they start from informal descriptions containing synopsis, narrative, and terminology [20]. Once obtained these informal descriptions, in general, each case followed its own approach to obtain the RSL specification, although of course they all considered the principles proposed in the RAISE method. We proposed and defined a concrete and detailed three-step process that could be applied in any domain, allowing to take profit of informal descriptions and reducing the gap between them and the final RSL specification.

Our proposal is based in the metamodel of LEL, Scenario Model, and Business Rules Model. The heuristics were defined considering the way in which the concepts of the UofD are described. We followed the structure proposed in each metamodel, but we allowed the use of a free style to describe each component. For example, we considered that each LEL symbol has a notion and a behavioural response but we followed a free style to express the content of notions and behavioral responses of LEL symbols. As a consequence, some heuristics take always a fixed decision; for example, definition of types is based on the classification of LEL symbols, modelling one type per each subject or object LEL symbol (Heuristic HIT1). However, other heuristics need human judgement to decide how to model some kind of requirement. For example, when developing abstract types into more concrete ones (Heuristic HDT1) or identifying verbal phrases modelling registration behaviour (Heuristic HDT2), the software engineer has to analyse the content of notions.

In the way to automate our strategy, one possibility to overcome some of the problems mentioned before would be to impose stronger standards or guidelines to describe each component of the models. For example, to express a property of a LEL symbol, the natural language structure "An x has a y" could be used, thus simplifying type attributes identification. There are some works that use controlled natural language [24]. Others, apply linguistic analysis ([4, 7, 8, 11]) or most sophisticated approaches (like CIRCE [1]). These works provide powerful information extraction techniques, maintaining the freedom of expression of natural language. Considering the importance that requirements models used in our approach give to stakeholders' active participation in the requirements definition process, we will incorporate linguistic analysis when developing a tool to implement our strategy.

5 VALIDATING THE RSL SPECIFICATION

The aim of validating the RSL specification is to check that we have written the right specification, i.e. that we have met the requirements. As we try to make the initial specification a contract between software engineers and stakeholders, the validation of the specification turns to be a very important step. Verification is concerned with checking whether the final implementation conforms to the initial specification. As it assumes the correctness of the initial specification, it must be done after validation. Together, validation and verification help assure we are "writing the right specification right". Validation needs requirements traceability, in order to relate them to where they are met either in the initial specification or in a later development. Once we are sure a requirement has been captured, we use verification to control whether it remains captured.

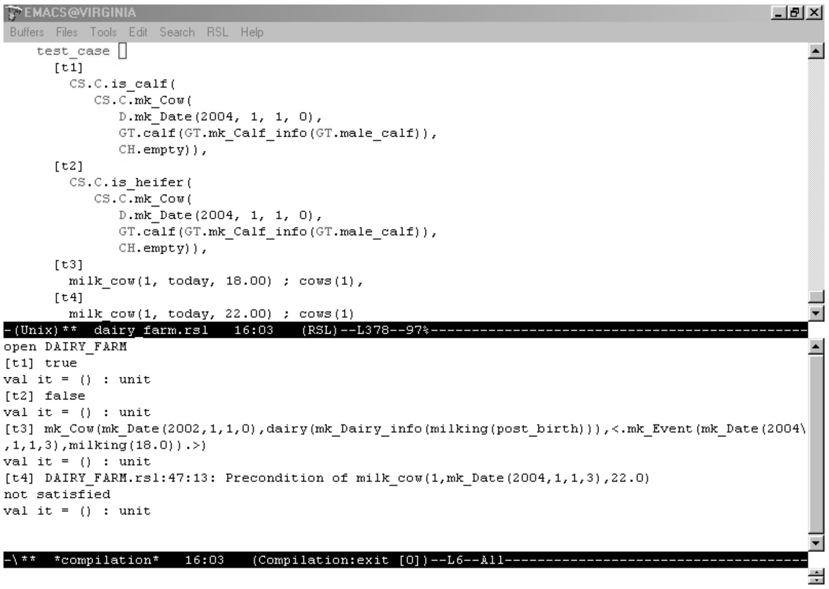
It is important to remark that the construction process of each natural language model used in our proposal has its own validation and verification stages. Previous to the application of the derivation strategy for the RSL specification presented in Section 4, natural language models were validated and verified. The LEL and the scenario model were verified following an inspection-based process [12, 16] while a more informal process was chosen for the business rules model [17]. After verification, all models were validated with stakeholders trough interviews.

To validate the RSL specification we considered the validation techniques proposed in [10] and we selected a combination of prototyping with system tests. With these techniques we could take advantage of the translators already implemented as part of the RAISE tools, such as the SML translator, thus minimising the development costs for the prototype. The SML translator maps a specification in RSL to the functional programming language Standard ML [28], giving as result a first prototype of the specification. RSL has a very rich set of features but not all of them can be translated into a functional programming language, like SML. RSL elements such as abstract types, axioms, post expressions, and implicit values and implicit functions cannot be currently translated into SML. So, sometimes it would be necessary to make some refinements in order to get a concrete RSL specification which could be translated into SML. A complete description of this translator can be found in [30] and, as the rest of the RAISE tools, it can be downloaded from UNU/IIST's web site.

The prototype obtained using the SML translator will not only help us in checking the specification against requirements, but it may also assist in clarifying the real requirements for the system, as the stakeholders may participate in this validation task. By running the prototype with appropriate test cases, stakeholders may find it easier to discover problems with poorly understood requirements.

5.1 Validating Our Specification

The initial specification for the Milk Production System derived applying the strategy described in Section 4 is an applicative one, i.e it is written in terms of definitions and applications of functions. Besides, it may have some abstract type and function definitions. So, to make use of the SML translator, we did a quick and simplified refinement of the abstract types we found to obtain a concrete applicative version of the derived specification. For some of the types, as we did not have enough information, we gave a temporary definition which could be replaced later by a more appropriate one.



```
    test_case []
      [t1]
        CS.C.is_calf(
           CS.C.mk_Cow(
              D.mk_Date(2004, 1, 1, 0),
              GT.calf(GT.mk_Calf_info(GT.male_calf)),
              CH.empty)),
      [t2]
        CS.C.is_heifer(
           CS.C.mk_Cow(
              D.mk_Date(2004, 1, 1, 0),
              GT.calf(GT.mk_Calf_info(GT.male_calf)),
              CH.empty)),
      [t3]
        milk_cow(1, today, 18.00) ; cows(1),
      [t4]
        milk_cow(1, today, 22.00) ; cows(1)
-(Unix)**  dairy_farm.rsl   16:03   (RSL)--L378--97%-------------------------------------------
open DAIRY_FARM
[t1] true
val it = () : unit
[t2] false
val it = () : unit
[t3] mk_Cow(mk_Date(2002,1,1,0),dairy(mk_Dairy_info(milking(post_birth))),<.mk_Event(mk_Date(2004\
,1,1,3),milking(18.0)).>)
val it = () : unit
[t4] DAIRY_FARM.rsl:47:13: Precondition of milk_cow(1,mk_Date(2004,1,1,3),22.0)
not satisfied
val it = () : unit


-\**  *compilation*   16:03   (Compilation:exit [0])--L6--All-----------------------------------
```

Fig. 2. Test cases execution

In addition, we defined an appropriate set of test cases in order to run the specification with them and check if the specification did what was required. Scenarios were of great help when designing test cases. The goal of a scenario contains the aim to be reached in the domain after performing the scenario. Then, to validate each function in the specification we suggest going to the scenario that motivated its definition, and analysing the goal to define one or more test cases. Besides, the scenario context may help define appropriate test cases to check partial functions. Test cases are always evaluated in order of definition, and this is particularly useful for imperative specifications with variables to store information [10]. As the information stored as a result of one test case is available for the next one, it is possible to test scenarios step-by-step by using a sequence of test cases. To achieve this, we formulated a concrete imperative specification from the concrete applicative one. Figure 2 presents the execution environment of RAISE tools, showing the definition of some test cases and the results of their execution for the Milk Production System specification.

6 CONCLUSIONS AND FUTURE WORK

In this paper we proposed a natural language-based strategy to be used in the first stages of development using the RAISE method. We defined a concrete and detailed four-step process that could be applied to any domain, to derive an initial formal specification in RSL from LEL, scenarios, and business rules. In this way, we could take profit of informal descriptions reducing the gap between them and the final RSL specification. In addition, we also take advantage of all the time and effort the definition of requirements and business model consumes.

By using our strategy, the effort to define requirements models is worth doing because, though partially, they could be later mapped onto a formal specification. The LEL provides structural features of the relevant terms in the UofD, thus limiting the definition of types to those that correspond to significant terms. Using the behavioural description represented in the scenarios, it is possible to identify the main functionality to model in the specification. Business rules allow to adjust the specification to reflect the policies of the organization.

The strategy gives as a result a set of modules hierarchically structured, that can be mapped onto a layered architecture by describing the structure of modules using the Layers pattern. This architecture is the basis to start applying the steps of the RAISE method and provides the specific properties all its developments should have. This means that, for example, any implementation or extension development step should preserve the layers and the relationships among them. The use of a layered architecture is particularly useful when designing complex systems, because it facilitates and encourages not only reuse but also separate and stepwise development.

As the heuristics we defined closely followed the principles the RAISE method proposes, the initial specification derived could be developed into a concrete one according to the steps provided by the RAISE method. With this concrete specification, the SML translator could be used to obtain a prototype and get a feeling of what the specification really does.

Concerning future work, we plan to improve our strategy by refining and completing the heuristics presented in this paper. One direction to address is the inclusion of linguistic approaches to achieve a better processing of the information as we mentioned in Section 4.5. In [20] we described a first approach to track traceability relationships. Although they contain the information in a "traced-to" way (for example, they show how a LEL symbol is modelled with a type or a function), the "traced-from" relationships could be added by including appropriate comments in the RSL specification derived. However, a more detailed and deeper analysis should be made because traceability relationships are not always one-to-one.

A complete automatic derivation is by no means possible, as LEL, scenarios, and business rules contain all the necessary and unavoidable ambiguity of the real world, while the specification contains decisions about how to model this real world. However, tool assistance is of great importance in the derivation process. We have developed a tool that implements the heuristics of derivation concerning LEL and scenarios. This tool is integrated with the RAISE tools, thus giving assistance in the RSL specification complete development process. We plan to add the heuristics related to business rules as well as to track traceability.

REFERENCES

[1] Ambriola, V.—Gervasi , V.: On the Systematic Analysis of Natural Language Requirements with CIRCE. Automated Software Engineering Journal, Springer-Verlag, Vol. 13, 2006, No. 1, pp. 107–167.

[2] Berry, B.—Bucchiarone, A.—Gnesi, S.—Lami, G.—Trentani, G.: A New Quality Model for Natural Languages Requirements Specifications. In: Proceedings of the Twelfth International Workshop on Requirements Engineering: Foundation for Software Quality, Luxembourg, 2006.

[3] Bjorner, D.: Software Engineering: A New Approach. Lecture Notes. Technical University of Denmark, 2000.

[4] Bryant, B. R.—Lee, B.—Cao, F.—Zhao, W.—Burt, C.—Raje, R.—Olson, A.—Auguston, M.: From Natural Language Requirements to Executable Models of Software Components. In: Proceedings of Monterey Workshop on Software Engineering for Embedded Systems: From Requirements to Implementation, 2003, pp. 51–58.

[5] Buschman, F.–Meunier R.–Rohnert, H.–Sommerlad, P.–Stal, M.: Pattern-oriented Software Architecture. John Wiley and Sons, 1996.

[6] Dang Van, H.—George, C.—Janowski, T.—Moore, R.: Specification Case Studies in RAISE. Springer-Verlag, 2002.

[7] Díaz, I.—Pastor, O.—Moreno, L.—Matteo, A.: A Requirements Engineering Linguistic Approximation for OO-Method (Una Aproximación Linguística de Ingeniería de Requisitos para OO-Method). Proceedings Workshop Iberoamericano de Ingeniería de Requisitos y Desarrollo de Ambientes de Software. Peru, 2004, pp. 270–281.

[8] Juristo, N.—Moreno, A.—Lopez, M.: How to Use Linguistic Instruments for Object-Oriented Analysis. IEEE Software, May–June, 2000, pp. 80–89.

[9] George, C.: RAISE Tools User Guide. Research Report 227. United Nations University/International Institute for Software Technology (UNU-IIST), Macau, 2001.

[10] George, C.: Introduction to RAISE. United Nations University/International Institute for Software Technology, Macau, Technical Report 249, 2002.

[11] Gervasi, V.—Nuseibeh, B.: Lightweight Validation of Natural Language Requirements. Software Practice and Experience Journal, John Wiley,& Sons, Vol. 32, 2002, No. 2, pp. 113–133.

[12] Kaplan, N.—Hadad, G.—Doorn, J.—Leite, J.: Language Extenden Lexicon Inspection (Inspección del Lexico Extendido del Lenguaje). In: Proceedings III Workshop on Requirements Engineering, WER 2000, pp. 70–91.

[13] Lee, B.—Bryant, B.: Prototyping of Requirements Documents Written in Natural Language. In: Proceedings of SESEC 2002, the 2002 Southeastern Software Engineering Conference, 2002.

[14] Leite, J.—Leonardi, M. C.: Business Rules as Organizational Policies. In: Proceedings of IEEE Ninth International Workshop on Software Specification and Design. IEEE Computer Society Press, Japan, 1998, pp. 68–76.

[15] Leite, J.—Hadad, G.—Doorn, J.—Kaplan, G.: A Scenario Construction Process. Requirements Engineering Journal, Springer-Verlag, Vol. 5, 2000, No. 1, pp. 38–61.

[16] Leite, J.—Doorn, J.—Hadad, G.—Kaplan, G.: Scenario Inspections. Requirements Engineering Journal, Vol. 10, 2005, No. 1, pp. 1–21.

[17] Leonardi, M. C.: A Strategy for Object Conceptual Modelling based on Natural Language Requirements Models (Una Estrategia de Modelado Conceptual de Objetos basada en Modelos de Requisitos en Lenguaje Natural). Master Thesis. Facultad de Informática, Universidad Nacional de La Plata, Argentina, 2001.

[18] Leonardi, M. C.—Leite, J.: Using Business Rules in Extreme Requirements. Lecture Notes in Computer Science 2348. Advanced Information Systems Engineering, 14th International Conference CAISE 2002. Springer Verlag, 2002, pp. 420–435.

[19] Leonardi, M. C.: Business Modeling with Client-Oriented Requirements Strategy. Encyclopedia of Information Science and Technology I–V, Dr. Mehdi Khosrow-Pour (Ed.), IRM Press, Idea Group Inc, 2005, pp. 339–344.

[20] Mauco, M. V.: A Technique for an Initial Specification in RSL. Master Thesis. Facultad de Informática, Universidad Nacional de La Plata, Argentina, 2004.

[21] Mauco, M. V.—Riesco, D.: Integrating Requirements Engineering Techniques and Formal Methods. Encyclopedia of Information Science and Technology, Volume I–V. Idea Group Inc., USA, 2005, pp. 1555–1559.

[22] Mauco, M. V.—Leonardi, M. C.—Riesco, D.: Formalizing a Derivation Strategy for Formal Specifications from Natural Language Requirements Models. In: Proceedings of 5th IEEE International Symposium on Signal Processing and Information Technology, Greece, 2005, pp. 646–651.

[23] Nuseibeh, B.—Easterbrook, S.: Requirements Engineering: A Roadmap. In: Proceedings of the Conference on The Future of Software Engineering, ACM, 2000, pp. 35–46.

[24] Schwitter, R.—Fuchs, N.: Attempt from Specifications in Controlled Natural Language towards Executable Specifications. In: Proceedings of Joint International Conference and Symposium on Logic Programming, 1996, pp. 536–549.

[25] Semantics of Business Vocabulary and Business Rules Specification (SBVR) OMG Adopted Specification dtc/06-03-02. Available on: `http://www.omg.org/technology/documents/spec_catalog.htm`.

[26] The RAISE Language Group: The RAISE Specification Language, BCS Practitioner Series, Prentice Hall, 1992.

[27] The RAISE Method Group: The RAISE Development Method, BCS Practitioner Series, Prentice Hall, 1995.

[28] Standard ML of New Jersey. Available on: `http://www.smlnj.org`.

[29] van Lamsweerde, A.: Formal Specification: a Roadmap. In: Proceedings of the Conference on The Future of Software Engineering, ACM, 2000, pp. 147–159.

[30] Wei, K.—George, C.: RSL to SML Translator. Technical Report No. 208, United Nations University/International Institute for Software Technology, Macau, 2000.

María Virginia Mauco is a computer science assistant professor in Universidad Nacional del Centro de la Provincia de Buenos Aires from Argentina. She is member of the Computer Science Department. She has a Masters degree in Software Engineering from Universidad Nacional de La Plata, Argentina. Her main research interests include software development methodologies, requirements engineering, and formal methods. She has been member of the program committee of national and international conferences related to software engineering.

María Carmen Leonardi is a computer science assistant professor in Universidad Nacional del Centro de la Provincia de Buenos Aires from Argentina. She is member of the Computer Science Department. She has a Masters degree in Software Engineering from Universidad Nacional de La Plata, Argentina. Her main research interests include software development methodologies, requirements engineering, and model-driven development. She has been member of the program committee of national and international conferences related to software engineering.