Towards a Better Integration of Requirements and Model-Based Specifications

Benoît Lebeaupin, Antoine Rauzy

ORIGINAL ARTICLE

Towards a Better Integration of Requirements and Model-Based Specifications

Benoît Lebeaupin[1] | **Antoine Rauzy**[2]

[1]LGI, CentraleSupelec, Gif-sur-Yvette, 91190, France

[2]MTP, Norges Teknisk-Naturvitenskapelige Universitet, Trondheim, 7491, Norway

As of today, most specifications of technical systems still rely on requirements written in natural language. However, this approach is known to be problem-prone, due to the inherent ambiguity of natural languages. On the other hand, fully formal or model-based approaches seem to be out of reach in many practical cases, especially in early design phases of systems.

In this article, we study how to combine in a pragmatic way natural language requirements with models. We propose to keep both formats and to link pieces of text in requirements with elements of models. In other words, corpuses of requirements are managed as hypertexts with links to models. For this approach to be fully efficient, the text of requirements is not free, but relies on controlled natural language techniques leading to a partial structuring of the text. We show that this makes it possible to design (semi-)automatic verifications on requirements and models, which would be impossible with unconstrained natural language. We illustrate here our approach on a small illustrative example and we report results obtained on a full size industrial application.

KEYWORDS

Requirements Engineering, Model-Based Systems Engineering, Controlled Natural Languages

1 | INTRODUCTION

According to several independent sources, a significant portion of projects fail or exceed cost or time constraints because of incomplete, unrealistic or changing requirements [56,18,63]. It is thus expected that good requirements engineering practices improve the quality of a project, reduce its cost and lower the risk of a failure. Recent empirical studies show that it is actually the case [18,13]. Moreover, regulatory bodies of life-critical systems (typically in aerospace and nuclear industries) impose practices to encourage high-quality requirements (including traceability to higher-level requirements, or justification when requirements are self-derived). However, because of a lack of empirical data [18] and because requirements engineering is still a relatively new domain, what should these good practices be and how to manage requirements throughout the life-cycle of systems is still an open question. In any case, various aerospace companies – such as our industrial partner (the aeronautic group SAFRAN) – are searching for robust methodologies to elicit and to manage requirements, and ways to integrate these methodologies with a model-based approach to systems engineering.

Before presenting our contribution, we should clarify what are requirements in the context of this work, and explain why natural language is still crucial for requirements engineering.

What Are Requirements in our Context?

According to the IEEE 1220 standard [27] for systems engineering, a requirement is "a statement that identifies a product or process operational, functional, or design characteristic or constraint, which is unambiguous, testable or measurable, and necessary for product or process acceptability", e.g. "The weight of the system shall be less than 100kg".

We shall take this definition as the basis for our work. It is however necessary to make more precise the type of systems we are considering. In this article, we consider that a "corpus" of requirements refers to a set of requirements which concern the same system and have, in principle, the same level of decomposition.

What Type of Systems Are We Considering?

Our work focuses on technical systems, such as the anti-lock braking system for the leading gear of an aircraft.

In general, the specifications of this type of systems do not evolve very fast (even if changes over the project life-cycle are unavoidable) and are strongly constrained by the physics.

Another important characteristic of these systems is that they interact with other technical systems, rather than directly with humans. It is therefore usually relatively easy to characterize their interfaces: for example, if the system of interest is electrically connected to another, we can specify that the current passing through this connection shall be lower than 1 Ampère.

Finally, they are usually decomposed into subsystems to keep a manageable complexity. Certain subsystems can be seen as black boxes, for their delivery is the role of suppliers. In this context, requirements are a communication tool between the client and the supplier. They are increasingly used as a basis to establish contracts.

Motivating Problems

Even though one tries to make requirements as "unambiguous, testable or measurable, and necessary" as possible, some ambiguity will necessarily persist because it is impossible to make explicit all of the knowledge about the systems, for at least two reasons. First, the process by which the knowledge would be made explicit would take forever. Second, the result of this process would be much too big to be usable in practice. To go on with our previous example, it must be assumed that the client and the provider know and agree on what is an Ampère. It is just not feasible and pointless to incorporate the laws of electricity in a corpus of requirements. More generally, requirements engineering in our

context has to rely on a vast amount of implicit knowledge, which cannot be made explicit in practice.
As all communication media, requirements distort the information. More precisely, a double distortion is involved: first, when the requirement is written (because of massive use of implicit knowledge by the writer); second, when the requirement is read (because of massive use of implicit knowledge by the reader). To avoid misunderstandings, this double distortion should be made as small as possible. This is the reason for guidelines on requirements, such as the ones given by Pohl [49], Badreau and Boulanger [3]. For example: Requirements should be relatively short to be easily understandable; Potentially ambiguous vocabulary and syntax should be avoided; One requirement should not contradict another requirement.
Guidelines emphasize also the necessity of managing requirements, e.g. to give each of them a unique identifier, to track changes, to trace between levels of requirements, to associate them with a level of maturity, to identify their authors… In this article, we shall focus on problems related to ambiguity of requirements, assuming they are managed in an efficient way.

The Role of Natural Language

As of today, most requirements are written in natural language, possibly using patterns such as:

> The ⟨*system*⟩ shall ⟨*do something*⟩ with ⟨*a certain level of performance*⟩ under ⟨*this and that circumstances*⟩.

But even restricted to such patterns (with which it is sometimes hard to comply), natural language requirements are subject to ambiguities.
Why therefore not use systematically formal specification languages such as B [1], Z [59] or Coq [7]?
There are actually several practical reasons not to do so.
First, a fully formal approach is often not suitable in the early phases of the design, where many things remain under-specified and evolving. Moreover, specifications of a system should concentrate on what the system should do (the problem) not on a what it should be (the solution). If the description of the problem, i.e. the customer needs, is informal, requirements involve thus necessarily a "gray zone", which is not possible to describe formally.
Second, although requirements must be in principle "testable or measurable", many remain more qualitative than quantitative, therefore not easily captured by purely mathematical means.
Third, formal languages are primarily designed to describe programs, which are, fundamentally, mathematical artifacts. Technical systems are not mathematical artifacts. As already pointed out, their descriptions necessarily involve a lot of implicit knowledge, which is hard to make explicit, in practice, such as "what an Ampère is".
Fourth, formal languages require a high level of mathematical expertise, which is usually not shared by all stakeholders. Requirements in our context are primarily a communication medium, which means that they must be readable by people with potentially widely different backgrounds. Moreover, experience shows that mathematical expertise required by formal languages is hard to maintain in companies, at least for non life-critical applications.

Our Approach

Requirements will thus continue to be written in natural language and we have to cope with that. This does not mean however that natural language cannot be constrained or complemented by formal artifacts. This is the direction we decided to explore with our industrial partner. More exactly, we explored a pragmatic approach relying on two pillars.
The first pillar of our approach is the co-design of corpuses of requirements and model-based descriptions of the architecture of the system under study. Integrating requirements written in natural language with model-based systems engineering is indeed *a priori* quite hard, as pointed out by Do *et al.* [15]. However the model-based approach plays a

steadily increasing role in systems engineering [41]. Our idea is therefore to link pieces of text in requirements, such as the "*(system)*" in the above pattern, with elements of models, such as a block in a SysML internal block diagram [22]. In other words, the idea is to consider a corpus of requirements as a hypertext specification, with hyperlinks pointing both to other parts of the corpus, and to elements of models. We use the term co-design because it is really the way we worked: requirements and models were designed concurrently and made iteratively more precise and complete.

The second pillar of our approach consists in using techniques stemmed from controlled natural language [38]. The idea is to write (the most formal) parts of the requirements, such as inequalities over certain quantities, in a structured way. This idea echoes attempts to formalize constraints in specifications with formalisms such as the Object Constraint Language [64].

The combination of these ideas makes it possible to write scripts that parse both requirements and models in order to perform various tests, enhancing in this way the quality of both requirements and models.

The contribution of this article is thus twofold. First, it describes a pragmatic approach to integrate corpuses of requirements and models. Second, it reports some experiments performed with this approach and draws lessons from these experiments.

Organization of the Article

The remainder of this article is organized as follows. Section 2 describes a simplified anti-lock braking system that will serve as illustrative example throughout the article. Section 3 shows how requirements can be represented as partially structured hypertexts and presents the benefits of such structuring. Section 4 introduces the underlying ideas and shows the advantages of concurrently designing corpuses of requirements and models of physical and functional architectures of the systems under study. Section 5 reports results we obtained on a full scale industrial specification. Section 6 discusses related work. Finally, Section 7 concludes the article and provides some thoughts on future directions.

2 | ILLUSTRATIVE EXAMPLE

This section presents a technical system, the anti-lock braking system (ABS) installed in cars, that we shall use throughout the article as an illustrative example. We chose this system because it can be understood relatively easily but nevertheless shares important characteristics with avionic systems we studied: it involves both hardware and software parts, it interacts mainly with other technical systems, it is composed of subsystems and it is life-critical.

An ABS, or anti-lock braking system, is a mechatronic system that aims at preventing the wheels of a vehicle from locking up during braking. An ABS automatically releases the brake when it detects a possible locking. Braking/release cycles can be repeated several times per second.

In modern cars, an ABS is usually installed on each wheel. For the sake of simplicity, we shall consider here only one ABS acting on one wheel, independently from the other ABS installed on other wheels. Moreover, we assume that the activation of the ABS is binary: either the braking fluid is transmitted to the brakes as if the ABS was not there, or the braking pressure is null.

2.1 Architecture and Behavior

Fig. 1 shows the environment as well as a simple decomposition of the ABS. The internal architecture of the hydraulic modulator, adapted from [52], is detailed in Fig. 2.

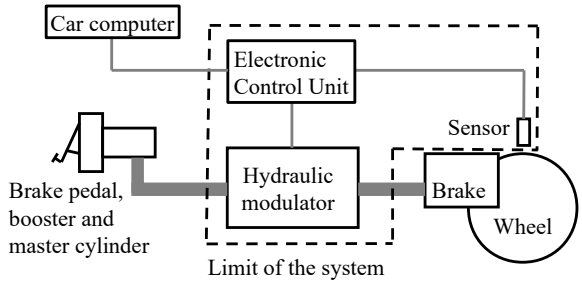

FIGURE 1 The anti-lock braking system (ABS) and its context

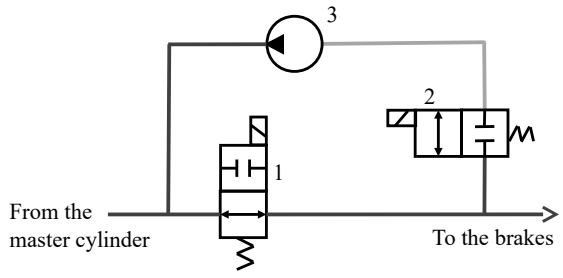

FIGURE 2 A possible hydraulic circuit showing brake fluid flow in the hydraulic modulator (accumulator and reservoir not shown)

When the electronic control unit (ECU) detects a risk of lock-up of the wheel, the hydraulic modulator shuts off the hydraulic connection between the master cylinder and the brake cylinder and depressurizes the latter portion. The connection is shut off by activating the electrovalve (1), while the depressurization is performed by activating the electrovalve (2), which lets the brake fluid flow to the pump (3).

The system detects when the brake should be released by calculating the wheel slip. This slip is calculated using the rotational speed of the wheel and the speed of the vehicle. The speed of the vehicle is approximated using the rotational speeds of each wheel, themselves measured by sensors. The ECU communicates also with the central computer of the car, which communicates eventually with the driver (e.g. via a warning lamp on the dashboard) and with other embedded systems.

2.2 Typical Requirements

We give below 10 requirements which could be part of the specification of the ABS. The objective is not to give a full description of the system. Rather, these requirements exemplify the form and content of real requirements, including their shortcomings, that we found in industrial specifications. They will serve as the basis of our discussion.

To start with, the following requirement sets up a general goal of the ABS.

Req. 1 *The system shall be designed to minimize the potential for driver errors that would significantly reduce safety during driving.*

The second requirement is more concrete. It specifies that, when there is no/low power, the braking circuit should transmit pressure as if the ABS was not there.

Req. 2 *When the power supply is lower than 10 mW, the brake pressure shall be equal to the master cylinder pressure.*

As systems have different modes of operation, it is often the case that requirements describe modal behavior, as follows.

Req. 3 *In state Off, when the master cylinder pressure is lower than 50 bars, the brake pressure shall be equal to the master cylinder pressure.*

A significant part of the specifications we studied consisted of "environmental requirements". These requirements describe the environmental conditions the system should be able to withstand and/or the performance it must ensure within these conditions. E.g.

Req. 4 *The system shall not be damaged by an ambient temperature above -30°C and below 60°C.*

Req. 5 *The system shall not be damaged by a maximum master cylinder pressure of 60 bars.*

Safety plays indeed a very important role in the design of systems. A subset of requirements are thus related to this topic. E.g.

Req. 6 *The structural items of the ABS shall remain within their own design and integration envelope after a failure leading to a partial or total part detachment.*

Req. 7 *While in state On, the probability of a failure leading to the driver being unable to steer shall be lower than* 10^{-8} *per hour.*

Req. 8 *The probability of a non-detected failure leading to the system being unable to apply a brake pressure of more than 20 bars in state Braking shall be lower than* 10^{-8} *per hour.*

The following requirement specifies performance in terms of energy consumption.

Req. 9 *The system shall not demand a current of more than 1A on the power supply.*

Finally, here follows a typical requirement about the maintainability of the system.

Req. 10 *It shall be possible to replace the hydraulic parts of the ABS in less than 2 hours.*

2.3 Discussion

In most of the corpuses of requirements we had access to, one can find a few high level requirements such as Req. 1. According to best practices, this requirement is certainly not a good one: it is rather vague and hardly measurable. It is thus tempting to discard it, purely and simply. However, this requirement specifies the overall objective of the system, i.e. the problem at hand. Moreover, it helps to understand the environment of the system. Together with high level system descriptions, such as those provided by Fig. 1 and the associated comments, it is thus definitely

useful. Whether such requirements should be kept in the requirements corpus or managed separately is probably a matter of corporate culture. We shall come back to this topic when discussing about architectural models in Section 4. Additionally, we mention some articles focusing on these high-level goals in Section 6.

At a first glance, Req. 2 looks on the contrary precise and testable. A more detailed review reveals however a number of issues. To start with and in the wake of the above discussion, it is questionable whether Req. 1 and Req. 2 should belong to the same corpus, as their levels of abstraction differ widely. Moreover, it can be argued that Req. 2 describes already the solution, and not the problem (Req. 10 makes also a strong assumption of what the system consists of). Several authors, e.g. Zave and Jackson [69], suggest that "*a specification should contain nothing but information about the environment*". In other words, requirements should not specify what is inside the system but only describe its environment. Our experience is that such an uncompromising "black box" approach is hard to implement in practice, at least in our context. For this reason, requirements such as Req. 2 and Req. 10 are probably acceptable. *De facto*, we found many such requirements in the corpuses we looked at.

It remains that Req. 2 relies on a lot of implicit knowledge, both about the physics (What is a mW? What is a pressure?) and about the system itself: Where does the power come from? What it is used for? Is the brake part of the ABS? What is the master cylinder?… These points have indeed to be made more precise in order to avoid misunderstandings. As already pointed out, it is impossible to describe the laws of the physics in a corpus of requirements. The system itself should probably be described aside this corpus, with figures and even models, as sketched Section 2.1.

Req. 2 contains moreover another more subtle ambiguity: Does it describe a hard real-time condition (whenever the power supply is low, the brake pressure shall *at the same time* be made equal to the master cylinder pressure), or is this time constraint more flexible (if the the power supply gets low, the brake pressure shall be *eventually* be made equal to the master cylinder pressure)?

Req. 3 is also precise and testable, at least much more precise and testable than would be the corresponding high level goal "*When the system is not activated, it shall not prevent braking*". Still, it raises the same kind of issues as Req. 2. It says implicitly that the system has several modes of operation, including the mode "Off" and that it can be in several physical conditions within each mode (e.g. the master cylinder pressure can be lower or higher than 50 bars). It is often the case that a group of requirements describe implicitly a state machine, without the benefits of a model in terms of conciseness, completeness and verifiability. Moreover, Req. 3 assumes implicitly, even though this is not mentioned in the text, that a pressure in the master cylinder is the expected to be lower than 50 bars and that any other condition should be considered as abnormal.

In many specifications, we can find expressions such as "shall not be damaged by", as in Req. 4 and Req. 5, or "the probability of a failure leading to X", as in Req. 7 and Req. 8. Such expressions are very frequently used and engineers have no problem understanding them. There is thus *a priori* no reason to banish them from corpuses of requirements. It remains that "to be damaged" is a rather vague concept. The "probability of a failure" looks more concrete, but is also problematic: does this refer to the probability of a failure over the whole life-time of the vehicle? Is it on the contrary a probability of failure per unit of time (given that a car is in general used by "slices" of maximum one hour)? In our case, the precise definition of what this expression stands for can be hopefully found in safety standards such as ISO 26262 [28]. These two examples show the imperative need of references to external documents.

In the 10 requirements presented in the previous section, the system under study is sometimes designated by "the system", like in Req. 9, sometimes by "the ABS", like in Req. 10, and sometimes kept implicit, like in Req. 7. Moreover, in some cases, only a part of the system is concerned and not the whole system, like in Req. 3 and Req. 10. This may cause problems because some requirements keep implicit what they refer to, such as Req. 7. Similarly, designating the system here as "the system" and there as "the ABS" is probably questionable. The important point here is to ensure that there is no ambiguity about which expression designates what.

In the same vein, note that these 10 requirements do not fit in the same template, such as the one proposed in the introduction. Forcing all requirements of a corpus to fit into a small set of templates, defined once for all, proves in many cases to be infeasible and leads to awkward descriptions. But here again, it is wise to avoid styles of writing that are too unstructured, especially if we want to perform automated tests and verifications.

We can finally note that Req. 10 is a soft technical constraint. It says implicitly that if something goes wrong with the hydraulic parts of the system, these parts can be replaced in an appointed garage, by some competent mechanic, within about two hours. This requirement is mostly here for commercial purposes, which are indeed extremely important, but of different nature than purely technical requirements such as Req. 3 or Req. 2.

We could fill many more pages with this discussion. But at this point, the reader has understood the essence of our message:

- Corpuses of requirements contain very different types of requirements and appropriately so.
- Not everything can be formalized, but it is worthwhile formalizing as much as possible, to avoid ambiguity.
- It is hard to fit all requirements into the same template (or into a small set of templates), even though many of them can be advantageously structured by means of such templates.
- Corpuses of requirements cannot be self-contained. They rely necessarily on implicit knowledge that cannot be part of any description (such as the laws of the physics), on implicit knowledge that is already described in external documents (such as safety standards), and finally on implicit knowledge that should better be described by means of models.

The next two sections describe the pragmatic approach we used to tackle these issues.

3 | PARTIALLY STRUCTURED HYPERTEXTS

This section presents the first pillar of our proposal to reduce the ambiguity of requirements, which consists in managing corpuses of requirements as partially structured hypertexts.

3.1 Linking Terms to their Definition

As discussed in the previous section, in our context, a corpus of requirements is never self-contained. It always involves references to external documents, such as:

- Glossaries that define concepts used in the requirements;
- Technical documentations that gives precise information on functions and parts of the system;
- Standards and regulations that frame its operation;
- Multi-media documents helping to understand what the system should be and should do;
- Models describing the architecture of the system or its behavior, possibly with several points of view;
- …

Making these references explicit and making it possible to reach them via hyperlinks is a very natural idea. Hyperlinks can also refer to other parts of the corpus.

A source of problems in requirements engineering rests in misunderstandings of terminology. The sooner such mis-

understandings are detected, the less costly it is for the project [26].

Consider for example, Req. 2. In this requirement, what does the term "power supply" refer to? If the corpus of requirements is the only source of information at hand, the reader has to interpret its meaning. Hopefully, a description of the environment comes with the corpus of requirements, which contains some information on terms used in requirements. Still, one has to infer that the term "power supply" refers to this information (and to locate the information in the documentation, which may be tedious). If there is only one power supply and the term "power supply" is always used in the requirements to refer to it, this inference is probably not too difficult to make. However, things are often more complicated than that. In the ABS case, there may be for instance one power supply for the control unit and one for the hydraulic modulator. Therefore, one has to guess which one is referred to.

A first idea is thus to link individual words or groups of words in requirements with their definitions standing outside the corpus of requirements. These hyperlinks may refer to model elements rather than entries in a glossary. They are written directly within the text (for example as hyperlink tags in an XML-like format such as ReqIF [46]). As we consider corpuses of requirements that contain at most a few thousands of requirements, recreating the links in the other direction (from models and glossaries to requirements) is easy and fast.

One of the advantages of such hyperlinks is that they make it possible to detect when different terms actually refer to the same thing. They also make it possible to extract statistical elements, such as where or how many times a part of the system is referred to in the requirements.

Note that the links we mention here, and the syntax we introduce in the next section, are independent from the traceability links between requirements and higher/lower-level requirements. A corpus may have only traceability links between requirements, or only the hyperlinks defining terms inside requirements, or both. Our work focuses on how individual requirements are structured, while we do not deal with traceability to higher/lower-level requirements, which usually considers individual requirements as atoms.

3.2 Structuring Requirements

Hyperlinks are extremely useful to avoid misunderstandings. However, they do not solve all of the problems identified in Section 2.3, such as ambiguities on the meaning of logical properties, errors in units, etc. A solution could be to write requirements as mathematical formulas. This objective is however in general unreachable, for all the reasons mentioned in the introduction.

The idea is therefore to structure requirements as much as possible, without making them fully formal. It is also important to provide as much flexibility as possible, i.e. to make it possible to write anything from free text to fully formal formulas, going through requirements written according to some templates.

Formal Syntax

To describe the structure of requirements, we propose thus to use typed terms similar to those of programming languages. A requirement is thus represented as a pair (S, P), where S is the system under study, or part(s) of this system, and P is the property S must fulfill. Properties P are written as terms built over a set of constants (including free pieces of text, integers, real numbers, Boolean constants…), and a set of built-in functions, like arithmetic and logical operations, inequalities… Available functions may depend on the context.

We 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.

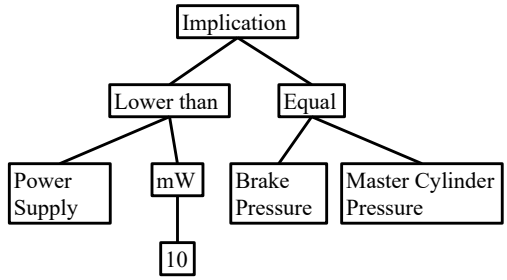

FIGURE 3 Syntax tree for Req. 2

Let us now introduce types: each n-ary function of F has one output type and n argument types. For a term to be well-typed, the types it contains must respect constraints which depend on the functions used (e.g. a Boolean "And" accepts only Boolean arguments).

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: for example, a Boolean "AND" is a binary function which accepts only Boolean arguments.

This decomposition is relatively similar to the grammatical analysis of a natural language (NL) sentence, which yields parse trees. The decomposition is also quite close to the Abstract Syntax Trees used in computer science.

Eventually, each requirement is thus encoded as a syntax tree. Consider again Req. 2. The syntax tree for this requirement could be as shown in Fig. 3.

In this tree:

- "Implication" is a binary function, with a Boolean output type and two arguments of type Boolean.
- "Lower than" is a binary function, with a Boolean output type and two arguments of the same type. More precisely, this type, here "power expressed in Watts", needs to be comparable.

A measure (a pair number-unit) is built using a unit function which takes a number as argument. For example, the element "10 mW" is built with a function "mW" taking 10 as argument.

The syntax proposed is flexible and we can modify the requirement, for example by updating the condition: if we want to write that "When the power supply is lower than 10 mW and higher than 0mW, the brake pressure shall be equal to the master cylinder pressure", we may modify Req. 2 so that it appears as in Fig. 4.

In this framework, each term has a type: for example, "10 mW" has the type "power expressed in mW". It is therefore possible to check automatically whether a term is well-typed (in a broad sense). For instance, a comparison between two quantities with different MKS units can be rejected. We did not implement a complete unit system, which would be able to detect that, for example, "Watt" and "Joule per second" are the same unit. Some ontologies for units of measurement can be found in Keil and Schindler [34].

Note that there may be different ways to write this requirement in plain English, just like there are usually several ways of reading a mathematical formula. However, whichever way the formula is read, its meaning is clear.

Note also that markup languages such as XML simplify the structuring of requirements according to the above principle.

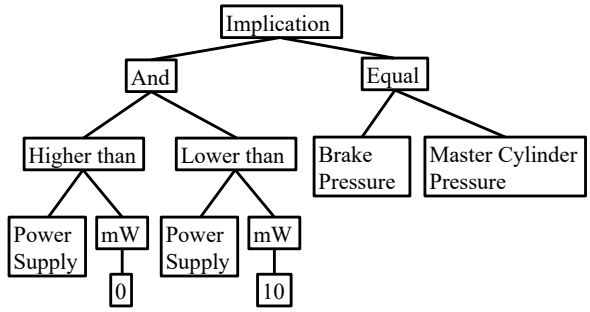

FIGURE 4 Syntax tree for a variant of Req. 2

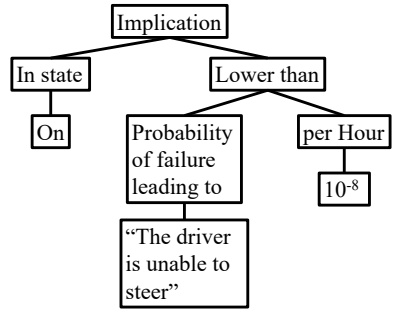

FIGURE 5 Syntax tree with a non formalized leaf

Note finally that this structuring can be progressive: a requirement with a low maturity can be less formalized, i.e. less structured, than its subsequent version with a higher maturity. For example, systems where requirements change often would probably have a less formalized specification, as spending time to perfect requirements which will change anyway may not be practical. Authoring requirements and models is an iterative process.

Free Text

As already mentioned, one of our key observations is that it is not possible to completely formalize requirements. This is the reason why the above formal syntax accepts pieces of free text as leaves of syntax trees. In the worst case, the syntax tree is reduced to a unique leaf. A key point is that even such a leaf can have a type and possibly several attributes that make it possible to check whether the leaf fits in the requirement.

Consider for example Req. 7. If analysts cannot or do not want to formalize "the driver being unable to steer", they can simply leave this as a free text in the requirement and give it the type Boolean, meaning that either the driver is able to steer or he/she is not. This makes it possible to integrate this piece of text as a leaf in the syntax tree pictured in Fig. 5.

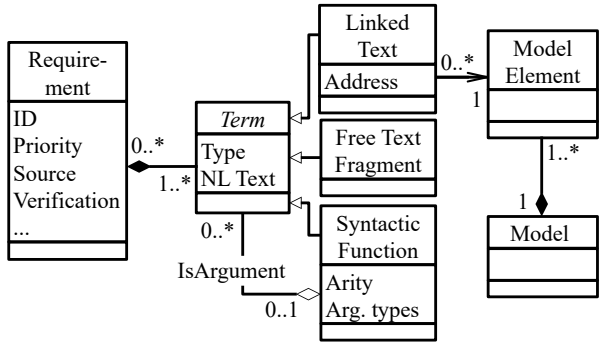

FIGURE 6 Class diagram for the meta-model of requirements

Meta-Model

The UML class diagram pictured in Fig. 6 describes the interactions between the different elements we introduced in this section. Note that establishing the set of possible relationships between requirements is beyond the scope of this paper. According to this meta-model, a requirement is composed one or more terms, each term being either: a free text fragment, text linked to a model element or a syntactic function.

For example, consider Req. 7: "*While in state On, the probability of a failure leading to the driver being unable to steer shall be lower than* 10^{-8} *per hour.*" An object diagram corresponding to this requirement is given in Fig. 7, as an instance of the class diagram of Fig. 6. Req. 7 is composed of 8 terms in gray (note that for clarity reasons, we did not write all the composition links to the requirement object, nor the attributes other than "Type", "NL Text" and "Address").

Each term has a type, which will be used to check that the syntax of the requirement is correct and for other automatic checks. Each term also has a "NL text" attribute, which is used to generate a textual representation of the requirement. In Fig. 7, one term is text linked to a model element, here the state "On" defined in the state machine given in Fig. 11. This term has an "address" attribute, specifying precisely what it is referencing. This term also has a type and a textual representation. Ideally, these two attributes should be defined directly in the linked model, as they stay the same between requirements: when referring to the state "On" in a requirement, it is always a state, and always using the same text helps the reader understand the specification.

The "the driver is unable to steer" term is a free text fragment as introduced in the previous subsection. The type and NL text of a free text fragment is the only information we have on this portion of the requirement. The values of these two attributes are defined by the user each time he or she uses a free text fragment in a requirement.

The rest of the terms are syntactic functions. As noted earlier, these functions have output types, (e.g. Boolean for "Lower than"). Only these syntactic functions may have arguments. The types and number of these arguments are constrained: for example, the "Implication" function accepts only two arguments of type "Bool". Types as well as functions may be defined by the user: to define a type, we need only a string, e.g. "FailRate", and possibly a parent type (for example, a "Force expressed in Newton" is a subtype of "Force"). In general, we use mainly the following types : Boolean, State, System (to refer to elements of an architectural model, such as "the ABS" or "the hydraulic modulator") and various measures (i.e. a number associated with a physical unit, such a Newton, meter, Ampere…).

These functions also have textual representations, which are used to get the textual, human readable, version of a requirement. These textual representations are modular: for example, an "Implication" will be written as "while <X>, <Y>" in the textual requirement, where <X> and <Y> are replaced by the textual representations of the first and second arguments of the implication respectively. Similarly, the function "In state" is written "in state <X>", where <X> is the

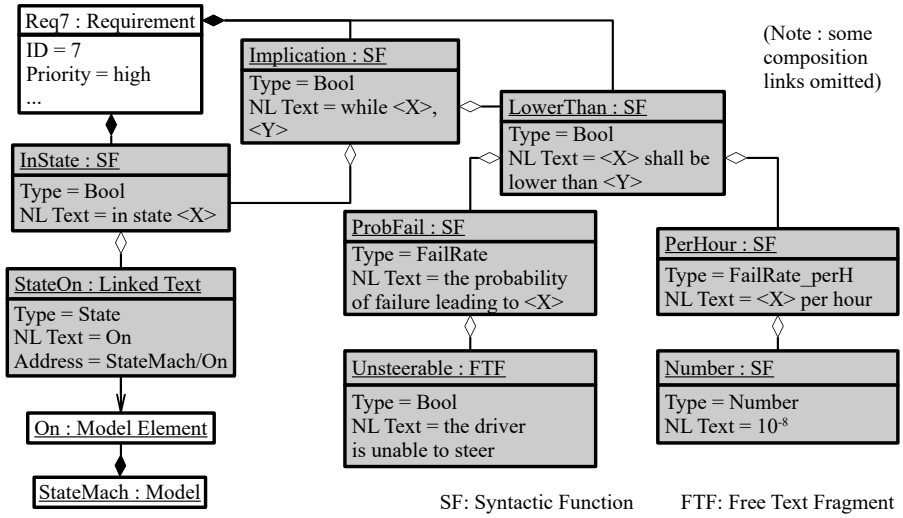

FIGURE 7 Object diagram for Requirement 7

textual representation of the function argument. We noticed that having always the same textual representation for a function could make some requirements unclear, so we have defined a few rules to adapt the text of a function to its context.

Another example of an object diagram is given for Req. 2 in Fig. 8 (only the attributes "Type" and "NL Text" are shown).

We can use a language such as XML to encode these structured requirements. We have written a script allowing extraction of a pure textual representation from these structured requirements, for human readers. This generated text may feel awkward for a reader and may not respect English rules, however it is understandable and we think the text generation can be improved with additional work.

Next, we will see how computers can use the information in these structured requirements (including hyperlinks, types, structure and NL text) to verify properties of requirements.

3.3 Automated Verification of Corpuses of Requirements

This trusted interpretation of requirements makes it possible to check automatically properties of corpuses of requirements. For example, one may want to ensure that a concept is always referred to with the same text.

Best practice guides, standards (such as IEEE 29148[29]) and reference books on requirements engineering provide criteria of quality of requirements and corpuses of requirements. Some of these criteria, such as traceability, cannot be checked automatically. Some others can. The rest of this section describe tests that can be performed automatically on formalized specifications.

Correctness

Before doing other types of tests, requirements have to be parsable. In particular they must respect the meta-model given in Fig. 6. This guarantees that the requirement is at least syntactically correct, for example the requirement "*The system shall not be damaged by an ambient temperature above -30°C and*" would be recognized as incorrect, since "and"

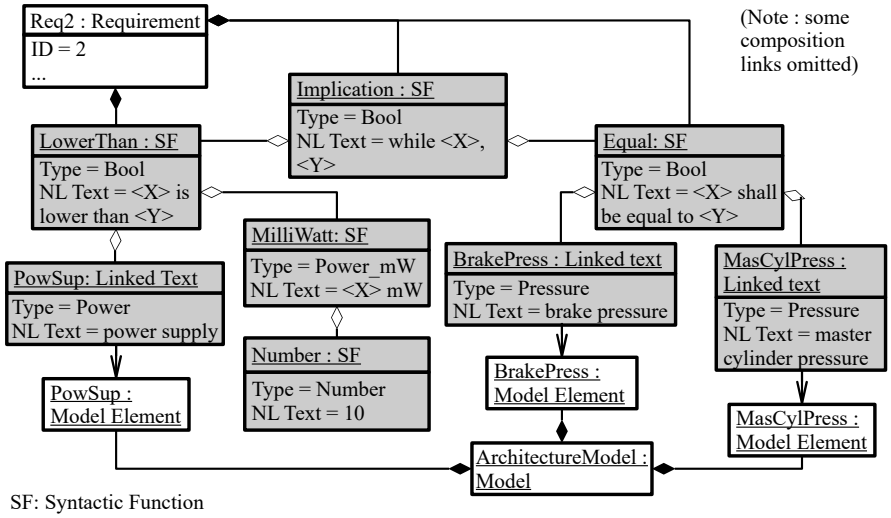

FIGURE 8 Object diagram for Requirement 2

has only one argument here.

Consistency

It is not in general possible to check whether a given corpus of requirements is consistent in a logical sense, i.e. to prove that there is no contradiction in the corpus. The underlying mathematical problem at stake is actually undecidable.

It is however possible to check consistency at the lexical level: for example that the same concept is always referred to with the same text (e.g. not "the system" as in Req. 10 and "the ABS" as in Req. 6). As already pointed out, it is also possible to check the types of terms (e.g. it should not be allowed to compare a measure of electric power with a pressure) as well as units of measures (e.g. pressure measures as in Req. 5 are all given in bars).

Completeness

For the same reasons as before, it is not possible to check for logical completeness of a corpus of requirements, i.e. to prove that all possible situations have been considered.

It is, however, possible to check that all concepts are defined. For example, it is possible to list all interfaces defined in models and to check whether they are referenced in the requirements: an undefined interface is obviously a potential source of problems. Reciprocally, it is possible to check that all pieces of text such as "the probability of a non-detected failure" in Req. 8 are defined by some artifact.

It is also possible to verify that each link declared in the requirements refers actually to some existing artifact: entry of a glossary, element of a model…

Testability

In a more subtle way, it is possible to check whether a requirement is concrete and precise enough by looking at the number of tests this requirement reacts to.

Consider for example Req. 4. It could be written, in a rather vague way, as "⟨*the system*⟩ *must* ⟨*resist to normal environ-*

mental conditions⟩.". This requirement would probably react to very few tests as the property is written as a piece of free text. In Req. 4 however, automated tests will check that temperature measures are given in degree Celsius, that inequalities compare comparable quantities, and so on.

4 | CO-DESIGN OF REQUIREMENT AND MODELS

This section presents the second pillar of our proposal to reduce the ambiguity of requirements, which consists in co-designing the corpus of requirements and models of the physical and functional architecture of the system.

4.1 The Paradigm S2ML+X

The first thing we have to agree on with the reader, is the definition of a model. Can, for example, Figs. 1 and 2 be considered as models? They are simplified representations of the ABS, used for a purpose, which, if we take the general definition given by Stachowiak [60] suffices to make them models. We shall use however a more restrictive definition for models, which excludes Figs. 1 and 2.

Recall that we want eventually to perform automated tests on models, i.e. to load models into computer memories and run programs on them. To do so, we need to distinguish four aspects of a model. First, a model has a semantic, i.e. it denotes a mathematical object, for example a structure like a tree or a graph. Without a formal definition of the semantics of the models we consider, it would be impossible to define operations to be performed on these models. Second, a model has a syntax, i.e. is encoded by a sequence of symbols, or equivalently in a data structure in the computer memory. This syntax obeys a grammar, i.e. a set of rules telling whether a given sequence of symbols is a valid model or not. A key issue is to associate each syntactically valid model with its semantics. Third, a model may be represented graphically, but one should not confuse the model with its graphical representation. Most of the time, graphical representations are only partial views of the model. Representing graphically all of the details of a model would actually overload the picture and thus miss the point of graphical representations as a communication means. Finally, a model is pragmatic, i.e. it represents some aspects of a system existing in the real world. Automated tests cannot deal with the pragmatism of models, just with their syntax. These ideas are developed by Rauzy and Haskins [51].

The models we used for the present study were designed in small domain specific languages themselves designed according to the S2ML+X paradigm [51]. This paradigm relies on the idea that any behavioral modeling language in systems engineering is the combination of a mathematical framework in which the behavior of the system is described (the X), and a set of constructs used to structure the model (S2ML). S2ML, which stands for system structure modeling language, gathers in a unified framework structuring constructs stemmed from both object-oriented and prototype-oriented programming [4]. The combination of S2ML with any mathematical framework X gives raise to a modeling language with a well defined syntax and semantics. Graphical representations can then be used to create views on models. Parsing S2ML+X languages is easy (for they are just textual encoding of models) and technology independent, conversely for instance to the XMI format in which SysML models are encoded.

For the purpose of this study, we created two small domain specific languages: one to represent physical and functional decompositions of the system (similar to, but more powerful and more formal than, SysML internal block diagrams) and one to represent a fragment of Harel's state charts [25] (similar to SysML state machines). Their particular grammar has no importance here. We focused on these two types of models because they are used extremely often in requirements specifications; however, other types of models could also be used. Parsers were designed in Python,

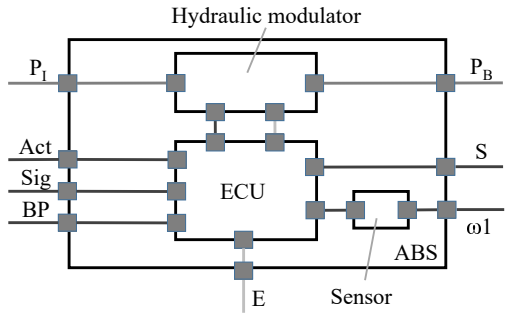

FIGURE 9 Graphical representation of the model of the physical architecture of the ABS

which makes it possible to prototype tools quickly and efficiently.

4.2 Architectural Models

One of the most basic needs to complement requirements is a description of the environment of the system under study, as well as of its physical and functional architectures (decompositions). Again, Figs. 1 and 2 illustrate this need. To represent such structural decompositions, one needs a structural modeling language, with notions of blocks (to represent components), possibly classes (when several components of the same type are involved), ports (to represent interfaces of components) and connections (to represent interactions between components). S2ML suits perfectly this need.

The fundamental remark here is that models representing the environment of the system, its physical and its functional decompositions do not need to be completed before writing requirements. It is much better to design these models while writing requirements.

For example, in order to understand Req. 10, we need to have a description of what are "*the hydraulic parts of the ABS*". Fig. 9 shows the graphical representation of a model of the physical architecture of the ABS. With such a model, it is possible to give a precise meaning to the above piece of text by linking it to the "*hydraulic modulator*" component of the model.

This iterative process is especially important when defining interfaces of components.

Assume for instance we designed the model of the environment of the ABS pictured in Fig. 10 and consider Req. 9. In the model represented in Fig. 10, the power supply is modeled as a value "*E*". *E* is expressed in Watts. But our requirement refers to the current coming from the power supply, not to the power. If we keep the model as is, there is no way to link the concept "*current*" of the requirement to an element of this model.

To solve this issue, we can consider that the power "*E*" is the current multiplied by the voltage in the supply line. If we assume (and document this assumption) that this voltage is constantly equal to 12V, we can rewrite Req. 9 as follows.

Req. 11 *The system shall not demand a power of more than 12W on the power supply.*

Alternatively, we could update the model and define two interfaces "*I*" and "*U*", replacing the power "*E*". and representing respectively the current and the voltage of the power supply. Doing so would potentially break references to "*E*" that were made in other requirements. However, since these references are explicit and easily explored by computers, it is possible to detect and correct them.

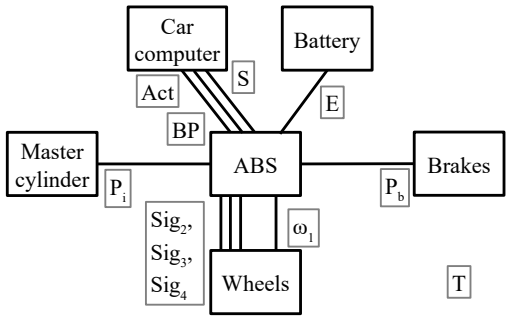

FIGURE 10 Graphical representation of the detailed model of the environment of the ABS

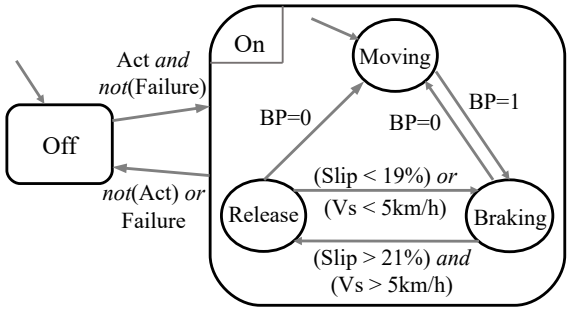

FIGURE 11 State machine defining the states of the ABS

4.3 Behavioral Models

In real specifications, requirements expressing purely combinatorial properties such as Req. 2 are rather unusual. Antecedents, such as "*the power supply is lower than 10 mW*" are often completed with a condition such as "*in state X*". Considering modes of operation, or states, makes it possible to express complex properties, especially properties describing temporal chaining.

Fig. 11 shows a state machine describing operational states of the ABS.

To define the transitions of this state machine, one needs direct inputs from the environment of the ABS such as "*Brake pedal pressed*" (BP) as well as abstractions like "*Vehicle speed*" (Vs).

Once such a model designed, it is possible to refer to states of the state machine in requirements, e.g. "*When in ⟨state Braking⟩, the system shall do X*".

In the industrial specification we had to deal with (see next section), this kind of state machine was defined implicitly using textual requirements. By making explicit the state machine, it is possible to clarify the meaning of requirements as well as to avoid unexpected interactions between requirements.

4.4 Examples of Possible Tests

We present here some examples of (semi-)automatic verifications that can be done on a formalized specification. Additional examples and discussion can be found in the first author's PhD thesis [40].

A common quality criterion of requirements is that they should be atomic, i.e. they express only one fact. It is possible to automatically raise a flag on requirements which may not be atomic. For example, the requirement "*In state Off, the brake pressure shall be equal to the master cylinder pressure and the signal S shall be false*" expresses two different results: "*the brake pressure shall be equal to the master cylinder pressure*" and "*the signal S shall be false*". If the structure of the requirement is formalized, this problem can be detected because it is possible to detect a structure of the form: "Z implies (X and Y)". It would probably be better to rewrite this requirement as two separate requirements: "*In state Off, the brake pressure shall be equal to the master cylinder pressure*" and "*In state Off, the signal S shall be false*".

As mentioned earlier, different terms can be used to refer to the same concept, and this can cause problems. By adding the hyperlinks proposed in Section 3, we know that Req. 4 and Req. 6 respectively use the words "the system" and "the ABS" to refer to the same model element, the "ABS" box of Fig. 9, because the same address is used in both hyperlinks. Once this problem is detected, there are different possible solutions, such as: raising a warning on one or both requirements, enforcing the use of one authorized term (e.g. "the ABS") by replacing all other terms, defining a list of allowed synonyms for some concepts, etc.

Queries, whether on the hyperlinks or on the structure of requirements, could be a useful tool for requirements engineers to find problems in requirements. For example, by looking at the addresses in hyperlinks, it is relatively easy to write a query which returns all the requirements which mention any state of the state machine. For the ten requirements given in Section 2.2, the result would be Req. 3, Req. 7 and Req. 8. Similarly, a simple query would show that the state "Release" is not mentioned by any requirement, which would probably indicate a problem if the requirements of Section 2.2 constituted the complete corpus.

5 | EXPERIMENTS ON AN INDUSTRIAL SPECIFICATION

The primary objective of our study was to enhance the specification of a new avionic system, i.e. to formalize it so as to remove ambiguities and possible mistakes. This section reports results of experiments performed on this specification with the method described in the previous sections.

The System and its Specification

For confidentiality reasons, we cannot say much here about the system under study. Nevertheless, we can provide some of its characteristics: It is a complex component of an aircraft with hardware (sensors, controllers and actuators) and software parts. It is safety-critical. Its design involved a large aircraft manufacturer and our partner SAFRAN.

The original specification was a 65-page document with 171 requirements. It was neither a first draft nor a completely polished specification.

Requirements were very similar to those presented Section 2.2: in fact, some of those are essentially anonymized versions of requirements from our system under study.

A model of the system and its environment was provided with the specification. However, interfaces between them were inconsistently defined: some were relatively precise and formal, some others were not.

We worked off-line without access to the authors of the specification (who are working for the aircraft manufacturer). However, we had access to experts from SAFRAN when we had technical questions.

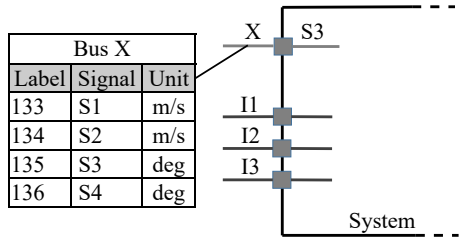

FIGURE 12 Example of information in an architecture model: the bus "X" is an object which contains various signals and their attributes

Experimental Protocol

The experiment protocol we defined with our industrial partner consisted in a step-by-step translation of the existing specification into a more formal one. This translation also involved the design of architectural models and the definition of links between formalized requirements and models.

The formalized specification we produced at the end of this process is certainly not the only one that could have been obtained. First, we had to interpret some ambiguous elements of the original specification for which even the experts we were in contact with could not decide which interpretation should be taken. Second, we had to make some arbitrary decisions about where to stop the formalization process. We tried to limit ourselves to a "realistic" effort of formalization.

Experiment

The 171 original requirements of the industrial specification can be divided into a few groups.

A first group of 23 requirements were essentially the textual description of a state machine: e.g. "The system shall transition from state X to state Y under conditions Z". We replaced these requirements by the corresponding model and added links to this model in other requirements when necessary. For example, this requirement would be represented by a transition between two states as in Fig. 11.

A second group of 12 requirements gave some details on the interfaces of the system, such as "The system shall acquire signal S3 from the bus X label 135". We replaced these requirements by additional information in the architecture model of the system. An example of representation for this requirement can be found in Fig. 12.

A few (5) requirements were not atomic because they had a structure similar to "when in state A, the system shall do X and Y". We separated each of these requirements in two distinct requirements, adding 5 requirements to the total.

Of the 141 remaining requirements, 40 were similar to Req. 2, in that they stated relatively simple properties on inputs, outputs and states of the system. These requirements were formalizable fairly easily.

A close group of 14 requirements were too vague and could be reformulated to use concrete, measurable quantities. For example we replaced "isolating the system from its power supply" by "the input power is equal to zero".

The rest of the requirements (87) were harder to formalize, particularly for the syntax we propose.

Of this subgroup, 30 might have been formalized, but used concepts which were either too hard to model properly, or which were used only in a few requirements. For example, some requirements mentioned integration envelopes, defining where components could be installed. For these requirements, a 3D model could be useful, be we did not

produce one. The original writers probably had such a 3D model available. One requirement defined constraints on noise using physical, measurable quantities. However, as it was the only requirement of this type, we considered that formalizing such a requirement would generally not be worth the additional effort.
34 requirements mainly defined references to external documents (e.g. "The system shall comply with standard X"). These requirements are relatively simple to write, so we did not try to formalize their syntax. However, we added hyperlinks to link the "standard X" text to the document describing the standard.
Finally, 23 requirements were simply too vague to be improved much by the concepts we propose (for example, requirements similar to Req. 1)
The requirements among these 87 which where hard to formalize were simply kept as (mostly) free text. We tried however to rewrite these requirements according to minimal patterns such as the one presented in the introduction. E.g.

Req. 12 *The ⟨system⟩ shall ⟨avoid design features that may generate intrusive noise in the cabin or cockpit⟩.*

We drew 376 links from these 141 requirements to external artifacts. Target artifacts were either elements in architectural models, states of the state machine, entries into the specification document such as tables or figures, or external documents such as standards, norms, etc.
The distribution of these links among the requirements is quite uneven. Some requirements involve six or seven links, while others involve only one. For example, Req. 12 involves only one link from the text item "*System*" to the corresponding block in the architectural model.

Results

We created automated tests (implemented by Python scripts), which use both the links to models presented in Section 3.1 and the syntax presented in Section 3.2, to detect problems in requirements. Once these problems are known, it is easier for requirements engineers to improve the requirements. Additionally, by using the concepts proposed in this work, requirements are improved "by construction": for example, a requirement using the structure we propose cannot be syntactically ambiguous (e.g. cannot have an undetermined structure such as "A or B and C", which can be read either "(A or B) and C" or "A or (B and C)").
According to SAFRAN experts, the final result (corpus of requirements plus models) provides a significant improvement in the quality of the specification. Different aspects of the requirements were improved.
Consistency is one of these aspects: The hyperlinks we added to requirements allow precise determination of the meaning of a term in a requirement, which reduces so-called lexical ambiguity (e.g. naming the same concept with different names). We link terms to their definition, typically an element of an architectural model. By doing that, a script can now explore, for each element linked in the requirements, what is the text used to refer to this element. For example, if two terms "*HMI discrete ORDER_BKWD*" and "*HMI backward command*" are in fact linked to the same element, the script compares these two character strings and raises an alert, since they are different. We detected 10 instances of inconsistent naming of concepts in the formalized specification.
Additionally, the requirements using the structure we propose cannot be syntactically ambiguous since their structure is explicitly described. We can estimate that around 20 requirements were poorly structured in the specification and were given the formalized structure presented in Section 3.2. (Other requirements had an ambiguous structure but we were not able to formalize them).
The correctness of requirements can also be improved. We can detect cases of incoherent typing (such as comparing a torque with a force). When we structure the requirements as we presented in Section 3.2, we give types to the terms in a requirement. For example, "10N" has the "force in newton" type, which a sub-type of the "force" type. Additionally,

we define in an architectural model that the output torque of a motor has the type “torque”. Now, suppose we write in a requirement that “the output torque of the motor shall be higher than 10N”. The “higher than” function expects two arguments which either have the same type, or where one type is a sub-type of the other. This is not the case here, so the type checker will raise an error. However, if we write “the output torque of the motor shall be higher than 10Nm”, we compare a “torque” with a “torque in newton-meter”, which is valid. Using this type system, we can also restrict torques to be expressed using only MKS units, or even only “Nm”, by specifying, in the architecture model, what is the type of “the output torque of the motor”. In the formalized specification, we did not detect such cases of incoherent typing. Additionally, even if the units used were not all MKS units, they were used consistently (e.g. a speed was always expressed using knots).

The structure of requirements can also hint at possible non-atomicity of requirements. We can check automatically if a particular pattern, such as “A implies B and C”, fits a requirement. If this is the case, it is possible the requirement is not atomic.

Another quality criterion for a corpus of requirements is completeness. In general, completeness of a specification is too hard to verify automatically. However, we can check that if we defined an interface, there should be at least one requirement which mentions it. By modeling the system and its environment as in Fig. 10, we get a list of all the interfaces between the system and its environment. We can count how many requirements include a link to each interface, and raise an alert if the total is zero for an interface, since it means it is not specified. In the formalized specification, we did not find any interface defined in the architectural model that was not mentioned in the requirements. However, some interfaces were only mentioned in one or two requirements, which may indicate underspecification of these interfaces.

Finally, an important improvement in the specification is the place of models in it. As noted earlier, a state machine was defined as a series of requirements in the original specification. The original writers were conscious of this, but chose to describe this state machine using text rather than with a model. By using a model with its visual representation, this state machine is far easier to understand than a group of 23 textual requirements. In general, we think models should be used in specifications as definitions for the terms used in requirements.

Lessons learned

As we designed the methodology while working on the industrial specification, it is difficult to give a precise estimate of the time spent on formalization processes. It took about one man-month to understand the most part of the specification, with the help of SAFRAN experts. It took about another man-month to formalize the 171 requirements of the specification and two additional weeks to design models.

This effort is both significant and worth doing, given the quality assurance it provides. in our case study, it has been made *a posteriori*. We believe that it would have been much less work if the formalization and testing would have been done by the authors of the specification while writing the requirements.

This suggests the possibility of designing agile methods for requirements engineering, as it exists for software development [57].

6 | RELATED WORK

Requirements engineering covers several different processes. In this paper, we are interested mostly in the “authoring process”, i.e. the process of writing down clearly and precisely the specifications of a system as a corpus of requirements. Moreover, we focus on concrete and relatively low level requirements of technical systems. Most of the

requirements we had to deal with represented quantifiable properties (for example "The weight of the system shall be less than 100kg"). These requirements are quite different from high-level "goals" and should be treated differently, as pointed out by several authors [62,67], which does not mean however that corpuses of requirements should not contain such high level goals.
As pointed out by Bijan *et al.* [8], ambiguity is a major problem with requirements, and this problem is not well covered in the literature:

> *"While there are many lists of defining what is a good requirement and many ideas on facilitating discussions with the customer and ways to trace requirement changes for impact analysis, the actual formulation of a requirement seems fuzzy. …*
> *One of the major problems with requirements is ambiguity, but many seem to think that ambiguity is simply about vague words such as appropriate and when necessary, but it is more than that. …*
> *There is no literature on quantifying how good a requirement specification is, based on the quality of the requirements taken as a whole."*

We can divide approaches to tackle these issues into two groups: methods focusing on requirements on the one hand, which we called requirement-centric, and, on the other hand, methods completing requirements with models. We review below these two categories of methods.

6.1 Requirement-Centric Approaches

Quality Criteria
Different lists of criteria of quality that a (set of) requirement(s) should have can be found in the literature [49,3,29,50,53]. These criteria are usually a short sentence stating a recommendation, e.g. "Potentially ambiguous vocabulary and syntax should be avoided". They describe errors that should be avoided when writing requirements. However, as pointed out by Bijan *et al.* [8], they do not necessarily guide writers for the formulation of requirements and they are relatively informal. Despite their shortcomings, these criteria of quality serve as a starting point for more refined methods, including the approach we propose.

Criteria Evaluations
Manually checking these criteria is time-consuming and error prone. Some authors propose methods to evaluate them automatically [53]. Some of these methods rely on general natural language processing techniques, such as the readability formula proposed by Kincaid *et al.* [35]. However, Kamsties and Paech [33] argue that using general methods both raise false alarms and miss problems linked with the specific context of requirements engineering or with the particular domain of the specified system.
Several authors propose to complement general natural language processing methods with specific tests dedicated to requirements engineering [16,17,6]. These tests range from simple word checking procedures to complete syntactic analysis. However, capturing the meaning of requirements is not feasible, especially without information on the context, which limits severely the ability to check quality criteria.

Ontologies
Some works aim at enhancing natural language processing methods by means of ontologies [36,33,32]. An ontology, in the context of information science, is a representation of concepts and relations between these concepts in a specific

domain. Ontologies are currently a very active area of research. Among other works on this topics, we can mention the standardization effort of the Object Management Group (OMG), aiming at linking UML models with the Web Ontology Language [11]. In Castet *et al.* [9], ontologies regroup components of a system, variables of these components, the behavior of these variables, as well as interactions between components. Models are included in the ontology to represent the behavior of the variables and the constraints due to the interactions.
In the context of requirements engineering, the problem stands in the design of relevant ontologies, i.e. ontologies that are both useful for a particular corpus of requirements and not specific to this corpus. In the case study presented by Kaindl [31], even if the word ontology is not mentioned, the author uses a "domain model" as a way to define concepts appearing in the requirements. It is possible to train ontologies directly with requirements documents instead of general texts on the domain [48,47]. Fraga *et al.* [21] propose to generate templates from ontologies, in order to provide more flexibility. Although all these works are interesting, it seems difficult in practice to obtain enough training material to get good results. Managing ontologies from project to project is also a complex process.
Ferrari *et al.* [19] aim at measuring and improving the completeness of specifications by checking whether some concepts and relations are present in the requirements. The interest of their work is that these concepts and relations are extracted from the documents from which the specification is derived (e.g. transcripts of meetings, standards, legacy documents). While this approach is not perfectly reliable, it may be helpful in our context.

Templates

A more modest but probably also more realistic approach to improve the quality of requirements consists in using templates, also called "boilerplates" [42]. We gave a typical example of template, recommended by the CESAMES method of systems engineering [37], in the introduction. After choosing an adequate template, the analyst fills in the blanks with the relevant information, e.g.

> **Req. 13** *The ⟨ABS⟩ shall ⟨release braking pressure⟩ within ⟨less than 100ms⟩ in case of ⟨a risk of wheel lock⟩.*

Note that we had to change slightly the third and fourth keywords of the template not to make the requirement sound awkward.
Reference books on requirement engineering, e.g. Pohl [49], Badreau and Boulanger [3], often mention templates, but they also warn about their limitations. They usually advise not to make templates mandatory. The main problem is that templates tend to be too "rigid": they prevent analysts from expressing what they really need. On one hand, multiplying templates causes management problems and decreases the value of templates. On the other hand, overly generic, multi-purpose templates tend to miss the goal of guiding the writing and reducing the ambiguity of requirements.
The Requirement Specification Language [12] can be considered as a set of templates. It makes it possible to write various types of elements (glossary terms, system goals, user stories…), including requirements. This language is relatively generic and seems focused on the attributes of requirements (priority, stakeholder, type…) rather than on the text itself.

Controlled Natural Languages

Templates aim at restricting the type of sentences that can be written in natural language. With that respect, they can be seen as a particularly rigid form of controlled natural language [38]. The so-called technical controlled natural languages, such as ASD-STE100 [61], are usually based on unrestricted natural language, by removing elements which may hinder understanding. In practice, however, systems engineers rarely use such technical controlled natural language when writing requirements. Rather, they follow "common sense" guidelines (e.g. keeping requirements short, avoiding passive voice, …) which are fundamentally not too far from these languages. There exist software tools

to check the compliance of a text with ASD-STE100 (e.g. HyperSTE, sold by Etteplan). Such tools could be indeed applied to corpuses of requirements. However, their interest is probably limited in our context as they perform only syntactic checks and do not "understand" the text.

As pointed out by Kamsties and Paech [33], methods that do not rely on specific information about the domain and even the system under analysis have a very good chance of missing problems in the requirements. Moreover, they tend to raise false alarms on elements which are unambiguous for the stakeholders. Therefore, we think requirements engineering methodologies should include contextual elements and more specifically, models. This is the topic of the next section.

6.2 Model-Based Approaches

Fully Formal Approaches

Most of the requirements engineering literature is dedicated to software systems. Unlike the physical systems we are interested in here, software systems are intrinsically authored specifications. This means that, for example, the inputs and outputs of a program are, by definition, formalizable. This is not the case for physical systems: a mechanical load can sometimes be modeled as a simple force vector, but it is usually much more complex. Even if the behavior of a program can be complex, it is at least deterministic. On the other hand, physical systems can and do fail randomly.

A fully logical approach to requirements engineering, such as the transition axiom method [39] or the Event-B process [1] cannot be applied to technical systems. Nonetheless, requirements engineering of software systems is a great source of inspiration. For instance, the Stimulus tool [30], developed by Argosim, aims at improving real-time requirements for embedded software by simulating them. The Stimulus approach assumes that requirements can be made completely formal. It is however similar in many respects to ours. In particular, it relies also on a co-design of requirements and models.

Logic-Based Controlled Natural Languages

Logic-based controlled natural languages are essentially natural language words which hide a formal language, e.g. a variant of first-order logic for ACE [23]. Writing requirements within such frameworks consists thus essentially in designing a mathematical model of the system under study. The natural language can be seen as an interface to this mathematical model. For example, in the CIRCE method [2], requirements must obey specific construction rules in order to build the internal model. Similarly, the templates of RSL [12] are used to generate views of the specification.

The use of logic-based controlled natural language raises two questions: first, are the underlying mathematical frameworks suitable for the particular needs of the system under study? second, does the use of natural language (instead of bare logical formulas) help to write and to understand requirements?

Regarding the first question, the challenge is to find a reasonable trade-off between the expressive power of the logical framework and the ability one has to perform automated demonstration in this framework, given the fundamental undecidability of the problem. Our experience is that logics that are used to specify software systems are too specific to be used for physical systems.

Regarding the second question, our answer is much more positive. Several tools exist that help the analyst to write requirements in logic-based controlled natural language, e.g. RAT [21] and PROPEL [10]. Our approach could benefit from inputs of these efforts.

Generation of Models from Textual Requirements and Vice-Versa

As pointed out Section 2.3, it is often the case that a group of textual requirements just describes with words a model such as a hierarchical diagram or a state machine. Ideally, it should be possible to obtain the model directly from these requirements. In practice, this goal is extremely difficult to achieve. Yue *et al.* [68] give a systematic review of approaches proposed in the literature. Among noticeable contributions on this topics, we can cite those done around CIRCE [24,2], PROPEL [10], NL-OOPS and LOLITA [44], and some others [14,54,58].

Writing down textual requirements from models can be also useful. Experts can define specialized models and, on demand, generate textual requirements which can be understood by non-expert stakeholders. Nicolás and Toval [45] present a systematic literature review of these approaches. Fockel and Holtmann [20] combine two methods, allowing the translation from (controlled) text to models and vice-versa. The generation of text from models seems technically easier than the reverse way. Meziane *et al.* [43] pointed out however that the generated text might look unnatural.

Some authors proposed to replace fully textual requirements by models, e.g. Bernard [5]. As explained in the previous section, we do not believe that such an approach can be generalized to all types of requirements.

Acknowledging the contributions of all these efforts, we believe that it is more pragmatic to co-design requirements and models because they complement rather than compete with each other.

Linking Requirements and Models

In SysML [22], it is possible to create links between requirements and other elements. Our work focuses on studying and dissecting the text of requirements, but SysML treats the text of requirements as a black box. Therefore, it does not allow to link specific parts of a requirement to a model. This feature seems however extremely useful. For instance, in Req. 2, it is important to link expressions "*power supply*", "*brake pressure*" and "*master cylinder pressure*" with their definition.

The Requirement Interchange Format (ReqIF) [46] and commercial requirements management tools such as DOORS make it possible to create hyperlink tags to the text of requirements. However, this feature is not necessarily used a lot in industry, probably because few tools are making use of these hyperlinks.

Kaindl [31] proposes a method named RETH (Requirements Engineering Through Hypertext), which he applied on an industrial project. Two aspects are mainly illustrated: linking scenarios (or more precisely actions in the scenarios) with requirements, and explicitly linking words in the requirements to their definitions. The former is similar to traceability links which are now relatively common, in academic papers and in the industry. In this experience report, the domain objects and concepts are mainly defined using paragraphs of natural language text. Other elements, such as models, can efficiently replace these definitions.

Design contracts for heterogeneous systems [65,55] are relations linking assumptions on the environment of a system to requirements for this system. These relations are implications: the requirements must be verified only if the assumptions are true. From this point of view, Req. 3 could be considered as (part of) a contract: with "When the master cylinder pressure is lower than 50 bars" an assumption on the environment, and "In state Off, the brake pressure shall be equal to the master cylinder pressure" the linked requirement. The formalism we propose allows us to write isolated assumptions as well as requirements, since they both are logical properties on the interfaces of systems. We did not focus on how to manage sets of assumptions and their relations to sets of requirements, but *a priori*, our work is not incompatible with what Westman and Nyberg [65], Sangiovanni-Vincentelli *et al.* [55] propose.

7 | CONCLUSION AND PERSPECTIVES

In this article, we have proposed a pragmatic approach to enhanced requirements engineering. This approach relies on two pillars: writing corpuses of requirements as partially structured hypertexts and co-designing these corpuses and models of the functional and physical architectures of the systems under study. It makes it possible to write less ambiguous requirements as well as to carry out a number of automated tests to check quality criteria. We reported experiments showing that it does not add a major burden to writers and readers of requirements. Moreover, it can be implemented progressively, by formalizing requirements step by step.

We identified two issues we would like to address in future work.

First, the industrial deployment of our approach should be supported by tools (with suitable graphical user interfaces) to author requirements and models, to link between different elements and to design and to call (libraries of) automated tests. There is nothing to invent here, as all these elements already exist. But it will involve some work to integrate them into a unified, user friendly framework.

Second, we focused in our work on architectural models and state machines. Other types of models are certainly needed for complete specifications. For example, we may need models describing use case scenarios for systems. Candidate formalisms to design these models include Business Process Modeling Notation (BPMN) [66] or UML/SysML sequence diagrams [22].

Acknowledgments

The present work has been realized in the framework of the first author's PhD thesis. This PhD thesis has been financed by the chair Blériot-Fabre at CentraleSupelec. The chair Blériot-Fabre is sponsored by SAFRAN group.

References

[1] Abrial, J.-R. (2010) *Modeling in Event-B: system and software engineering*. Cambridge, United Kingdom: Cambridge University Press.

[2] Ambriola, V. and Gervasi, V. (2006) On the systematic analysis of natural language requirements with CIRCE. *Automated Software Engineering*, **13**, 107–167.

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

[4] Batteux, M., Prosvirnova, T. and Rauzy, A. (2018) S2ML for structuring models. In *IEEE International Symposium on Systems Engineering (ISSE 2018)*. Roma, Italy: IEEE.

[5] Bernard, Y. (2012) Requirements management within a full model-based engineering approach. *Systems Engineering*, **15**, 119–139.

[6] Berry, D., Gacitua, R., Sawyer, P. and Tjong, S. F. (2012) The case for dumb requirements engineering tools. In *Proceedings of Requirements Engineering: Foundation for Software Quality* (eds. B. Regnell and D. Damian), vol. 7195 of *LNCS*, 211–217. Essen, Germany: Springer.

[7] Bertot, Y. and Castéran, P. (2004) *Interactive theorem proving and program development. Coq'Art: the calculus of inductive constructions*. Berlin Heidelberg, Germany: Springer.

[8] Bijan, Y., Yu, J., Stracener, J. and Woods, T. (2013) Systems requirements engineering–state of the methodology. *Systems Engineering*, **16**, 267–276.

[9] Castet, J.-F., Rozek, M. L., Ingham, M. D., Rouquette, N. F., Chung, S. H., Jenkins, J. S., Wagner, D. A., Dvorak, D. L. and Karban, R. (2015) Ontology and modeling patterns for state-based behavior representation. In *AIAA Infotech @ Aerospace*.

[10] Cobleigh, R. L. (2008) *PROPEL: An approach supporting user guidance in developing precise and understandable property specifications*. Ph.D. thesis, University of Massachusetts Amherst, Amherst, MA, USA.

[11] Colomb, R., Raymond, K., Hart, L., Emery, P., Welty, C., Xie, G. T. and Kendall, E. (2006) The object management group ontology definition metamodel. In *Ontologies for software engineering and software technology* (eds. C. Calero, F. Ruiz and M. Piattini), 217–247. Berlin Heidelberg Germany: Springer Verlag.

[12] da Silva, A. R. (2017) Linguistic patterns and linguistic styles for requirements specification (I): an application case with the rigorous RSL/business-level language. In *Proceedings of the 22nd European Conference on Pattern Languages of Programs*, 22:1–22:27. Irsee, Germany: ACM.

[13] Damian, D. and Chisan, J. (2006) An empirical study of the complex relationships between requirements engineering processes and other processes that lead to payoffs in productivity, quality, and risk management. *IEEE Transactions on Software Engineering*, **32**, 433–453.

[14] de Almeida Ferreira, D. and da Silva, A. R. (2009) A controlled natural language approach for integrating requirements and model-driven engineering. In *Proceedings of the fourth International Conference on Software Engineering Advances (ICSEA'09)*, 518–523. Porto, Portugal: IEEE.

[15] Do, Q., Cook, S. and Lay, M. (2014) An investigation of MBSE practices across the contractual boundary. *Procedia Computer Science*, **28**, 692–701.

[16] Fabbrini, F., Fusani, M., Gnesi, S. and Lami, G. (2001) The linguistic approach to the natural language requirements quality: benefits of the use of an automatic tool. In *26th Annual IEEE Computer Society - NASA Goddard Space Flight Center Software Engineering Workshop*, 97–105. Greenbelt, MD, USA: IEEE.

[17] Femmer, H., Méndez Fernández, D., Wagner, S. and Eder, S. (2017) Rapid quality assurance with requirements smells. *Journal of Systems and Software*, **123**, 190–213.

[18] Fernández, D. M., Wagner, S., Kalinowski, M., Felderer, M., Mafra, P., Vetrò, A., Conte, T., Christiansson, M.-T., Greer, D., Lassenius, C., Männistö, T., Nayabi, M., Oivo, M., Penzenstadler, B., Pfahl, D., Prikladnicki, R., Ruhe, G., Schekelmann, A., Sen, S., Spinola, R., Tuzcu, A., de la Vara, J.-L. and Wieringa, R. (2017) Naming the pain in requirements engineering: Contemporary problems, causes, and effects in practice. *Empirical Software Engineering*, **22**, 2298–2338.

[19] Ferrari, A., dell'Orletta, F., Spagnolo, G. O. and Gnesi, S. (2014) Measuring and improving the completeness of natural language requirements. In *Proceedings of Requirements Engineering: Foundation for Software Quality*, no. 8396 in LCNCS, 23–38. Essen, Germany: Springer.

[20] Fockel, M. and Holtmann, J. (2014) A requirements engineering methodology combining models and controlled natural language. In *Proceedings of the 4th International Model-Driven Requirements Engineering Workshop (MoDRE)*, 67–76. Karlskrona, Sweden: IEEE.

[21] Fraga, A., Llorens, J., Alonso, L. and Fuentes, J. M. (2015) Ontology-assisted systems engineering process with focus in the requirements engineering process. In *Proceedings of Complex Systems Design & Management (CSDM'14)* (eds. F. Boulanger, D. Krob, G. Morel and J.-C. Roussel), 149–161. Switzerland: Springer.

[22] Friedenthal, S., Moore, A. and Steiner, R. (2011) *A Practical Guide to SysML: The Systems Modeling Language*. San Francisco, CA 94104, USA: Morgan Kaufmann. The MK/OMG Press.

[23] Fuchs, N. E. and Schwitter, R. (1996) Attempto Controlled English (ACE). In *Proceedings of the First International Workshop on Controlled Language Applications*, 124–136. Katholieke Universiteit Leuven.

[24] Gervasi, V. and Nuseibeh, B. (2002) Lightweight validation of natural language requirements. *Software: Practice and Experience*, **32**, 113–133.

[25] Harel, D. (1987) Statecharts: a visual approach to complex systems. *Science of Computer Programming*, **8**, 231–274.

[26] Haskins, B., Stecklein, J., Dick, B., Moroney, G., Lovell, R. and Dabney, J. (2004) Error cost escalation through the project life cycle. In *INCOSE 14th Annual International Symposium*, 1723–1737. Toulouse, France: Wiley Online Library.

[27] IEEE 1220-2005 (2005) Standard for Application and Management of the Systems Engineering Process. Standard, IEEE, New York, NY, USA.

[28] ISO 26262 (2012) Functional Safety - Road Vehicle. Standard, International Standardization Organization, Geneva, Switzerland. URL `http://www.iso.org/iso/home.html`.

[29] ISO/IEC/IEEE 29148 (2011) International Standard - Systems and software engineering – Life cycle processes – Requirements engineering. Standard, ISO/IEC/IEEE, New York, NY, USA.

[30] Jeannet, B. and Gaucher, F. (2016) Debugging Embedded Systems Requirements with STIMULUS: an Automotive Case-Study. In *Proceedings of the 8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*. Toulouse, France. URL `www.erts2016.org`. Archive hal-01292286.

[31] Kaindl, H. (2005) A scenario-based approach for requirements engineering: Experience in a telecommunication software development project. *Systems Engineering*, **8**, 197–210.

[32] Kaiya, H. and Saeki, M. (2005) Ontology based requirements analysis: Lightweight semantic processing approach. In *Proceedings of the Fifth Quality Software International Conference (QSIC'05)*, 223–230. Melbourne, Australia: IEEE.

[33] Kamsties, E. and Paech, B. (2000) Taming ambiguity in natural language requirements. In *Proceedings of the thirteenth International Conference on Software and Systems Engineering and Applications (ICSSEA'00)*. Paris, France.

[34] Keil, J. M. and Schindler, S. (2019) Comparison and evaluation of ontologies for units of measurement. *Semantic Web*, 33–51.

[35] Kincaid, J. P., Fishburne Jr, R. P., Rogers, R. L. and Chissom, B. S. (1975) Derivation of new readability formulas for navy enlisted personnel. Tech. rep., Naval Technical Training Command, Memphis, TN, USA.

[36] Körner, S. J. and Brumm, T. (2009) RESI - A natural language specification improver. In *Proceedings of International Conference on Semantic Computing (ICSC'09)*, 1–8. Berkeley, CA, USA: IEEE.

[37] Krob, D. (2017) *CESAM: CESAMES Systems Architecting Method: A Pocket Guide*. CESAMES, www.cesames.net.

[38] Kuhn, T. (2014) A survey and classification of controlled natural languages. *Computational Linguistics*, **40**, 121–170.

[39] Lamport, L. (1989) A simple approach to specifying concurrent systems. *Communications of the ACM*, **32**, 32–45.

[40] Lebeaupin, B. (2017) *Vers un langage de haut niveau pour une ingénierie des exigences agile dans le domaine des systèmes embarqués avioniques*. Ph.D. thesis, Université Paris-Saclay (ComUE).

[41] Madni, A. M. and Sievers, M. (2018) Model-based systems engineering: motivation, current status, and needed advances. *Systems Engineering*, **21**, 172–190.

[42] Mavin, A., Wilkinson, P., Harwood, A. and Novak, M. (2009) Easy approach to requirements syntax (EARS). In *Proceedings of the 17th International Symposium on Requirements Engineering*, 317–322. Atlanta, Georgia, USA: IEEE.

[43] Meziane, F., Athanasakis, N. and Ananiadou, S. (2008) Generating natural language specifications from UML class diagrams. *Requirements Engineering*, **13**, 1–18.

[44] Mich, L. and Garigliano, R. (2002) NL-OOPS: A requirements analysis tool based on natural language processing. In *Proceedings of third International Conference on Data Mining (Data Mining III)*, 321–330. WIT Press.

[45] Nicolás, J. and Toval, A. (2009) On the generation of requirements specifications from software engineering models: A systematic literature review. *Information and Software Technology*, **51**, 1291–1307.

[46] OMG (2016) Requirements Interchange Format. Tech. rep., Object Management Group. URL `http://www.omg.org/spec/ReqIF/`.

[47] Ormandjieva, O., Hussain, I. and Kosseim, L. (2007) Toward a text classification system for the quality assessment of software requirements written in natural language. In *Proceedings of the fourth international workshop on Software Quality Assurance (SOQUA'07)*, 39–45. New York, NJ, USA: ACM.

[48] Parra, E., Dimou, C., Llorens, J., Moreno, V. and Fraga, A. (2015) A methodology for the classification of quality of requirements using machine learning techniques. *Information and Software Technology*, **67**, 180–195.

[49] Pohl, K. (2010) *Requirements engineering: Fundamentals, principles, and techniques*. Berlin Heidelberg, Germany: Springer.

[50] Rajan, A. and Wahl, T. (2013) *CESAR: Cost-efficient methods and processes for safety-relevant embedded systems*. Vienna, Austria: Springer.

[51] Rauzy, A. and Haskins, C. (2018) Foundations for model-based systems engineering and model-based safety assessment. *Journal of Systems Engineering*.

[52] Reif, K., ed. (2015) *Automotive Mechatronics: Automotive Networking, Driving Stability Systems, Electronics*. Wiesbaden, Germany: Springer Vieweg.

[53] Saavedra, R., Ballejos, L. C. and Ale, M. (2013) Quality properties evaluation for software requirements specifications: An exploratory analysis. In *Proceedings of the XVI Workshop de Ingeniería en Requisitos, WER'13*, 6–19. Universidad ORT Uruguay.

[54] Samarasinghe, N. and Somé, S. S. (2005) Generating a domain model from a use case model. In *Proceedings of 14th International Conference on Intelligent and Adaptive Systems and Software Engineering* (eds. R. T. Hurley and W. Feng), 278–284. Toronto, Canada: ISCA.

[55] Sangiovanni-Vincentelli, A., Damm, W. and Passerone, R. (2012) Taming Dr. Frankenstein: Contract-based design for cyber-physical systems. *European journal of control*, **18**, 217–238.

[56] Schmidt, R., Lyytinen, K., Keil, M. and Cule, P. (2001) Identifying software project risks: An international delphi study. *Journal of Management Information Systems*, **17**, 5–36.

[57] Schwaber, K. and Beedle, M. (2001) *Agile software development with Scrum*. Upper Saddle River, NJ, USA: Prentice Hall.

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

[59] Spivey, J. M. (1992) *The Z notation: a reference manual*. Upper Saddle River, NJ, USA: Prentice Hall.

[60] Stachowiak, H. (1973) *Allgemeine Modelltheorie*. Vienna, Austria: Springer-Verlag.

[61] STEMG (2017) ASD Simplified Technical English. Tech. rep., ASD, Brussels, Belgium. URL `http://www.asd-ste100.org/`.

[62] van Lamsweerde, A. (2001) Goal-oriented requirements engineering: A guided tour. In *Proceedings of the fifth international symposium on Requirements Engineering*, 249–262. Toronto, Ontario, Canada: IEEE.

[63] Verner, J., Cox, K., Bleistein, S. and Cerpa, N. (2005) Requirements engineering and software project success: an industrial survey in australia and the us. *Australasian Journal of Information Systems*, **13**.

[64] Warmer, J. B. and Kleppe, A. G. (1998) *The Object Constraint Language: Precise Modeling With UML*. Object Technology Series. Hoboken, New Jersey, USA: Addison-Wesley.

[65] Westman, J. and Nyberg, M. (2018) Conditions of contracts for separating responsibilities in heterogeneous systems. *Formal Methods in System Design*, **52**, 147–192.

[66] White, S. and Miers, D. (2008) *BPMN Modeling and Reference Guide: Understanding and Using BPMN*. Lighthouse Point, FL, USA: Future Strategies Inc.

[67] Yu, E. (1995) *Modelling strategic relationships for process reengineering*. Ph.D. thesis, University of Toronto, Department of Computer Science.

[68] Yue, T., Briand, L. C. and Labiche, Y. (2011) A systematic review of transformation approaches between user requirements and analysis models. *Requirements Engineering*, **16**, 75–99.

[69] Zave, P. and Jackson, M. (1997) Four dark corners of requirements engineering. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, **6**, 1–30.