A Language Proposition for System Requirements

Benoît Lebeaupin*, Antoine Rauzy† and Jean-Marc Roussel‡
*LGI, École Centrale Paris, Université Paris-Saclay, Châtenay-Malabry, France
benoit.lebeaupin@centralesupelec.fr
†Departement of Production and Quality Engineering, NTNU, Trondheim, Norway
antoine.rauzy@ntnu.no
‡LURPA, ENS Cachan, Université Paris-Saclay, 61 Avenue du Président Wilson, 94230 Cachan, France
jean-marc.roussel@lurpa.ens-cachan.fr

***Abstract*—Natural language is currently the basis of the majority of system specifications, even if it has several drawbacks. In particular, natural language is inherently ambiguous. In this article, we propose a way to complete the natural language text of requirements by giving a formal syntax to this text. We introduce and use an example to illustrate our ideas.**

I. Introduction

According to reports such as the one by Schmidt et al. [1], an important part of IT projects fail or exceed cost or time constraints because of reasons related to stakeholder needs (such as incomplete, unrealistic or changing requirements). Arguably, this is also the case for systems engineering in general. For that reason, requirements engineering constitute an important part of the design process of systems. Additionally, life-critical systems are oversaw by regulatory bodies which require companies to perform requirements engineering.

In the context we are interested in, a requirement on a system is a sentence defining a property, for example "`The weight of the system shall be less than 100kg`[1]".

Most requirements are written using natural language (NL) text even if it has several drawbacks for requirements engineering:

- We want to communicate requirements with the less distortion possible between stakeholders. But unconstrained natural language is inherently ambiguous.
- We cannot use natural language requirements for automated reasoning: for example, it's hard to detect automatically that two requirements contradict themselves.
- Model-Based Systems Engineering (MBSE) is a developing trend, but it is not clear how to articulate MBSE and NL requirements.

We however think that requirements will continue to be written using NL, but it is not necessary that we use only natural language to specify requirements. We propose to complete the natural language requirements, first by defining a syntax for requirements, second by using artifacts such as architectural models and linking them to the text of requirements. We will only talk about the former in this article. Our contributions are a requirements language proposition and how to effectively use it.

[1]Sentences written in `this font` are requirements or parts of requirements, usually inspired by an industrial specification document. They were modified to make them generic for obvious intellectual property issues

In the next section, we present a review of related works. In the section 3, we introduce an example of system which will serve as illustration in the rest of this article. In section 4, we will discuss generally about requirements and requirements engineering and present the outline of the proposed language. We then illustrate the proposed idea in sections 5 with examples of requirements. Finally, we conclude this paper and give an outlook in section 6.

II. Related Work

Requirements engineering (RE) covers a lot of different processes and here we are mostly interested by the "specifying" step of requirements engineering. This "specifying" step is the process of documenting clearly and precisely the requirements in a requirements document. Even then, "requirement" does not mean the same thing for everybody: in this article, we focus on concrete and relatively low level requirements of a physical system, for example a component in an airplane. These requirements have few things in common with the high-level "goals" of Goal-Oriented Requirements Engineering (GORE) [2].

Additionally, since requirements engineering was first developed for software systems, a large part of RE works concern software-only systems. However our work concerns physical systems which may include software. What difference does it make? First, software systems are usually considered as discrete event systems and various RE methods are based on this (such as the transition axiom method [3], or Event-B [4]). Yet the inputs and outputs of a motor or the ambient temperature for example, are better modeled as continuous variables. Another difference is that physical systems directly interact with the physical world, which adds less predictable and less formalizable parameters.

The Stimulus tool [5], developed by Argosim, aims to improve real-time requirements for embedded software by allowing to simulate these requirements. Stimulus focuses on real-time requirements while our work aims to include different types of requirements, including real-time ones. That being said, our work and the work by Argosim have relatively similar principles, notably the link between models (architecture models and state machines) and requirements. However, requirements need to be completely formal to be

simulated with Stimulus, while we aim for a less powerful, but easier (and less expensive), goal.

One approach for writing requirements focuses on the use of templates (also called "boilerplates") [6] [7]. An example of template could be "`The <system> shall <do something> with <a given level of performance> in <a given context>`". After choosing an adequate template, the writer fills in the blanks with the relevant information, for example "`The <ABS> shall <release braking pressure> in <less than 100ms> in <case of a risk of wheel lock>`".

Fundamental RE books often mention templates [8] [9], but they also warn the reader of their limitations and usually advice that the use of templates should not be mandatory. The main problem is that templates tend to be too "rigid" and prevent the writer from expressing what he needs. One solution would be to add more templates but that requires to manage them, something which can become difficult when there are too many templates. On the other hand, if the templates are more generic, they allow more expressiveness, but they are also less useful for guiding writers and reducing ambiguity.

In the work of Fraga et al. [7], the templates can be generated from ontologies, which allows more flexibility. The problem of creating and managing these ontologies can however become complicated.

Templates are a restriction of natural language, and as such are part of Controlled Natural Languages (CNLs) [10]. Essentially, the goal of our work is to define a language for requirement which should be 1) less ambiguous than natural language (NL) 2) close enough to NL that using it would not require extensive training. This is also the goal of CNLs, which can be separated into two types: the "technical" CNLs, to improve human-human communication (ex [11]), and the "logic-based" CNLs, to improve the human-computer communication (ex [12]).

Some works [13] [14] aim to integrate requirements and models by automatically translating NL requirements into models using Natural Language Processing (NLP) methods. These approaches' reliance on automatic NLP translation requires starting from textual requirements which are already using a very precise language and are highly structured and regular. The context of the work by de Almeida Ferreira [13] is also relatively different: since it focuses on software development, the authors consider that the requirements they work with can come from stakeholders of various backgrounds, including completely non-technical backgrounds. We mainly studied aeronautical systems, particularly systems which have an impact on the safety of passengers. In this context, requirements come mostly from sources such as other technical specifications (for example the specification of the upper-level system) or legal documents.

III. Motivating Example

An ABS, or anti-lock braking system, is a mechatronic system aiming to prevent the wheels of a vehicle from locking up during braking. Wheel locking should be avoided because it means wheels are slipping on the road, which leads to several problems:

- uncontrollability of the vehicle
- decreased adhesion on most surfaces
- possibility of burned or burst tires (particularly for aircrafts)

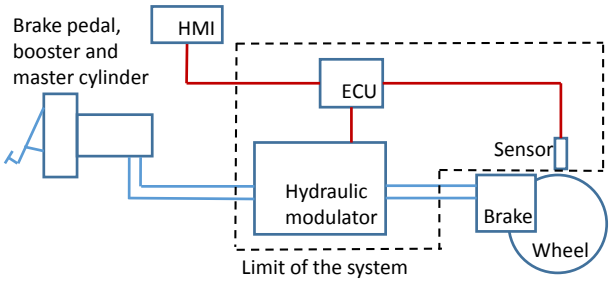

Fig. 1. The Anti-Lock Braking System and its context

The principle of the ABS is that when it detects certain conditions indicating an imminent wheel lock up, it releases the brake to avoid the lock up. Normal braking is resumed once the dangerous conditions are not satisfied anymore. This braking/release cycle can be repeated several times per second.

In this work we will consider an ABS installed in an automobile. In modern cars, an anti-lock braking system is usually installed for each wheel. For the sake of simplicity, in this work, we will only consider one ABS acting on one wheel, independently from the other wheels and anti-lock braking systems. Additionally, we consider that the activation of the ABS is binary: either it simply transmits the braking fluid to the brakes as if the ABS was not there, or it sets the braking pressure to zero.

In fig. 1, we present the environment of the ABS we consider and a very basic decomposition of the system. Compared to a classic braking circuit, when the electronic control unit (ECU) detects a risk of wheel lock-up, the hydraulic modulator shuts off the hydraulic connection between the master cylinder and the brake cylinder and depressurize the latter portion. We will consider that the system detects when the brake should be released by calculating the wheel slip, i.e. the relative speed between a point on the exterior of the wheel and a point on the road. This value is calculated using the rotational speed of the controlled wheel and the speed of the vehicle, which is itself approximated using the rotational speeds of the other wheels. The rotational speeds of the wheels are measured using sensors. The ECU also communicates with other systems, such as a warning lamp on the car dashboard.

The behavior of the system is the following:

- The system is deactivated if a failure is detected or if the activation is somehow not demanded
- When the ABS is deactivated, the braking circuit behavior is identical to the same circuit without the ABS (the pressure in the brake cylinder and the pressure at the output of the master cylinder are the same)

- When the ABS is activated:
  - And when the brake pedal is not pressed, the pressure in the brake cylinder is the same as the pressure at the output of the master cylinder, i.e. 0 bar
  - And when the brake pedal is pressed:
    * And when the conditions indicating a risk of wheel lock up are false, the pressure in the brake cylinder is the same as the pressure at the output of the master cylinder
    * And when the conditions indicating a risk of wheel lock up are true, the pressure in the brake cylinder is 0 bar, regardless of the pressure in the master cylinder

The system is able to change between braking and release several times per second. The system also monitors its status and reports it to a central computer in the car.

Reasons for Using the ABS as Example

The ABS, in its most basic configuration, is a relatively simple and understandable system. However, it can serve as an example of more complex systems. It comprises hardware parts and software parts, which is increasingly the case in recent systems. It is also a system which interacts directly mostly with other technical systems, rather than with human users. This context allows to have a relatively predictable and formalizable environment, which allows to have more efficient methods for designing systems and their specifications.

The ABS is complex enough to be decomposed into several subsystems, which make it an interesting example in a world where systems are more and more complex. This increasing complexity leads project managers to decompose systems in smaller parts to keep them understandable by a single team.

The anti-lock braking system is also a life-critical system. Non critical systems are not necessarily designed using requirements engineering since it can be relatively time-consuming. However, methods such as requirements engineering are required for critical systems. Even if all systems can benefit from (good) requirements engineering, the context of physical, life-critical systems is particular. For a such systems, we would expect requirements to change relatively slowly compared to, for example, requirements for a smartphone application developed using an agile method.

IV. Theoretical and Conceptual Developments

A. About Requirements

We decompose systems into subsystems to keep a manageable complexity so that different teams can focus on different subsystems. This goal defines the limits between subsystems: we will decompose a system according to a customer/supplier point of view, whether the supplier and customer are in the same company or not. This point of view is not necessarily the same as a "functional" point of view, or even an "physical" point of view.

Requirements are a communication tool: their goal is that the supplier understands what the customer wants. Of course, as all communication tools, they distort the information, both when writing requirements and when reading them.

As a high level goal, these distortions should be the smallest possible: ideally, the writer of requirements wants to be able to express exactly what he wants, and wants the reader to understand exactly what he, the writer, meant. This ideal goal can be decomposed into sorts of guidelines we find in RE fundamentals books [8] [9] on what the requirements should be like, for example:

- Requirements should be relatively short to be more easily understandable
- Potentially ambiguous vocabulary and syntax should be avoided
- One requirement should not contradict another requirement

Other guidelines stem from, for example, the necessity of managing requirements (for example, the requirements must be uniquely identified), but we will focus here on the ambiguity aspect.

B. Natural Language

The goal of requirements is that everyone should understand the same thing when reading them, in other words, requirements should not be ambiguous. It could seem surprising that most requirements are written using natural language: Natural languages are ambiguous, so why do we use them for requirements?

One of the answers, quite simply, is that other possible languages, such as formal languages, are less ambiguous only for those who know them. One advantage of natural languages is that everyone know at least one of them. Additionally, organizations usually have a common natural language. On the other hand, formal languages are generally known only by a small minority of people and requirements should be readable by people from potentially widely different backgrounds.

Another advantage of natural language is that it allows to express extremely varied things, whereas formal languages can be severely restricted. Even if specifications are only an abstraction of real systems, the systems we want to specify exist in the real world. The authors are convinced that, for any given formal language, a non-negligible proportion of requirements in an industrial specification are practically impossible to write using this language.

We thus consider that, in our context, natural language must be part of any realistic method for requirements engineering. However, we think that natural languages should not be the only way to express requirements.

C. Additional Information to Disambiguate Natural Language Text

We saw that the goal of requirements is communication. Misunderstandings of requirements are an important source of problems and the sooner these misunderstandings are detected, the less costly it is to the project [15]. Therefore, we want to help engineers avoid writing bad requirements. Requirements can be bad for different reasons, but in this work we will focus

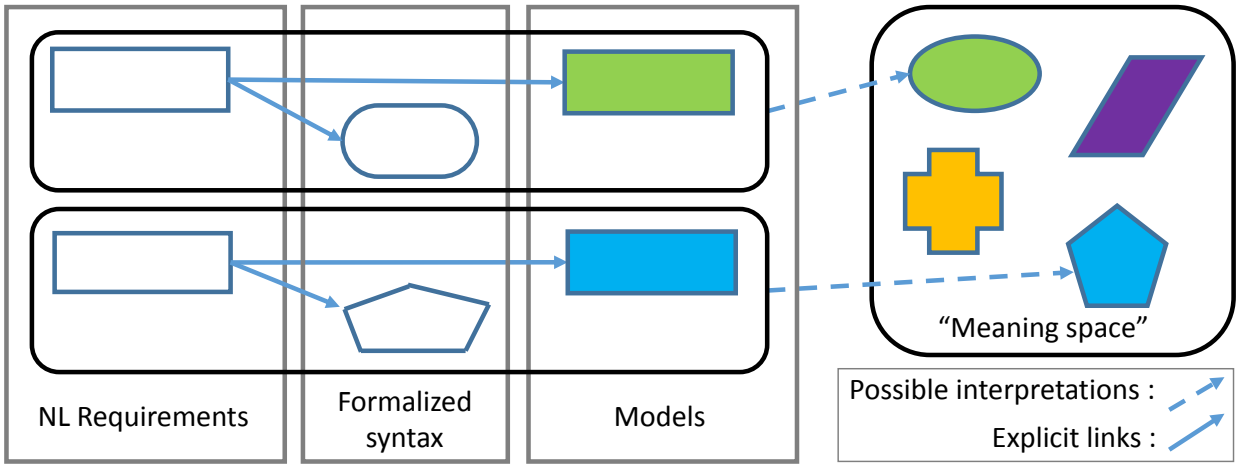

Fig. 3. Models and formal syntax as a tool against ambiguity

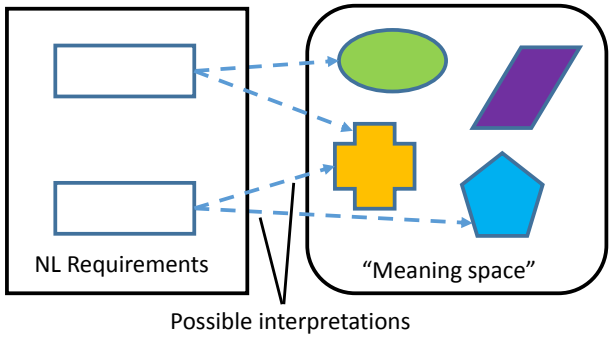

Fig. 2. Representation of ambiguity

mainly on the natural language text of requirements and on reducing the ambiguity of this text.

Let us consider a "meaning" space as in fig. 2. When someone reads a text, she associates (interprets) the text with a part of the meaning space. An ambiguous text is a text which can be associated with different parts of the meaning space by different readers. Our goal is to have a one-to-one correspondence between text and meaning. Of course, this "meaning" space does not exist physically, or even as bits in a computer, and we cannot refer directly to it, but we can restrict possible interpretations by providing a "preferred way" to interpret a text.

As illustrated in fig. 3, the requirements are linked to syntactic information and to model elements. This additional information precises the text in the requirements. If the text, the syntactic information and the models are consistent and correct, the additional information should allow readers to choose the correct meaning when reading the specification. Potential ambiguity, such as one text with several meanings or several texts sharing one meaning, is then hopefully removed. Concretely, a requirement will be written as a succession of hyperlinks: some will refer to model elements and others will refer to information on the syntax of the requirement.

In this article, we will focus on the syntax part. What we are describing here is a logic-based Controlled Natural Language, but seen from another point of view. The formal language, instead of being the base of the means of communication, is seen as a way to disambiguate the syntax of a text by completing it with additional information. In another article, we will present how to use models as references to help disambiguate the lexical parts of requirement.

D. Formalizing the Structure of Requirements

We sought for the specific rules governing the structure of requirements. Requirements should respects the rules of natural language to be easily understandable, but we can find other rules that should be respected by requirement writers.

A requirement on a system is a property, a property which should be true when the system is installed. If the property is false, it means the system does not respects the requirement. Requirements, at least in the context we study, should be precise enough so that they are either true or false. This means that requirements are, from a computer science point of view, Boolean-valued functions.

Additionally, it is usually possible to decompose requirements in smaller fragments: for example, the requirements we studied were often in the form of implications: "`When the power supply is lower than 10mW, the brake pressure shall be equal to the master cylinder pressure`".

"`the power supply is lower than 10mW`" is called the antecedent and "`the brake pressure is equal to the master cylinder pressure`" is called the consequent. The implication is also called material conditional. For a Boolean equation $R = A \implies B$, where A and B are Boolean-valued functions, R is false if and only if A is true and B is false: $A \implies B$ is equivalent to $\neg A \vee B$.

In the previous example, we can continue the decomposition: "`the power supply is lower than 10mW`" is a Boolean-valued function and is composed of "`the power`

`supply`", "`is lower than`" and "`10mW`". We can consider that it is not necessary to decompose "`the power supply`" further: we arrived at an atomic expression. Additionally, when we study the words in requirements, and the models they refer to, we notice that "`the power supply`" is a reference to a well defined and delimited atomic concept.

To describe the structure of requirements, we propose a type system similar to those existing in programming languages. A requirement is represented as a double (S, P), where P is the property the customer wants to be true and S is the system to which we request that property. In the specification of a system called T, S is usually the same as T, but not always: we may want to express a property on a subsystem of T rather than on the complete system.

We will recursively define the set of terms from a set F of functions:

- 0-ary functions (constants) are terms.
- If f is a n-ary function, $n > 0$, and $(t_1, \ldots, t_n)$ are n terms; $f(t_1, \ldots, t_n)$ is a term.

Let us now introduce types: each n-ary function of F have one output type and n input types. For a term to be well-typed, the types it contains must respect a constraint satisfaction problem.

The type of a term is the output type of its root function: the type of a term of the form $f(t_1, \ldots, t_n)$, where f is a n-ary function, is the output type of f. The constraints on the types depend on the function f. Additionally, if a requirement is a double (S, P), S should be of type "system" and P of type "Boolean".

This decomposition is relatively similar to the grammatical analysis of a NL sentence, which yields parse trees: a sentence ("The chicken crosses the road") can be composed by a noun phrase ("The chicken") and a verb phrase ("crosses the road"). A verb phrase can itself be composed of a verb ("crosses") and a noun phrase ("the road"). A noun phrase can be composed of a determiner ("the") and a noun ("road"), etc.. The decomposition is also quite close to the Abstract Syntax Trees used in computer science.

For our previous example "`When the power supply is lower than 10mW, the brake pressure shall be equal to the master cylinder pressure`", we can define the tree in fig. 4.

We detail some of the functions used:

- "Implication" is a binary function, with a Boolean output type and two inputs of type Boolean.
- "Lower than" is a binary function, with a Boolean output type and two inputs of the same type (we do not want to define a "Lower than" function for each physical unit (N, m, W...))

We consider that a measurement (a number-unit pair, for example: 10mW, 15 meters) is constructed using a unit function (there "mW" or "meter") which takes a number as argument (10, 15).

This typed language allows to formalize the syntax of a part of the requirements, but it is not enough to formalize all of them.

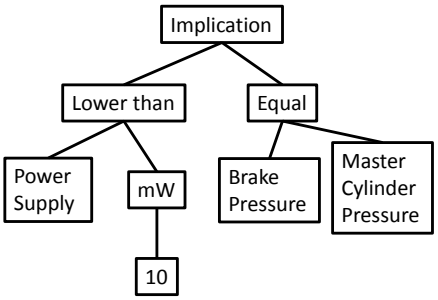

Fig. 4. Syntax tree of a requirement

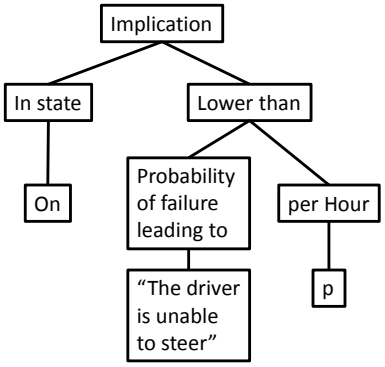

Fig. 5. Syntax tree with a non formalized leaf

To avoid limiting expressiveness, we need to allow engineers to write free text requirements or fragments of requirements. To do this while still keeping the syntax we described, we propose that the engineer can write the free text he wants to write, but he needs to add a type to this free text fragment.

For example, we consider the requirement "`While in state On, the probability of a failure leading to the driver being unable to steer shall be lower than p per hour`".

If we cannot or do not want to formalize "`the driver being unable to steer`", we can simply let this free text in the requirement and give it the type Boolean (we consider here that either the driver is able to steer or he is not). We can however decompose the rest of the requirement as seen in fig. 5.

It is also possible to write an entire requirement as free text, without any syntax information. Obviously, such a requirement will not appear in a query depending on syntax information, for example a query selecting the antecedent of each requirement if it exists.

E. Testing Properties on Requirements and Sets of Requirements

We proposed to add a formal structure to requirements. The goal is to reduce ambiguity by providing a trusted way to interpret the text: for example, a formal syntax will allow to know exactly the scope of a conjunction ("and", "or"). We

can also use this syntax information to realize automatic tests, queries and reasoning.

There are various criteria on requirements and on sets of requirements we would like to check. Lists of quality criteria are defined in norms (such as IEEE Std. 29148-2011 [16]), industrial standards (such as the ones gathered for the CESAR project [17] or fundamental RE books. Some of these criteria are covered by other methods (such as traceability) or cannot be checked automatically, but other are more suitable for our method. We present a few tests which can be done automatically on formalized specifications.

Some of these tests aim to validate consistency criteria. Inconsistency can happen at different levels. We do not try to "understand" automatically the meaning of a requirement, so we cannot detect that two or more requirements have incompatible meanings. However, we can detect inconsistency at a syntactic level, such as trying to compare a force with a torque.

Other criteria focus on completeness: even if it is not possible to detect automatically that a specification does not omit anything, we can detect potential problems. We can use syntactic information to extract particular requirements or parts of requirements. For example we can select the antecedent (if it exists) for each requirement. We wrote a pattern search to that end.

V. Examples of Requirements

A. First Example

Let us take a high level goal of the ABS, such as "When the system is not activated, the system shall not prevent braking". This kind of goal is useful to better understand the context of the specified system, but if we want requirements to be precise and non-ambiguous, a goal like this is not a requirement. A more explicit requirement inspired from this goal could be "`In state Off, when the master cylinder pressure is lower than 50 bar, the brake pressure shall be equal to the master cylinder pressure`". (We assume that a master cylinder pressure lower than 50 bar constitute expected behavior of the environment and that everything else is considered as anomalous behavior. What the system shall do in case of anomalous behavior can be specified in other requirements.)

Of course, since this is the point of this work, the requirement "`In state Off, when the master cylinder pressure...`" is not limited to this textual component. We can detail the syntax of this requirement as in fig. 6. Here, the elements of the tree are generic functions, they could be found in any specification, except "`Off`", "`Master Cylinder Pressure`" and "`Brake Pressure`". We identify these three text fragments as specific references to the architecture of the system (for "`Master Cylinder Pressure`" and "`Brake Pressure`") or to a description of the behavior of the system (for the state "`Off`"). More details about these references will be given in a coming article.

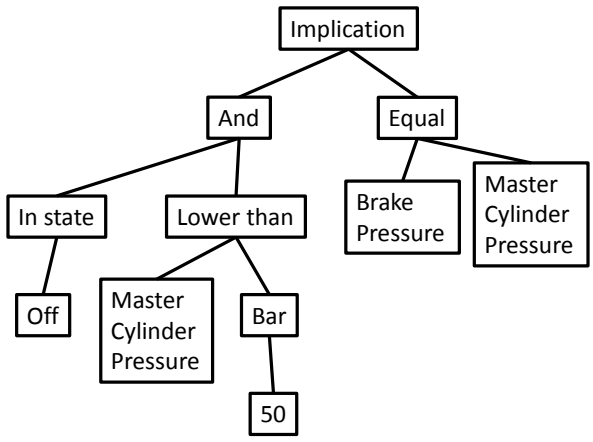

Fig. 6. Syntax tree of another requirement

B. Second Example

In specifications, we can find requirements similar to "`The system shall not be damaged by a maximum master cylinder pressure of 60 bar`". Some expressions such as "`shall not be damaged by`" are found repeatedly in industrial specifications. These expressions are currently used and engineers have no problem understanding them, so we do not consider them particularly ambiguous. We can include them directly in the language by creating syntax functions for these expressions: here, "Not damaged" takes a system as argument and outputs a Boolean.

What "Not damaged" exactly means can be defined in the preamble of the specification or in an external document or norm if it is needed.

C. Third Example

We consider the goal "We do not want failures which would prevent the driver from braking". We can express a detailed safety requirement such as "`The probability of a non-detected failure leading to the system being unable to apply a brake pressure of more than 20 bar in state Braking shall be lower than` 10^{-8} `per H`"[2]. Here too, "the probability of a non-detected failure leading to X" was often found in the industrial specifications we studied and we defined a corresponding syntax function.

The "20 bar" limit is chosen as a threshold: under 20 bar we consider that the driver cannot brake. This limit can be seen as arbitrary, but we need a limit to be able to write a complete and precise requirement. One way to avoid this threshold effect could be to add more requirements (30 bar, 20 bar, 10 bar...) for different levels of failure.

Additionally, here we look only at one dimension, whether pressure is sufficient or not, but we can also add requirements about a temporal dimension. For example we could want to

[2]this requirement does not necessarily cover completely the goal, and we may need other requirements

specify that a delayed output is less dangerous than no output at all, or that transient violations of a threshold are acceptable.

Adding these precisions to a specification requires either writing and managing more requirements or using more expressive ways to describe the properties we want. For example, instead of having one or several threshold and writing a requirement for each of them, we could define a continuous function which would link the acceptable probability of an event happening with the severity of this event. However this approach is not without drawbacks. First, the goal of requirements is communication: we have to ensure that whatever model or artifact we use is understood the same way by everyone, preferably without needing important training. Second, adding precise details is not useful if customer or supplier have no method to test, verify or calculate these details.

Other, Less "Cooperative", Examples

The three previous examples were relatively fit to be analyzed and formalized, however, that is not the case for all real requirements.

D. Fourth Example

"`The system equipment or structural items shall remain within their own design and integration envelop after a failure leading to a partial or total part detachment.`" In this requirement we can identify 3 parts:

- one antecedent: "`after a failure leading to a partial or total part detachment`"
- one consequent: "`remain within their own design and integration envelop`"
- which parts are concerned : "`The system equipment or structural items`"

If it is possible, we could use a Computer Aided Design model to use as a reference for "`design and integration envelop`", but apart from that, formalizing this requirement seems to be complicated.

E. Fifth Example

"`The system shall be designed to minimize the potential for human errors that would significantly reduce safety during maintenance and operation.`" We can identify a condition "during maintenance and operation" but not much more. We note that this requirement is not necessarily a good requirement (it is not exactly precise nor complete). But since it is the kind of requirement we find in industrial specifications, *a priori* they are not useless and engineers want to be able to express such properties.

F. General Remarks on these Last Examples

It may be possible to formalize more the syntax of these last two requirements, but the problem is more whether it would be useful rather than whether it would be possible. maybe we can rewrite these requirements to be perfectly formal, but if the cost of writing this formal definition is too important compared to the gain, formalizing would not be profitable.

From a more general point of view, we do not think that completely formal specifications are practically possible or profitable for systems we studied. However, we think that writing more formalized specifications than the current, natural language-only ones could be useful. We can see the situation as a slider between non-formal and completely formal: different contexts will have different optimal levels of formalization. We need tools to be able to write these different levels of formalization.

We think that one of the advantages of our approach over other works is that it is gradual: contrary to methods such as Event-B [4], we do not need to have a completely formal specification in order to see the benefits of our work. Relatively small steps can already be beneficial: for example, we can write partially formalized requirements or formalize only a part of the specification and get some advantages, such as reduced ambiguity and the ability to realize some automatic tests. Less formalized requirements means less reduction in ambiguity, but also less work.

For these requirements which are not formalizable, or whose formalization would not be profitable, other methods could ease requirements engineering. For example, the requirement "`The system equipment or structural items shall remain within their own design and integration envelop after a failure leading to a partial or total part detachment`" is not necessarily specific to the anti-lock braking system. If all other specifications by the same company include this requirement, it could make sense to have a set of unchanging requirements automatically included in new specifications, or other similar reuse strategies.

VI. Limitations and Perspectives

A. Summary

In this article we present ideas on how to make the text of requirements less ambiguous without adding a major burden to the writers and readers of a specification. In addition to examinations of the relevant scientific literature, we based our work on studies of industrial specifications and discussions with RE practitioners in the industry. We think that it is possible to improve the whole process of RE by defining more precisely the syntax of the text of requirements. We can then use the additional information to realize automatic and quick tests on the requirements.

We experimented the ideas we proposed by modifying an industrial specification and by trying to write a completely new specification for an example system.

B. Coherence Between Natural Language and Formal Structure

A crucial problem of having co-existing natural language and formal language is to ensure the coherence between the information included in these two media. As in any other domain where we have more than one description of the same thing, we have to ensure that these descriptions do not

contradict themselves. When we modify one of them, the change should affect the other descriptions, or at least the impact should be easily tractable.

One way to ensure the coherence is to have one description as the source description, and generate the other from it. This generation should preferably be done automatically to avoid additional work for requirements engineers and to ensure that the two descriptions are actually coherent. This is relatively difficult to do with natural language either as the source or as the generated description:

- If natural language is the source, it means we need to be able to generate formal language from it directly. Some works [13] focus on this issue, but automatic NLP is difficult and not necessarily well developed enough for the applications we envision here.
- If the formal language is the source, it means that the requirements writer needs to be able to read and write formal language. This is an important drawback if we want our method to be relatively easy to use. Additionally, the generated text may feel unnatural to readers.

The principle of logic-based CNLs is to use the formal language as the source and to "hide" this formal language using natural language words. A writer still needs to master, or at least to understand, the underlying formal logic, to know what he can or cannot write.

This coherence problem is essentially unavoidable: we cannot hope to get formal language text automatically from non-formal text. Hopefully, a good requirement editor could help a writer to keep coherent all the elements we want to introduce in addition to the text.

Helping the engineer with writing the formal structure part of requirements will probably be complex. We can look at how other works, particularly concerning CNL, approach this problem. For example, a requirements editor could include an autocomplete program which, once it detects a known word, would write the corresponding syntactic function automatically.

C. Ambiguity of the Lexical Part of Requirements

We also studied the words which are used in requirements. These words, when they are not the functions we defined earlier (implication, and, lower than...), usually refers to concepts peculiar to the specified system. These concepts are usually defined using models. For example, the interface of a system (here for the ABS, elements such as "the Brake Pressure" or "the Master Cylinder Pressure"), are usually defined in an architecture model. In the same way, the desired behavior of a system can be described using a state machine.

In various domains of systems engineering, including requirements engineering, knowing how to use efficiently these models is the subject of active research. We suggest to add, to the words of requirements, explicit links referring to the corresponding models or elements of models. This will help reducing the lexical ambiguity in the requirements, such as one word having two or more possible meanings.

D. Generalization to Other Types of Specifications

Another perspective is to look at how we could apply what we propose to other types of specifications: the industrial specifications we studied concerned different systems, which were however relatively similar (aeronautical context, mixed software/hardware systems). We can wonder how the same ideas applied to different contexts would work. For example, if we are not in an aeronautical context anymore, we obviously need to adapt some elements (FH, flight hour, is not a relevant unit anymore). But the real question is whether the basic principles of what we propose in this work are still sound. We think the construction process we did with the ABS example makes sense, but we could not study actual automotive specifications.

ACKNOWLEDGMENT

The author is supported by the Blériot-Fabre chair, co-directed between CentraleSupelec and Safran.

REFERENCES

[1] R. Schmidt, K. Lyytinen, and P. C. Mark Keil, "Identifying software project risks: An international delphi study," *Journal of management information systems*, vol. 17, no. 4, pp. 5–36, 2001.

[2] A. Van Lamsweerde, "Goal-oriented requirements engineering: A guided tour," in *Fifth IEEE International Symposium on Requirements Engineering*. IEEE, 2001, pp. 249–262.

[3] L. Lamport, "A simple approach to specifying concurrent systems," *Communications of the ACM*, vol. 32, no. 1, pp. 32–45, 1989.

[4] J.-R. Abrial, *Modeling in Event-B: system and software engineering*. Cambridge University Press, 2010.

[5] B. Jeannet and F. Gaucher. (2015) Debugging real-time systems requirements: Simulate the "what" before the "how".

[6] A. Mavin, P. Wilkinson, A. Harwood, and M. Novak, "Easy approach to requirements syntax (EARS)," in *Requirements Engineering Conference, 2009. RE'09. 17th IEEE International*. IEEE, 2009, pp. 317–322.

[7] A. Fraga, J. Llorens, L. Alonso, and J. M. Fuentes, "Ontology-assisted systems engineering process with focus in the requirements engineering process," in *Complex Systems Design & Management*. Springer, 2015, pp. 149–161.

[8] K. Pohl, *Requirements engineering: fundamentals, principles, and techniques*. Springer Publishing Company, Incorporated, 2010.

[9] S. Badreau and J.-L. Boulanger, *Ingénierie des exigences: Méthodes et bonnes pratiques pour construire et maintenir un référentiel*. Dunod, 2014.

[10] T. Kuhn, "A survey and classification of controlled natural languages," *Computational Linguistics*, vol. 40, no. 1, pp. 121–170, 2014.

[11] ASD Simplified Technical English. [Online]. Available: http://www.asd-ste100.org/

[12] N. E. Fuchs and R. Schwitter, "Attempto Controlled English (ACE)," in *Proceedings of the First International Workshop on Controlled Language Applications*, 1996.

[13] D. de Almeida Ferreira and A. R. da Silva, "A controlled natural language approach for integrating requirements and model-driven engineering," in *Software Engineering Advances, 2009. ICSEA'09. Fourth International Conference on.*, 2009, pp. 518–523.

[14] V. Gervasi and B. Nuseibeh, "Lightweight validation of natural language requirements," *Software: Practice and Experience*, vol. 32, no. 2, pp. 113–133, 2002.

[15] J. M. Stecklein, J. Dabney, B. Dick, B. Haskins, R. Lovell, and G. Moroney, "Error cost escalation through the project life cycle," 2004.

[16] "ISO/IEC/IEEE International Standard - Systems and software engineering – Life cycle processes – Requirements engineering," *IEEE Std. 29148-2011*, 2011.

[17] A. Rajan and T. Wahl, *CESAR: Cost-efficient Methods and Processes for Safety-relevant Embedded Systems*. Springer, 2013, no. 978-3709113868.