A Formal Framework for the Formalization of Informal Requirements

Florent Peres, Jing Yang, Mohamed Ghazel

A Formal Framework for the Formalization of Informal Requirements[1]

Florent PERES, Jing YANG, Mohamed GHAZEL
Univ. Lille Nord de France, F-59000 Lille,
IFSTTAR, ESTAS, F-59650 Villeneuve d'Ascq
Email: {florent.peres, jing.yang, mohamed.ghazel}@ifsttar.fr

Abstract. **Systems' requirements are usually written in a natural language since it generally means a greater understanding among the various stakeholders. However, using an informal language potentially gives rise to interpretation problems, which are to be resolved prior to using (automated) verification techniques. This article tackles an important issue pertaining to requirement engineering: how to guide and help requirements' formalization? In order to support the formalization process, we propose a methodology based on a formal structure, which is the corner stone of the refinement process. The operating mode of the refinement process is highly iterative: the aforementioned structure is constructed incrementally until its validity is formally obtained. Although this process is formally backed up, it is a fundamentally subjective one, which means that interpretation errors can still occur. In case of errors, it is essential to be able to backtrack refinements until an interpretation error is found. This is why we require that each refinement be associated with a justification which may subsequently be analyzed in case an error occurred during the verification phase. This formalization process was designed to be used alongside an (unspecific) engineering process, in charge of the implementation. Once the formalization is complete, it is checked against the implementation using testing techniques, or directly against an implementation model via model-checking.**

Keywords: *formal framework, requirement engineering, requirement formalization, requirement traceability, critical systems, software engineering.*

1. Introduction

When engineering a system, it is recommended that goals, functionalities and constraints are identified as precisely as possible. This is what constitutes the requirement documents or the specifications, for short. In this (or these) document(s), we may find recommendations, but also requirements: sentences that must be valid on the final system.

Requirements are usually written in natural language (e.g. English), either because formal languages are not known by the requirement engineers, or because it is too early in the conception process to use such a specification form (i.e. there are not enough details to successfully write a formal requirement). The main issue with writing requirements in a natural language is that they cannot serve as inputs for automated verification techniques.

Indeed, formal languages are mandatory to bring into play automated verification techniques, since they are the only languages computers can understand. Moreover, formal languages are not ambiguous: this means a sentence cannot be understood in different ways. This implies that humans too can benefit from using a formal language since the requirement will be fully understood without a prior interpretation from the user. The main problem is that both writing and reading activities are not simple to accomplish when it comes to formal languages.

The formalization process consists in writing requirements using a mathematical formal notation. It can be a hard task if done directly from an informal requirement and there cannot be any true guide to formalization: only expertise and experience (and maybe some heuristics like in [31]) can help to know how to get a sound formal writing.

[1] This research has been partially supported by Region Nord Pas de Calais and European fund Feder under the FUI National project FerroCOTS, labelled by i-Trans (N° 10030031M081).

To ease the process, the generally recommended solution is to perform some kind of refinement: either working on informal requirements as in the case of KAOS methodology [6], or on formal abstract requirements like for B method [1]. Each solution has its limits.

The methodology proposed in this article only concerns a small part of requirement engineering and may be integrated in other higher level methodologies (like KAOS for example). Here we deal mainly with software engineering, i.e. the logical/control part of systems. As hypotheses, we suppose that the system implementation is ready (or at least, that the required low level information of the future implementation are available), and that a requirements' document comprising all the requirements to be formalized is readily available. What we want to achieve is to be able to automatically check the validity of each requirement on the given implementation. The verification step is outside the scope of this work: we only want to focus on formalization using a refinement method. We propose a refinement methodology supported by a formal structure: the Pseudo-Requirement Graph. It consists in two types of nodes: refinements and pseudo-requirements. Refinements can help to clarify or on the contrary to abstract some requirements parts. They can also help to decompose a requirement into small pieces which may be useful to re-use some already written part of requirement and also to come closer to a formalization. In summary, the idea is to proceed by iterations in which the requirement is modified so that it becomes closer and closer to a formal logical assertion that can be used by an automatic verification tool (a model-checker for example).

Although expertise is necessary to successfully accomplish the requirement formalization, it may not be sufficient in light of the safety objectives of safety critical systems such as trains, planes, etc. Indeed, an error can never be excluded, even if the system is implemented by the best experts. When an error is detected, it is imperative that the source of the problem be backtracked, as soon as possible, so that the error can be quickly corrected. But also, to possibly learn from this mistake and therefore take necessary measures for a similar error not to occur again. This involves using traceability mechanisms, to be able to backtrack the refinement process, in order to analyze the interpretation choices and their consequences.

The paper is structured as follows: Section 2 gives an overview of the related works about requirement refinement methods. In Section 3, we present the formal framework for requirements' formalization by introducing the formalization process firstly then the *pseudo-requirement graph* which backs our refinement process. In Section 4, we give an informal explanation about the *pseudo-requirement graph.* In Section 5, a case study featuring a train gates control system is presented. Finally, Section 6 recalls the main contributions and gives an overview of the future work.

2. Related Works

To get a better idea of how our approach compares to others, we will restate our problem while emphasizing some key-words which will help us to guide this comparison.

Our main goal is to *automatically* check requirements. The requirements we are using are written in an informal form, while automation requires that a *formal specification* be provided. Then, we need to refine the input *informal* requirements, so that in the end each of them (in the ideal case) is entirely *formalized*. Because refinement from an informal to a formal language is a highly subjective process prone to (incorrect) interpretations, we also want to keep a *trace* of such interpretations, so that in the case the verification tool states that the requirement is not fulfilled by the system, it is easier to backtrack the refinements and check each of them and find what could have gone wrong.

Due to the quite strong (and not new) interest in finding techniques to transform an informal statement into a formal one, this section does certainly not pretend to be exhaustive. Instead, we wanted to give a good grasp about the available techniques and methodologies.

In Table 1, the related works are sorted according to 1) **Formal**: after the methodology is applied, the requirement is formalized; 2) (resp. 3)) **Informal** (resp. **Formal**) Refinement: refinements are done while the requirement is in an informal (resp. formal) form; 4) **Documentation**: associate the refinement with some documentation (typically to explain the interpretation choices); 5) Natural Language Processing (**NLP**): (semi-)automatic processing of the informal requirement.

Table 1. Quick Taxonomy of related Works

Paper	Formal	Requirement			Tr. links	NLP	Type
		Informal	Formal	Doc.			
[38]	*						A
[17]	*	* (1 lvl)			~		A(B)
[3]	*						A
[6],[7]	*	*			*		AB
[35]		*			*		B
[41]		*			*		B
[2]		*			*		B
[26]	*						A
[32]	*						A
[39]	*						A
[4]	*						A
[5]	*	* (1 lvl)			~		A(B)
[37]	*		*				A
[40],[25]	*						A
[36]		*					B
[27]	*						A
[30]	*						A
[43]	*		*				A
[19]	*						A
[33]						*	D
[11]		*		*	~ (rationale)		BC
[21]						*	D
[18]	*		*				A
[23]		*			*		B
[24]						*	D
[10]		*			*		B
[29]						*	D
[42]	*	(links only)			*		AB
[16]						*	D
[28]	*	*			~ (orig. req)		AB
[34]	*		*	~	~		ABC
[31]	*	*		*	*		ABC
[15]		* (colors)		~	~ (color trail)		B
[8]						*	D

First, lots of works propose to directly formalize the requirements. These works essentially target software engineering (and this is why direct formalization can actually be used). They are identified by Type A in Table 1. Some of these works propose to make the refinement inside the formal world. The requirement is then written incrementally: at each refinement step, at least one constraint is added.

Then comes the Goal-Oriented Requirement Engineering (GORE) methodologies and the likes (Type B or AB). GORE methodologies advocate to focus on the question *why* while analyzing and constructing requirements (i.e. to study a requirement based on its utility and purpose in the system). These methodologies are highly iterative, and then natively support informal refinement and traceability (of refinements). These methodologies are very generic: they can be applied on a very wide

range of systems; they were designed to be a unique framework: (ideally) all aspects concerning requirement engineering should be handled by such methodologies. One of them, KAOS [6], has a stronger focus on software engineering (although it is far from being restricted to that area) and allows formalization and formal refinements (using the extension proposed in [7]). [17] and [5] are A(B)-type because the refinement part is only done using an intermediate language (i.e. there is only one refinement step).

Automatic or semi-automatic techniques (Type D) can be applied on informal requirements using natural language processing techniques. These techniques can parse a text written in a natural language (English is the most common) and are capable of identifying agents, roles, etc. They can build a class diagram or even a behavioral model. Although automation is a great feature, these techniques are not capable of taking design-decisions like precising an ambiguity: so they cannot automatically produce a sound formal form (otherwise this would mean that the "informal" requirement was in fact "formal"), but they are valuable tools for early analysis of requirements.

Finally, works whose type is ABC comply with our documentation requirement: in [34], the authors use *breadcrumbs*, or a new (formal) knowledge about the system to modify the existent requirements whose behavior may have to be changed according to that new *breadcrumb*. Breadcrumbs are thought to be a kind of documentation and therefore must be kept along with the requirements. In [31], a *Goal Argumentation Method* is proposed: it comprises a decision procedure, clarification techniques and an argumentation model. First, problems/weaknesses/ambiguities are checked using some predefined *techniques*. From this problem, one has to explore the alternative solutions, then pick one along with arguments in its favor. As for the clarification technique, it consists in labeling the words of the goal according to their type of fuzziness: ambiguous, over-general, vague, synonymous. For each type of fuzziness, the authors propose some heuristics to help clarify the goal. Finally, an argumentation must be given with an *argumentation model*, which is a graph-like structure storing relation between arguments (implication, counter-implication).

As for ~, this symbol is used in Table 1 to denote an implicit or unclear feature of the work. For example, in [28], requirements are sorted and refined using pattern storing whose original requirement(s) it is clarifying. So, there is some kind of traceability, even if it remains limited.

3. A Formal Framework for Requirements Formalization

Before going into more details about the framework, let us review how we propose to use it.

3.1. Usage of the Framework

The formalization process will be done inside a data structure called *pseudo-requirement graph*. It is constituted of two types of objects: *pseudo-requirements* and *refinements*. A pseudo-requirement is either a requirement or a part of a requirement (or an *atom*, as atoms are generally parts of a requirement). This is a *top-down* process: it starts from the high level requirements (i.e. rough pseudo-requirements that are actually true requirements, directly taken from the requirements' document) and end with directly formalizable pseudo-requirements (which we call *atomic pseudo-requirements*).

The activity diagram of Figure 1 shows the control flow. First, the graph validity must be tested. If it is indeed valid, then the requirements of the graph are checked. To do the actual checking, we use *model-checking*, but nothing prevents one to use other techniques, as soon as they are compatible with the formal language used (here CTL* - CTL for Computation Tree Logic).

On the one hand, if the verification phase confirms that all requirements are met, then everything is fine and the process ends. On the other hand, if an error is found by the verification tool, the graph must be backtracked from the pseudo-requirements which have no children (at this step, the graph must be valid, which means that they are atomic pseudo-requirements). At each step of the backtracking, the user must check (with its own expertise, as this process *cannot* be automatized) whether the refinement rationale leading to the current pseudo-requirements is justified. Should an error be found, the faulty sub-graph, whose root is the faulty refinement, must be erased and the formalization must be resumed from that point. If there is no error in the current refinement, then the same checking must be operated on the parents of the current pseudo-requirement.

When the graph is not valid, it must be corrected either by modifying/deleting existing refinements/pseudo-requirements or by adding a new requirement using a new refinement. First, the user must pick a valid pseudo-requirement: any requirement — which has not been already refined and which is not a formalized atom — would do. When an appropriate pseudo-requirement is found, a refinement category must be picked amongst the following: *precision, abstraction, correction, decomposition* or *generic*. These categories will be further explained in section 4. A refinement contains necessary information such as: parts involved in the refinement, justification of the refinement, etc. This information has to be filled according to the chosen category (some fields may be omitted, depending on the category, except for *generic*). Once the refinement has been fully specified, the requirement (or the set of requirements if the category is a *decomposition*) holding the result of the refinement has to be written. The final step is to link the refinement to the resulting requirement and add it to the graph.

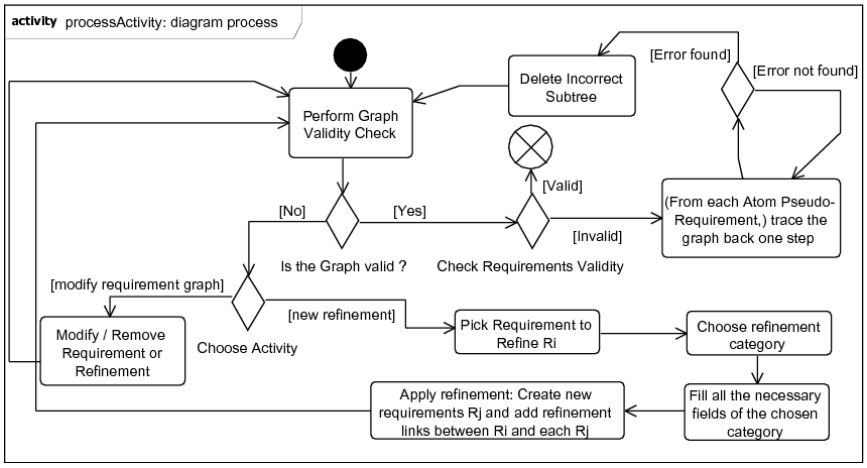

Figure 1. Formalization Process

Practically speaking, the formalization coherence between the pseudo-requirements does not have to be done manually. In fact, *atomic* pseudo-requirements are the only objects for which formalization is actually manipulated by the user. Once all the *atoms* are formalized, it is possible to automatically fill the formalization fields of all the pseudo-requirement parents until the root.

Two limitations, coming from the formal verification methodologies, are to be discussed here. In a previous paragraph we explained how the graph allows us to backtrack refinements when an error is detected at the verification phase. It was implied that the error was coming from the formalization itself, but it could also come from an error in the model. In that case, one would not find any error in the refinement process, but before knowing where the error actually comes from (the implementation model or the formalization?), there may be a lot of tedious review work to do. Another -even more serious- problem would be that an error could exist in the implementation model (or in both the implementation model and the formalization), but the (potentially incorrect) formalization is still valid on the incorrect model: there, the verification tools are not capable to actually detect any deficiencies in the implementation model and/or in the formalization. Besides formal verification techniques, test could also be used based on the basis of the obtained properties

3.2 Pseudo-Requirement Graph

Now that the general usage of a requirement graph has been explained, we will focus on formally defining the structure itself.

The Formal Logic. We will first present the formal language we chose as the target language.

Definition 1. *Let* Prop *be the set of atomic propositions. Let* Pred *be a set of variables. A* requirement formalization *is a formula whose grammar is defined as follows:*

$$f ::= \mathbf{Prop} \mid f \wedge f \mid \neg f \mid f \mathbf{U} f \mid f \mathbf{S} f \mid \mathbf{X} f \mid \mathbf{Y} f$$

The set of all possible formulas over Prop *is denoted* F_{Prop} *or the shorter* F *whenever* Prop *need not be precised. Please note that we have used the set* Prop *as a grammar rule: any element of* Prop *can be chosen from the rule* Prop. *The operator U is called* until *operator, and X is the next operator. S and Y, called since and* yesterday *operator are respectively the past-dual of U and X.*

Presenting the semantics of such a composite logic (it is an assembling of various logics) is not interesting with regard to the subject of the article. Operators *U* and *X* are quite usual and their semantics may be found in [13]. As for *S* and *Y*, please refer to [20].

The other temporal operators are defined from those previous four and the usual Boolean connectors:

- *finally:* $\mathbf{F}\phi = \mathbf{true}\ \mathbf{U}\phi$
- *globally:* $\mathbf{G}\phi = \neg\mathbf{F}\neg\phi$
- *weak until:* $\alpha \mathbf{W} \beta = (\alpha U \beta) \vee (G\alpha)$
- *previously:* $\mathbf{P}\phi = \mathbf{true}\ \mathbf{S}\phi$
- *historically:* $\mathbf{H}\phi = \neg\mathbf{P}\neg\phi$
- *before:* $\alpha \mathbf{B} \beta = (\alpha \mathbf{S} \beta) \vee (\mathbf{H}\alpha)$

If necessary, the logic can also be extended with predicates and with the usual existential and universal quantifiers.

$$f ::= \ldots \mid (\exists x)(f) \mid (\forall x)(f) \mid Pr(x_1, \ldots, x_n)$$

where $x, x_1, \ldots, x_n$ are variables and *Pr* is a predicate name.

The Structure

Definition 2. *A* Pseudo-Requirement Graph *is a tuple*
<*P*, *R*, **Pre**, **Post**, **Link**, **What**, **Why**, **How**, **Desc**, *F*, **Type**>, *in which:*

- *P is the set of pseudo-requirements;*
- *R is the set of refinements, such that* $P \cap R = \phi$*;*
- **Pre**: $R \rightarrow P$ *is the refinement* source-link
- **Post**: $R \rightarrow 2^P$ *is the refinement* target-link
- **Link**: $R \rightarrow F$ *is the refinement* formal-link
- **What**: $R \rightarrow$ *String is the* what-property *of the refinement*
- **Why**: $R \rightarrow$ *String is the* why-property *of the refinement*
- **How**: $R \rightarrow$ *String is the* how-property *of the refinement*
- **Desc**: $P \rightarrow$ *String is the* desc-property *of the pseudo-requirement*
- *F*: $P \rightarrow F$ *is the* formalization *of the pseudo-requirement*
- **Type** $\in$ *{Req, Part, Atom} is the* type *of the pseudo-requirement*

A *requirement graph* is composed of two types of nodes: *pseudo-requirements* and *refinements*. A pseudo-requirement "holds" three items: *Desc*, the informal description of the pseudo-requirement; *F* its formalization and *Type*, as its name suggests it, its type. A pseudo-requirement can be of type *Req*, in which case it is a *genuine requirement*. It can be of type *Part*, in which case it is no longer a *requirement* but only a *part* of a requirement. And it can finally be of type *Atom*, which means it is directly formalizable.

A refinement holds the rationale information which are useful for traceability: *Why*, *What* and *How*? To a refinement is also associated the *link* between the pseudo-requirement it refines and the pseudo-requirement bearing the result of the refinement.

Finally, the relation between pseudo requirements and refinements is given by the two relations *Pre* and *Post*. Pre associates a refinement to the pseudo-requirement is it refining, while Post associates a refinement to the set of pseudo-requirements which are the result of the refinement.

Definition 3. *The set of ancestors* $pred_\infty(p)$ *of a pseudo-requirement p is defined as follows:*

- $pred_1(p) = \{\ t \in P \mid (\exists x \in R)(t = \mathbf{Pre}(x) \wedge p \in \mathbf{Post}(x))\ \}$[2]
- $pred_i(p) = pred_{i-1}(p) \cup \{\ t \mid (\exists x)(t = \mathbf{Pre}(x) \wedge (\mathbf{Post}(x) \cap pred_{i-1}(p)) \neq \phi)\ \}$
- $pred_\infty(p) = \bigcup_{i=1}^{\infty} pred_i(p)$

The computing stops because P is bounded and pred is monotonic.

$pred_1(p)$ gives the direct ancestors of p: if $p \in \mathbf{Post}(x)$ and $t = \mathbf{Pre}(x)$ then p refines at least a part of t by the refinement x and then $t \in pred_1(p)$.

$pred_i(p)$ gives the set of ancestors, distant from p by at most i "generations". Being at most of i^{th} generation means being of $(i-1)^{th}$ generation *or* being the direct ancestor of any $(i-1)^{th}$ generation pseudo-requirement.

The set of all ancestors of p is then the infinite union of all the ancestors up to i^{th} generation. Please note that this set is not infinite and must reach a fixed point as there can only be a finite amount of pseudo-requirements and because $Card(pred_i) \geq Card(pred_{i-1})$.

Definition 4. *The set of successor* $succ_\infty(p)$ *of a pseudo-requirement p is defined as follows:*

- $succ_1(p) = \{\ t \in P \mid (\exists x \in R)(t \in \mathbf{Post}(x) \wedge p = \mathbf{Pre}(x))\ \}$
- $succ_i(p) = succ_{i-1}(p) \cup \{\ t \mid (\exists x)(t \in \mathbf{Post}(x) \wedge \mathbf{Pre}(x) \in succ_{i-1}(p))\ \}$
- $succ_\infty(p) = \bigcup_{i=1}^{\infty} succ_i(p)$

$succ_1(p)$ gives the direct successors of p: if $p = \mathbf{Pre}(x)$ and $t \in \mathbf{Post}(x)$ then t refines at least a part of p by the refinement x and then $t \in succ1(p)$.

$succ_i(p)$ gives the set of successors, distant from p by at most i "generations". Being at most the i^{th} successor either means being at most the $(i-1)^{th}$ successor *or* being the direct successor of any at most $(i-1)^{th}$ pseudo-requirement successor.

As previously the set of all successor is given by an infinite union, but the set $succ_\infty(p)$ is finite because it eventually reaches a fixed point (the proof uses the same arguments as previously).

Definition 5. *Validity of Requirement Graph*

Let $G = \langle P, R, \mathbf{Pre}, \mathbf{Post}, \mathbf{Link}, \mathbf{What}, \mathbf{Why}, \mathbf{How}, \mathbf{Desc}, F, \mathbf{Type}\rangle$ *be a requirement graph. G is a* valid Requirement Graph *iff all the following sentences hold:*

- $(\forall x \in R)(Link(x): F_{\mathbf{Post}(x)})$*, i.e. the formula, which gives the link between* **Post** *pseudo-requirements, only uses these* **Post** *pseudo-requirements as terms.*
- $(\forall x \in R)(\forall y \in Post(x))(y$ *is used in* $Link(x))$*. Informally, this means that all the pseudo-requirements obtained after the x refinement are used in* $Link(x)$*.*
- **Desc** *is an injective function. This means that whenever two (pseudo-) requirements hold the same informal description then they must be the same (pseudo-)requirement.*
- $(\forall p)(p \notin pred_\infty(p))$*. This property ensures that there is no refinement loop.*
- $(\forall p)(Type(p) = Atom \Rightarrow (\forall x \in pred_\infty(p))(Type(x) \neq Atom))$*. In other words, if a pseudo-requirement is an* Atom*, then none of its ancestors can be an* Atom*. An* Atom *pseudo-requirement cannot be further refined as it is directly formalizable.*
- $(\forall p)(succ_\infty(p) = \phi \Rightarrow Type(p) = Atom)$*. A pseudo-requirement that does not have any successor must be of type* Atom*.*

[2] Note that $(\forall p)(Card(Pred_1(p)) = 1)$

- $(\forall p)(\exists x)(\exists p \in Post(x) \wedge Link(x) = \bigwedge_{i \in Post(x)} i) \Rightarrow (\forall q \in succ_{\infty}(p))\ (Type(q) \neq Req)$. *This signifies that whenever a refinement x does not link* **Pre**(*x*) *with* **Post**(*x*) *by a logic formula in conjunctive form, then every ancestor is of type* Part *or* Atom*, but not* Req. *Indeed, as soon as a link is more complex than a conjunction, the verification of any* **Post**(*x*) *is meaningless when performed alone. In other words, this means that the pseudo-requirement of* **Post**(*x*) *are no longer genuine* requirements *because they do not specify a complete requirement (only a* part*).*

- $(\forall p)(\exists r)(Pre(r) = p \Rightarrow F(p) = (Link(r)$ *in which every* $q \in Post(r)$ *is replaced by* $F(q)))$. *This property ensures that the formalization feedback is done correctly: the formalization f of a pseudo-requirement p must be equal to the formula given by the link of the refinement r, in which every pseudo-requirement* $q \in Post(r)$ *is replaced by its formalization* $F(q)$*. The formalization* $F(p)$ *of a pseudo-requirement p, such that* $Type(p) \in \{Req, Part\}$*, is computed from the set* $succ_{\infty}(p) \cap \{p \mid Type(p) = Atom\}$*, i.e. the pseudo-requirements which are the successors of p* but also *which are of type* Atom.

- $(\forall x)(\forall y)(Pre(x) = Pre(y) \rightarrow x = y)$*, if two refinements refine the same requirement, then they must actually be the same refinement.*

4. A Closer View on the Pseudo-Requirement Graph

Three items are associated with a pseudo-requirement: **Desc**, its informal description; **F** its formalization in terms of CTL*; and finally its **Type**. Let us review the different types that a pseudo-requirement can be: **Req**, the pseudo-requirement is actually a requirement: if such a pseudo-requirement is formalized, then it means that it can be passed to the verification tools; **Part**, in that case, the pseudo-requirement is only a *part* of a requirement, which means that, although it could be passed to a verification tool, the result of the verification would not be useful. In other words the verification of a pseudo-requirement whose type is *"part"* is meaningless; **Atom**, in the ideal case, an atomic pseudo-requirement is reducible to a variable: in all rigor, if an atomic pseudo-requirement is not a variable, this means that a decomposition could be applied, but sometimes, this decomposition may be a bit too "pedantic" and would not bring much (or even no) useful information. Anyway, when a pseudo-requirement is atomic, this means that its formal and informal descriptions are actually very close semantically (and often syntactically): no interpretation has to be done and no implicit fact has to be known in order to understand the passage from the informal text to its formalization.

Four elements are associated with a refinement *r*: **What**, it describes what part of the pseudo requirement *Pre*(*r*) is refined. What a **what**-property can hold is hard to explain because of the various style of informal writings. Ideally we think of it as a "conceptual unit" that should be small (the more precise the **What** property of the refinement, the more precise the review in case of an error). But more importantly, it must be coherent with the **Why** and **How**-property; **Why**, this property explains why the refinement is helpful: this can often be answered by first asking why *this* choice of refinement instead of one another, or why *this* part of the refinement instead of one another. In any case, this should be a practical information of an implicit fact about the system's domain or implementation; **How**, describes the result of modifying what is pointed out by the **What**-property according to what is explained by the **Why** property. This is the result of the refinement on the **What** part (the actual result of the refinement being *Post*(*r*)); finally, **Link** describes how the *Pre*(*r*) pseudo-requirement is *formally* related to the requirement(s) in *Post*(*r*): the *link*-property is a CTL* formula whose terms all belong to *Post*(*r*), and for which all the elements of *Post*(*r*) are terms of **Link**.

Pragmatically speaking, there are categories of refinement of whose some properties may be hard or useless to fill. We will exhibit some of such categories of refinement, but others can be defined according to the type of requirement one has to formalize. We think the following four are quite generic and should therefore be generally applicable and useful: *Precision*: when a part must be disambiguated and/or precised; *Abstraction*: when a part is described in too much details regarding the system which is being studied (in other words, when a part is out of scope); *Correction*: when the pseudo-requirement is incorrect; *Decomposition:* when the pseudo-requirement can be decomposed in several parts. Any other type of refinement is considered *generic*.

These categories are only a shortcut to avoid using *"generic"* refinements. Their choice is purely subjective but must be coherent with the information given and the result of the refinement (although nothing can help us to ensure this coherence, except manual proof reading).

Property 6. The attributes **What**, **Why** and **How** of *generic* refinements are mandatory. If any of those is not set, then the requirement graph is invalid. Moreover *Link*(*r*) = *Post*(*r*) and Card(*Post*(*r*)) = 1.

Property 7. The attribute **Why** of *Precision/Abstraction*-refinements may be omitted, as the category explicitly tells the reason of the refinement. However, this attribute can still be used to further the precision of the motivation for the refinement and, in so doing, to add some implicit knowledge that would otherwise stay hidden to the backtrack review process in case of an error.

Property 8. The attributes **What** and **How** of a *decomposition*-refinement *r* may be omitted, as *Pre*(*r*) is **What**, except for some operators which will be formalized in **Link** and **How** is the combination of *Post*(*r*) and **Link**.

Definition 9. *A Pseudo-requirement whose type is Req is called a* requirement. *Only* requirements *are to be checked.*

5. Graphical Representation and Example

A graphical representation has the advantage — over a linear textual form — to emphasize on the link between pseudo-requirements and refinements but also to present an overview of the overall process. Before going further and show the formalization process through an example, let us choose a graphical convention to represent the pseudo-requirement graph.

Pseudo-requirements are drawn using a rectangle. Pseudo-requirements are pictured with a different border according to their type: a dashed line for true requirements, a thin line for requirement parts and a thick line for atomic requirements.

Refinements are drawn using a rounded rectangle. Whenever a refinement is applied on the whole source requirement, the **What** and **How** properties are completely omitted; if not, the **What** property is represented by a gray zone in the source requirement and the **How** is normally included in the refinement. **Links** are only precised in refinement *r* when *Link* ≠ *Post*(*r*).

The requirement whose pseudo-requirement graph is pictured in Figure 2 is relative to a train gates control system. The requirement is the following:

> *The driver shall be informed of the deployment authorization state of the bridge plate[3] before the opening of the doors, after having detected the presence of a low railway platform of 840mm at a train station and the train speed is low.*

The first important remark is that the property *F* of the requirement *Rq* is (obviously) not filled until the end. *Rq* is decomposed into two requirements *Rq*1 and *Rq*2, which are linked by the CTL* formula: []($Rq2$ ***U*** $Rq1$). The requirement *Rq* will then be formalized as soon as both *Rq*1 and *Rq*2 are formalized. Backtracking a formalization can easily be automatized, so the person in charge of writing this pseudo-requirement graph would not actually write the formalization of anything else other than atomic pseudo-requirements.

*Rq*1 is first abstracted then formalized, because we then arrived at the implementation level and *Auth_D* is the Boolean variable representing the signal. *Rq*2 is decomposed twice, leading to four atomic pseudo-requirements. We will not go into more details, as we think reading the pseudo-requirement graph should be quite straightforward. One interesting point to notice is the refinement R_{22111} (on the bottom left), whose category is *generic*. This could be considered as decomposition, but this would be a bit awkward, so here is a typical usefulness of a *generic* refinement. If such a refinement would often appear, it could be advantageously replaced by a new type of refinement category.

[3] Bridge plates are devices helping to get in or out of the train.

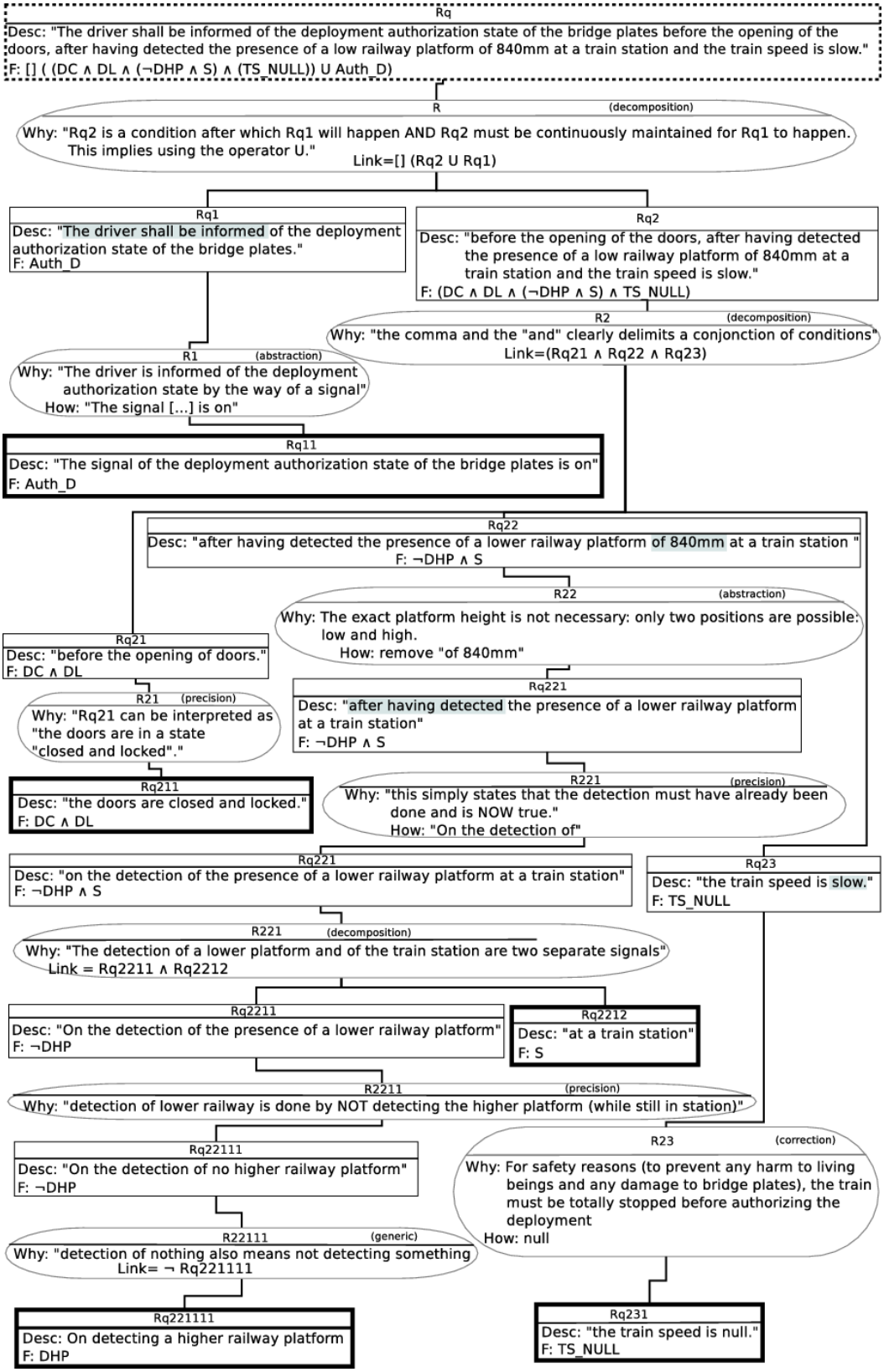

Figure 2. Example of a Pseudo-Requirement Graph

6. Conclusion

We have proposed a refinement process supported by a formal graph structure: the *requirement graph*. A requirement graph consists in two types of nodes: pseudo-requirements and refinements. Pseudo-requirements can be a part of a requirement or a requirement itself, and refinements relate two or more pseudo- requirements. Refinements store information of what exactly is refined, why the refinement must take place and how it takes place. In the end, the pseudo-requirements are formalized using the CTL* logic. Our proposal is only a shallow analysis of a requirement in the sense that it formalizes a requirement in a form that is more descriptive than operative (even though the bounds between the two are quite fuzzy). In particular, and contrary to other techniques, it does not propose a way to build a behavioral model from a set of requirements, because basically, we only seek to obtain a *property* to hold on the implementation. This is why our formalization technique could be incorporated in methodologies focusing on software systems (or similar systems), like KAOS.

Although this approach is quite usable independently, it seems preferable to allow some interaction points with the team in charge of the implementation in order to achieve a sound formalization. Indeed, at the formal level, it may be necessary to possess some knowledge about the architecture of the implementation and its interactions. In this perspective, this approach should be better used as a late requirement engineering tool.

As perspectives, we seek to explore how to better manage re-engineering, and/or necessary corrections/changes in requirements. We will also investigate other potential refinement categories and heuristics. The integration of the approach inside UML/SysML is also studied. Moreover while analyzing system specification, even if the requirements are correct when taken individually, they may have some inconsistency between each-others, then we aim at developing some mechanisms to investigate requirements' consistency [22].

As for the choice of our formal language, even if we think that CTL* is a language that is not hard to learn and that one get used to it rather quickly, it may be worthwhile to study using specification patterns instead of directly CTL*, as presented in [12].

7. References

[1] Abrial, J.R., "The B-book: assigning programs to meanings", Cambridge University Press, New York, NY, USA, 1996.

[2] Antón, A.I., "Goal-based requirements analysis", In Proceedings of International Conference on Requirements Engineering (ICRE '96, pp. 136–144, 1996.

[3] Atlee, J., Gannon, J., "State-based model checking of event-driven system requirements", Software Engineering, IEEE Transactions on 19(1), pp. 24 –40, jan 1993.

[4] Bois, P., Dubois, E., Zeippen, J.M., "On the use of a formal requirements engineering language: The generalized railroad crossing problem", Requirements Engineering, vol. 2, pp. 171–183.

[5] Dano, B., Briand, H., Barbier, F., "A use case driven requirements engineering process", Requirements Engineering, vol. 2, pp. 79–91, 1997.

[6] Dardenne, A., van Lamsweerde, A., Fickas, S., "Goal-directed requirements acquisition", In Proceedings of IWSSD'93, pp. 3–50. 6IWSSD, 1993.

[7] Darimont, R., van Lamsweerde, A., "Formal refinement patterns for goal-driven requirements elaboration", In Proceedings of SIGSOFT FSE'96, pp. 179–190, 1996.

[8] Deeptimahanti, D.K., Sanyal, R., "Semi-automatic generation of UML models from natural language requirements", In Proceedings of ISEC'11, pp. 165–174, 2011.

[9] Dhiauddin M., Suffian M., Ibrahim S., "A Prediction Model for System Testing Defects using Regression Analysis", International Journal of Soft Computing and Software Engineering [JSCSE], Vol. 2, No. 7, 2012.

[10] Donzelli, P., "A goal-driven and agent-based requirements engineering framework", Requirements Engineering, vol. 9, pp. 16–39, February 2004.

[11] Dutoit, A.H., Paech, B., "Rationale-based use case specification", Requirements Engineering, vol. 7, pp. 3–19, 2002.

[12] Dwyer, M.B., Avrunin, G.S., Corbett, J.C., "Patterns in property specifications for finite-state verification", In Proceedings of ICSE'99, pp. 411–420, 1999.

[13] Emerson, E.A., Halpern, J.Y., "*sometimes* and *not never* revisited: on branching versus linear time temporal logic", J. ACM, vol. 33, pp. 151–178, January 1986.
[14] Erfanian A., Nima Karimpour Darav N., "CBM-Of-TRaCE: An Ontology-Driven Framework for the Improvement of Business Service Traceability, Consistency Management and Reusability", International Journal of Soft Computing and Software Engineering [JSCSE], Vol. 2, No. 7, 2012
[15] Erfurth, I., Rossak, W., "CUTA4UML: bridging the gap between informal and formal requirements for dynamic system aspects", In Proceedings of ICSE'10, vol. 2, pp. 171–174.
[16] Fliedl, G., Kop, C., Mayr, H.C., Salbrechter, A., Vöhringer, J., Weber, G., Winkler, C., "Deriving static and dynamic concepts from software requirements using sophisticated tagging", Data Knowledge Engineering, vol. 61, pp. 433–448, June 2007.
[17] Fraser, M., Kumar, K., Vaishnavi, V., "Informal and formal requirements specification languages: bridging the gap", IEEE TSE, vol. 17(5), pp. 454 –466, 1991.
[18] Fuhrman, C.P., "Lightweight models for interpreting informal specifications", Requirements Engineering, vol. 8, pp. 206–221, 2003.
[19] Fuxman, A., Pistore, M., Mylopoulos, J., Traverso, P., "Model checking early requirements specifications in tropos", In Proceedings of RE'01, pp. 174 –181, 2001.
[20] Gabbay, D., "The declarative past and imperative future: Executable temporal logic for interactive systems", LNCS 398, pp. 409–448, 1987.
[21] Gervasi, V., Nuseibeh, B., "Lightweight validation of natural language requirements", Software: Practice and Experience, vol. 32(2), pp. 113–133, 2002.
[22] Ghazel M., Mekki A., "Assisting Specification and Consistency-Check of Temporal Requirements for Critical Systems", In the Proceedings of WorldComp2010- SERP, 2010.
[23] Grünbacher, P., Egyed, A., Medvidovic, N., "Reconciling software requirements and architectures with intermediate models", SoSyM, vol. 3(3), pp. 235–253, 2004.
[24] Harmain, H., Gaizauskas, R., "Cm-builder: A natural language-based case tool for object-oriented analysis", Automated Software Engineering, vol. 10, pp. 157–181, 2003.
[25] Heymans, P., Dubois, E., "Scenario-based techniques for supporting the elaboration and the validation of formal requirements", Requirements Engineering, vol. 3, pp. 202–218, 1998.
[26] Hussak, W., Keane, J.A., "Expressing requirements on a parallel system formally", Requirements Engineering, vol. 1, pp. 199–209, 1996.
[27] Hussak, W., Keane, J.A., "Formal analysis of memory requirements", Requirements Engineering vol. 4, pp. 188–197, 1999.
[28] Ilic, D., "Deriving formal specifications from informal requirements", In proceedings of Computer Software and Applications Conference, pp. 145–152, 2007.
[29] Ilieva, M.G., Ormandjieva, O., "Models derived from automatically analyzed textual user requirements.", SERA 2006, pp. 13–21.
[30] Isazadeh, A., Lamb, D., Shepard, T., "Behavioural views for software requirements engineering", Requirements Engineering, vol. 4, pp. 19–37, 1999.
[31] Jureta, I., Faulkner, S., Schobbens, P.Y., "Clear justification of modeling decisions for goal-oriented requirements engineering", Requirements Engineering, vol. 13, pp. 87–115, 2008.
[32] Kohring, C., Lefering, M., Nagl, M., "A requirements engineering environment within a tightly integrated SDE", Requirements Engineering, vol. 1, pp. 137–156, 1996.
[33] Overmyer, S.P., Lavoie, B., Rambow, O., "Conceptual modeling through linguistic analysis using lida", In Proceedings of the 23rd ICSE, pp. 401–410, 2001.
[34] Seater, R., Jackson, D., Gheyi, R., "Requirement progression in problem frames: deriving specifications from requirements", Requirements Engineering, vol. 12, pp. 77–102, 2007.
[35] Sutcliffe, A.G., Maiden, N.A.M, "Bridging the requirements gap: policies, goals and domains", In Proceedings of the 7th IWSSD, pp. 52–55, 1993.
[36] Sutcliffe, A., "Scenario-based requirements analysis", Requirements Engineering, vol. 3, pp. 48–65, 1998.
[37] Turner, K., "Incremental requirements specification with lotos", Requirements Engineering, vol. 2, pp. 132–151, 1997.
[38] Vissers, C.A., Scollo, G., Sinderen, M.V., Brinksma, E., "Specification styles in distributed systems design and verification", TCS 89, pp. 179–206, 1991.

[39] Wieringa, R.J., Saake, G., “Formal analysis of the shlaer-mellor method: Towards a toolkit of formal and informal requirements specification techniques”, Requirements Engineering, vol. 1, pp. 106–131, 1996.
[40] Wieringa, R., Dubois, E., Huyts, S., “Integrating semi-formal and formal requirements”, In the Proceedings of the 9th Int. Conference CAiSE’97, LNCS, vol. 1250, pp. 19–32, 1997.
[41] Yu, E.S.K., Mylopoulos, J., “Using goals, rules, and methods to support reasoning in business process reengineering”, IJISAFM, vol. 5, pp. 1–13, 1994.
[42] Zhang, W., Mei, H., Zhao, H., “Feature-driven requirement dependency analysis and high-level software design”, Requirements Engineering, vol. 11, pp. 205–220, 2006.
[43] Zhu, H., Jin, L., “Scenario analysis in an automated tool for requirements engineering”, Requirements Engineering, vol. 5, pp. 2–22, 2000.