Seamless development of complex systems : a multirequirements approach

Florian Galinier

▶ **To cite this version:**

Florian Galinier. Seamless development of complex systems : a multirequirements approach. Software Engineering [cs.SE]. Université Paul Sabatier - Toulouse III, 2021. English. NNT : 2021TOU30130 . tel-03599456

HAL Id: tel-03599456
https://theses.hal.science/tel-03599456

Submitted on 7 Mar 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE

En vue de l'obtention du
DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par l'Université Toulouse 3 - Paul Sabatier

Présentée et soutenue par

Florian GALINIER

Le 8 novembre 2021

Développement sans rupture de systèmes complexes : une approche basée multi-exigences

---

Ecole doctorale : **EDMITT - Ecole Doctorale Mathématiques, Informatique et Télécommunications de Toulouse**

Spécialité : **Informatique et Télécommunications**

Unité de recherche :
IRIT : Institut de Recherche en Informatique de Toulouse

Thèse dirigée par
Jean-Michel BRUEL et Bertrand MEYER

Jury

Mme Jeanine SOUQUIERES, Rapporteure
M. Richard PAIGE, Rapporteur
Mme Régine LALEAU, Examinatrice
M. Sébastien GÉRARD, Examinateur
Mme Ileana OBER, Examinatrice
M. Jean-Michel BRUEL, Directeur de thèse
M. Bertrand MEYER, Co-directeur de thèse
Mme Sophie Ebersold, Co-encadrante de thèse

Abstract

Proving that a system satisfies its requirements is an important challenge of Requirements Engineering. On the one hand, formal approaches provide a way to express requirements mathematically and prove that a system satisfies its requirements. However, if formalization offers additional possibilities such as verification, or even validation, it often proves to be too difficult to use in practice by the stakeholders involved in the development of systems. On the other hand, in most cases, requirements are written and sometimes traced in Natural Language for communication and mutual understanding purposes. Moreover, this remains during the whole development process. Thus, it is necessary to consider the need to address all these stakeholders during the development process.

The main objective of this thesis is to provide a seamless methodology that allows benefiting from the formalization of requirements while being understandable by all stakeholders. We propose an approach that considers requirements as parts of the system's code, which, as such, contributes to improving quality assessment. In addition, integrating the requirements into the code guarantees a seamless development. The contributions target three main benefits. First, there is no need to switch from one tool or environment to another: a single framework supports the development from analysis to implementation. Second, changes and reversibility become a regular occurrence, directly supported by the method, language, and tools, facilitating round-trips. Third, the different levels of abstraction remain inside the object-oriented paradigm.

We apply this vision to the development process itself with the same expected advantages. The development life-cycle can then benefit from this strong integration of requirements into the code. These artifacts help in software development by providing support and guidelines for analysis or decision support and reinforcing the software quality. Besides, reusability, evolutivity, and maintainability are enhanced. Traceability between requirements and code allows an easy impact analysis when any of these artifacts evolve.

However, if this paradigm is familiar to developers and even if we put an effort in providing expressivity, they are not addressed to other stakeholders that used to work with several tools. Since we also want non-experts to use our approach to validate systems in the early stage of their development, we propose a Domain-Specific Language: (i) close to natural language and (ii) based on formal semantics. Using Model-Driven Engineering techniques, this language bridges the gap between the several stakeholders involved in a project (considering their different backgrounds) and between the requirements and the code.

We finally put a research effort into defining relationships between requirements. We provide their formal definitions and properties on the propagation of the satisfaction state. These definitions can help engineers verify requirements (by checking the validity of the semantics of the relationships between two requirements) and verify the system compliance (thanks to satisfaction propagation).

This work is a step towards introducing formal semantics into traceability, making it possi-

ble to automatically analyze requirements and use their relationships to verify the corresponding implementation of the system.

Résumé

Prouver qu'un système satisfait à ses exigences est un défi important de l'ingénierie des exigences. D'une part, les approches formelles fournissent un moyen d'exprimer les exigences mathématiquement et de prouver qu'un système satisfait ses exigences. Cependant, si la formalisation offre des possibilités supplémentaires telles que la vérification, voire la validation, elle s'avère souvent trop difficile à utiliser en pratique par les acteurs impliqués dans le développement des systèmes. D'autre part, dans la plupart des cas, les exigences sont écrites et parfois tracées en langage naturel à des fins de communication et de compréhension mutuelle. De plus, cela reste le cas tout au long du processus de développement. Ainsi, il est nécessaire de considérer le besoin de s'adresser à toutes ces parties prenantes pendant le processus de développement.

L'objectif principal de cette thèse est de fournir une méthodologie sans rupture qui permet de bénéficier de la formalisation des exigences tout en étant compréhensible par toutes les parties prenantes. Nous proposons une approche qui considère les exigences comme des parties du code du système, ce qui, en tant que tel, contribue à améliorer l'évaluation de la qualité. De plus, l'intégration des exigences dans le code garantit un développement sans rupture. Ces contributions visent trois avantages principaux. Premièrement, il n'est pas nécessaire de passer d'un outil ou d'un environnement à un autre : un cadre unique prend en charge le développement de l'analyse à la mise en œuvre. Deuxièmement, les changements et la réversibilité deviennent un phénomène régulier, directement pris en charge par la méthode, le langage et les outils, ce qui facilite les allers-retours. Enfin, les différents niveaux d'abstraction restent dans le cadre du paradigme orienté objet.

Nous appliquons cette vision au processus de développement lui-même avec les mêmes avantages attendus. Le cycle de vie du développement peut alors bénéficier de cette forte intégration des exigences dans le code. Ces artefacts aident au développement du logiciel en fournissant un support et des lignes directrices pour l'analyse ou l'aide à la décision et en renforçant la qualité du logiciel. En outre, la réutilisabilité, l'évolutivité et la maintenabilité sont améliorées. La traçabilité entre les exigences et le code permet une analyse d'impact facile lorsque l'un de ces artefacts évolue.

Cependant, si ce paradigme est familier aux développeurs et même si nous faisons un effort d'expressivité, il ne s'adresse pas aux autres parties prenantes qui ont l'habitude de travailler avec d'autres outils. Puisque nous souhaitons également que des non-experts utilisent notre approche pour valider des systèmes dans la première phase de leur développement, nous proposons un langage spécifique au domaine : (i) proche du langage naturel et (ii) basé sur une sémantique formelle. En utilisant les techniques de l'ingénierie dirigée par les modèles, ce langage permet de combler le fossé entre les différents acteurs impliqués dans un projet (compte tenu de leurs différentes expériences) et entre les exigences et le code.

Nous avons enfin consacré un effort de recherche à la définition des relations entre les exigences.

Nous fournissons leurs définitions formelles et leurs propriétés sur la propagation de l'état de satisfaction. Ces définitions peuvent aider les ingénieurs à vérifier les exigences (en vérifiant la validité de la sémantique des relations entre deux exigences) et à vérifier la conformité du système (grâce à la propagation de la satisfaction).

Ce travail est une étape vers l'introduction de la sémantique formelle dans la traçabilité, permettant d'analyser automatiquement les exigences et d'utiliser leurs relations pour vérifier l'implémentation correspondante du système.

Acknowledgements

Each thesis is a slice of life, and it is appropriate to thank all those who participated in the development of this project.

I would like to start by thanking Sophie, Jean-Michel and Bertrand. They gave me the opportunity to work on an exciting project, and helped me by guiding me while letting me explore my ideas. I learned a lot from them, both from a research and teaching point of view.

Thanks to Sophie for all the discussions, over a whiteboard and a coffee that helped clarify and advance my work. I really enjoyed working with you, and I hope we can continue to work together in the future.

Thanks to Jean-Michel, for guiding my research and allowing me to clarify many points that deserved it. I couldn't have hoped for a better thesis advisor, whenever I had a question, I knew that you would be able to answer me or guide me to the person who would have the answer. It was a very pleasant working environment. Special mention to the trip to RE in Korea with you which was reduced to a round trip to Shanghai. I was very disappointed not to be able to attend the biggest conference in our field because of this typhoon, but at least I can say that I made a round trip to China to eat Chinese food.

Thanks to Bertrand who introduced me to many fields of research at the beginning of my thesis, through the Vericlub in particular, and for the always constructive criticisms which allowed my work to evolve. Thank you also for having provided us with an exceptional working environment during our meetings in Villebrumier.

I also want to thank the Kazan-Toulouse research group. The exchanges are always enriching and I no longer count the ideas that have emerged from our meetings. Thank you for the proofreading, for the hard work on the papers that sometimes took a long time to come out, and there is no doubt that the future works will be as interesting.

Thanks to all the SM@RT team, PhD students or tenured researchers, who gave me precious advice.

Thanks also to Clément, Jimmy and Manuel. You have supported me during my thesis, and you will have to support me as your boss in our entrepreneurial adventure. I sometimes wonder if you have all your head.

Finally, some words are more effective when they are expressed in the language that the target person understands best. So, I apologize to those who will not be able to read these next words, addressed to my family, in French.

Merci à mes parents et à mes soeurs, ainsi qu'au reste de ma famille, qui m'ont toujours soutenus, bien qu'ils n'aient certes pas tous compris ce que je fais exactement !

Merci à mes amis, qui ont été là pour m'aider à décompresser et à décrocher, que ce soit lors des discussions sur discord ou lors des week-ends du bonheur et autres barbecues. Vous m'avez sans

doute empêché d'exploser en plein vol.

Et bien sûr merci à Marjorie. Tu es devenue ma femme durant cette aventure et tu m'as toujours soutenu, tu as servi beaucoup trop souvent de support de discussion où tu ne faisais que m'écouter dire des choses sans doute incompréhensibles. Nul doute que je n'aurai pas pu arriver au bout de ce travail sans ton aide.

À René, Michel, Virginie, Robert et Marc,
qui ne sont plus là pour voir la fin de cette aventure,
mais qui ont laissé une empreinte indélibile sur mon passé,

À Eyleen,
qui n'était pas là pour voir le début de l'aventure,
mais qui est mon avenir,

Je dédie cette thèse.

Contents

List of Figures

List of Tables

Part I

Introduction to the problem

Chapter 1

Introduction

"The most important single aspect of software development is to be clear about what you are trying to build."

Bjarne Stroustrup

Contents

How can we ensure that a system is the "right" system? Complex systems, software-intensive systems, embedded systems are around us every day: smartphones, cars, airplanes, satellites are among other examples of objects that we use daily, sometimes unconsciously. Thus, these systems' quality is crucial: how to be confident in such devices? If the question seems minor for smartphones' applications, we quickly understand that it becomes crucial for machines that can exceed a ton and be launched at several tens of kilometers per hour. Furthermore, if everyone has now been confronted with a software bug at one time or another, we probably identified it because the system was not doing what it was supposed to do.

However, what is the system supposed to do? This is the first step of every system development: determine what the customer wants the system to do. To rephrase it: what are the customer requirements for the system? These requirements are the basic bricks used to build the system. Hence, ensuring that the system is the right system will "simply" be to check that its requirements are met.

In the next sections, we will first develop the context, and then we will explain why it is not so "simple" to ensure that a system is the "right" system.

1.1 Context of the thesis

1.1.1 What are requirements?

According to SWEBOK [3], a requirement is "*a property that must be exhibited by something in order to solve some problem in the real world.*" While the SWEBOK is intended to be a reference work on software engineering, this definition is quite confusing.

First, this definition states that a requirement is "*a property*" but does not define the scope of this property (is it a system property? a property of the environment? a property of the project? something else?).

Secondly, according to this definition, a requirement "*must be exhibited by something.*" Every studies made in Requirements Engineering (RE) agree that a requirement must be made explicit (since implicit requirements can introduce inconsistencies in the development). Besides, there should be a way to check that a requirement is effectively satisfied. As a consequence, the definition shall not be prescriptive but only descriptive. Moreover, the vague term *something* introduces more ambiguity in the statement than elements of answers.

Finally, the definition is talking about "*problem in the real world.*" The notion of "*the real world*" is never defined elsewhere in the document and is imprecise. This notion can include anything and cannot be used for a precise definition.

The following sections in the SWEBOK document present some distinctions between categories of requirements:

- product and process requirements: this section introduces a distinction between requirements about the product itself (the system, using our vocabulary) and the process (project using our vocabulary). This section helps to define the scope of property from the previous definition, but environment properties are still ignored in the SWEBOK definition;
- functional and non-functional requirements: the SWEBOK uses the well-known distinction between requirements describing the system's behavior and non-functional ones, which are requirements that "act to constrain the solution." The SWEBOK does not include a complete list of non-functional requirements and refers to several works for these requirements, such as [4, 5, 6];
- emergent properties of a system, which are requirements according to the SWEBOK definition. However, such properties are not defined as requirements in standard literature since requirements are something expected while, by definition, it is not the case for emergent properties;
- quantifiable requirements: this category is orthogonal to the previous ones: while others present how to categorize requirements, this section is stating about the need to have a precise requirement, using quantifiable terms to avoid ambiguity;
- system and software requirements: in this section, the authors of SWEBOK made the distinction between system requirements that are about the system as a whole and software requirements that are requirements about software components derived from system requirements. This classification is again orthogonal with the previous ones.

Other definitions, such as the definition given in [7], widely cited in the literature, are quite ambiguous, and there is no strict definition of what a requirement is. In [8], we made an effort to provide such a definition.

So what is a requirement? We will try to give a descriptive and straightforward definition that we already used in [8]. As previously said, a requirement is the expression of a stakeholder need. Most of the time, this expression is made in Natural Language (NL) [9, 10] – in English, French, German, etc. The sentence *"The drone shall avoid obstacles"* is thus a requirement, and so is *"At any time, the sensors shall be able to detect an object in less than twice the stopping distance and the drone must be able to stop in less than a second."* These two requirements express the same idea (the drone shall be able to detect an obstacle and avoid it) at several levels of abstraction, but both are requirements. Similarly, requirements can be expressed in several ways. Thus, the requirement *"At any time, if the sensors detect an object less than twice the stopping distance then the drone must be able to stop in less than a second."*, the expression $(sensors\_range \geq 2 \times (speed + \frac{acceleration}{2})) \wedge (\frac{speed}{acceleration} < 1)$, and Fig. 1.1 are three different representations of requirements, all expressing the same property.

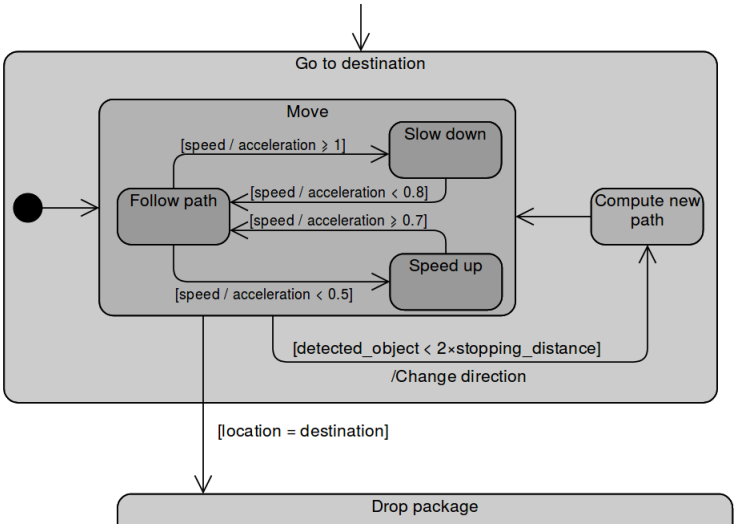


Figure 1.1: Excerpt of a state diagram representing the *"avoid obstacle"* requirement

A requirement is so the expression of a need. But what is a need? In the sentence *"The drone shall avoid obstacles"*, we express the need for the *system* (the "*drone*") to own a property ("*avoid obstacles*"). In this case, a requirement is the expression of a (desired) property – a predicate – of the system. Thus, a need is a direct expression of what a stakeholder wants, while the requirements we are particularly interested in in this thesis are derived from this need, and are expressed after defining the scope of the system.

Indeed, in [11], the authors emphasize the need to express requirements regarding the environment (i.e., anything that is not the system but can have an effect on it). The environment is crucial, and in particular in our example, "*obstacles*" are not objects of our system but of the environment, and the requirement describes how the system shall react to the environment. The Problem Frames approach [12] which has influenced many more recent approaches, precisely defined that each requirement must be related to a specific problem domain, which is an element outside the system that will propose the solution to that requirement. Moreover, a requirement can even be represented as a relationship between different artifacts in the environment. Jackson describes

a number of problem frames that correspond to these classes of relationships. In the context of this thesis, we will thus emphasize this need to express a requirement in the context of the system environment.

Another essential element in system development is the *development* itself, the project, and requirements can also be about the project (in [13], such requirements are instead called *constraints*). Using these elements, we give the following definition for requirement:

Definition (Requirement)**.** A requirement is the expression of a property of the system, the project, or the environment.

1.1.2 Why are requirements so crucial?

The criticality of requirements has often been addressed in the literature. Indeed, the ambiguity, the lack of consistency, and poorly treated requirements can lead to bad design and bad implementation. In the context of this thesis, we will focus on software-intensive systems – i.e., systems where the software plays an important part. However, even in this more restricted context, lots of failures can exist. Even if the consequences of a code issue can be less evident than a structural defect in the architecture of a bridge, there are more and more software in our lives in everyday objects, and most of the time, requirements are poorly treated. Moreover, even if issues may seem unimportant and only minor, failures caused by specification errors are numerous, costly and sometimes, tragic [14]. In [15], a list of majors and costly failures is given and, if in the better cases the failures only lead to an economical cost, in most critical systems, these failures can put lives in danger.

This is the case with the Therac-25 incident. This machine was designed for radiation therapy, but in mid-80s, a failure led to several people's death, massively irradiated. The investigation highlighted, among other problems, a defect in the specification and poor traceability to the previous versions of the software. Not taking into account the user's real needs and expertise introduced several biases that led to this tragedy.

Furthermore, even in systems that seem not to have any chance to injure people, the impact may be difficult to predict. This is the role of risk analysis to detect where are the crucial parts of systems. However, one can ask why requirements are not systematically treated critically. We will develop the question in chapter 2, but in a nutshell, it is costly and can take a considerable amount of time. All the parts of the system cannot be treated as critically as the autopilot mode of a plane. However, if we can provide tools to process more formally and systematically the requirements, this can reduce the number of failures in software engineering. In the next section, we will present the significant issues from our perspective to the requirements engineering.

1.2 Issues in Requirements Engineering

Requirements are the basis of every project and there exists, in one way or another, a set of requirements (in a Requirements Specification, in Users Stories, in Tickets in a collaborative revision control system, in emails between stakeholders, etc.). So if these requirements exist, what does introduce deficiency in a specification? This section will present three of the major issues of the Requirements Engineering that we want to address in this thesis.

1.2.1 Expressing requirements

First of all, even if the requirements are expressed, it is necessary to emphasize how they are expressed. Indeed, if the requirements are poorly expressed and are ambiguous, even with a good requirements management tool, the output system can be different from the one expected by the customer. When the customer expresses its needs for the system, it is essential, in the elicitation phase, to provide a complete set of requirements. In chapter 2, we will provide a set of quality criteria for requirements, but we can emphasize that the way to express requirements is crucial. Indeed, it is easy to understand that a simple, clear, and unambiguous requirement is preferable to a more complex and convoluted one.

The language used to express requirements is thus crucial. The language here means the field of the lexicon used and the grammar, the organization, and the way to write the requirements. From our point of view, it is essential to provide a clear methodology on how to express requirements and support this approach by tools. In the context of this thesis, even if we will always take into account traceability to all kind of requirements and specification, we will more precisely address system requirements, requirements that describe the system in relation to its environment.

1.2.2 Improving requirements traceability (and why?)

Since critical issues arise from requirements problems, good management of requirements in systems realization is fundamental [16]. We can consider that, since that is the job of the engineer, he will do it. However, it does not take into account that there is not one person who will collect the requirements, design the system, and implement it. Most of the time, a team that works on the system does not have a complete view of the system. To ensure that all the requirements are met, it is crucial to link parts of the system with the requirements.

This is one of the objectives of traceability. Many examples of systems crashes emphasized the necessity of requirements traceability through the whole development process [17] [18]. Traceability helps to ensure, in one way, that a requirement has been satisfied. On the other way, it helps to ensure that a part of the system is required to satisfy a requirement. The traceability in software engineering is, however, at best poorly treated. From our point of view, it remains a significant effort to do to provide a clear methodology and tools for traceability.

1.2.3 Keep the link with all the stakeholders

In the two previous sections, we emphasize providing tools and methodology for requirements expression and traceability. Such tools exist. However, they are confronted with the lack of expertise of users or to the force of the habits. In our opinion, one of the major obstacles for the adoption of new tools and methodology is that they are not close enough to the stakeholders' habits, which do not widely use new approaches. More precisely, most of the tools are either addressed to engineers or to other non-expert stakeholders. The approach proposed in this thesis aims to involve engineers in requirements management as early as possible in order to allow an early introduction of a formalism in the expression of requirements. However, if this approach aims to be anchored in a classic engineering approach (V-cycle, agile methodology), it is important to keep the approach as accessible as possible to other stakeholders. Indeed, it is important to remember that requirements are a communication tool between different stakeholders, and especially with customers, who are not engineers. Thus, even if the approach presented in this thesis is primarily addressed to engineers,

we have emphasized the importance of keeping the representation as readable as possible for non-expert stakeholders. In particular, it is essential to provide tools that fit on stakeholders' habits or link to other standard tools. These three issues will lead us to propose a methodology, as close as possible to the developers' habits, providing links to existing tools.

1.3 Detailed plan

To answer the issues raised in section 1.2, this thesis is decomposed into three parts. The current part (**Introduction**) introduces the problem (chapter 1) and some good properties for requirements (chapter 2). We will use these good properties to analyze existing approaches that try to answer to the identified issues. These approaches will be grouped by categories to compare approaches with similar objectives, before emphasizing what is, according to us, the best way to provide solutions. The next part (**Contributions**) will first detail the SIRCOD approach (chapter 3), introducing a seamless approach to expressing requirements and tracing them, from the first step of a project lifecycle to the complete realization of the system. We will then present RSML in chapter 4, a language and a tool to support the SIRCOD approach and provide all stakeholders with better means to be as close as possible to the user's habits. In chapter 3 and chapter 4, while we will apply to specific use cases, but also to a running example: an automatic delivery drone[1]. In chapter 5, we will propose a set of new navigation links, including traceability links, to enrich the methodology and ease the analysis of requirements. To conclude this chapter, we will apply the complete approach to our running example. Finally, the **Conclusion** part will summarize the contributions and open on the perspectives that remain.

[1]User Requirements Document of this example can be found here: `https://gitlab.com/fgalinier/rsml-examples/-/blob/master/AutomaticDeliveryDrone.md`

Chapter 2

State of the art

"None of us knows that which we know all together."

Euripides

Contents

In this chapter, we introduce (section 2.1) how RE tries to ensure good quality of systems. We analyze some advice and good properties of requirements given in the most prominent standards (section 2.1.1). We also provide a state of the art of approaches that attempt to lead to better quality requirements, based on NL (section 2.1.2), on formal methods (section 2.1.3), or trying to bridge the gap between these two worlds (sections 2.1.4 and 2.1.5). Next, we introduce what can be the benefits of using Model-Driven Engineering (MDE) in the context of RE (section 2.2), giving approaches that address some of the issues of RE, such as traceability of requirements through several levels of abstractions (section 2.2.1) or dealing with several stakeholders (section 2.2.2).

2.1 Requirements Engineering

In this section, we survey several RE approaches based on criteria that we define in section 2.1.1. A complementary work to this section can be found in [19].

2.1.1 What are "good" requirements?

As a first step, it is necessary to define what is a "good" requirement. Indeed, talking about improving requirements without defining the objective to reach would make little sense. Defining qualities for requirements has been addressed many times in the literature, both from an academic and an industrial perspective.

Standards such as the often-cited IEEE 830-1993 [20] or the more recent ISO/IEC/IEEE 29148-2018 [21] defined the characteristics for good requirements. Among these criteria, a particular emphasis is placed on the need to have a set of requirements that are:

- **necessary**: all the requirements are needed to describe the system;
- **complete**: a requirement is sufficient and does not need more information to describe the desired system property;
- **unambiguous**: the requirements are simple to understand and do not introduce any ambiguity (for example by using pronouns such as 'it' or 'this');
- **singular**: a requirement is about a unique property of the system;
- **consistent**: there are no two requirements stating two contradictory properties;
- **feasible**: a requirement shall be realistically satisfiable;
- **correct**: a requirement is a correct transformation of the entity from which it was derived – these characteristics was linked to the **traceable** criterion in older version of the IEEE standard[1];
- **verifiable**: evidence shall be available to prove that a requirement is actually satisfied by the system.

The International Council on Systems Engineering (INCOSE) also emphasizes in [22] the need for complete **traceability** between artifacts from several stakeholders.

These criteria are guidelines to write requirements and apply to the way requirements are expressed. Indeed, to match these criteria, standards often propose templates to constrain the way of expressing requirements.

In [23], the authors propose a list of properties for a good requirement language for software-intensive systems. In their opinion requirements approaches should:

- ease communications between stakeholders (e.g., by providing a graphical modeling way to express requirements, by being the more human-readable possible, etc.);
- allow to model common relationship existing between requirements themselves and between requirements and other system artifacts;
- provide common properties of requirements, such as risks, priority, type;
- provides a way to express non-functional requirements;
- respect recommendations from IEEE standard (i.e., requirements should be unambiguous, consistent, modifiable, correct, complete, and traceable);
- to a lesser extent, be machine-readable and have a strongly defined semantics.

[1] Actually, the *traceability* is indeed a tool to check this correctness rather than a real needed characteristic for a requirement.

In [24], Schneider and Buede propose 15 properties for informal requirements, based on an analysis of properties from several works. They aim to ease the analysis by avoiding overlapping properties while proposing a sufficient set to provide good requirements.

Another good property for expressing requirements is, according to [25], the use of a unique formalism. Indeed, "Single Model Principle" should lead to avoiding issues of classical approaches where different formalisms are used: inconsistencies, semantic gaps, transformation efforts. This *seamless* approach aims to link different phases of the development lifecycle, from design to implementation, to prevent failures [26].

In the context of this thesis, we will focus on the following properties for requirements approaches:

(a) **Intended audience**: does the approach require an expertise, or is the approach understandable by every stakeholder? This shall answer to the properties that require stakeholders to take position on requirements (is the requirement *necessary*, *complete*, *feasible*?) and reduce the *ambiguity* of requirements for stakeholders.

(b) **Traceability support**: is there a way to link requirements to other artifacts (such as requirements or other parts of the system) and what is the type of these links? This property shall help answer the need of *correctness*, *consistency* and *absence of ambiguity*.

(c) **Coverage**: what kinds of requirements can be expressed with the approach (all the requirements? only functional ones?)? This property is required to state the *completeness* of the set of requirements, i.e. all requirements are needed and sufficient to describe the system (not to be confused with the completeness of a single requirement).

(d) **Semantic definition**: are the requirements semantically defined in the approach? A semantic definition shall help avoid *ambiguity* and check the *consistency*, *correctness*, and *singularity*. Moreover, the power of formalization allows rigorous management of expressions endowed with semantics [27].

(e) **Verifiability**: can a requirement be formally verified? This property aims to answer the need for *verifiability* for requirements.

There are mainly two worlds that are confronting nowadays for the expression of requirements:

- in critical systems, formal methodologies, based on a mathematical foundation, are widely used to prove the safety of a system;
- in other systems, requirements are often expressed as sentences that described users' needs, with no real links with the system itself.

To compare these approaches, we analyze them in the following sections through the prism of the previously defined properties.

2.1.2 Natural Language-based approaches

We present in this section the approaches that are designed to be as close as possible to NL. These approaches are commonly designed to be as readable as possible by non-specialists and are thus addressed to several stakeholders. The objectives of such approaches are mainly to involve all the stakeholders during the system's lifecycle.

Commercial solutions

Description: There are several industrial approaches that are proposing to express requirements and to links them (some of them can be found in [28]). Since they are probably the most used approaches, we cannot talk about tools for requirements without citing the well-known IBM Rational DOORS [29] or Dassault Systems Reqtify [30]. These tools allow the users to manage requirements, expressed using several different tools (such as text documents or spreadsheets) and to manage links between requirements and other artifacts.

These tools do not consider the way that requirements are expressed: they are addressed to non-specialists and thus are easy to handle. By working with requirements as units – atomic artifacts – that can be manipulated, these solutions are abstract enough to work with any type of requirements, expressed in several representations. They are however mainly used with NL requirements and so, can be used to express all kinds of requirements.

In these approaches, the users can express relationships between requirements, introducing traceability links such as refinement, as well as relationships between requirements and specifications, such as code (Reqtify allows to create a link between a NL requirement expressed in a Microsoft Word Document and C code portion).

However, and regardless of their proprietary formats, these tools do not provide a strong semantics for the links. These tools are not formally defined in any way and therefore do not support verification.

Properties:

(a) **Intended audience**: These tools are addressed to all stakeholders.

(b) **Traceability support**: Traceability is the core behavior of these tools.

(c) **Coverage**: All kinds of requirements can be considered with these tools.

(d) **Semantic definition**: There is no semantic definition.

(e) **Verifiability**: There is no verification mechanism.

KAOS

Description: Some of the best-known approaches to requirements are the Goal-Oriented Requirements Engineering (GORE) approaches. These approaches aim to provide a framework to requirements' elicitation, expressing in the first time the *goal* of the stakeholders, to refine in a second time into sub-goals and finally, requirements that the system has to meet.

The KAOS [31] methodology is one of these approaches (as well as i* [32]). Supported by several tools, like Objectiver [33], KAOS allows the users to create *requirements* and to link them to other artifacts, such as *goals* they refine, *agent* that are responsible of them or even to the *operation* that implement the requirements (in Fig. 2.1 the requirement *"Avoid the obstacles"* is refined from the need *"Transport safely the package"*).

In the basic form of KAOS, goals and requirements are pieces of NL text and so, both functional and non-functional requirements can be expressed and are understandable by all stakeholders. Even if a focus is put on traceability in KAOS, this traceability is more about elicitation artifacts and does not keep a link in downstream phases. However, there is a current research work (see below

in section 2.2.2) that leads to creating traceability between KAOS and textual representation of requirements.

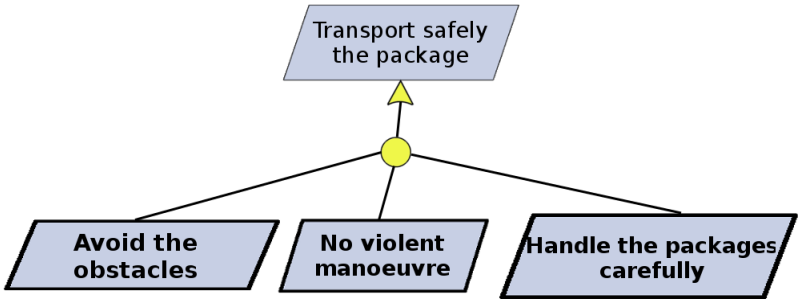


Figure 2.1: Example of a goal and the requirements refined from this goal in KAOS

If there is a clear semantics definition of KAOS, requirements themselves are expressed in NL and so, are not semantically defined. Some research works aim to provide a formal semantics to requirements expressed in KAOS (in LTL [34] or Event-B [35]), but it is not natively supported by the approach, as well as the verification.

Properties:

(a) **Intended audience**: These tools are addressed to all stakeholders.

(b) **Traceability support**: Traceability between elicitation artifacts and to specification documents.

(c) **Coverage**: All kinds of requirements can be considered.

(d) **Semantic definition**: Not natively.

(e) **Verifiability**: There is no verification mechanism.

SysML

Description: The Systems Modeling Language (SysML) [36] extension of Unified Modeling Language (UML) [37] is designed to model complex systems. Among the different diagrams of this extension, the *Requirements Diagram* allows engineers to represent requirements as artifacts, that can be connected to other artifacts. These other artifacts can be other specification elements (such as Block for example) as well as other requirements artifacts, allowing quick and visual traceability. Requirements' artifacts themselves are composed of a unique identifier and a text, that contains the NL requirement.

Both functional and non-functional requirements can be represented only in NL, but cannot be verified. The SysML model does not integrate links to other paradigms and so, there is no native possibility in SysML to create relationships with other non-SysML representations of requirements

(contrary to the aforementioned tools). The ReqCycle[2] plugin for the PolarSys[3] development tool offers this possibility.

In [23], Soares et al. propose to extend SysML to provide a better-structured way to express requirements. By adding an explicit relationship between requirements, especially in the requirements table, they improve the traceability notion in requirements expression, making explicit (and semantically defined via metamodel) links between requirements. Indeed, they highlight that the type of relationships existing between requirements is important to measure the impact change. The solution proposed still requires human expertise to determine the change effect.

Properties:

(a) **Intended audience**: Requirements expressed in SysML's requirements diagram can be understood by all stakeholders.

(b) **Traceability support**: Traceability between artifacts of several phases is supported.

(c) **Coverage**: All kinds of requirements can be considered.

(d) **Semantic definition**: There is no semantic definition for requirements.

(e) **Verifiability**: There is no verification mechanism.

URML

Description: User Requirements Modeling Language (URML) [38, 39] is a UML profile. This language provides a graphical representation of requirements and other concepts linked to requirements (such as threats, hazards, mitigations, product lines, and stakeholders). This approach proposes to add some semantics on links between requirements and other artifacts (e.g., refines, constrains, etc.). In their work, the authors focused on the existing relationships between requirements and other elements, not on the way of expressing requirements. Note however that the requirements can be linked to *use cases* or *processes*. These models should introduce a kind of formal representation of the requirements.

This approach is focused on the elicitation process, considering concepts from feature-oriented, goal-oriented, process-oriented, risk-oriented approaches (for example, the danger notion is used to justify the need for some requirements). Even if some semantics is defined on the links, there are no semantics on requirements themselves and no verification mechanism associated with this approach.

Properties:

(a) **Intended audience**: The icon-based notation should be understandable by all stakeholders.

(b) **Traceability support**: Traceability between concepts is inside the language.

(c) **Coverage**: All kinds of requirements can be considered.

(d) **Semantic definition**: There is some semantics on links between artifacts.

(e) **Verifiability**: There is no verification mechanism.

[2] `https://www.polarsys.org/projects/polarsys.reqcycle`
[3] `https://www.polarsys.org/`

Note: There exist other UML profiles that extend the notion of requirements. For example, the Modeling and Analysis of Real Time and Embedded systems (MARTE) [40] profile for real-time and embedded applications allows using a more refined approach to requirements, introducing in its Design Model temporal logic elements that can be used to express requirements.

This profile has been used in the methodology proposed by [41] in coordination with SysML and Electronic Architecture & Software Tools - Architecture Description Language (EAST-ADL2) – a language to design automotive systems. The approach aims to keep the idea of traceability between requirements and components proposed by SysML, while they use notions of MARTE to complete EAST-ADL2.

These approaches are however domain-centric (real-time and embedded applications, automotive embedded systems) and are thus not detailed here.

URN

Description: User Requirements Notation (URN) is a standard from the International Telecommunication Union (ITU-T) [42]. The language and its tool, jUCMNav [43], aim to provide a visual notation for modeling requirements. The use of a formally defined notation should thus help to elicit and analyze requirements.

The language itself is composed by two parts: the Goal-oriented Requirement Language (GRL) and the Use Case Map (UCM). The GRL can be used to express non-functional requirements and goals, and the very first steps of the requirements elicitation process, while UCM enables to refine these non-functional requirements and goals into more functional requirements (called System Requirements), through scenarios. The expressiveness of the language is one of the crucial points according to the authors, and both of the non-functional and functional requirements are thus understandable by all the stakeholders. Indeed, the GRL expresses requirements using NL, while the scenarios that allow the user to formalize functional requirements is a simple way to keep a kind of expressiveness.

Traceability between artifacts is a concern of the notation. Indeed, it is possible to use several relationships between the elements (like contribution, correlation, decomposition, etc. for GRL). Each link can be qualified to give an additional information (for example, it is possible to indicate if a soft-goal contribute to help or to make another soft-goal). Moreover, the connection between URN and other languages is made possible by two ways: first, by providing a metamodel of URN, it is possible to make transformations to other modeling languages, then, the tool allows integrating URN into DOORS, providing a way to benefit from the traceability to other paradigms.

Properties:

(a) **Intended audience**: Both non-functional and functional requirements are understandable by all stakeholders.

(b) **Traceability support**: Traceability between internal artifacts in the approach is supported. The external traceability is supported by the tool and integrated into DOORS (and so has drawbacks of this tool).

(c) **Coverage**: Non-functional requirements are addressed in GRL while functional ones can be expressed in UCM.

(d) **Semantic definition**: While the language syntax is defined in the standard, the semantics is informal.

(e) **Verifiability**: No formal verification mechanism.

2.1.3 Formal approaches for requirements

In this section some formal methods used in requirements engineering are presented. Contrary to the approaches of the previous section, formal approaches aim to be rigorously defined to allow verifying a design according to requirements. They are addressed to a more expert public, with a formal background, and are mainly used in industry for critical parts of systems.

VDM

Description: Developed by the IBM laboratory, the Vienna Development Method (VDM) [44] is one of the oldest formal methods.

It first proposes a Specification Language called VDM-SL [45]. Using this language, the users can model the system (using *modules*) at a high level of abstraction and then refine it successively to a more detailed design (through a *reification* process).

A more recent version of VDM, VDM++ [46], proposes to use *classes* and *inheritance* instead of the VDM-SL module structure.

Only functional requirements can be expressed using VDM-SL and VDM++ languages, but such requirements can then be verified through proof-obligation.

Properties:

(a) **Intended audience**: As a formal method, VDM is addressed to experts.

(b) **Traceability support**: There is no traceability support.

(c) **Coverage**: Only functional requirements are considered.

(d) **Semantic definition**: A formal semantics is available for VDM.

(e) **Verifiability**: Requirements expressed in VDM can be verified.

Alloy

Description: Alloy [47] is a formal modeling language based on first-order logic, inspired by Z [48]. The language proposes to specify systems, using a syntax similar to object modeling languages, such as Object Constraint Language (OCL).

The user can describe data structures in Alloy (using the keyword *sig* and *extends* mechanism) and define relationships to other artifacts, like in classical object-oriented. It is also possible to express constraints on the model in order to tend towards a correct representation of the system. These constraints can be used to express the functional requirements of the system.

Specification designed using Alloy can then be automatically checked, using the *Alloy Analyzer*, a constraint solver. If the model constraints can be correctly satisfied, the solver can give the user a valid instance of this specification.

Properties:

(a) **Intended audience**: As a formal method, Alloy is addressed to experts.

(b) **Traceability support**: There is no traceability support.

(c) **Coverage**: Only functional requirements are considered.

(d) **Semantic definition**: A formal semantics is available for Alloy.

(e) **Verifiability**: Requirements expressed in Alloy can be verified.

Event-B

Description: Event-B [34] is an event-based version of the B formal method [49] that has been proposed to model a system.

The user can express states of the systems (as sets and functions) and define events that will affect these states. This specification can be constrained by invariants, used to express functional requirements.

A valid model is thus a specification that satisfies these invariants. When a model is valid, it can be refined into a more detailed model (of lower abstraction level) and, using the proof obligations, ensure that the refinement is correct regarding its invariants. The proof-based refinement thus ensures that the derived models respect the properties from a higher level of abstraction models.

Properties:

(a) **Intended audience**: As a formal method, Event-B is addressed to experts.

(b) **Traceability support**: There is no traceability support.

(c) **Coverage**: Only functional requirements are considered.

(d) **Semantic definition**: A formal semantics is available for Event-B.

(e) **Verifiability**: Requirements expressed in Event-B can be verified.

FORM-L

Description: FORM-L [50] is a language extension for MODELICA [51]. The FORM-L language proposes a refinement-based process to specify a system, its environment, and its requirements.

In this approach, the system is first considered as a black-box, with goals, that will then be successively refined into requirements that will be more and more detailed. The system itself will be detailed, going from a black-box to its components. The mathematical notation and the refinement process should lead to a correct model of the system. Probabilities are an important part of this formalism, each element expressing a probability to satisfy a requirement, and each requirement is expressed with a require probability to be satisfied. The verification process is based on probabilistic models to check the validity of requirements. There is a current effort to provide a formalization of the notation to obtain a complete semantics.

Since the approach starts from the goals and goes to the system, there is a support of traceability from goals to the requirements that shall satisfy these goals. However, the traceability is internal to FORM-L and there is no traceability to other tools.

Properties:

(a) **Intended audience**: The notation is addressed to users with a specific training – even if more affordable than some other formal approaches.

(b) **Traceability support**: Traceability can be established between formal requirements, informal requirements and goals.

(c) **Coverage**: Only functional requirements are considered.

(d) **Semantic definition**: There is a current effort to define a formal semantics for FORM-L.

(e) **Verifiability**: Thanks to transformation to Stimulus (see section 2.1.5), model-checking is possible.

2.1.4 Natural Language Analysis

Some approaches propose to analyze NL requirements and to provide a more formal representation. Contrary to the aforementioned approaches, approaches presented in this section propose to bridge the gap between requirements expressed in NL and a more formal representation of these requirements.

Natural Language Processing (NLP)

Description: Some of the most popular approaches for Natural Language Analysis are Natural Language Processing (NLP) techniques. Indeed, such techniques allow stakeholders to write requirements in NL and to obtain a formal representation thanks to the processing of NL text. The aim of this section is not to detail how all these approaches are working, but the basic idea of approaches such as [52, 53, 54] is to provide tools that allow analyzing NL texts, extracts requirements (in finer or coarser grain depending on the approach). These requirements can be then expressed more formally, through ontological representations or in formal mathematical notations. Most of the time, these works do no go that forward into formalization. For example, in [55], the authors propose to analyze the compliance of requirements to a requirement template. In [56], the work of analysis should help to detect what are the requirements into a document. Thus, if these works allow for better requirements document, most of them do not address the formalization of requirements. Moreover, even for the approach that proposed formalized requirements, they co-exist with the original NL requirements and there is still a gap between both representations since they are not expressed in the same language and are independent of each other. Note that the analysis of NL is a quite difficult task, and there is currently a major research effort in this field, trying to overcome this difficulty.

Properties:

(a) **Intended audience**: The source documents are addressed to all stakeholders, but the resulting ones require most of the time specific skills.

(b) **Traceability support**: There is a direct correspondence between the Natural Language text and the formalized requirements but, since there are expressed in two different languages, it is necessary to keep a link between these representations. However, the work on these approaches

is focusing on how to analyze and formalize requirements analyzed in such a way, not on the treatment of requirements after this analysis.

(c) **Coverage**: Approaches that formalize requirements through ontological representations allow expressing all kinds of requirements.

(d) **Semantic definition**: The objective of NLP is to extract semantics from NL.

(e) **Verifiability**: NLP is an entry point for a formal representation of requirement. It gives artifacts that are classified and can be used. There is no verification of requirements at NLP level.

Natural Language Analysis: from NL to STD

Description: The methodology proposed by [57] aims to iteratively transform NL requirements into State Transition Diagram (STD). Based on the idea that functional requirements can be expressed as a $\{state_1, transition, state_2\}$ triplet, this approach is focusing on functional requirements,. It provides a transformation pattern to create, from a requirement, a partial STD that will be iteratively completed with other requirements.

The analysis of the final STD should lead to detect lacks in requirements (for example if a state is missing). This representation, even if less understandable by non-specialist, still be very simple and can be used as communication support for most of the stakeholders. Moreover, STD are quite common, and since STD are formal, they can be analyzed by states machine verification tools.

Properties:

(a) **Intended audience**: While the natural language requirements are accessible to every stakeholder, the STD representation is less affordable.

(b) **Traceability support**: Relationships between requirements are partially covered, but there is no upstream and downstream traceability.

(c) **Coverage**: Thanks to the level of abstraction of requirements, all kinds of requirements can be expressed.

(d) **Semantic definition**: There is no strict semantics definition of the approach.

(e) **Verifiability**: Resulting STD can be analyzed to detect deficiencies in the specification.

Natural Language Analysis: from NL to OWL

Description: In [58], the authors propose a methodology to transform NL requirements into an ontology based representation in OWL [59]. For this purpose, they introduce an intermediate modeling language to express requirements. This language still be, according to the authors, addressed to requirements engineers and cannot be used for communication with all stakeholders.

The proposed ontology for requirements integrate notions such as *functional goals* and *quality goals* (respectively functional and non-functional requirements), as well as a domain representation through the *domain assumptions*.

Requirements expressed in this ontological representation can be manipulated by tools like Protégé[4] or libraries (such as OwlReady[5], owlcpp[6], ...). These requirements, however, cannot be verified.

Properties:

(a) **Intended audience**: The use of this approach requires a specific training.

(b) **Traceability support**: Ontologies can be linked to other ontologies, providing a partial traceability.

(c) **Coverage**: All kinds of requirements, functional and non-functional, can be addressed.

(d) **Semantic definition**: OWL semantics only is supported.

(e) **Verifiability**: There is no support of verifiability.

2.1.5 Constrained Natural Language

In this section, we present some methodologies that propose to express requirements in a kind of NL, constrained by a grammar. The objective of these approaches is to provide languages quite similar to NL, easy to understand by all stakeholders, but constrained enough to automatize analysis.

Requirements Grammar

Description: Several grammar have been proposed to requirements. In fact, most of the standards incite to respect a clear and precise grammar. However, tools and approaches described in this section constrained the language to obtain a subset of NL, still understandable by all the stakeholders but easier to analyze, since the grammar is controled. For instance, in [60], the authors define a context-free grammar for requirements to ease the elicitation process. By positioning their approach in the upstream phases, Scott and Cook are focusing on the analysis of lexical clauses to detect inconsistencies. This approach is part of the elicitation phase and aims to ease this step, removing some ambiguities or inconsistencies, but does not provide a precise semantics to the requirements and so, does not offer a formal verification mechanism. Therefore, both functional and non-functional requirements can be represented, as well as domain knowledge, without any distinction.

Properties:

(a) **Intended audience**: Every audience can read requirements expressed in the constrained NL.

(b) **Traceability support**: There is no traceability support.

(c) **Coverage**: Given the level of abstraction of requirements, all kinds of requirements are addressed.

(d) **Semantic definition**: No semantics is associated to the language.

[4] `https://protege.stanford.edu/`
[5] `https://pypi.python.org/pypi/Owlready`
[6] `http://owl-cpp.sourceforge.net/`

(e) **Verifiability**: Verifiability is limited to the analysis of the consistency of lexical clauses of requirements.

NL/OCL

Description: In [1], Hähnle et al. propose to express parts of a systems constraints in a constrained NL. These constraints are translated in OCL [61], a constraints formal language for UML. In the example on Fig. 2.2, the authors present a matching between a constraint on an OCL operation – an operation that returns the first element of a queue – and its NL translation.

Operation getFirst

OCL:
```
context Queue::getFirst() : Integer
    pre:  self.size() > 0
    post: result = self.asSequence()
                          ->first
```

English: for the operation getFirst() : Integer of the class Queue, the following precondition should hold:
the size of the queue is greater than zero
and the following postcondition should hold:
the result is equal to the first element of the queue.

Figure 2.2: Example from [1] of a matching between English and OCL

This work is focusing on operational requirements and all the concepts should be related to the system itself. OCL is a formal language, and there exist approaches for verification [62]. The goal of the authors is to integrate the approach in the KeY Java-oriented formal verification project [63].

Properties:

(a) **Intended audience**: Input can be read by all stakeholders, but OCL is addressed to software engineers.

(b) **Traceability support**: There are links between requirements and specification, but no specific support.

(c) **Coverage**: Only functional requirements are addressed.

(d) **Semantic definition**: Semantic definition is given in OCL.

(e) **Verifiability**: OCL verification tools can be used to verify the consistency of requirements.

Relax

Description: The Relax language [2] is a requirements language for Complex Adaptive System (CAS). As environmental factors are crucial in CAS, Relax proposes to explicitly express both properties (with `ENV` keyword) and monitors that can provide these properties (with `MON` keyword). Both notions can then be linked through relationships with `REL` keyword. Another keyword, `DEP`, allows creating relationships between requirements. The example Fig. 2.3 let's appear all these keywords. The requirement itself is expressed in the first lines and is identified by `R1.1'`. The other

entries describe environmental knowledge (the food location and the food information) and monitors that can be used to acquire this knowledge. In the end, relationships with other requirements are also provided.

R1.1': The fridge SHALL detect and communicate information with AS MANY food packages AS POSSIBLE.
ENV: Food locations & food information.
MON: RFID tags; Cameras; Weight sensors.
REL: RFID tags provide food locations/food information; Cameras provide food locations; Weight sensors provide food information (whether eaten or not).
DEP: R1.1' negatively impacts R1.2'; R1.1' positively impacts R1.4 and R1.5.

Figure 2.3: Example of a smart fridge requirements from [2]

Both functional and non-functional requirements can be expressed, through the help of temporal notions. The language itself is semantically defined with Fuzzy Branching Temporal Logic (FBTL) [64] (a generalization of Propositional Linear Temporal Logic (PLTL)).

Properties:

(a) **Intended audience**: Relax is close to NL and thus, is readable by all stakeholders.

(b) **Traceability support**: Relationships between requirements can be established, but there is no specific traceability support.

(c) **Coverage**: Only functional requirements are addressed.

(d) **Semantic definition**: Rigorous definition using FBTL is supported.

(e) **Verifiability**: Requirements expressed in Relax can be verified using the FBTL semantic definition.

Stimulus

Description: The approach proposed in the Stimulus[7] tool from Argosim also provides a constrained NL [65]. Functional requirements and environmental assumptions can be expressed in a non-specialist understandable way.

This approach is focusing on verification – traceability is not considered. For this purpose, Argosim based its approach on a programming language, inspired by Lucid Synchrone [66] and Lutin [67], and verification itself is based on model-checking.

Properties:

(a) **Intended audience**: Close to NL, Stimulus is readable by all stakeholders.

[7] `http://argosim.com/product-overview/`

(b) **Traceability support**: Traceability is not addressed in Stimulus.

(c) **Coverage**: Only functional requirements are addressed.

(d) **Semantic definition**: There is a precise definition in a programming language inspired by Lucid Synchrone and Lutin.

(e) **Verifiability**: System behavior is model-checked by simulating inputs.

2.1.6 Summary and discussions

The table 2.1 summarizes the approaches surveyed in section 2.1. As stated in section 2.1.1, we can identify that there are two worlds:

- NL-based approaches (DOORS, Reqtify, KAOS, SysML, URML) are focusing on the readability of requirements and the traceability, notwithstanding the semantics and the verifiability of requirements themselves;
- formal approaches (VDM, Alloy, Event-B, FORM-L) provide a strong semantics and mechanism of verification but are not addressing the traceability.

In the table, we analyze for each of the properties if the approach support it (✓), does not support it (✘) or partially supports it (∼). For the intended audience, we emphasize if the approach allows expressing requirements understandable by all stakeholders (A) or only by experts (E). For the approaches that transform NL to another language, we only take the output language to analyze this property.

	Intended Audience	Coverage	Traceability support	Semantic definition	Verifiability
DOORS	A	✓	✓	✘	✘
Reqtify	A	✓	✓	✘	✘
KAOS	A	✓	∼	✓	✘
SysML	A	✓	∼	✘	✘
URML	A	✓	∼	✓	✘
URN	A	✓	✓	✘	✘
VDM	E	✘	✘	✓	✓
Alloy	E	✘	✘	✓	✓
Event-B	E	✘	✘	✓	✓
FORM-L	E	∼	∼	✓	✓
NLP	E	✓	✘	✓	✓
NL to STD	E	✓	✘	✘	✓
NL to OWL	E	✓	✘	✓	✘
Requirements grammar	A	✓	✘	✘	✘
NL to OCL	E	✘	✘	✓	✓
Relax	A	✓	∼	✓	✓
Stimulus	A	✘	✘	✓	✓

Table 2.1: Summary of the approaches

NLP and Constrained NL aim to bridge the gap between these two worlds, but none of them is covering all the properties. One of the properties that is the most poorly treated is the traceability: (i) approaches that formalize traceability links only consider links between artifacts of the approaches and (ii) approaches that provide links to other formalisms, do not provide a clear semantics for these links.

Indeed, traceability in the most popular tools is considered as simple links, connecting to artifacts of the system, with no specific semantics. While most of the widely used tools are focusing on traceability, the most formal tools presented in this section do not address this issue. It is, however, crucial to be able to link several representation of a requirement, in several tools. As IT and software are more and more present in everyday life, projects often involved stakeholders from different backgrounds, that are used to work with different tools. It is thus crucial to give a real importance to these traceability links, and we will analyze how can we do that in the next section.

2.2 Model-Driven Engineering

One of the main issues leading to failure in Requirements Engineering is the need to express requirements in several languages. As previously introduced, requirements are most of the time expressed at first in NL. They are then refined and formalized in a concrete realization. Even if not using the strict formal representation of requirements, there are several levels of abstractions to express requirements, and we present in section 2.2.1 how these levels of abstractions can be addressed. In section 2.2.2, we present approaches that try to overcome not only the problem of using several levels of abstraction but also the use of different formalisms, specific to different stakeholders.

2.2.1 Representation of artifacts through several abstractions

Development based on waterfall processes or V-model is built around the idea of creating artifacts that are less and fewer abstracts. The first artifacts give a general idea of what the product shall be, and own a high level of abstraction, while programs and code are very concrete. Even in more modern methods like Agile [68, 69], these several levels of abstraction subsist (from user stories that express requirements to code). Interweaving these several levels is crucial for traceability; indeed, it is important to keep consistency between these several levels to avoid misconceptions. We will quickly introduce in two following sections how some of the approaches mentioned in section 2.1 are considering these several levels of abstraction, and we will then analyze how some other approaches try to overcome the issue of expressing the same requirements at several levels of abstraction.

Modeling languages

Modeling Languages, such as UML or SysML, is one of the main kinds of approaches that come to mind when talking about levels of abstractions. These languages provide several diagrams, each of them addressing a specific design problem. For instance, UML owns several views, with a view dedicated to express "*what*" shall be the system (use-case view), while the logical view expresses *how* the system shall meet its requirements. Even if several levels of abstractions are expressed in several diagrams, these diagrams are part of a whole unique model and are thus linked.

It is even clearer in SysML. As was introduced in section 2.1.2, the *Requirements Diagram* of SysML allows the user to link requirements artifacts to other model elements. Thus, it is easy to create a link from a requirement, very abstract, to block elements that describe a specific part of the system's behavior. However, as analyzed in section 2.1.6, such links do not own formal semantics and depending on the approach or the tool used by the development team, a *derived* link can take a very different meaning.

Formal methods

The formal approaches introduced in section 2.1 (such as Event-B or Alloy) also address several levels of abstractions. Indeed, it would be utopian to expect an engineer to write, from scratch, a complete and detailed description of the system, even more using formal notations. That is why formal methods are based on the idea of refinement: the user will express at first a very general and abstract description of elements of the system, and will gradually introduce less abstract and more detailed elements.

Contrary to models introduced in the previous section, formal methods are semantically defined and thus, links between these several levels of abstractions are also defined (and are important to keep the consistency of the design).

However, in such methods, links to two essential levels of abstraction are missing: the very first level, the requirements expressed in NL, and the very last one, the code. If most approaches allow code to be generated from a more detailed design, this generation does not keep a link between elements of the formal model and the implementation. Any change in the specification thus implies regenerating the whole code completely.

Behavior-Driven Development

To interweave several levels of abstraction, the Behavior-Driven Development (BDD) approach [70] aims to express the functionality of the software in natural language and automatically translate these parts into software fragments. This approach aims to ensure that all expressed requirements will be implemented in the system. The generation process provides an empty system specification – that engineers can complete – and tests that will fail at first because of the empty specification. The first canvas is thus an abstract representation of behavior, in a kind of NL, while the body written by the developer will be a more concrete representation. By completing the specification, engineers will satisfy requirements and have traceability on which requirements are not yet satisfied.

```
Scenario 1: Account is in credit
Given the account is in credit
And the card is valid
And the dispenser contains cash
When the customer requests cash
Then ensure the account is debited
And ensure cash is dispensed
And ensure the card is returned
```

Listing 2.1: Example of a BDD scenario (taken from [70])

BDD is thus an approach that provides a way to express functional requirements. Each functional requirement is called a scenario that takes part in a story. A story aims to provide objective and added value of scenarios. A scenario of a functionality is composed of three parts:

- the context, *i.e.*, constraints of the system that must be held to realize the functionality;
- the trigger event of the functionality;
- the action to realize.

In a classical structure of a BDD scenarios, the first part is introduced by `Given`, the second part by `When`, and the last part by `Then` (see an example in List. 2.1). Contrary to previous approaches, this approach focuses on implementation and check the realization of requirements by tests. It does not provide any verification mechanism. Moreover, even if the requirements are not considered as a textual unit as in NL approaches, the several parts of a BDD scenario are treated as a unit. The semantics of these units should then be defined by engineers directly into source code.

Multirequirements

The idea of interweaving several levels of abstraction can remind the Single Model Principle [25], mentioned in section 2.1.1. In the paper, R.F. Paige and J.S. Ostroff argue for an approach that expresses requirements in the same formalism expressed in NL, in modeling notations, and in code. This shall lead to reduce the gap that can exist between these several representations.

This idea is taken up in the multirequirements approach introduced by Meyer [71]. In this approach, a unique programming language, Eiffel, is used to write the requirements, the specification, and the implementation. The approach proposes to combine specification expressed in natural language, using code comments, diagrams (embedded into the Eiffel main IDE, EiffelStudio), and design by contracts, to express requirements formally. The implementation itself can then be expressed through this specification canvas.

One of the main advantages of such an approach is that it uses, like BDD, a paradigm that will be necessarily used during the development: the programming language. Moreover, expressing it in a unique paradigm can improve traceability: the user does not have to navigate through several tools, but only have to use a unique dedicated tool: the language IDE.

Requirements expressed on the form of contracts are formal: they are semantically defined, they can be analyzed with static verification tools, and can even be used for dynamic verification.

However, the link to other paradigms is not widely explored in this approach: the object-oriented paradigm is well-known by software engineers, and they can easily understand specification written with this formalism, but large projects do not only involve software engineers.

It thus lacks some research efforts on how to keep the link between all the stakeholders, including their usual tools and viewpoints.

2.2.2 Addressing multiple viewpoints

Addressing multiple viewpoints is not an issue specific to RE. Indeed, conciliating several stakeholders is a common problem in large projects. Such projects imply several stakeholders, with several tools, and the information can be expressed in several heterogeneous ways. However, it is important to keep a method to conciliate these several viewpoints; they are all part of the same system and thus need to be able to be conciliated. There is a large research effort in this domain; in [72], a systematic review is made and the authors find more than 8600 research papers addressing this problem. Similarly, in [73], the authors analyzed some of the most recent and most used approaches. In this section, we will list some of these approaches used to overcome the issue of addressing several viewpoints.

Model-based approaches

Model-based provide several mechanisms to create dedicated views on a problem. Indeed, as mentioned in section 2.2.1, several diagrams can be used for the same model. Such diagrams can be seen

as views, dedicated not only for a level of abstraction but also for a specific point of view. Thus, the SysML requirements diagram can be understood by most of the stakeholders, while the blocks diagram is more dedicated to engineers. In the same way, the use case diagrams of UML can be used as a communication tool between the development team and customers, while class diagrams are addressed to engineers. These several views (or diagrams) are possible due to the distinction between the abstract syntax and the concrete syntax; while the abstract syntax has to store all the information of the models, several concrete syntaxes can be linked to the same abstract syntax, each representing a partial view of the system.

Some tools, such as Sirius [74], allow the user to define concrete syntaxes associated with an abstract syntax (a model). The user can thus define several views for the same model. However, the source model still requires to be complete and represent all the project facets. Moreover, this implies having access to all the data, which can be a problem in industrial projects that involved several companies (for reasons related to intellectual property protection policy).

The GEMOC initiative [75] aims to globalize the use of model-based tools and methodologies to ease these approaches' transition. This interface can be involved to connect different Domain-Specific Modeling Languages, used to express the specific needs of different stakeholders. These can thus continue to work with their usual tools and Domain-Specific Languages – languages addressed to a specific domain –, and models can be used as based artifacts to bridge the gap between the several DSL. Moreover, the acceptance of this approach can be eased by the growth of interest for Model-Based Systems Engineering (MBSE) [76] in the systems engineering industry.

EMF approaches

Eclipse Modeling Framework (EMF) is one of the most popular frameworks to express and manipulate models. Since it is one of the most used tools for model manipulation, several approaches have been designed to integrate this notion of viewpoints into the framework.

In [73], the authors cite EMF Profiles [77] as an approach dedicated to the multiple views problem. This approach takes the idea of "profile", widely used in UML world, and integrates them into EMF. Profiles, in UML, are used to extend models to domain-specific problems. It gives stereotypes to specific elements; for instance, a class can be stereotyped as a requirement or a goal. The profile is however stored in a new model (the original one is conserved). A profile for a model can thus be seen as a specific view of this model, and several profiles applied to a single model as several views of this model. Such an approach is, however, not primarily addressed to define views and needs specific dedicated tools.

The EMF Views approach [78] provides a language and a tool to define views, similarly to database systems. Contrary to EMF Profiles, these views are not new models with a copy of elements from the original model, but a kind of embedded query language that will ask the original models. Thus, a view defined with EMF Views does not modify the original metamodel. Several views can thus be associated with a single model that federates the elements of each view. However, such an approach still needs the user to define a complete model with the elements of each of the views.

OpenFlexo

In [79], the authors propose a new method to link specifications expressed in several ways. Contrary to the traditional MDE approach that proposes using model transformations to change from one model to another, they propose a "space" vision. Stakeholders' models can be seen as technical

spaces, used to express specific needs of a domain, and to compose several subsystems in a common one, a conceptual space is used to link interfaces of different domains.

They so mapped concepts expressed with several paradigms (EMF, XML, Word documents, etc.) in a common interface. If a stakeholder chooses to change specifications, the common system will be impacted, but contrary to the model transformation approach, other stakeholders will not have to transform models again. They provide open source tools to their methods, grouped in the Openflexo project[8].

In [80, 81], the same authors propose to apply their model federation principle to requirements. They consider the different ways to express requirements as technological space and link them in the conceptual space. For example, requirements expressed in a Word Document can be linked to other requirements expressed in KAOS.

The use of a central interface language, their conceptual space, facilitates the translucence between several stakeholders. The transparent model transformation mechanism can measure the change impact from a paradigm to the others.

However, it is needed to implement several technological connectors to bridge the gap between their virtual models and the technological spaces. Thus, if a technological connector does not exist, the writing of such a connector is needed. Even if there exists an open-source tool, there is a need for a software expert to implement it. Another research direction can be to explore automatic models transformations to deduce technological connectors.

2.2.3 Summary and discussions

MDE gives several solutions to address different points of view, whether for several levels of abstractions or several tools used by users. In order to express several levels of abstractions, most of the approaches are proposing to use a unique language. However, these approaches are not end-to-end approaches, and gaps are still existing between some representations of requirements: NL requirements, for example, are not considered in most of the approaches or, at least, not in their usual initial form: in textual documents.

Approaches that try to conciliate several points of view are usually considering a unique model that integrates all the information of the project. The problem of such approaches is that if a new point of view is added, the model shall be modified with several new information. Openflexo does not have this problem: new models are plugged to existing ones, creating a common virtual model. However, this technology is quite new. Moreover, contrary to the classical model transformations, it is not mastered by many people for now.

2.3 Conclusion

Ensuring the quality of a project by tracing requirements is a main problem of engineering. The approaches presented in this section all try to provide good requirements and good traceability between the several points of view that can co-exist.

However, all these approaches are facing the gap existing between:

- NL requirements, affordable and understandable by all stakeholders but, obviously, not formal and consequently, potentially ambiguous, inconsistent, not complete, etc.;

[8] `https://www.openflexo.org/`

- and more formal representation of requirements, in a dedicated language, but affordable by only some experts in the project.

Formal approaches should be considered: the only way to automate the analysis of requirements will go through the formalization process, either by an end-to-end formal approach or by a transformation from non-formal to formal approach (through NLP or approaches that provide a more abstract view of the system). However, one can ask why not all engineers use formal approaches? Indeed, industrial practices usually do not enact formalization of requirements [82], and they use natural or modeling languages to express requirements and manual processes to trace them from analysis to implementation through design [83]. An effort shall, however, be made to ease the acceptance of such methods.

On the one hand, the skills required to work with classical formal methods are not widely known, and not all the engineers can understand such approaches. On the other hand, approaches based on the programming language have the advantage of being formal while remaining accessible to engineers. From a multi-view perspective, these approaches can be used to express several levels of abstraction in the same language. Moreover, the more-and-more used agile approaches could include these approaches since they need frequent exchanges with customers and directly impact source code changes.

Furthermore, the approaches presented in this state of the art are poorly addressing the traceability problem and how a change can impact other artifacts (either requirements or parts of the system).

There also exists a lack of a bridge between existing approaches and practices. Indeed, it is utopian to propose a single model for projects. While this may work on fairly simple projects, in complex systems, where many stakeholders are involved, tools adapted to specific parts of the system should be proposed. This would require a considerable amount of work, while not providing tools as adapted as those already existing. Moreover, the various stakeholders' uses and habits would undoubtedly hinder the acceptance of a single model. Therefore, it seems more realistic to propose a model dedicated to requirements engineering while creating bridges to existing tools. The different stakeholders could thus continue to use their usual tools while benefiting from this model's contributions.

Part II

Contributions

Chapter 3

The Seamless Integration of Requirements in CODe (SIRCOD) approach

"Any fool can write code that a computer can understand. Good programmers write code that humans can understand."

Martin Fowler

Contents

As mentioned in the introduction of this thesis, requirements are the base of an engineering process. However, while there is a real need to express them in a way that is affordable by all the stakeholders – after all, everyone has to clearly understand what the developed system is –, these requirements have to be defined in a way that is clear enough to leave no ambiguity – and ambiguity is a property of NL. So, the research in RE proposed several ways to try to overcome this issue: strict syntax for requirements document, analysis tools, formal languages. From our point of view, the proposed approaches suffer from their quality: either they are not formal enough and thus, can lead to the introduction of ambiguity, either there are formal enough, but engineers that are non-expert are reluctant to use such approaches. In this chapter, we propose an intermediate

way: the use of programming language. This shall lead to introducing some formalism affordable by the software engineers that work on a project.

3.1 Why use a programming language?

Requirements and code are closely linked to the development of software systems. Indeed, each piece of code should answer a need expressed in a requirement, and each requirement should be satisfied by (at least) one piece of code. However, most of the time, developers are asked to write code in dedicated editors, that can provide tools for comments and documentation, but requirements remain artifacts in other formalisms, disconnected from the written code (Requirements documents written in Microsoft Word, issues in an issue-tracking manager or even user stories). If a change is introduced in code or requirements, there is no simple mechanism to analyze and impact this change.

By interweaving requirements and code, approaches based on the Single-Model Principle try to overcome this issue. The multirequirement approach [71] emphasizes the need to express requirements and code in the same formalism: the programming language. An advantage of such an approach is allowing the developer to access the requirements when programming in the same environment.

However, this approach suffers from several issues: first, the relationship between the several levels of abstraction of requirements (from the requirement itself to the code that realizes it) are not clearly defined. Then, even if some links can exist between these several representations in the multirequirement approach, they are not semantically defined. To ease the analysis of the code regarding the requirement, a semantic link shall exist between them.

This chapter will present the Seamless Integration of Requirements in CODe (SIRCOD) approach, which proposes exploring the multirequirement principle to overcome the issues mentioned above. In the following sections, we will present the basis of SIRCOD, developing the methodology and the tools introduced in the EiffelStudio IDE to support this methodology. This approach aims to allow developers to link informal requirements and their more and more formal realization, along with the system's implementation.

3.2 Overview of the approach

As mentioned in the introduction, requirements are the basic bricks of the system. As such, they are some of the first artifacts introduced during development. This early introduction leads to several facts: (i) the first phases involve several stakeholders that are not all experts of requirements or code, and (ii) requirements can exist in several representations but are, most of the time, expressed in NL.

Based on that assumption, it should be considered that NL requirements are the entry point for any development approach. This is why, even if we consider that early formalization is a good practice (as aforementioned in section 2.1.1), we propose in SIRCOD to start from requirements documents.

The process presented in Fig. 3.1 is thus quite classical, starting with the extraction of requirements and leading to the implementation. However, applying the multirequirement approach (and the Single Model Principle) means that all these steps are made in the programming language environment.

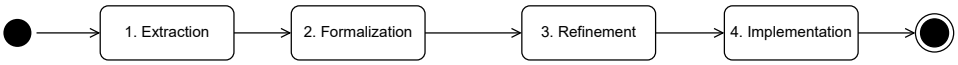


Figure 3.1: The four steps of the SIRCOD process

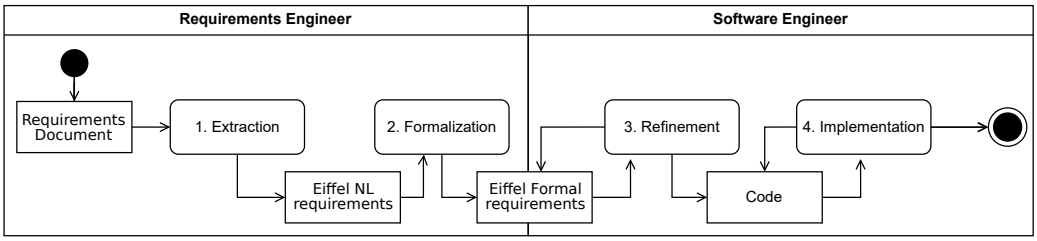


Figure 3.2: Detailed steps of the SIRCOD process

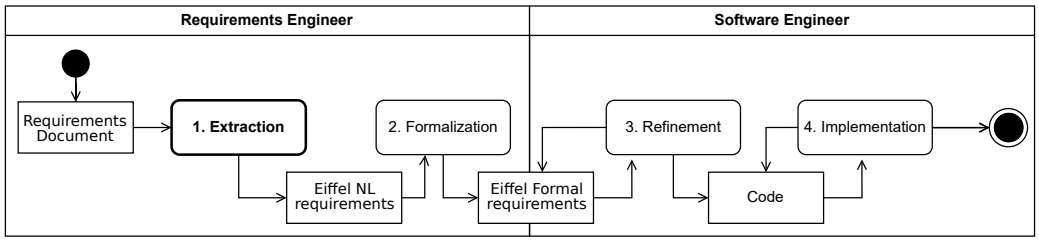
In Fig. 3.2, we present the several artifacts that will be implied in the process. SIRCOD thus start from a requirements document (in NL), and will progressively introduce them into the Eiffel language, first as NL artifacts and then, as formal artifacts. Finally, the last output is the code itself.

In the classical processes of RE, the very first step is the elicitation of the requirements. The purpose of this step is to obtain a first draft of the user's needs which can then be refined. In the context of this section, we do not address the elicitation step.

While in classical approaches the requirements will be refined in a system requirements document, the idea here is to introduce the Eiffel representation of the requirements as soon as possible, the refinement process to obtain system requirements taking place in the code. Thus, in a similar way to formal approaches, refinement will take place in a formal framework and verification and validation process will take place in the approach. Unlike these approaches, however, there will be no code generation step, as the use of the programming language itself allows to avoid this step.

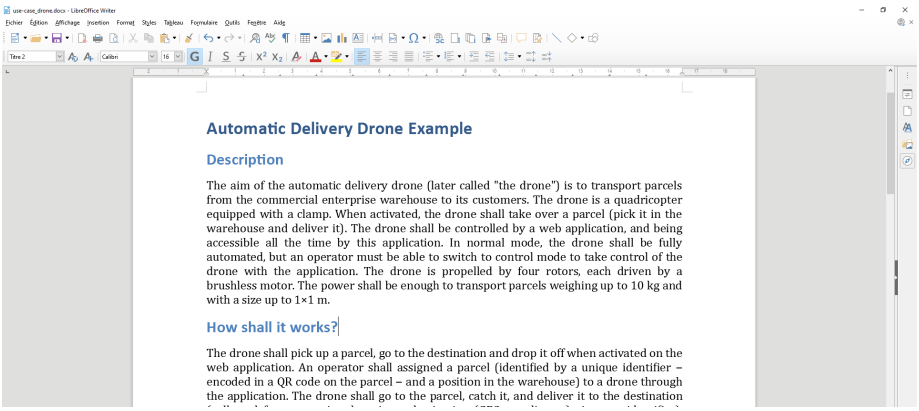
In the following sections, we will detail each activity and present the changes we made in the EiffelStudio environment to support the approach.

3.3 Requirements: from Natural Language documents to code artifacts



3.3.1 From Requirement-in-documentation to Requirement-in-code

Once the elicitation stage is over, the requirements are most of the time expressed in a user requirements document. In the case of the GORE approaches discussed in section 2, which are part of this elicitation stage, the main tools (such as Objectiver) also allow the generation of a requirements document. Thus, whether the elicitation is done using classical method or with a GORE approach, it is necessary to start by extracting these requirements and express them into a programming language (activity 1.a).


Automatic Delivery Drone Example

Description

The aim of the automatic delivery drone (later called "the drone") is to transport parcels from the commercial enterprise warehouse to its customers. The drone is a quadricopter equipped with a clamp. When activated, the drone shall take over a parcel (pick it in the warehouse and deliver it). The drone shall be controlled by a web application, and being accessible all the time by this application. In normal mode, the drone shall be fully automated, but an operator must be able to switch to control mode to take control of the drone with the application. The drone is propelled by four rotors, each driven by a brushless motor. The power shall be enough to transport parcels weighing up to 10 kg and with a size up to 1×1 m.

How shall it works?

The drone shall pick up a parcel, go to the destination and drop it off when activated on the web application. An operator shall assigned a parcel (identified by a unique identifier – encoded in a QR code on the parcel – and a position in the warehouse) to a drone through the application. The drone shall go to the parcel, catch it, and deliver it to the destination

Figure 3.3: User Requirements Document of the Automatic Delivery Drone

The methodology presented in this section supposes a manual extraction of requirements (and their relationships) and their translation into the chosen programming language. In chapter 4, we will present the automatic translation of requirements into code and the tool that supports it.

The Fig. 3.3 illustrates a Microsoft Word document containing a set of requirements. The very first step of SIRCOD consists of creating an Eiffel method (called *feature*) for each requirement. These features – that we called **requirements features** – returns a sentence describing the requirement (e.g., in List. 3.1, the feature *requirement_1_1_doc* returns the expression in NL of the corresponding requirement).

feature

```
requirement_1_1_doc: STRING
do
  Result := "[
    The drone shall pick up a parcel, go to the destination and drop
        it off when activated on the web application.
  ]"
      end
```

Listing 3.1: Eiffel feature describing a requirement

By convention, we named this feature with a suffix *_doc* to differentiate them from other code artifacts – the *doc* here, meaning that the feature is for documentation purposes. List. 3.2 gives the pattern to apply. Even if requirements are still expressed in NL, they are now artifacts of the programming language and can be used as they are.

```
feature
  <identifier of the requirement>_doc: STRING
  do
    Result := "[
      <text of the requirement>
    ]"
        end
```

Listing 3.2: Eiffel pattern for a requirement feature

3.3.2 Linking requirement-in-code and requirement-in-documentation

The second step (activity 1.b) consists of linking the requirement-in-code and its original document to allow the navigation from one to the other. Thanks to Eiffel Information System (EIS), an EiffelStudio mechanism, a code element (a feature, for example) can be linked to an external document. More precisely, it can be linked to a piece of text of this document, thanks to bookmarks.

As shown in Fig. 3.4, the user can choose a bookmark on the document – to ease the addition of such bookmarks, if some paragraphs of the documents do not have bookmarks, generic ones are added by the tool.

The selection of a bookmark will add a **note** to the given feature. Eiffel notes can be used to enrich the features that express requirements. These notes can be compared to Java annotations and do not embed any semantics, except the one given by tools using them.

For example, the link between the original source of the requirement of List. 3.1 (the red dashed underlined sentence in the MS Word document of Fig. 3.4), and the Eiffel requirement feature (*requirement_1_1_doc*) is expressed by the note:

```
EIS: "src=use-case_drone.docx", "bookmark=1.1".
```

The produced code is given in List. 3.3, and the updated pattern is given in List. 3.4.

```
feature
  requirement_1_1_doc: STRING
  note
    EIS: "src=use-case_drone.docx", "bookmark=1.1"
  do
    Result := "[
```

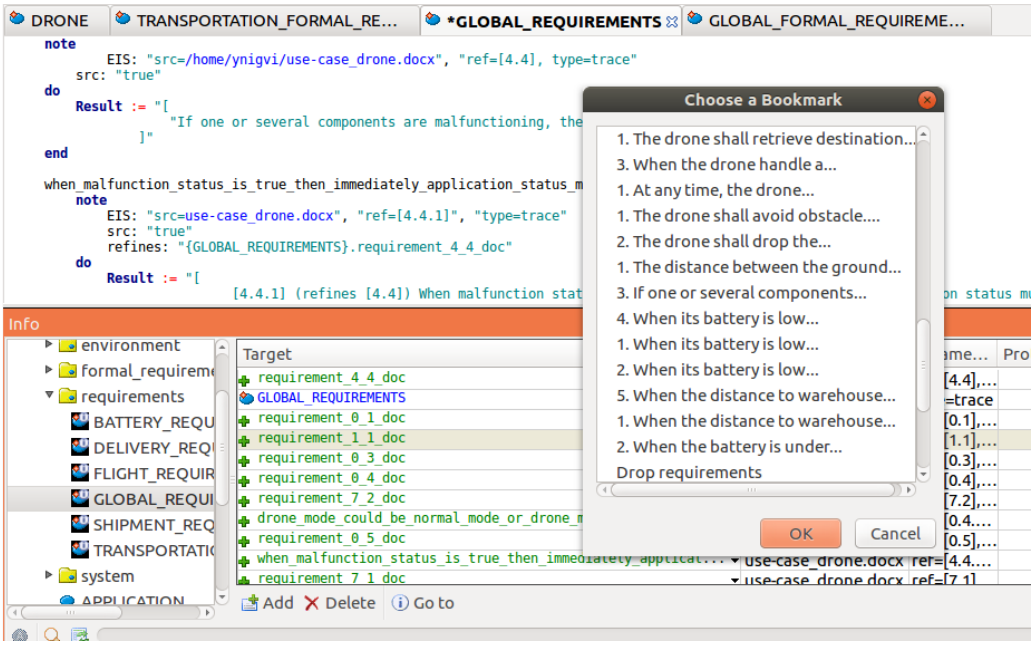


Figure 3.4: The "Info" view that allows the user to graphically add a source

```
    The drone shall pick up a parcel, go to the destination and drop
        it off when activated on the web application.
]"
    end
```

Listing 3.3: Eiffel feature documenting a requirement

```
feature
  <identifier of the requirement>_doc: STRING
  note
    EIS: "src=<path to source document>", "bookmark=<bookmark of the
        requirement>"
  do
    result := "[
      <text of the requirement>
    ]"
    end
```

Listing 3.4: Eiffel updated pattern for a requirement feature

3.3.3 Navigating from requirement-in-code to requirement-in-documentation and vice versa

As introduced in the previous section, we emphasize creating an Eiffel representation of requirements and a link to the source. If the introduced representation is quite simple and easy to read, it is utopian to use a unique language to write requirements. First, due to the force of habit, most of the commonly used representations such as MS Word documents or spreadsheets will probably remain used for a long time. Then, different representations can be useful to address several stakeholders: some of them will be more comfortable with natural language, other with graphical representations, and so on.

Thus, it is important to find a way to keep the links between these representations and the code, which is supported by EIS. This mechanism that we introduced in section 3.3.2 allows creating links from (and to) source code to (and from) other documents. These links are encoded directly into the code, through notes, and take several arguments. The bookmark is one of the most interesting arguments since it allows the user to define precisely into the document where the referenced extract is. Since bookmarks can be put at any place in a document's structure, such a link allows users to create relationships between any nodes of two trees: the syntax tree of the code and the document's tree structure.

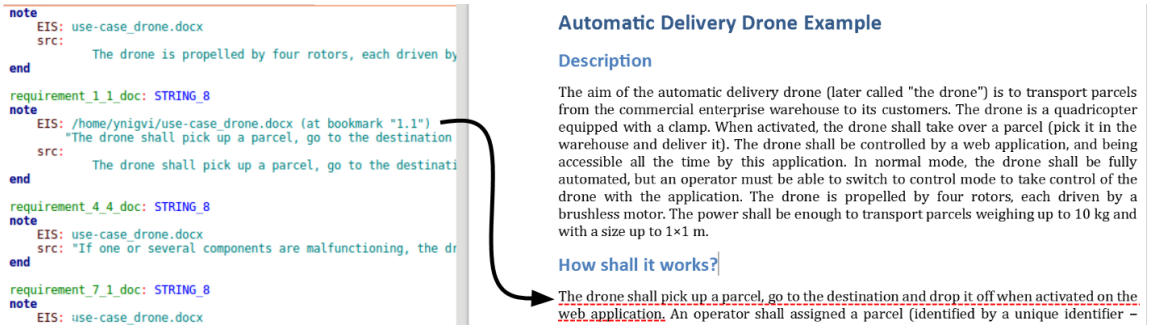


Figure 3.5: EIS link from code to a requirement

As already mentioned, it is quite simple to create a link to an existing – or a generated – bookmark. Thanks to EIS, if one of the end of the link between code and document is modified, the developer is notified and can check and update the change. Fig. 3.6 gives an example of a change notification. Since this feature has been modified, the *Affected source* list (in the hierarchy tree at the left of the figure) shows the user that he should check the change's impact on the document. Similarly, the *Affected target* list will warn the user when an external source, linked to a part of code, has been modified.

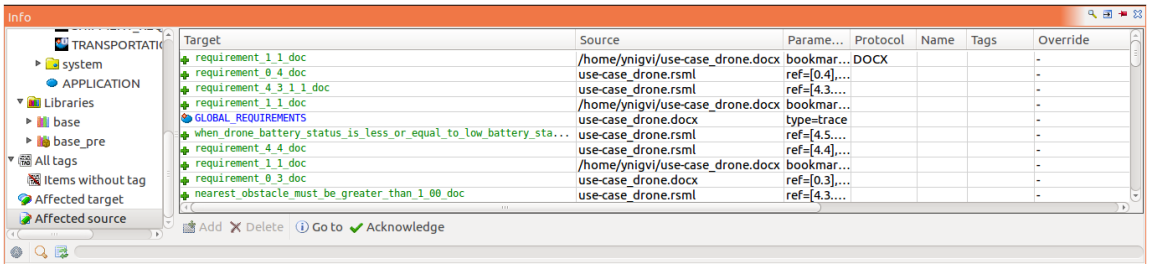
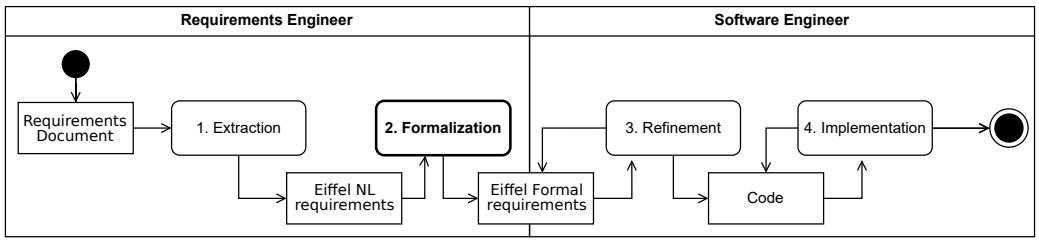


Figure 3.6: Notification of several affected sources after modification in code

3.4 Linking requirements and their formalization



After expressing the NL requirements in code, thanks to the **requirements features**, the second step of the process (activity 2) aims at introducing some formality. This step's objective is to ensure that the future system will meet its requirements. The very first step is, in our opinion, before any other consideration, to enforce the developer to consider the requirement during development. To this purpose, we based the SIRCOD approach on a refinement process, starting from the requirements, at the higher level of abstraction, going to the implementation. This is the common approach in formal methods, from the classical VDM [44] with data reification to more recent approaches like Event-B [34]. This idea to start from abstract to go to concrete is also something existing in Object-Oriented programming, and is made possible thanks to the notion of *abstract* methods and classes (*deferred* in Eiffel).

The very first step of formalization is thus to provide, for each requirement, a *deferred feature* that is a formal representation of this requirement. The developer who will have to inherit from the class where the requirements are formalized will have to consider this feature. Otherwise, the system will not compile (we detailed this below in section 3.5). These *formal requirement features* can be obtained by applying the pattern given in List. 3.5.

To keep complete traceability to the original requirements, we created an **src** note that can be used to link every code artifacts. Code artifacts are identified as given in List. 3.5: first, the class where is the **requirement feature** between brackets ({`<class containing the requirement feature>`}), followed by a dot and the identifier of the **requirement feature** (`<identifier of the requirement>_doc`).

```
feature
  <identifier of the requirement>
```

```
    src: "{<class containing the requirement feature>}.<identifier of the
        requirement>_doc"
  deferred
  end
```

Listing 3.5: Eiffel pattern for a formal requirement feature

By applying this to the requirement 1.1 given in List. 3.3, we thus obtain the **formal requirement feature** in List. 3.6. Note that both features (*requirement_1_1_doc* and *requirement_1_1*) are linked through the **src** note of List. 3.6.

```
feature
  requirement_1_1
  note
    src: "{GLOBAL_REQUIREMENTS}.requirement_1_1_doc"
  deferred
  end
```

Listing 3.6: Formal representation of a requirement in Eiffel

If this pattern can apply to all the requirements, it is possible to go further in formalizing some of them. To this end, it is important to define what is a formal requirement. Formal approaches (such as Event-B [34] or VDM [44]) are based on the idea that a requirement is satisfied if and only if, given some assumption A, we are able to prove that predicate P expressed by the requirement is met (that can be formalized as $A \vdash P$). A requirement is thus a pair of one set of constraints (the assumptions) and one set of properties (the predicates).

3.4.1 Definitions

A requirement R is a set of properties, $\mathcal{P}_R = \{P_1, ..., P_m\}$ (each property $\mathcal{P}_i$ is a predicate) that the system shall meet given a certain context $\mathcal{C}_R = \{C_1, ..., C_n\}$ (i.e., when all the constraints $\mathcal{C}_i$ of the set of constraints $\mathcal{C}_R$ are held). To rephrase, a requirement R is satisfied ($sat(R) \equiv \top$) iff all the properties are met ($hold(\mathcal{P}_R) \equiv ((P_1 \equiv \top) \wedge ... \wedge (P_m \equiv \top))$) when all the constraints are held ($hold(\mathcal{C}_R) \equiv ((C_1 \equiv \top) \wedge ... \wedge (C_n \equiv \top))$).

Fundamental property of satisfaction: Using the previous definition, we can formalize the *satisfaction* of a requirement R as:

$$sat(R) \equiv hold(\mathcal{C}_R) \rightarrow hold(\mathcal{P}_R)$$

Note: If a requirement R has no constraint, it is a global requirement and the context can be understood as "always". So, we can simplify:

$$sat(R) \equiv (\top \rightarrow hold(\mathcal{P}_R)) \equiv hold(\mathcal{P}_R)$$

On the contrary, a requirement R shall have at least one property ($\mathcal{P}_R \neq \emptyset$), since a requirement without property makes no sense.

Comment: Requirements properties are analyzed only when constraints are held. Indeed, the given definition implies that if the constraints are not held, then the requirements can be satisfied whether the properties are satisfied or not. For example, a requirement *"When the drone is off, the altitude shall be 0."*, if the drone is on, no matter if the altitude is 0 or more, the requirement is satisfied in this context. That implies that a requirement is not satisfied only if its properties are not satisfied when its constraints are held: $\neg sat(R) \equiv \neg(hold(\mathcal{C}_R) \rightarrow hold(\mathcal{P}_R)) \equiv hold(\mathcal{C}_R) \wedge \neg hold(\mathcal{P}_R)$.

Using this definition of a requirement as a pair of assumptions and assertions to held, we can match this to the concept of Design By Contracts in programming language as already introduced in [84] or even [71].

Indeed, introducing preconditions (assumptions) and postconditions (assertions) allows us to provide a formal way to express requirements while keeping a programming style that shall ease accepting such an approach. In the same way, it is possible to use invariants to express elements that will be part of the environment: they will be checked before and after the requirements. Note that, contrary to what we developed above, the aim here is to provide a **formal expression** of the requirement, in addition to the **formal representation**.

This cannot be applied to all requirements and, for instance, the requirement 1.1 already presented is a bit too large to be formalized as it is. We thus chose to formally express the requirement given in List. 3.7. This requirement is in the context of the handling phase of the drone activity. It states that when a drone is activated, it shall immediately be able to go to work and so, has to know what is the parcel that it has to take in charge.

```
feature
  requirement_1_2_1_doc: STRING
  note
    EIS: "src=use-case_drone.docx", "bookmark=1.2.1"
  do
    Result := "[
      1.2.1. When the drone is activated then parcel and its position
         shall be assigned to the drone.
    ]"
end
```

Listing 3.7: Feature requirement of requirement 1.2.1

The constraint and the properties of the requirement are here quite simple to found: the constraint requires that *the drone is activated*, and the properties to check are *a parcel is assigned to the drone* and *the position of this parcel is known by the drone.* We apply the pattern from List. 3.8 to obtain the new formalization.

```
feature
  <identifier of the requirement>
    src: "{<class containing the requirement feature>}.<identifier of the
        requirement>_doc"
         require
           <FOREACH constraint c DO>
                          <identifier of c>: <formal expression of c>
           <END FOREACH>
```

```
deferred
      ensure
        <FOREACH property p DO>
                        <identifier of p>: <formal expression of p>
        <END FOREACH>
end
```

Listing 3.8: Formally expressed requirement in Eiffel

Note that, at this step, the feature is still deferred. Indeed, even if the specification is given, the implementation here is not yet written. The formalization of *requirement_1_2_1_doc* is given in List. 3.9.

```
feature
  requirement_1_2_1
  note
    src: "{SHIPMENT_REQUIREMENTS}.requirement_1_2_1_doc"
  require
    when_drone_is_activated: (drone_status = activated)
  deferred
  ensure
    check_assigned_parcel: (assigned_parcel /= Void)
    check_parcel_position: (parcel_position /= Void)
  end
```

Listing 3.9: Formal expression of a requirement in Eiffel

The formal requirements thus expressed are limited to state verification. However, in work done in [85], a way to express timing requirements is given. This work, that has been developed by A. Naumchev in [86] (where a complete set of Eiffel templates to express requirements based on the Dwyer patterns [87] is given) can be used to express more formal expression of requirements. Note that these patterns however imply writing non-deferred features. If this allows for a better expressiveness of formal requirements, this prevents applying a refining approach as already described. A solution is to use two features to express such requirements: the first to express the requirement itself, and the second to be used as a refining artifact. This is the solution given in List. 3.10.

```
feature
  check_<identifier of the requirement>
  note
    src: "{<class containing the requirement feature>}.<identifier of the
        requirement>_doc"
  do
    <Apply the Autoreq pattern to <identifier of the requirement>>
  end

  <identifier of the requirement>
  deferred
  end
```

Listing 3.10: Formally expressed requirement in Eiffel

This is, however, not a satisfying solution, and more investigation in this field is needed.

Nevertheless, the formal expression of requirements has several advantages, whether using contracts or Autoreq templates. The first advantage of such a representation is to be used as a canvas for development. Indeed, the developer that will refine and implement the requirement will have to comply with these contracts. The formal expression avoids any ambiguity, and the link made with the requirement in natural language makes it possible to keep the expression of the language simple to understand. Of course, classical design by contracts can be used, and the written contracts will be used to check the code dynamically.

Moreover, in addition to this dynamic analysis, it is also possible to use static analysis tools, like the Eiffel prover AutoProof [88]. This tool analyzes the written code and checks if it conforms to the assertions. The feedback thus obtained (an example is given in Fig. 3.7) can help the developer detect if the requirement is correctly satisfied.

```
HANDLING_CONTROLLER.search_parcel                    Successfully verified.
HANDLING_CONTROLLER.on_activate                  147 Postcondition check_assigned_parcel may be violated.
```

Figure 3.7: Sample of AutoProof output

3.4.2 The documentation view

Integrating requirements and documentation directly in code shall improve traceability. However, the readability of requirements written in a programming language is not the best possible, with the several code artifacts (such as do, end, Result, etc.). The purpose of the documentation view is to overcome this issue. EiffelStudio already integrates a mechanism of views. For example, the plain text view is the classical view, with the features of a class, while the flat view integrates all the elements of the class' ancestors.

We first added the documentation view as a way to print code documentation available in another class. The idea was that most of the time, in well-documented class (using notes or other doc comments), the tangled expression of the code and the documentation does not allow a clear and simple reading of the code on the one hand and the documentation on the other hand. If the documentation's readability can be improved through HTML generation, for example, with a long excerpt of documentation in code, it can be made difficult to read it. We so decided to extract code documentation in dedicated classes, in the same way, we split requirement features and formal requirement features. However, during development, the engineer still needs to have access to the documentation, and forcing him to switch classes made the task harder.

We thus had a documentation view that allows, in a kind of literate programming way [89], printing both the documentation and excerpts of code. While the process can be used to document the code, it seemed to us that it was particularly interesting in the context of SIRCOD. Indeed, it is thus easy when in a formal requirement feature to switch to the NL representation, improving the readability and the understandability of requirements while keeping the formality of the code expression.

In Fig. 3.8, a screenshot of the documentation view for List. 3.1 is given. In the feature `requirement_1_1_doc`, only the identifier of the feature, the source of the requirement (EIS note) and the content of the requirement appear.

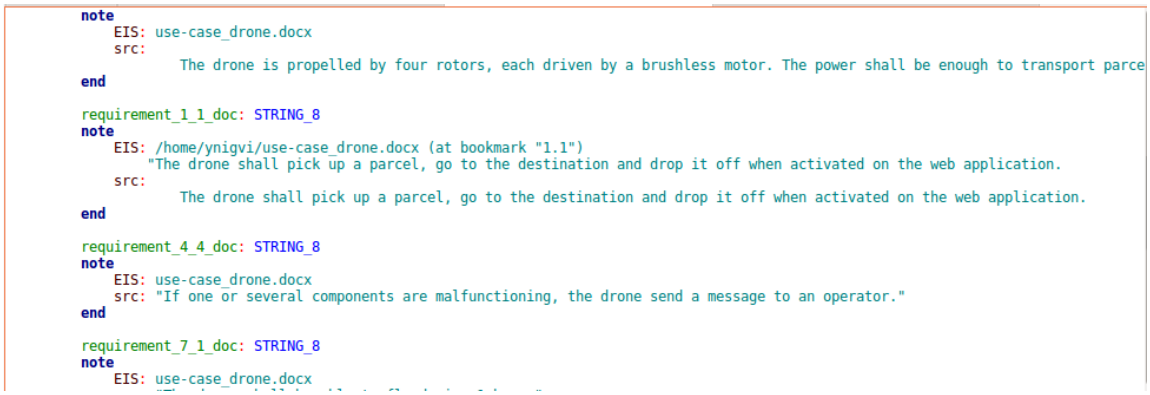


Figure 3.8: Documentation view of requirement features

The text of the source, specified in the EIS note, is retrieved from the docx document and printed. It so simplifies and relaxes the development by preventing the user from opening an external editor. In Fig. 3.8, the EIS link to the bookmark 1.1 in the document `use-case_drone.docx` is used to extract the first sentence of the section 1.1. This view allows better readability by clearing the editor of all elements that could interfere with reading (like the body of the features).

To ease the navigation, the documentation view is also clickable. This implies that all references to code elements, like other classes' features, can be clicked and automatically to open in the editor. This again contributes to the fluidity of development by preventing the user from searching for classes in the file system.

In Fig. 3.9, the documentation view for the List. 3.6 gives an example of how the documentation view can be used by engineers to trace the requirement and the code documentation, putting in the current context the text of the link. Thus, instead of browsing the requirement's source, the user can switch to the documentation view to look at the requirement. This contributes to the development's improvement. Let us notice that if the requirement includes a composition, the documentation view also includes this composition.

Such a view allows for better integration of requirements in both code and documentation. Requirements, code, and documentation are all artifacts of the same model and are linked together by clickable and bidirectional navigable links. However, to avoid the overload of unnecessary information that could impair the readability of the code, in the classical programming view, the elements of documentation or requirements still exist but are just ”links” to other artifacts that are navigable but do not print all the unnecessary text for the developer. Details related to requirements or code documentation are printed in the dedicated view for documentation. This shall ease to handle the information, hiding details when and where they are unnecessary, and highlighting them when useful.

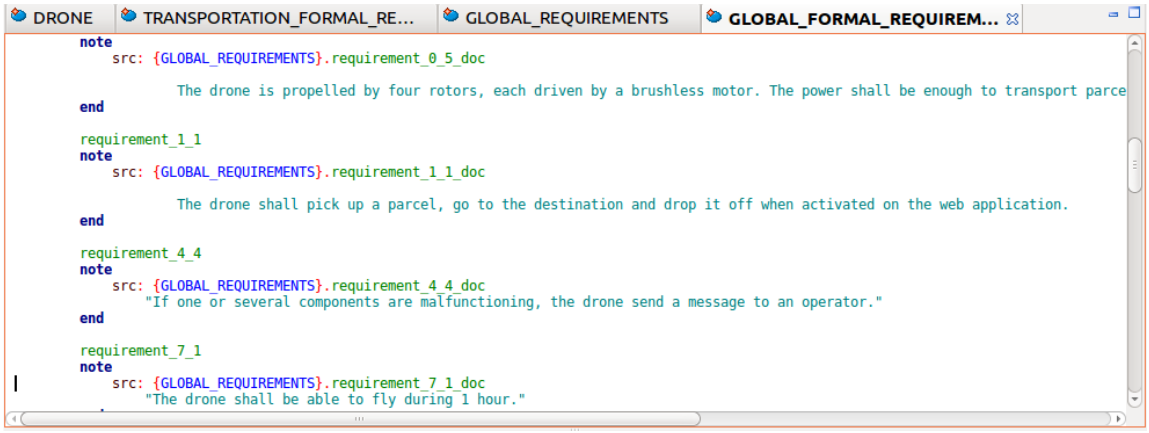
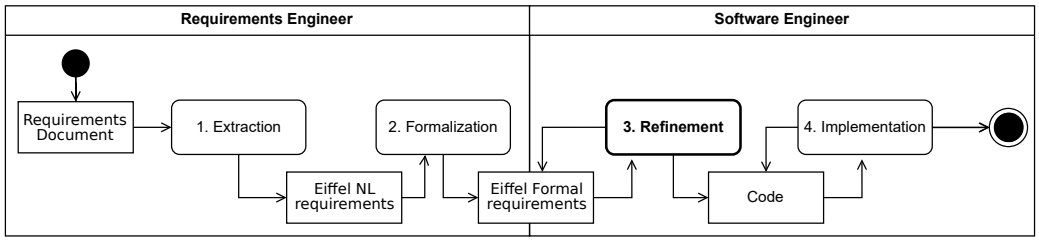


Figure 3.9: Integration of documentations artifacts into code

3.5 Refining requirements



As mentioned in section 3.4, **formal requirements features** can be used as first canvas for a correct-by-construction development. Since these features are deferred, every class that will inherit from the class that contains **formal requirements features** will have to consider these deferred features: either by implementing them directly or by implementing them in their child classes.

This is done in the third activity, called refinement, which is iterative. This process can be compared to the well-known approach of refinement in requirements engineering [31, 90]. The idea is to start from quite simple and abstract requirements and enrich them iteratively, making them more concrete. Object-Oriented paradigm and Design by Contracts are quite suitable for this approach. Following the Liskov substitution principle [91], preconditions and postconditions own the same properties as refinement as defined by Abrial in [34].

```
class HANDLING_CONTROLLER
  inherit
    SHIPMENT_FORMAL_REQUIREMENTS
  rename
    requirement_1_2_1 as on_activate
  end
```

```
  feature
    on_activate
    do
      assigned_parcel := Void
      parcel_position := get_position(assigned_parcel)
    end
end
```

Listing 3.11: Incorrect implementation of requirement_1_2_1

This methodology ensures that all of the requirements have not only been considered by the engineer but also satisfied. Indeed, an incorrect specification, like in List. 3.11, will result in an error during static analysis or else during testing. The **implementation feature** `on_activate` of List. 3.11 shall implement the **formal requirement feature** `requirement_1_2_1` declared in List. 3.6. That means that pre and post-conditions shall be verified. In this example, the assigned parcel is set to `Void`, while it is required to have an assigned parcel when activated, according to `requirement_1_2_1`. Two cases can happen here: the requirement is badly formalized (or even expressed in NL), and the engineer can discuss with the other stakeholders to propose a new good formalization of the requirement, or, more probably, the implementation is false. This is the case for our example: after setting the assigned parcel to `Void`, the engineer tries to get such a non-existing parcel position. This is corrected in List. 3.12.

```
class HANDLING_CONTROLLER
  inherit
    SHIPMENT_FORMAL_REQUIREMENTS
  rename
    requirement_1_2_1 as on_activate
  end

  feature
    on_activate
    do
      assigned_parcel := get_new_assigned_parcel
      parcel_position := get_position(assigned_parcel)
    end
end
```

Listing 3.12: Corrected implementation of requirement_1_2_1

Moreover, it still exists a redefinition link between the **formal requirement feature** and the **implementation feature**. By keeping the link between the different artifacts related to requirements, we ensure traceability from the first NL representation of requirements to their final implementation. There is moreover a strong dependancy mechanism introduced by this refinement step, between an **implementation feature** and a **formal requirement feature** (and so between the classes that contains the two features). This implies several things: first, any feature that would have a dependency on the **implementation feature** would have by transitivity a dependency on the **formal requirement feature**. This is a desired mechanism, as it ensures that any artifact in the system that depends on a requirement will be impacted by the change of the requirement. Furthermore, the dependencies that could exist between a **formal requirement feature** and other

formal requirement feature will also impact a **implementation feature** that would redefine a **formal requirement feature**. Thus a **implementation feature** will be dependent on any **formal requirement feature** that has a direct link or not, impacting the whole dependency hierarchy in case of change. Finally, an analysis of dependencies will allow to find methods which do not depend on any **formal requirement feature**, which can constitute a superfluous functionality, or a contrario of the **formal requirement feature** not redefined, which imply unsatisfied requirements.

3.6 Organizing requirements

The previous sections present a methodology to integrate requirements into code and documentation. Such an integration shall reduce the gap between requirements and code, improve the adequacy of code to requirements, ensure traceability and ease the verification mechanism. In previous sections, we do not emphasize the classes that contain features.

However, in large projects, grouping all the same type features in a single class will not lead to easy management. First, it will become difficult to find a specific feature in a class and read it, despite the documentation's display. Second, the analysis of satisfaction will be complicated.

A solution is to introduce several classes that will address specific parts of the requirements, in the same way, classes in Object-Oriented Programming addressed several parts of the system. However, to ease the management, we recommend separating the concerns and not to mix **formal requirements features** and **implementation features** declaration in the same class. These artifacts stand at two different levels of abstractions, and thus, even if some **formal requirements features** can still be implemented in deferred classes of the system, the declaration of such **formal requirements features** shall be made in requirements dedicated classes. Furthermore, to ease the readability of NL requirements in code, it is better to separate **requirements features** and **formal requirements features** into different classes.

Moreover, we advise not to mix classes of requirements (classes that only contain **formal requirements features** or **requirements features**) and implementation classes in the same cluster or package. Using this decomposition, it is possible to provide the following requirements coverage properties:

- a requirement is satisfied if the **formal requirement feature** of this requirement is implemented;
- a class of requirements is satisfied if all the **formal requirement feature** of this class are satisfied;
- a cluster of classes of requirements is satisfied if all the classes of requirements of the cluster are satisfied.

A new inheritance tree for our running example can thus be for example the one given in Fig. 3.10. Requirements are thus grouped into a cluster of requirements classes, and each of the different classes of requirements are used to separate them, based on their concern.

These definitions, the methodology, and the tool described in previous sections provide a methodology to a kind of *Requirement-Driven development*. Requirements are transformed to artifacts that shall be implemented, and thus ensure that a system developed according to this canvas will:

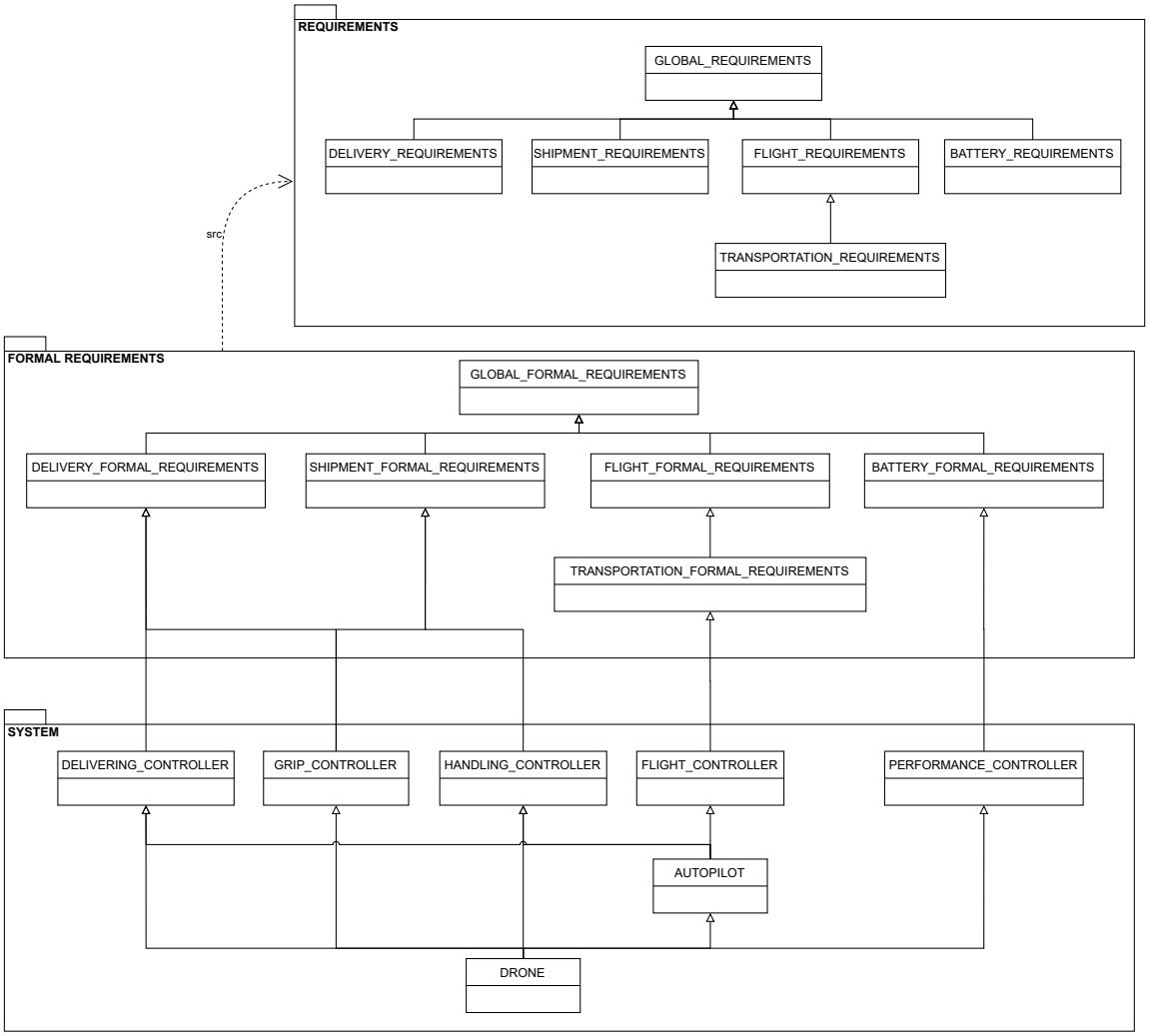


Figure 3.10: Excerpt of the inheritance hierarchy of the running example

- satisfies all the requirements, using the coverage;
- correctly satisfies the requirements, using the formalism of assertions.

3.7 Use case: the Landing Gear System

In this section, we apply the methodology described in the previous section to an excerpt from the well-known LGS case study [92]. This case study describes an airplane landing set, composed of

the landing gear itself, a box that contains the gear, and the doors of the box. The goal here is not to provide in this section a complete approach to LGS but to focus on a few key requirements of the door and landing gear deployment system in order to provide a use case from the original requirements to the implementation.

3.7.1 From NL to Eiffel (Step 1)

The very first step of our methodology is to transpose the natural language requirements, expressed in a document, directly into source code (step 1.a). The resulting requirements, as shown in Fig. 3.11, are linked through EIS to the source (with the finest grain available: a bookmark to the section that contains the requirement – step 1.b) to keep traceability and to prevent user if a change occurs in the file, thanks to the EIS mechanism. This can be done using the graphical interface of EIS to reduce the cost of adding such links. These requirements features can now be used as the other artifacts of the language.

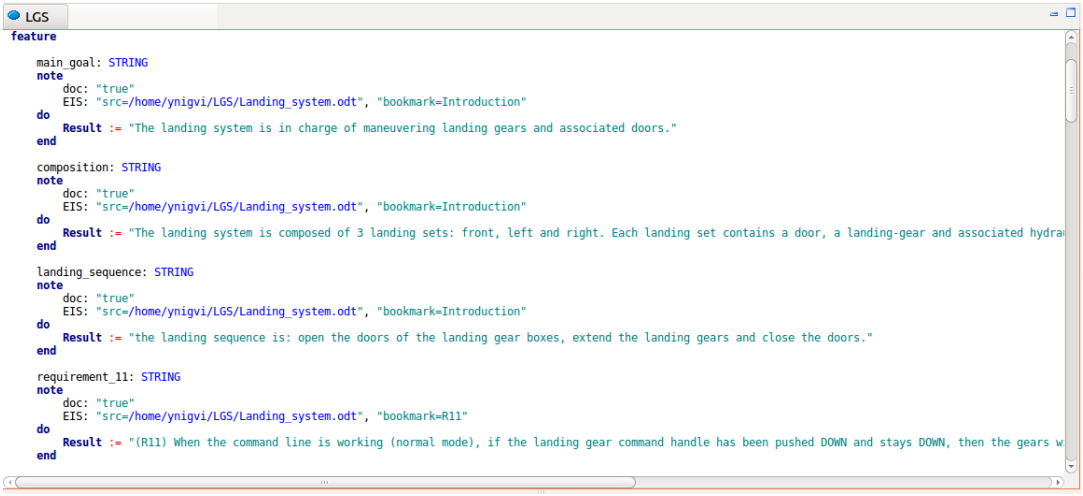


Figure 3.11: LGS requirements in Eiffel code

Moreover, it is possible to switch to the documentation view to ease the readability, as shown in Fig. 3.12, and to compare to the source of requirement.

Indeed, if there is a change in the source file, the EIS mechanism warn the user to check that there is no incidence in the *REQUIREMENTS* class. Using the documentation view, we can see if there is a modification between the textual source requirement and the requirement feature.

3.7.2 Formalization (Step 2)

The following step of the methodology is to introduce some formality in requirements. In Fig. 3.13, a formalization of *requirement_11_bis* is proposed. This feature uses the feature's preconditions to state that we are in the context of normal mode and that the handle has been pushed down. Postconditions shall then ensure that the system meets the requirements (gear extended and doors closed).

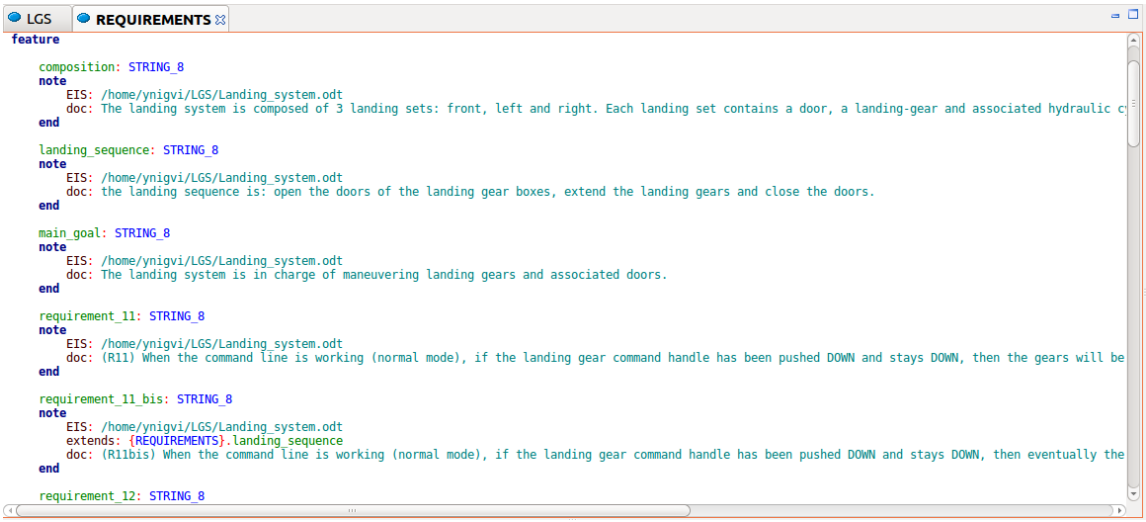


Figure 3.12: Documentation view of LGS requirements

```
requirement_11_bis
note
    doc: "{REQUIREMENTS}.requirement_11_bis"
require
    normal_mode: mode = normal
    handle_pushed_down: handle = down
deferred
ensure
    gears_down: gear = down
    doors_closed: doors = closed
end
```

Figure 3.13: Formal representation of requirement_11_bis

The feature here is declared deferred, i.e., the class extending the $FORMAL\_REQUIREMENTS$ class shall implement this feature. Such an approach aims to ensure requirements coverage. The engineer will have to implement each formalized requirement in the subclasses, ensuring good coverage of the requirements.

According to the methodology, a note link to the documentation (the feature $REQUIREMENTS.requirement\_11\_bis$) is kept here, allowing the switch to the documentation view (as in Fig. 3.14) to print the textual requirement. This traceability link thus ensures a good understandability (by providing a view with a formal representation of requirement and a view with a textual one), but ensures that the feature is up to date with the requirement.

```
requirement_11_bis
note
    doc: {REQUIREMENTS}.requirement_11_bis
         (R11bis) When the command line is working (normal mode), if the landing gear command handle has been pushed DOWN and stays DOWN, then eventua
end
```

Figure 3.14: Documentation view of the formal requirement_11_bis

3.7.3 Refinement (Step 3)

Once requirements have been formalized, the refinement process can be done. Like in Event-B, for example, the idea is to start from the high-level requirements and refine them into executable code. This is done through the inheritance mechanism. It thus inscribes the methodology into the classical object-oriented mechanism, already widely used, reducing the development cost increase that can be introduced by adding the formalization of requirements. However, a good practice encourages contracts to ensure the quality of the code, and the writing of a formal version of requirements, as base classes of the system can be seen as a good way to encourage this kind of development. Moreover, using the documentation view will allow the users to see the requirements using a single click, while he had to refer to the source requirements document in a classical development process.

REQUIREMENTS | LGS

```
    revision: "$Revision$"

class
    LGS

inherit
    ARGUMENTS_32
    FORMAL_REQUIREMENTS
    rename
        requirement_11_bis as extension_sequence
    end

create
    make

feature
    extension_sequence
    note
        doc: "{REQUIREMENTS}.landing_sequence"
        extends: "{FORMAL_REQUIREMENTS}.requirement_11_bis"
    do
        open_doors
        extends_landing_gears
        close_doors
    end
```

Figure 3.15: Refinement of requirement_11_bis

For example, in Fig. 3.15, *requirement_11_bis* has been refined into the feature *extension_sequence*. This feature shall satisfy the *requirement_11_bis* formal requirement feature (since it shares the preconditions and postconditions) and, as shown by the documentation in Fig. 3.16, the extension sequence is decomposed in: opening the doors, extending the landing gear, and closing the doors. Note that although this is not detailed here, the features $open_doors$, $extends_landing_gears$ and $close_doors$ are themselves refinements of requirements, notably concerning the pilot interface (the feature $extends_landing_gears$ must, for example, modify the state of a light according to the position of the gears) or even the electrovalves' commands.

This is represented by the call to functions that shall realize these operations.

```
extension_sequence
note
    extends: {FORMAL_REQUIREMENTS}.requirement_11_bis
    doc: {REQUIREMENTS}.landing_sequence
        the landing sequence is: open the doors of the landing gear boxes, extend the landing gears and close the doors.
end
```

Figure 3.16: Documentation view of extension_sequence feature

Moreover, thanks to the inheritance mechanism, if a formal requirement is modified, the change automatically impacts the redefining features contracts. All the features that redefine this requirement shall comply with new requirements – otherwise, a static or dynamic analysis will fail to prove the system correctness. This thus ensures to automatically take into account any change in the Eiffel requirements on the system.

Since formalized in a programming language, such requirements features can then be reused in several ways, other projects can easily import requirements (or at least, the relevant part of requirements). Thus, it is possible to write a library of requirements that can be used for several projects, thus reducing the cost of writing new requirements in the following projects.

However, we shall note that the increased number of relations between requirements can be a drawback to the approach: without adapted tooling support, propagating a change to all the related requirements can be a time-consuming task.

3.8 Conclusion

The use of the SIRCOD approach, if it introduces an increase of work in the beginning (with the transformation from NL requirements to code and with the formalization of requirements), is quite similar to a classical Object-Oriented development process from the third stage and does not introduce an increase in the work effort required. Moreover, we address several of the properties that we defined in section 2.1.1:

(a) **Intended audience**: if requirements for the first time are expressed in a NL way, the approach in the actual form is more addressed to software developers.

(b) **Traceability support**: the traceability, from the source to the code, passing by the several refinement processes, is ensured by the approach.

(c) **Coverage**: NL requirements allow to express all kind of requirements. The **formal requirements features** only allow the user to express the functional parts of the system formally. However, it is still possible to write an empty formal requirement, linked to non-functional ones, to express the need to take them into account.

(d) **Semantic definition**: Using the semantic of the programming language, requirements are semantically defined once formalized. There is no semantic definition.

(e) **Verifiability**: Contracts allow to verify that the system is correct regarding the requirements.

SIRCOD thus allows the coverage of a large part of the properties that we aimed to address. There are, however, some issues that remain to address in the approach. First, the use case allows

us to see a big effort to be done in the first step of the approach. Translating from NL to Eiffel is time-consuming. This problem is one of the main issues addressed by NLP, and if we do not pretend to provide tools that support full automation of this process, we will propose in chapter 4 a way to ease this step. Then, if traceability from source code to requirements document is ensured, some links can exist between requirements (that can be found in approaches such as KAOS or SysML, for example – see section 2) can be used to provide a more fine analysis of requirements. We will address this issue in chapter 5. Finally, more work should be done on the dependency between a requirement and its implementation introduced by the inheritance mechanism, and especially on the impact this can have on the complexity of the code.

Chapter 4

RSML: a modeling language for requirements

"During the process of stepwise refinement, a notation which is natural to the problem in hand should be used as long as possible."

Niklaus Wirth

Contents

As mentioned in section 2, several tools are addressed to different stakeholders, depending on their background and the context. Thus, some stakeholders can prefer to work with textual approaches, while others can prefer to use mathematical-based approaches. For instance, an electrical engineer will not work in the same perspective than a lawyer, even if involved in the same project. As introduced in section 2.2, addressing these several points of view is, however, crucial. We cannot expect users to work with a given approach without any existing bridge to their existing familiar approaches.

Some approaches tried to propose a representation for the requirements and the links between requirements, like ReqIF [93]. However, there is no standard representation of requirements, and such approaches are not widely used.

To overcome this difficulty, we propose to use MDE and defining a metamodel based on the semantics that we defined in previous chapters. The advantages of such an approach are multiple: on

the one hand, it allows us to define a DSML based on this metamodel, that we named Requirements-Specific Modeling Language (RSML), more accessible to non-experts than Eiffel, and on the other hand, by fitting into the idea of the globalization of modeling languages [75], it can be used to bridge the gap to other approaches.

In this chapter, we first present in section 4.1 the metamodel defined to fit the semantics proposed in previous chapters. In a second time, in section 4.2, we describe RSML, the DSML associated with this metamodel. In section 4.4, we present how the existing MDE tools can be used to bridge the gap with the existing approaches. Then, in section 4.5, we apply the methodology to a use case, from [94], to validate our approach, to, finally, conclude in section 4.6. Objectives of such an approach are multiples: in addition to offering a simple tool to allow transformation to other paradigms, it allows us to propose a syntax close to NL. Such a syntax will be more understandable to all stakeholders, especially to non-experts, allowing for a better communication and feedback on the requirements.

4.1 A requirements meta-model

The RSML metamodel[1] (an excerpt is given in Fig. 4.1) is organized in two main parts:

(i) Domain Knowledge (cf. Fig. 4.2);

(ii) Requirements themselves, and the distinction between Natural Language requirements and formal ones (cf. Fig. 4.4).

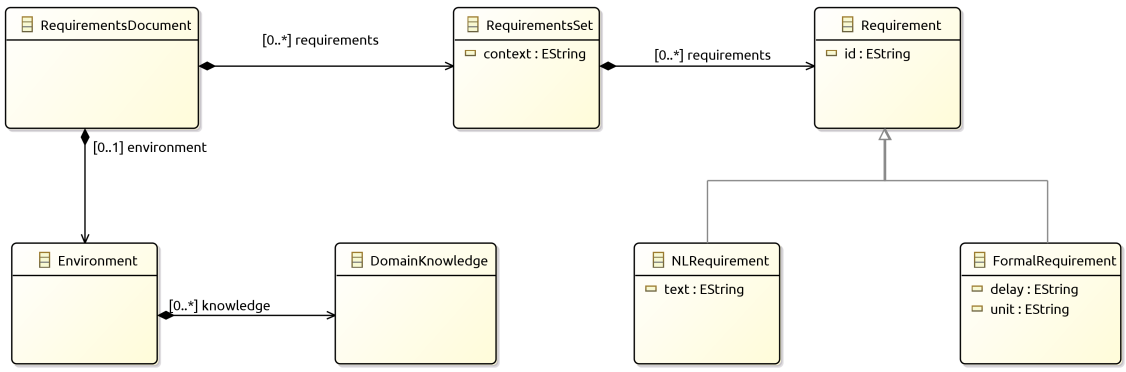


Figure 4.1: Principal parts of the RSML metamodel

Each of the metamodel elements is precisely semantically defined. Some of the notions of the metamodel are linked to the *Element* concept. This concept is used to define an abstract artifact, identified by a string, that can be an element of the system or the environment. In the following sections, we will define what they represent and to which semantics they are matched for each of these elements.

[1]The complete metamodel can be found in Appendix A.1

4.1.1 Domain Knowledge

In [11], Jackson and Zave made a distinction between the Environment and the System. The Environment is more precisely composed of two sets: the Requirements and the Domain Properties. If the first one expresses the needed properties for the System, the Domain Properties are properties that can impact the system and thus, shall be considered. These Domain Properties will thus be assumptions that shall be considered while writing the specifications of the system.

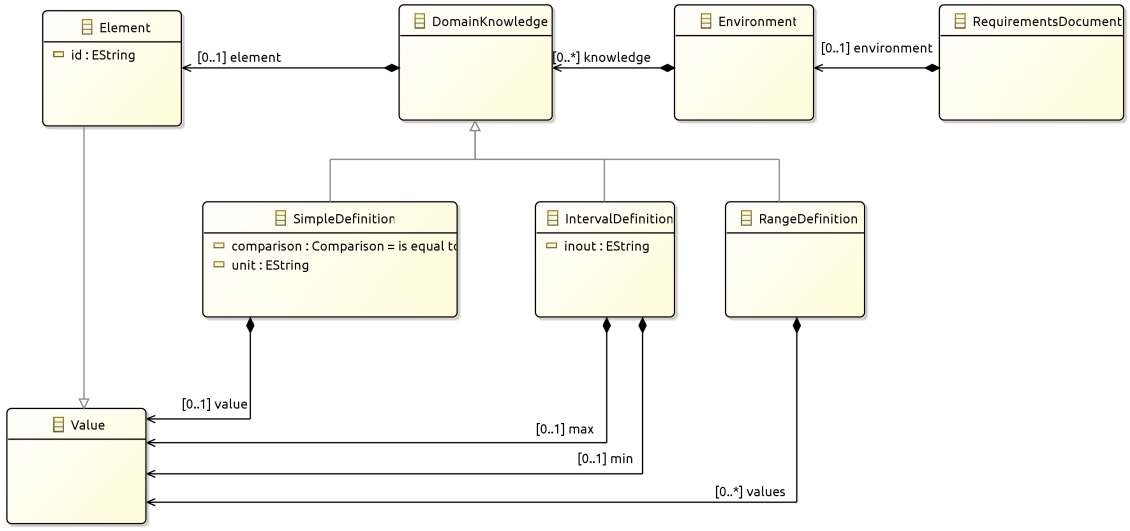


Figure 4.2: Domain Knowledge definition (i) of the RSML metamodel

The part (i) of the metamodel given in Fig. 4.2 is used to model these domain properties. It allows defining the value or the state of an *Element*, that can be later used during requirements writing or specification.

Domain knowledge is formalized as assumptions using the semantics that we defined in chapter 3. Such assumptions shall be considered for all requirements during analysis. It is thus possible to express them as preconditions required for all the features of the system. If a precondition is a required state for all features, it is more convenient to express it as an invariant. To allow the use of these assumptions in the whole system, we place it in a specific class inherited by all the other classes of the system.

For instance, a law fixing that *"the maximal authorized flight altitude for a drone is 150 meters"* can be expressed by instantiating this part of the metamodel (with *"the maximal authorized flight altitude for a drone"* an Element of the system) – like in Fig. 4.3. This example will be formalized by the invariant: `maximal_authorized_flight_altitude = 150`.

4.1.2 Requirements

Requirements themselves are expressed in part (ii) of the metamodel, given in Fig. 4.4. They are divided into two categories: natural language requirements and formal requirements. This

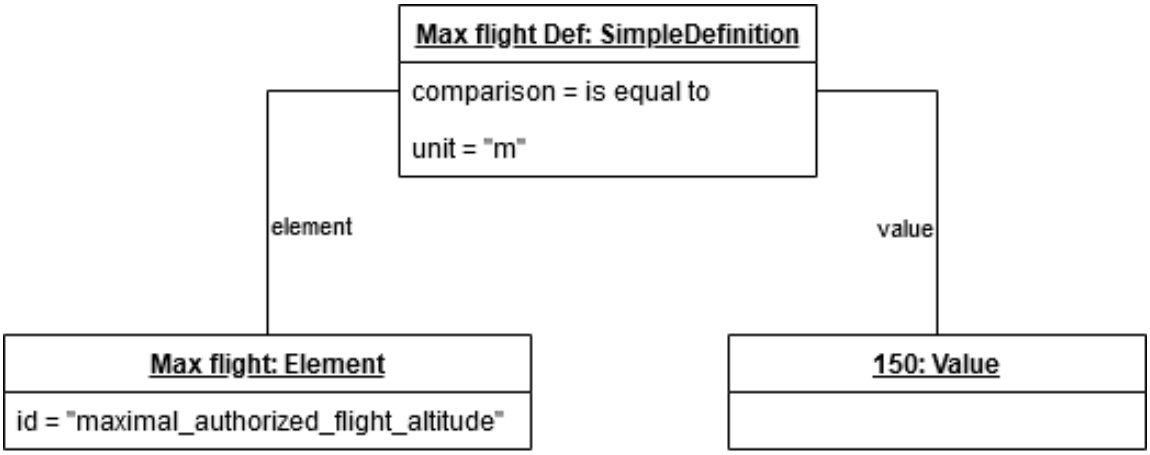


Figure 4.3: Example of an instantiation of part (i) of the RSML metamodel

distinction allows expressing formal requirements while keeping possible to express some of the requirements in a NL way (for those that cannot be formalized). Indeed, the classical distinction is made between functional and non-functional requirements. While the first ones can – more or less – easily be formalized, the formalization of non-functional ones is harder. As we already mentioned in chapter 3, it is hard to write as assumptions and assertions requirements like *"The system shall be efficient"*. This kind of requirement will most of the time be derived into others, easier to analyze (e.g., *"The consumption of the battery shall be less than 700 mAh"*). However, it is crucial to keep a trace of the original requirement, and even more crucial when no other requirements have .g., been derived from this one. To this purpose, in RSML, whether or not they have been formalized, all requirements have to be identified by a unique identifier to ease the traceability.

The NL requirements are similar to SysML requirements or KAOS goals: they have a free text attribute. They will be formalized as NL requirements in section 3: as features with no body nor contracts, linked to textual documentation. Based on SIRCOD, the user will have to consider them, even if they are not automatically formalized.

Formal requirements are used to express properties the system has to hold. As detailed in section 3, requirements are expressed as assertions to be verified in a given context. The context is expressed by *Constraint*s, which are the assumptions of the requirements (e.g., preconditions). The *Properties* are treating about *Element*s and are compared to *Value*s. A *Value* can be a primitive value (a number or a string), a state, or a reference to another *Element*. In the metamodel, two intermediate classes respectively named *ConstraintsDisjunction* and *PropertiesDisjunction* are used to represent the *logical or*. Indeed, by default, we assume that every *assertion* is to hold when all the *constraints* are satisfied. These artifacts (*ConstraintsDisjunction* and *PropertiesDisjunction*) are thus introduce to ease the composition using an *or* structure. This, however, imposes to express these constraints and assertions as conjunctive normal forms. It simplifies the bridge to the Eiffel semantics since it is the most natural way to express sets of contracts in Eiffel: a contract can be composed by a disjunction of predicates, while the conjunction of all contracts has to be satisfied.

Let us give an example. A formal requirement stating that *"In flight mode, the drone altitude*

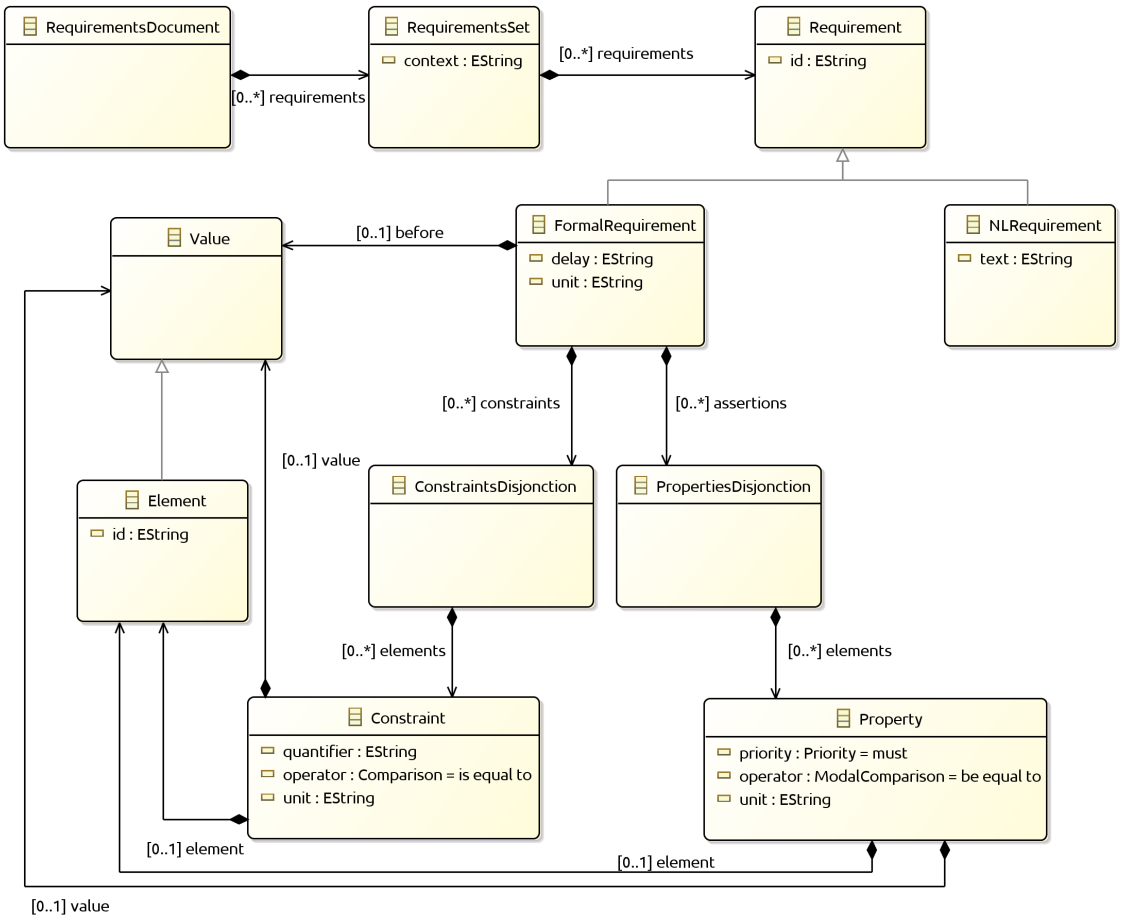


Figure 4.4: Requirements definition (ii) of the RSML metamodel

must be less or equal to the maximal authorized flight altitude." will be formalized by Fig. 4.5, semantically defined by:

- an assumption stating that the drone is in flight mode (the *Constraint*): `mode = flight_mode`;
- an assertion verifying that the altitude is less or equal to the previously defined maximum altitude (the *Property*): `altitude <= maximal_authorized_flight_altitude`.

The *Requirement*s are grouped in *RequirementsSet* that are used as *packages* of requirements to group them, given a *structural context*. Contrary to the context of *Formal Requirement*, this context gives information on how to organize requirements (allowing, for example, to group all requirements for a specific subsystem in a given context).

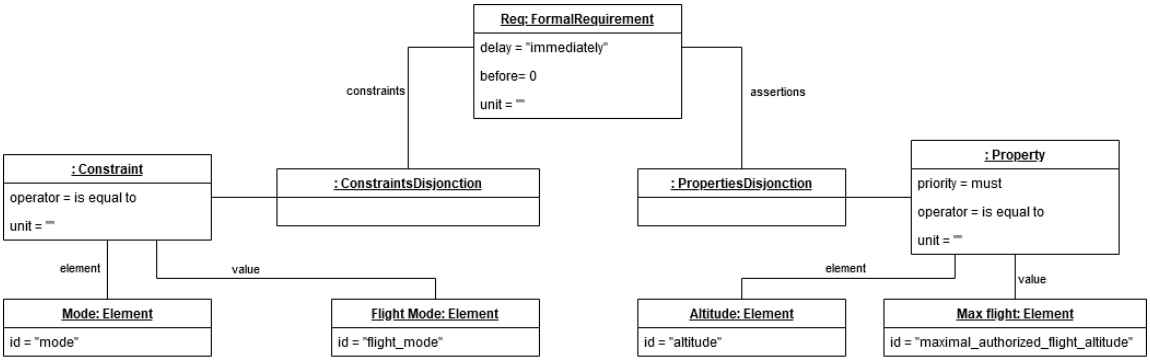


Figure 4.5: Example of an instantiation of part (ii) of the RSML metamodel

To summarize, requirements and domain knowledge can be expressed using the RSML metamodel. This shall help to give a complete representation of the Environment elements, according to the Jackson-Zave definition.

The semantics definition leads to a complete requirements validation environment:

- requirements themselves, if formal, can be used in the validation process using a prover or tests;
- using relationships and their semantics, the correctness of the relationship can be detected.

4.2 Requirements Specific Modeling Language (RSML): a constrained language for requirements

As previously said, the metamodel defined in section 4.1 supports the expression of domain knowledge, requirements, and relationships between requirements. The most natural way to write requirements still be, in most of the case, the textual writing. To ease the use of the metamodel, MDE give appropriate tools: the DSL and more specifically, DSML.

In this section we present RSML, a DSML close to NL[2]. Such a language can be used to allow non-specialists to express requirements in a similar way to Behavior-Driven Development (BDD) [95] tools such as Cucumber [96]. Contrary to these tools, RSML is semantically defined. All the artifacts described in this section match the semantics given in the previous chapter.

In Fig. 4.6, an RSML sample is given. This illustrates how RSML can be used to express domain knowledge, NL requirements, and formal requirements. **Requirements document** in RSML are, for their part, composed of two elements (that will be detailed below):

- the **environment**, that contains the domain knowledge;
- the **requirements set**, that contains requirements and their relationships.

[2]The language was developed in the GEMOC(`http://gemoc.org/studio.html`) environment [75],

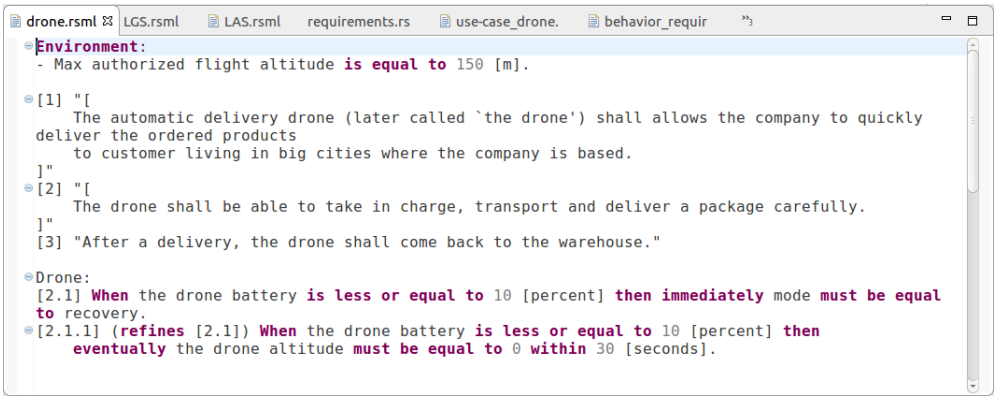


Figure 4.6: Example of the RSML DSL in practice

The root of the grammar of the language is thus:

⟨*requirements document*⟩ ::= ⟨*environment*⟩? ⟨*requirements set*⟩*

In the next section, we detail how these elements can be expressed, developing each of the parts as in the metamodel section.

4.2.1 Expressing domain knowledge

Elements of the domain knowledge are expressed in a list introduced by the *Environment* (or its shorten version *Env*) keyword, followed by a colon (*:*) and a newline. Each element is introduced by a dash (-). The example given in section 4.1 is thus expressed as "Max authorized flight altitude is equal to 150 [m].". This statement follows the grammar rule:

⟨*environment*⟩ ::= (‘`Env`’ | ‘`Environment`’) ‘`:`’ ⟨*newline*⟩ ⟨*domain knowledge*⟩+

⟨*domain knowledge*⟩ ::= ‘`-`’ (⟨*simple definition*⟩ | ⟨*interval definition*⟩ | ⟨*range definition*⟩) ‘`.`’ ⟨*newline*⟩

⟨*simple definition*⟩ ::= ⟨*element*⟩ ⟨*comparison operator*⟩ ⟨*value*⟩ (‘`[`’ ⟨*unit*⟩ ‘`]`’)?

⟨*interval definition*⟩ ::= ⟨*element*⟩ (‘`is in`’ | ‘`is out`’) ⟨*value*⟩ ‘`and`’ ⟨*value*⟩

⟨*range definition*⟩ ::= ⟨*element*⟩ ‘`can be`’ ⟨*value*⟩ (‘`,`’ ⟨*value*⟩)*

⟨*element*⟩ ::= ⟨*word*⟩+

⟨*comparison operator*⟩ ::= ‘`is`’ (‘`equal to`’)?
| ‘`less than`’
| ‘`less or equal to`’
| ‘`greater than`’
| ‘`greater or equal to`’
| ‘`different to`’

⟨*value*⟩ ::= ⟨*number*⟩ | ⟨*state*⟩ | ⟨*element*⟩

⟨*unit*⟩ ::= ⟨*word*⟩ (‘^’ ⟨*number*⟩)? (‘/’ ⟨*unit*⟩)?

The *Element*s are referenced using a noun phrase (excluding the reserved keywords of RSML). The articles are filtered to allow some variation of the formulation, and these *Element*s are case-insensitive.

The domain knowledge thus allows the users to define for a state:

- the relative value (equal, greater, less, different);
- an interval the value shall be in or out;
- a list of possible values.

These definitions can then be used in other parts of the RSML document.

4.2.2 Requirements in RSML

We will develop below how are expressed NL and formal requirements. As introduced in the meta-model, each requirement is uniquely identified to ease the traceability. This identifier is introduce in the grammar using square brackets ([and]):

⟨*requirement*⟩ ::= ‘[’ ⟨*id*⟩ ‘]’ (⟨*natural language requirement*⟩ | ⟨*formal requirement*⟩) ⟨*newline*⟩

⟨*id*⟩ ::= ⟨*digit*⟩ (‘.’ ⟨*id*⟩)?

As previously said in the introduction of this section, requirements are grouped in **requirements set**. These requirements sets are identified by a unique identifier that the user-defined (using a list of words). This shall ease the analysis of requirements, allowing to group them by semantics proximity (or by any other classification used by the user). The grammar for these sets are:

⟨*requirements set*⟩ ::= ⟨*word*⟩+ ‘:’ ⟨*newline*⟩ ⟨*requirement*⟩+

Natural Language Requirements

They are one of the two kinds of requirements of the metamodel. They are only composed, in addition to the identifier, by the text of the NL requirement itself. In RSML, we let the user the possibility to express NL requirements in a single line (surrounded by double-quotes), or in multi-line (surrounded by "[and]"). This is described in the following grammar:

⟨*natural language requirement*⟩ ::= ⟨*single line text*⟩ | ⟨*multiline text*⟩

⟨*single line text*⟩ ::= ‘"’ .* ‘"’

⟨*multiline text*⟩ ::= ‘"[’ (. | ⟨*newline*⟩)* ‘]"’

In Fig. 4.6, there are three examples of NL requirements: requirements 1, 2 and 3. Requirements 1 and 2 are multiline, while requirement 3 is a single line natural language requirement.

Formal Requirements

They are the other kind of requirements of the metamodel. They allow expressing constraints (the context of a requirement) and properties to be satisfied (the requirement itself). In RSML, we chose to use for them a syntax similar to cucumber:

- constraints are introduced by `when`,
- while properties are introduced by `then` (not mandatory if there are no constraints).

Moreover, each property has an attached priority, introduced by a modal (**must**, **should**, **could** or **would**).

In [85], a semantics is given to allow expressing temporal and timing properties in Eiffel. We thus let the possibility to add timing constraints in the requirements. When there is at least one constraint in the requirement, the user must specify if the requirement shall be met **immediately** or **eventually**. If the requirement is met **eventually**, the user can be precise in what interval the requirement shall be met with the keyword **within**, followed by the duration (number + time unit).

This is summarized in the following grammar:

⟨*formal requirement*⟩ ::= (⟨*properties disjunction*⟩ (‘`and`’ ⟨*properties disjunction*⟩)?) ‘`.`’
| ‘`When`’ ⟨*constraints disjunction*⟩ (‘`and`’ ⟨*constraints disjunction*⟩)? ‘`then`’ ‘`immediately`’ ⟨*properties disjunction*⟩ (‘`and`’ ⟨*properties disjunction*⟩)? ‘`.`’
| ‘`When`’ ⟨*constraints disjunction*⟩ (‘`and`’ ⟨*constraints disjunction*⟩)? ‘`then`’ ‘`eventually`’ ⟨*properties disjunction*⟩ (‘`and`’ ⟨*properties disjunction*⟩)? (‘`within`’ ⟨*integer*⟩ ‘`[`’ ⟨*Unit*⟩ ‘`]`’)?) ‘`.`’

⟨*constraints disjunction*⟩ ::= ⟨*constraint*⟩ (‘`or`’ ⟨*constraint*⟩)?

⟨*constraint*⟩ ::= ⟨*element*⟩ ⟨*comparison operator*⟩ ⟨*value*⟩ (‘`[`’ ⟨*unit*⟩ ‘`]`’)?

⟨*properties disjunction*⟩ ::= ⟨*property*⟩ (‘`or`’ ⟨*property*⟩)?

⟨*property*⟩ ::= ⟨*element*⟩ ⟨*priority*⟩ ⟨*modal comparison operator*⟩ ⟨*value*⟩ (‘`[`’ ⟨*unit*⟩ ‘`]`’)?

⟨*modal comparison operator*⟩ ::= ‘`be equal to`’
| ‘`be less than`’
| ‘`be greater than`’
| ‘`be less or equal to`’
| ‘`be greater or equal to`’
| ‘`be not equal to`’

4.3 A tool-supported language

The example introducing the previous section (Fig. 4.6) is a screenshot of the environment developed for the RSML language. Using the Gemoc studio, a complete Integrated Development Environment (IDE) dedicated to RSML has been realized[3]. This IDE proposes several functionalities that we will detail now.

[3]The tool is available at: `https://gitlab.com/fgalinier/RSML`.

First of all, the RSML editor itself gives a syntax highlighting to ease RSML documents' writing. An autocomplete menu is also available to propose to the user the best elements to complete an expression. In Fig. 4.7, for instance, the autocomplete menu proposes to the user the several comparison operators existing in RSML that can be inserted.

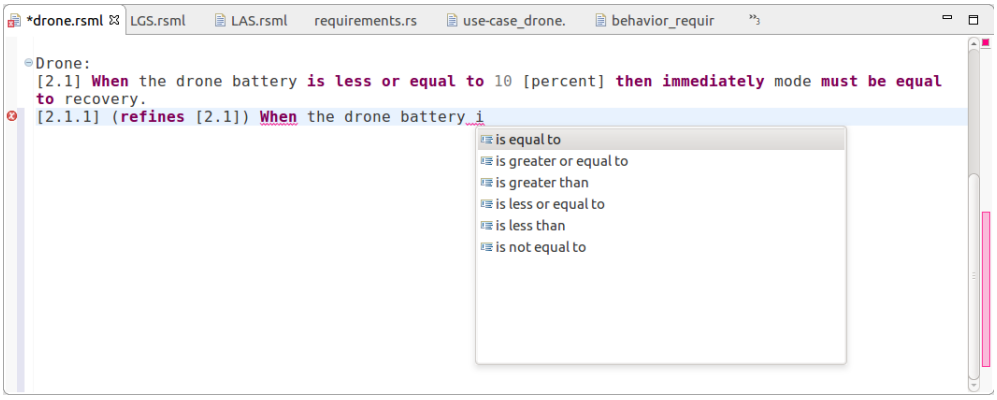


Figure 4.7: Autocomplete feature in the RSML editor

The editor also raises errors that can appear: grammar errors (in Fig. 4.7, since the formal requirement is not complete, a syntax error is raised), or errors when duplicating an identifier for example (see on Fig. 4.8, an example of such an error).

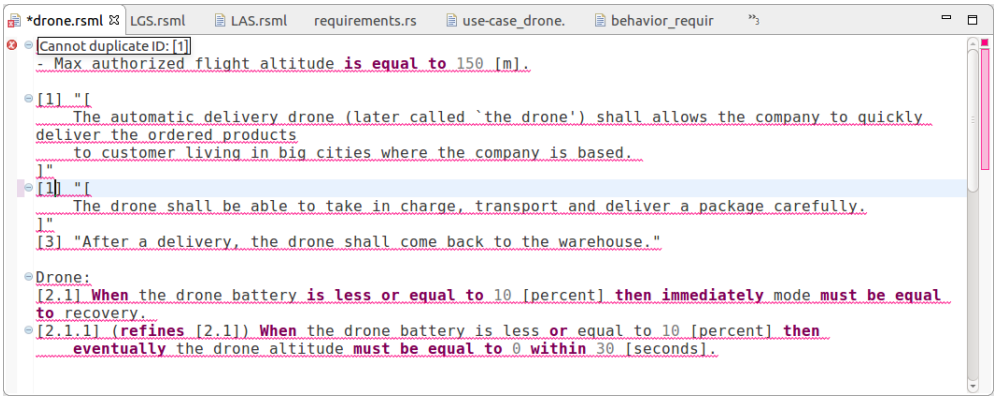


Figure 4.8: Error of duplicated identifier in the RSML editor

In addition to these editor's facilities, the environment allows the automatic transformation of the requirements expressed in RSML to the semantics of the requirements (in Eiffel), that is accessible in the *src-gen* directory. Other transformations can be possible (as detailed in section 4.4).

For example, the requirement 2.1 in Fig. 4.6 is formalized as given in List. 4.1.

```
when_drone_battery_is_less_or_equal_to_10_then_immediately_
mode_must_be_recovery
note
```

```
        EIS: "src=drone.rsml", "ref=[2.1]", "type=trace"
        Description: "[
                [2.1] When the drone battery is less or equal to 10
                [percent] then immediately mode must be equal to
                recovery.
        ]"
require
        when_drone_battery_is_less_or_equal_to_10_percent:
            drone_battery <= 10
deferred
ensure
        check_mode_must_be_equal_to_recovery: mode = recovery
end
```

Listing 4.1: Eiffel semantics of requirement 2.1

We can find in this formalization several elements introduced in the previous chapters: the EIS link to the source (here, the RSML document), the required constraint of the requirement (the precondition), and the property to ensure (the postcondition). A class that will contain a requirement in an Eiffel project will be based on the requirements sets. A class will thus model each set of requirements.

In List. 4.2, the set of requirements called "*drone*" is transformed in class `DRONE_REQUIREMENTS` – the `_REQUIREMENTS` suffix is systematically added to ease the distinction of classes of requirements. Moreover, all the classes generated by the editor inherit from the `DOMAIN_KNOWLEDGE` class, which contains the environment definitions.

```
class DRONE_REQUIREMENTS
inherit DOMAIN_KNOWLEDGE
```

Listing 4.2: Eiffel context for requirements 2.1 and 2.1.1

4.4 A modeling language to link other formalisms

Using a modeling language provides a way to transform from a model to another easily [97]. This can be done using dedicated languages, like ATL [98] or Kermeta [99], or even programmatically, using EMF[4]. Example transformation from this section has been done using Xtend[5] language with the EMF API. In the context of requirements, as introduced in section 1.1, this can help provide a bridge to different existing approaches, often used by the several stakeholders implied in a project.

In the context of RSML, we propose several transformations, from and to other languages, to ease the adoption of such an approach. Since the metamodel is built using EMF, it is possible to use the several approaches presented before (ATL, Kermeta, etc.) to realize these transformations.

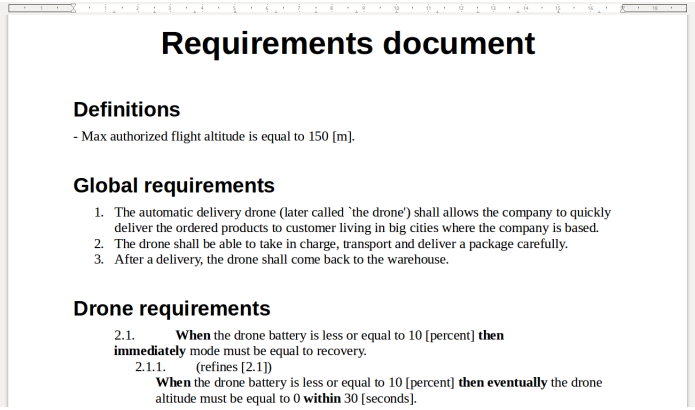
4.4.1 From RSML to textual representation

Despite the several approaches proposed to express requirements, the most used ones remain NL textual representations, and more specifically, Microsoft Word and Excel files. This can be explained

[4] `http://www.eclipse.org/modeling/emf/`
[5] `http://www.eclipse.org/xtend/`

by the apparent simplicity of such notations that can be handle by all stakeholders, or even for contractual reasons.

As mentioned in the introduction to this section, several tools allow transformations from a model to another. We present here some transformations from RSML to docx and excel files. Most of the representations used in these formats are specific to companies, so we propose possible representations for these formats. These representations can easily be adapted with some slight change in the transformations rules.



Requirements document

Definitions

- Max authorized flight altitude is equal to 150 [m].

Global requirements

1. The automatic delivery drone (later called `the drone') shall allows the company to quickly deliver the ordered products to customer living in big cities where the company is based.
2. The drone shall be able to take in charge, transport and deliver a package carefully.
3. After a delivery, the drone shall come back to the warehouse.

Drone requirements

2.1. **When** the drone battery is less or equal to 10 [percent] **then immediately** mode must be equal to recovery.

2.1.1. (refines [2.1])

When the drone battery is less or equal to 10 [percent] **then eventually** the drone altitude must be equal to 0 **within** 30 [seconds].

Figure 4.9: Example of docx document generated from the RSML example in Fig. 4.6

In the representation given in Fig. 4.9, RSML artifacts are matched to node of the docx document.

Both requirements (NL ones and formal ones) are written in their textual representations, each in a different paragraph introduced by their identifier. For example, in List. 4.3, a NL requirement is transformed into a paragraph in the docx document, inserting a bookmark to be able to link the requirement in the SIRCOD approach. The different requirements set are introduced by a section title, named by the identifier of the set. The domain knowledge is introduced in the first section, called *Definitions*, that lists all the environment definitions.

```
public void compile(final NLRequirement r) {
  Paragraph p1 = new Paragraph();
  // ...
  BookmarkStart bookmarkStart = new BookmarkStart();
  bookmarkStart.setName(bookmarkName);
  bookmarkStart.add(bookmarkText);
  BookmarkEnd bookmarkEnd = new BookmarkEnd();
  bookmarkEnd.setName(bookmarkName);
  p1.add(bookmarkStart);
  p1.add(bookmarkEnd);
  // ...
  p1.add(requirementId + " - ");
  if (r.getRelations().size() > 0) {
    p1.add("(");
```

```
        boolean first = true;
        EList<Trace> _relations = r.getRelations();
        for (final Trace rel : _relations) {
          {
            if ((!first)) {
              p1.add(" and ");
            } else {
              first = false;
            }
            this.compile(rel, p1);
          }
        }
        p1.add(") ");
      }
      // ...
      p1.add(requirementText);
      p1.setStyle("gtext");
      this.doc.getBody().add(p1);
    }
```

Listing 4.3: Excerpt of the xtend function that transform NLRequirement to a docx paragraph

	A	B	C	D	E	F	G	H	I	J	K
1	#	Context	Requirement description	Priority	Trace to	Addition to	Alternative to	Contained by	Refines	Constraints	Contradicts
2	1	Global	The automatic delivery drone (later called `the drone') shall allows the company to quickly deliver the ordered products to customer living in big cities where the company is based.								
3	2	Global	The drone shall be able to take in charge, transport and deliver a package carefully.								
4	3	Global	After a delivery, the drone shall come back to the warehouse.								
5	2.1	Drone	**When** the drone battery is less or equal to 10 [percent] **then immediately** mode must be equal to recovery.	MUST							
6	2.1.1	Drone	**When** the drone battery is less or equal to 10 [percent] **then eventually** the drone altitude must be equal to 0 **within** 30 [seconds].	MUST					2.1		

Figure 4.10: Example of MS-Excel document generated from the RSML example in Fig. 4.6

In the same way, in Fig. 4.10, requirements from RSML are presented in a spreadsheet. This representation emphasizes the relationships, listing all of them and their targets of the relationships in different cells. Moreover, the requirements' priority is also printed (using the priority of the highest priority property if a requirement owns several properties).

These two transformations are example of how the use of RSML can be matched to standard practices. Using RSML to write first requirements can thus be used to ensure a clear semantics for these requirements – and this semantics can be used in the following stages of projects, as introduced in section 3 – while allowing non-experts stakeholders to analyze and works with requirements in tools they used to work with.

4.4.2 Transforming RSML to (and from) other languages

Textual representations are certainly the most used support for requirements. However, some other tools presented in section 2.1 provides facilities in requirements analysis or even in design using requirements. It can thus be useful to provide bridges to (and from) these tools. For example, ones

can want to elicit requirements using a GORE approach, such as KAOS, and then switch to RSML to analyze the elicited requirements. Similarly, system engineers that are used to design systems with SysML can need to switch from an RSML representation to SysML.

We present in this section an example of transformation from RSML to a target modeling language: SysML. However, in the same way, several companies can use several standards for their requirements documents, the semantics of SysML is not the same depending on tools or even on the project.The proposed transformation is thus an example and shall be adapted to the team's current habits by a modeling expert that can change the transformations rules.

It is thus possible to write transformations from Fig. 4.6 to the SysML requirements diagram given in Fig. 4.11. In this case, each requirement is transformed into a requirement artifact in

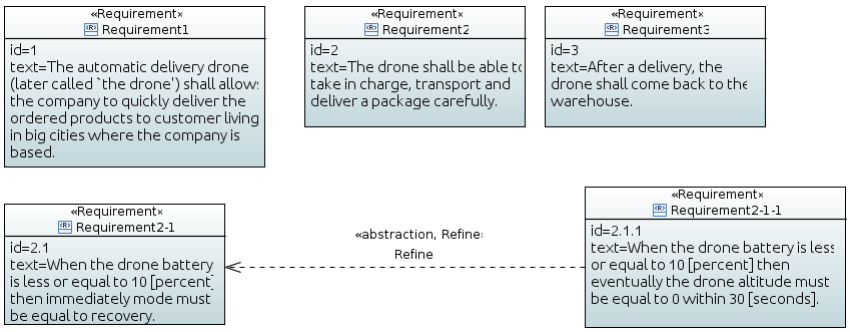


Figure 4.11: SysML representation of requirements from example in Fig. 4.6

SysML, composed by its identifier and its textual representation – no matter if it is a NL requirement or a formal one. In List. 4.4, the xtend function that allow transforming a NL requirement to a SysML requirement is given.

```
public void compile(final NLRequirement r) {
  // ...
  String reqId = RSMLRequirementId;
  String className = ("req_" + reqId);
  org.eclipse.uml2.uml.Class cl = this.model.createOwnedClass(
      className, false);
  EObject applyStereotype = UMLUtil.StereotypeApplicationHelper.
      getInstance(null).applyStereotype(cl, RequirementsPackage.
      eINSTANCE.getRequirement());
  org.eclipse.papyrus.sysml14.requirements.Requirement req = ((org.
      eclipse.papyrus.sysml14.requirements.Requirement)
      applyStereotype);
  req.setId(RSMLRequirementId);
  req.setText(RSMLRequirementIdText);
  this.model.getPackagedElements().add(cl);
  this.resource.getContents().add(req);
}
```

Listing 4.4: Excerpt of the xtend function that transform NLRequirement to a SysML requirement

Nevertheless, formal requirements can be transformed into more formal diagrams, such as state diagrams. Constraints can be seen as guards to reach a state, while the state is the property to satisfy – as proposed in [57], described in section 2.1.4.

An other advantage of a model representation is to allow transformation not only *from* RSML, but also **to** RSML. The level of details of requirements that we can obtain depends on the source's level of details. For example, if we take the requirements diagram of SysML, requirements are only textual. Consequently, we can only generate NL requirements. Relationships, however, are preserved by the transformation – assuming that the semantics of relationships in RSML have been aligned with those of SysML. Given the previous example in Fig. 4.11, we can obtain the RSML model in Fig. 4.12.

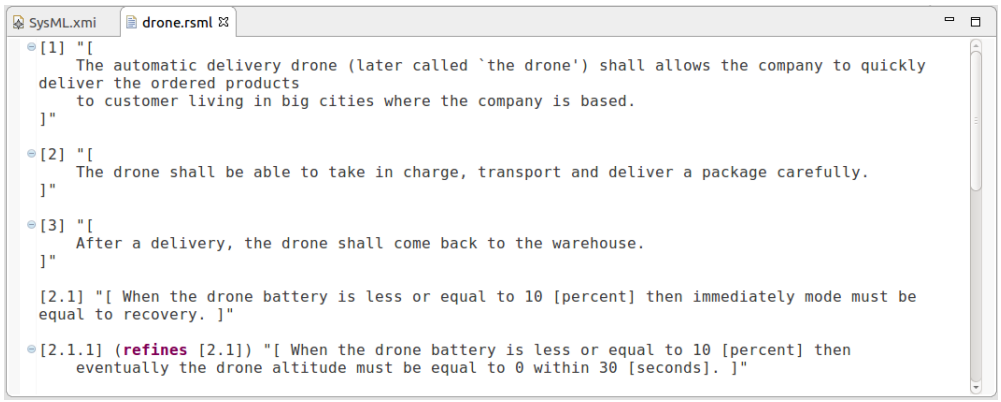


Figure 4.12: Resulting RSML of transformation from SysML of Fig. 4.11

Since requirements are not split into clusters or sets in SysML, the decomposition does not appear in the resulting model. Moreover, domain knowledge elements are not part of the requirements diagram of SysML and, thus, do not appear in the resulting model either. SysML requirements *id* and *text* have been matched to the corresponding field in RSML.

Such requirements are all translated in NL requirements; indeed requirements in SysML requirements diagram are all textual. NLP approaches can be seen as a possible improvement of such transformation; the text analysis can help transform textual requirements to formal ones, at least for those of the form *"when this then do that"*. This is, however, out of the scope of the current thesis.

Note that it is possible to easily transform requirements 2.1 and 2.1.1 from the example in formal requirements, by removing the quotation marks, since they respect the grammar of such requirements. A possible improvement should thus be to provide tools that allow modeling SysML requirements diagram, with a constrained editor for the requirements' text attributes.

There are several advantages to use a modeling language that allows its models to be transformed into other languages: first, as already mentioned, it allows users to work with their usual tools, even with requirements first expressed in RSML. Then, transforming from other languages to RSML can lead to work with the more adapted tool for specific parts – it is more convenient, for instance, to create relationships between requirements and use a graphical notation SysML or even KAOS. The transformation to RSML can then help verify the validity and the consistency of such links, using OCL rules or the semantics presented in the previous chapter. Finally, using a modeling language

can help integrate such an approach in more global use of modeling languages, easing the bridges to other approaches.

4.5 Use case: the London Ambulance Service system

We applied RSML on the already cited use cases. However, the two previous ones were embedded systems examples, and we want to confirm that the approach is domain-independent by applying it to an information system. In his thesis [94], E. Letier applied a GORE formal approach on the London Ambulance Service (LAS) system. This system shall manage ambulances and dispatch them to an emergency that is encoded in the system.

The first goal of this system, as defined by E. Letier is:

> *«For every urgent call reporting an incident, there should be an ambulance at the scene of the incident within 14 minutes.»*

From this goal, one higher level goal is deduced, which is:

> *«Every incident requiring emergency service is eventually resolved.»*

Starting from this new high goal, two other sub-goals are refined:

> *«Every incident requiring emergency service is eventually reported to the LAS.»*

> *«An incident is resolved by the intervention of a single ambulance.»*

The satisfaction of the goal *"Every incident requiring emergency service is eventually resolved."* is thus satisfied by the realization of the *report*, *ambulance mobilization* and *intervention* goal.

The aim is not to develop a complete and precise system, but to start from these goals and refine them into a framework that could be used to develop the system.

The first step is thus to write these goals in RSML, that is done in List. 4.5.

```
[1] "Every incident requiring emergency service is eventually resolved
    ."
[1.1] (parts of [1]) "Every incident requiring emergency service is
    eventually reported to the LAS."
```

Listing 4.5: First RSML requirements of the LAS

At first, to validate that this approach is domain-independent, we applied it to different well-known use cases such as the Landing Gear System (LGS) [92] and the London Ambulance Service (LAS) system [94]. These RSML case studies are publicly available[6]. In both cases, we noted that the available editor of RSML leads users to express requirements easily. Non-specialist users wrote the RSML version of requirements that were given in NL.

The formalizations that have been deduced were checked with AutoProof. It was used to control the specification of the system, and leads us to detect validation and verification errors:

[6] `https://gitlab.com/fgalinier/rsml-examples`.

- In the LAS, the issue revealed by AutoProof pointed out that a requirement was misunderstood and had to be corrected to obtain a set of valid requirements [100]: the formalization of the requirement `"An ambulance shall not be available when it is allocated"` was interpreted as `"After being allocated, an ambulance shall not be available"`. This formalization proved to be such that it was impossible to write a correct specification. The requirement rather means that the ambulance cannot be at the same time `available` and `allocated`. Once the correction is made, the system specification became correct regarding the new requirements[7]. This kind of validation still requires human expertise to understand why the requirement cannot be fulfilled, but the formal representation points to the requirement that needs to be further analyzed. Therefore, this case study allows us to **validate** the requirements of the system.

- In the LGS case study, the use of Autoreq led us to detect errors in an existing Abstract State Machine specification of the system [85]. Thus, the use of RSML semantics and AutoProof permit the **verification** of the system itself, regarding the requirements.

Secondly, we conducted a study to validate the usability of RSML itself. We worked with 11 users. Most of them were industrial users from a company like Airbus, Thales, Capgemini, or Orange (other practitioners from small startups). They usually express requirements in NL languages or modeling languages such as SysML. We asked them to express requirements in RSML, and we collected their feedback. Requirements were about a drone system and were described in NL in a document[8]. We provided the users with some explanations of the tool support and the syntax of RSML[9]. The feedback and the users' backgrounds were collected in a form. Among the testers:

- 81% were industrial engineers, the others 19% were academics (seniors or Ph.D. students);

- 54% were used to read requirements (every week), 46% less often;

- 18% were used to write requirements frequently, 63% once a month, 19% never;

- All agree on the fact that poorly formulated requirements have slowed down their work (63% sometimes, 27% often, 10% rarely).

- Only 27% answered that a misunderstanding of requirements did not (or rarely) led to quality problems in their work (for 18% it is often, 54% sometimes).

- Only 28% used requirements specific approaches (such as Gherkin, User Stories, ...).

- Finally, less than 10% used dedicated tools for requirements.

About the *usability* of RSML and its tool support, we asked for feedback on ease of use, quality of requirements obtained, prospects for adopting RSML. The grade range was from 0 (totally disagree) to 5 (totally agree).

The *ease of write RSML requirements* got a majority of 3s, while the *ease of read RSML requirements* got a majority of 4s. A majority of users stated that it is easier to write and read RSML requirements than formal ones. They generally admitted that they had learned to use RSML fairly

[7]The system specification is available here: `https://gitlab.com/fgalinier/LAS`.
[8]Cf. `https://gitlab.com/fgalinier/RSML/blob/master/use-case/ADDE.md`.
[9]Cf. `https://gitlab.com/fgalinier/RSML/blob/master/README.md`.

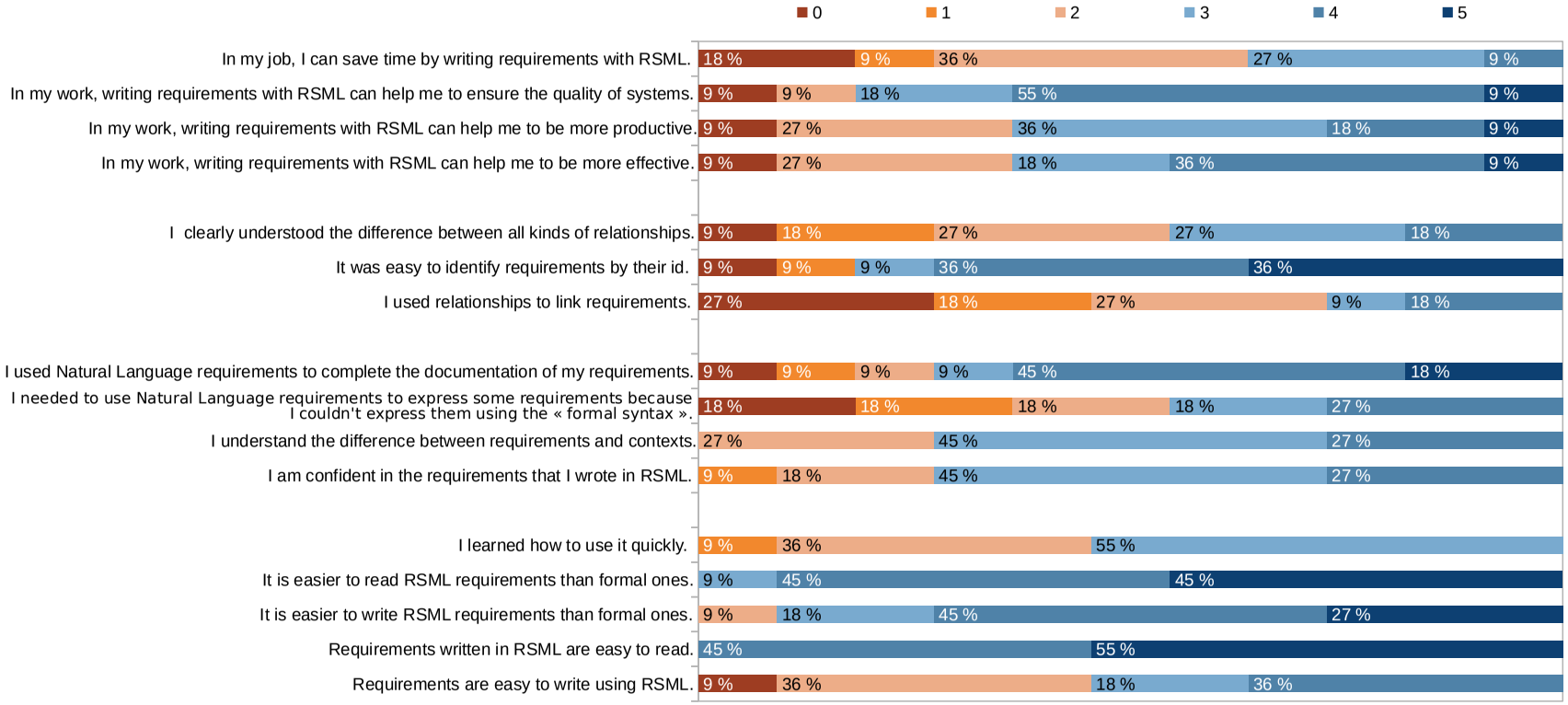


Figure 4.13: Answers to the survey about RSML

quickly. They generally admitted that they were quite confident in the quality of the requirements they wrote. A large majority understood the difference between requirements and contexts. At the question: "In my work, writing requirements with RSML can help me to be more effective" they answered 3 at 80%. At the "In my work, writing requirements with RSML can help me to be more productive" one, the average answer was 3. They found at 60% that "In my work, writing requirements with RSML can help me ensure the quality of systems."

About the *quality* of the RSML description of the drone system sent by the testers, we can say that compared to a description we made[10], the results were satisfying at 80%.

Finally, the results obtained are encouraging as users accustomed to working with requirements found the approach satisfactory and occasional users of requirements or accustomed to working in natural language (1) do not have any major difficulty in using the approach, and (2) thought it provides them an interesting level of formalization. We now intend to apply our approach to an industrial case study, the AIDA system, recently made available as open-source at the IRT Saint Exupéry's public forge[11].

Then, once our ongoing work on requirements relationships formalization is completed, we intend to validate the usefulness of translating such relationships into other approaches relationships. We want to use an existing SysML requirements diagram to translate it in RSML and proceed to its analysis to check it.

4.6 Conclusion

The approach proposed in this chapter leads to the easier use of formal representation of requirements. We propose RSML, a requirements dedicated language that is close to NL while supporting formal semantics. RSML is based on a metamodel of requirements allowing their expression in both NL and formal language. Since RSML is semantically defined on SIRCOD, RSML expressions can be used and analyzed, as introduced in chapter 3. Therefore, non-experts can benefit from the inherent formality of RSML.

Using the techniques and tools of MBSE, we also propose some bridges to existing languages such as SysML or MSWord/MSExcel, to allow engineers to work with their usual tools. We treat several requirements engineering issues, providing a way to transform usual requirements expressions both to and from RSML. RSML can then be seen as a pivot between various expression spaces. It is also a pivot between formal and not formal fields. RSML aims to integrate into a seamless approach, reduce the gap between NL requirements and formal representations, and provide ways to address several kinds of stakeholders (being modeling tools users, NL users, or even more formal approaches users). We are adding new transformations to other approaches, like KAOS, to confirm our approach's scalability in a more global context.

The first experiments gave us good feedback. Using RSML on case studies showed to be useful for several reasons:

1. It is easily usable by stakeholders used to working with NL.
2. It allows obtaining a formal expression for requirements and their validation through a prover.
3. It allows systems verification regarding their requirements.

[10] Cf. `https://gitlab.com/fgalinier/RSML/blob/master/use-case/drone.rsml`
[11] Cf. `http://www.irt-saintexupery.com/wp-content/uploads/2019/02/Annual_Report_2018_Web.pdf`.

This approach and its tool support will be used by more people, with various RE and industrial case studies. This would validate the approach scalability in terms of: number and confidentiality of stakeholders, number and types of requirements, and number and types of relationships between them.

Chapter 5

A semantics for requirements relationships

"Incorrect documentation is often worse than no documentation."

Bertrand Meyer

Contents

Relationships between requirements and other artifacts (including other requirements) are quite common in RE. Formal methods like Event-B emphasize the need to refine high-level requirements to lower-level requirements, more complex but more precise and closer to the solution. GORE approaches are also based on the relationships between these artifacts, and the most widely used tools in the industry are tools providing traceability [83].

In a classical development process, it is necessary to distinguish traceability and other relationships. While traceability is used between artifacts from several development steps (e.g., between a requirement and a piece of code), relationships are addressed to artifacts from the same step. Most of the research effort has been made in providing tools and methodology for traceability, with some automatic tools ([101] or [102]), but most of them still require human expertise to check traceability [103]. Moreover, there is a real need to clearly defined traceability. In [104], the authors reviewed several works on traceability and advocate to provide a standard metamodel for traceability. Indeed, it is all the more critical that in a classical context, it is important to be able to keep the traceability information when transforming one model into another (e.g., design to code). A certain amount of work has thus been done to try to automate and keep a coherent traceability between models. To do this, it is necessary to define a common definition for the traceability relations

[105, 106], i.e. to classify these relations.

If in the context of a seamless approach the problem of transformation does not arise, the classification of the internal relations to the requirements is important, even if it can be simplified (thus, the "satisfy" relation of SysML does not apply within the framework of the approach SIRCOD, an element of the system satisfying a requirement being a refinement). However, since we advocate an approach that considers all elements as parts of a single model (within the constraints of a real project), we need to consider all the internal relationships that exist between the artifacts of this single model that need to be analyzed.

As already said in the introduction of this section, several approaches have defined a set of requirements relationships, and the objective of this thesis is not to claim that we invent the relationships. Rather, we are trying to provide a more generic classification of these relationships, based on the existing, and provide some semantics to use them as a base tool for engineers.

In this chapter, we will first present in section 5.1 the classification that we propose, to give then in section 5.2 a formal definition of these relationships. In section 5.3, we present how these relationships can be used to ease the development of a system. In section 5.4, we apply the presented relationships to use cases to analyze the impact of such an approach, and in section 5.5, we will conclude on the benefits and the remaining tasks on this work.

5.1 Existing relationships related to requirements artifacts

As already mentioned, the more common language for requirements is the NL. It is thus important to put effort into the analysis of this kind of requirement. This is the work done in [8].

To try to provide a complete set of requirements relationships possible, we analyzed NL documents. During this analysis, we defined several categories for requirements and relationships between requirements. We presented this work several times during workshops to validate this work, allowing us to change the classification to the current list incrementally.

The goal here is not to criticize the quality of the requirements document. Such criticism can be interesting; indeed, if some objectives arguments can be used to judge a requirements document, they can be used to improve this document. However, it is necessary to analyze existing requirements documents without any prescription to state how they are. Based on this analysis, the second step of the work will propose some tools or methodology on how to use such a classification to improve existing or new requirements documents.

We finally defined ten categories for requirements (and 12 derived categories). Note that we consider each part of a requirement document as a requirement in the context of this work. We will not develop here this classification (the purpose of this section is mainly to present relationships), but here is a quick overview of the categories (for a more detailed version, please refer to [8]):

1. **Component**: a property that describes a part of the system, the environment or the project;
2. **Goal**: an objective of the project or the system;
3. **Behavior**: a property of an operation of a component (or the whole system);
4. **Task**: an activity of the project;
5. **Product**: an artifact created by a task of the project;

6. **Constraint**: a property of the environment that affect a part of the system (of or its behavior) or of the project;
7. **Role**: a responsibility for a task or a behavior;
8. **Limit**: a property that the system, the project or the environment does not include;
9. **Lack**: a property not defined in the requirements document, but that should be;
10. **Metarequirement**: a property of a requirement.

Using these categories, we defined several relationships that can exist between requirements. In a similar way to categories for requirements, we have incrementally obtained nine relationships, listed here:

1. **Disjoins**: two requirements R_1 and R_2 are disjoint iff they are unrelated;
2. **Belongs**: a requirement R_1 belongs another requirement R_2 iff R_1 is a subrequirement of R_2;
3. **Repeats**: a requirement R_1 repeats a requirement R_2 iff R_1 specifies the same property as R_2;
4. **Contradicts**: a requirement R_1 contradicts a requirement R_2 iff the properties specified by the requirements are not compatible;
5. **Follows**: a requirement R_1 follows a requirement R_2 iff R_1 is a consequence of R_2;
6. **Extends**: a requirement R_1 extends a requirement R_2 iff R_1 assumes R_2 and specifies a new property;
7. **Excepts**: a requirement R_1 excepts a requirement R_2 iff R_1 modify a specific property of R_2 in a specific context;
8. **Constrains**: a requirement R_1 constrains a requirement R_2 iff R_1 specifies a constraint on a property specified in R_2;
9. **Characterizes**: a requirement R_1 characterizes a requirement R_2 iff R_1 is a metarequirement of R_2.

As for requirements categories, several derived relationships were added. The **details** relationships is thus a derived case of the **extends** relations; in this case, the requirement that *extends* the other requirement only add detail in a specific property and does not add a new property.

The other derived relationships come from the question of the desirability of the **repeat**. As detailed in the introduction, the objective is not to be prescriptive, and so the presence (or not) of a **repeat** is not analyzed as such. However, to ease future prescriptive work, and since this is one of the major problem addressed to us during our presentations, we detailed the **repeat** into three other relations:

- the **shares**, when two requirements own a common sub-requirement (or two sub-requirements that are repeating);
- the **explains** when two requirements are repeating but are not of the same type (e.g., text and image or text and UML diagram);

- the **duplicates** when two requirements are repeating and are of the same type.

To complete this analysis, we also analyzed how formal requirements can be related to each other.

5.2 Formal definitions of these relationships

In section 3.4, we introduced a formal definition of requirements themselves. We also define the requirements' satisfaction since it is the property we want to trace to ensure systems compliance. These definitions are used in this section to define relationships between requirements formally. As a reminder, we defined a requirement as a pair of one set of constraints C_R and one set of properties P_R with $C_R \vdash P_R$. We thus obtained the equivalence:

$$sat(R) \equiv hold(\mathcal{C}_R) \rightarrow hold(\mathcal{P}_R)$$

We give here two new definitions that we will use in the next sections to complete this definition.

Definition 1

Two requirements R_1 and R_2 share a same context iff $\mathcal{C}_{R_1} = \mathcal{C}_{R_2}$.

Definition 2

An atomic requirement R is a requirement where $|\mathcal{P}_R| = 1$ (a requirement with a unique property).

Using these definitions, we can define relationships between requirements.

5.2.1 What are the relationships between requirements?

To construct the set of relationships between requirements, we considered two requirements R_1 and R_2, their constraints ($\mathcal{C}_{R_1}$ and $\mathcal{C}_{R_2}$) and their properties ($\mathcal{P}_{R_1}$ and $\mathcal{P}_{R_2}$). We analyzed what could be the relationships between constraints and between properties of both requirements. The matrix given in Tab. 5.1 shows the deduced relationships.

Table 5.1: Matrix of relationships between constraints and between properties of two requirements R_1 and R_2

	$\mathcal{P}_{R_1} = \mathcal{P}_{R_2}$	$\mathcal{P}_{R_1} \subseteq \mathcal{P}_{R_2}$	$\mathcal{P}_{R_2} \subseteq \mathcal{P}_{R_1}$	$\mathcal{P}_{R_1} \cap \mathcal{P}_{R_2} = \emptyset$
$\mathcal{C}_{R_1} = \mathcal{C}_{R_2}$	R_1 repeat R_2	R_2 is an addition to R_1	R_1 is an addition to R_2	R_1 and R_2 can contradict [1]
$\mathcal{C}_{R_1} \subseteq \mathcal{C}_{R_2}$	R_2 constrains R_1	R_1 is a part of R_2	R_1 refines R_2	R_1 and R_2 can contradict [1]
$\mathcal{C}_{R_2} \subseteq \mathcal{C}_{R_1}$	R_2 refines R_1	R_2 refines R_1	R_2 is a part of R_1	R_1 and R_2 can contradict [1]
$\mathcal{C}_{R_1} \cap \mathcal{C}_{R_2} = \emptyset$	/	/	/	/

We identified six one-to-one relationships:

[1] If a property P_i of R_1 can be expressed as $\neg P_j$, with P_j a property of R_2, R_1 contradicts R_2 (more details in section 5.2.2).

- *repetition*
- *addition*
- *contradiction*
- *constraint*
- *partition*
- *refinement*

Before entering into their details, let us explain why we limited ourselves to those six relationships:

(i) From our point of view, if two requirements apply in two different contexts, we cannot say anything formal about the relationships between these requirements. That is why the last line of Tab. 5.1 (where $\mathcal{C}_{R_1} \cap \mathcal{C}_{R_2} = \emptyset$) does not let appear any relations. However, we must note that traceability links could link such requirements. In SysML, for example one could use *rationale* to express *traces* (for example reading a requirement help to understand another one). Since we cannot attach a formal definition to this *trace* relationship, we do not treat it in this thesis' scope.

(ii) To be complete about traceability, we also have to talk about relationships between requirements and other development artifacts. Actually, some approaches such as KAOS or i* [32] link requirements to stakeholders (that will be in charge of these requirements), or even to part of the system that should satisfy a requirement (*operationalization* in KAOS or *satisfy* in SysML [36]). SysML moreover, can be used to express that a requirement is *verified* by a test case. Both last relationships should be taken into account for validation. The semantics of such links can be expressed as:

- if an predicate V of the project *verifies* a requirement R, it will have to check the validity of $\mathcal{P}_R$ ($hold(\mathcal{P}_R) \equiv \top$), assuming the constraints are hold ($hold(\mathcal{C}_R) \equiv \top$). To rephrase, $sat(R)$ will be equal to the boolean value of V;
- if an element E of the system *satisfies* a requirement R, any predicate V that *verifies* R will have to control E; i.e., a verifier V have to control if $hold(\mathcal{P}_R) \equiv \top$, assuming ($hold(\mathcal{C}_R \equiv \top$), given the behavior of E.

Note that $\mathcal{C}_{R_1} \subseteq \mathcal{C}_{R_2}$ can mean that:

- $\forall C_i \in \mathcal{C}_{R_1}, C_i \in \mathcal{C}_{R_2}$ (classical subset relation); but also that
- $\forall C_i \in \mathcal{C}_{R_1}, \quad \exists C_j \in \mathcal{C}_{R_2}, C_i = C_j \vee x$ since $P_1 \wedge ... \wedge P_m \wedge ... \wedge P_n \wedge (P_1 \vee x) \wedge ... \wedge (P_m \vee x) \equiv P_1 \wedge ... \wedge P_m \wedge ... \wedge P_n$, $\{P_1 \vee x, ..., P_m \vee x\} \subset \{P_1, ..., P_m, ..., P_n\}$.

So, a requirement R_1 is less constrained than R_2 (the context is larger) if R_1 applies with less constraints than R_2 or if R_1 applies with constraints of R_2 that are weakened. This formulation will demonstrate satisfaction's propagation on refinement and is commonly used in literature about refinement.

5.2.2 Formal definitions of relationships between requirements

In this section, we will consider each of the relationships listed below. We will give an example, provide a semantic definition, formalize the satisfaction propagation, and demonstrate it. In the following, we will consider:

- R_0, R_1, R_2, R_i, ..., R_n, requirements;
- $\mathcal{C}_{R_0}$, $\mathcal{C}_{R_1}$, $\mathcal{C}_{R_2}$, $\mathcal{C}_{R_i}$, ..., $\mathcal{C}_{R_n}$, constraints;
- and $\mathcal{P}_{R_0}$, $\mathcal{P}_{R_1}$, $\mathcal{P}_{R_2}$, $\mathcal{P}_{R_i}$, ..., $\mathcal{P}_{R_i}$, properties.

Note: For each relationship, we do not express any value judgments about them. For example, the contradiction is not something that is desired in a requirements set. However, it is interesting to detect this kind of link, and it corresponds to a reality that we have to handle. This is also the case for repetition. Repetitions do not add information about the system and can be sources of ambiguity – if two requirements are not clearly expressed as a repeat, one can ask the difference between them – and engineers want to avoid repetition. Furthermore, if a requirement changes and not the other one, inconsistency will be introduced in the system. However, sometimes repetition can be needed. For example, engineers may want to use a requirement expressed elsewhere (in another part of the system or even in another project) and to be informed if it changes. So we have to propose a way to support repetitions.

While we aim to support the management of these relationships (e.g., applying good principles, detecting unwanted relationships, etc.), we do not deeply analyze their usage and implications in this chapter.

Repetition

Example: "*When activated, the drone shall take over a parcel (pick it in the warehouse and deliver it)*" repeats "*The drone shall pick up a parcel, go to the destination and drop it off when activated on the web application*".

Definition: A requirement repeats another one if they are the same – i.e., a requirement is a repetition of another one if they share the same context and if their properties are equals: R_1 repeats R_2 iff $\mathcal{C}_{R_1} = \mathcal{C}_{R_2}$ and $\mathcal{P}_{R_1} = \mathcal{P}_{R_2}$.

Satisfaction's propagation : $sat(R_1) \equiv sat(R_2)$

Demonstration: By construction.

Note: These repetitions can be detected by simplifying constraints and properties to conjunctive normal form.

Constraint

Example: "*When delivery order is given and when the battery is low (< 10%), the drone shall send a notification and stay in the charge pad*" is a constrained version of "*When the battery is under 10%, the drone shall stay in charge and a notification of battery status shall be available for the user*".

Definition: A requirement is a constrained version of another one if the first one owns more constraints:
R_1 is a constraint of R_0 if $\mathcal{P}_{R_0} = \mathcal{P}_{R_1}$ and $\mathcal{C}_{R_O} \subseteq \mathcal{C}_{R_1}$.

Satisfaction's propagation : $sat(R_0) \rightarrow sat(R_1)$

Demonstration: Let's say $\mathcal{C}_{R_1} = \{C_1, ..., C_m, ..., C_n\}$ and $\mathcal{C}_{R_0} = \{C_1, ..., C_m\}$.

$$\begin{aligned} sat(R_1) &\equiv hold(\mathcal{C}_{R_1}) \rightarrow hold(\mathcal{P}_{R_1}) \\ &\equiv (C_1 \wedge ... \wedge C_m \wedge ... \wedge C_n) \rightarrow hold(\mathcal{P}_{R_1}) \\ &\equiv (C_1 \wedge ... \wedge C_m \wedge C_1 \wedge ... \wedge C_m \wedge ... \wedge C_n) \rightarrow hold(\mathcal{P}_{R_1}) \\ &\equiv (hold(\mathcal{C}_{R_0}) \wedge hold(\mathcal{C}_{R_1})) \rightarrow hold(\mathcal{P}_{R_1}) \\ &\equiv (hold(\mathcal{C}_{R_0}) \rightarrow hold(\mathcal{P}_{R_1})) \vee (hold(\mathcal{C}_{R_1}) \rightarrow hold(\mathcal{P}_{R_1})) \\ &\equiv sat(R_0) \vee sat(R_1) \end{aligned}$$

So we have:

$$\begin{aligned} & sat(R_1) \Leftrightarrow sat(R_0) \vee sat(R_1) \\ \equiv\ & (sat(R_1) \rightarrow (sat(R_0) \vee sat(R_1))) \wedge \\ & ((sat(R_0) \vee sat(R_1)) \rightarrow sat(R_1)) \\ \equiv\ & ((sat(R_1) \rightarrow sat(R_0)) \vee (sat(R_1) \rightarrow sat(R_1))) \wedge \\ & ((sat(R_0) \rightarrow sat(R_1)) \wedge (sat(R_1)) \rightarrow sat(R_1)) \\ \equiv\ & ((sat(R_1) \rightarrow sat(R_0)) \vee \top) \wedge ((sat(R_0) \rightarrow sat(R_1)) \wedge \top \\ \equiv\ & sat(R_0) \rightarrow sat(R_1) \end{aligned}$$

Addition

Example: "*The drone shall retrieve destination of handled parcel from an existing web service and transport it to this destination*" is an addition to "*When a parcel is handled by the drone, the drone shall transport it to its destination*".

Definition: A requirement is an addition version of another one if they share the same context and the first one has more properties:
R_1 is an addition of R_0 if $\mathcal{P}_{R_0} \subseteq \mathcal{P}_{R_1}$ and $\mathcal{C}_{R_0} = \mathcal{C}_{R_1}$ (Let's note that if the constraints are different, this is not an addition but just a different requirement).

Satisfaction's propagation : $sat(R_1) \rightarrow sat(R_0)$

Demonstration: Let's say $\mathcal{P}_{R_1} = \{P_1, ..., P_m, ..., P_n\}$ and $\mathcal{P}_{R_0} = \{P_1, ..., P_m\}$.

$$\begin{aligned} sat(R_1) &\equiv hold(\mathcal{C}_{R_1}) \rightarrow hold(\mathcal{P}_{R_1}) \\ &\equiv hold(\mathcal{C}_{R_1}) \rightarrow (P_1 \wedge ... \wedge P_m \wedge ... \wedge P_n) \\ &\equiv hold(\mathcal{C}_{R_1}) \rightarrow (P_1 \wedge ... \wedge P_m \wedge P_1 \wedge ... \wedge P_m \wedge ... \wedge P_n) \\ &\equiv hold(\mathcal{C}_{R_1}) \rightarrow (hold(\mathcal{P}_{R_0}) \wedge hold(\mathcal{P}_{R_1})) \\ &\equiv (hold(\mathcal{C}_{R_1} \rightarrow hold(\mathcal{P}_{R_0})) \wedge (hold(\mathcal{C}_{R_1} \rightarrow hold(\mathcal{P}_{R_1})) \\ &\equiv sat(R_0) \wedge sat(R_1) \end{aligned}$$

So we have:

$$\begin{aligned} & sat(R_1) \Leftrightarrow sat(R_0) \wedge sat(R_1) \\ \equiv\ & (sat(R_1) \rightarrow (sat(R_0) \wedge sat(R_1))) \wedge \\ & \quad ((sat(R_0) \wedge sat(R_1)) \rightarrow sat(R_1)) \\ \equiv\ & ((sat(R_1) \rightarrow sat(R_0)) \wedge (sat(R_1) \rightarrow sat(R_1))) \wedge \\ & ((sat(R_0) \rightarrow sat(R_1)) \vee (sat(R_1)) \rightarrow sat(R_1)) \\ \equiv\ & ((sat(R_1) \rightarrow sat(R_0)) \wedge \top) \wedge ((sat(R_0) \rightarrow sat(R_1)) \vee \top) \\ \equiv\ & sat(R_1) \rightarrow sat(R_0) \end{aligned}$$

Partition

Example: "*When the distance to warehouse is more than 500 meters and when the drone battery is under 10%, then the drone shall switch to recovery mode, send a location message and land quickly*" can be partitioned into

- "*When the battery is under 10% then the drone shall switch to recovery mode*" and "*When the battery is under 10% then the drone shall land quickly and send a location message*";
- or even into "*When the distance to warehouse is more than 500 meters then the drone shall send a position message*" and "*When the battery is under 10% then the drone shall switch to recovery mode and land quickly*".

Definition: A requirement can be split into other requirements that will together participate to the satisfaction of the original one.

R_0 (non-atomic) is partitioned into $\{R_1, ..., R_n\}$ where all requirements R_i in $\{R_1, ..., R_n\}$ own a set of properties $\mathcal{P}_{R_i}$ with $\mathcal{P}_{R_i} \subseteq \mathcal{P}_{R_0}$.

To ensure that all properties of the original requirement will be met, the partition has to be complete, i.e., the set of all requirements part of the original one have to cover the original one ($\bigcup_{i=1}^{n} \mathcal{P}_{R_i} = \mathcal{P}_{R_0}$).The constraints of these requirements can be a subset of the constraints of the original one ($\mathcal{C}_{R_i} \subseteq \mathcal{C}_{R_0}$).

Satisfaction's propagation : $sat(R_0) \equiv (sat(R_{11}) \vee ... \vee sat(R_{1m})) \wedge ... \wedge (sat(R_{n1}) \vee ... \vee sat(R_{nm}))$ with:

- $|\mathcal{C}_{R_0}| = m$ and $|\mathcal{P}_{R_0}| = n$;
- for all $i \in [1; n]$ and $j \in [1; m]$, $\mathcal{C}_{R_{ij}} \in \mathcal{C}_{R_0}$ and $\mathcal{P}_{R_{ij}} \in \mathcal{P}_{R_0}$;
- for all $i \in [1; n]$, and for all $j, k \in [1; m]$, $\mathcal{P}_{R_{ij}} = \mathcal{P}_{R_{ik}}$;
- for all $i \in [1; n]$, $\bigcup_{j=1}^{m} \mathcal{C}_{R_{ij}} = \mathcal{C}_{R_0}$.

Demonstration:

$$\begin{aligned}
sat(R_0) &\equiv hold(\mathcal{C}_{R_0}) \rightarrow hold(\mathcal{P}_{R_0}) \\
&\equiv (C_1 \wedge ... \wedge C_x) \rightarrow (P_1 \wedge ... \wedge P_y) \\
&\equiv (hold(\mathcal{C}_{R_1}) \wedge ... \wedge hold(\mathcal{C}_{R_m})) \\
&\quad \rightarrow (hold(\mathcal{P}_{R_1}) \wedge ... \wedge hold(\mathcal{P}_{R_n})) \\
&\equiv ((hold(\mathcal{C}_{R_1}) \wedge ... \wedge hold(\mathcal{C}_{R_m})) \rightarrow (hold(\mathcal{P}_{R_1})) \wedge ... \wedge \\
&\quad ((hold(\mathcal{C}_{R_1}) \wedge ... \wedge hold(\mathcal{C}_{R_m})) \rightarrow hold(\mathcal{P}_{R_n})) \\
&\equiv ((hold(\mathcal{C}_{R_1}) \rightarrow hold(\mathcal{P}_{R_1})) \vee ... \vee \\
&\quad (hold(\mathcal{C}_{R_m}) \rightarrow hold(\mathcal{P}_{R_1}))) \wedge ... \wedge \\
&\quad ((hold(\mathcal{C}_{R_1}) \rightarrow hold(\mathcal{P}_{R_n})) \vee ... \vee \\
&\quad hold(\mathcal{C}_{R_m}) \rightarrow hold(\mathcal{P}_{R_n})) \\
&\equiv (sat(R_{11}) \vee ... \vee sat(R_{1m}) \wedge ... \wedge \\
&\quad (sat(R_{n1}) \vee ... \vee sat(R_{nm}))
\end{aligned}$$

Refinement

Example: "*When the drone handle a parcel, then it shall transport it carefully*" can be refined in "*At any time, the drone shall avoid obstacles, fly at the minimum safe height and be able to stop in less than 1 second.*".

Definition: A requirement is a refinement of an other one if it introduces a new property and/or if it relax the context (the constraints are weaken):
R_1 refines R_0 if $\mathcal{P}_{R_0} \subseteq \mathcal{P}_{R_1}$ and $\mathcal{C}_{R_0} \equiv \mathcal{C}_{R_1} \vee \mathcal{C}'_{R_0}$ (where $\mathcal{C}'_{R_0}$ is another set of constraints).

Satisfaction's propagation : $sat(R_1) \rightarrow sat(R_0)$.

Demonstration: Let's consider two requirements R_0 and R_1, with $\mathcal{C}_{R_0} = \{C_1, ..., C_m\}$ and $\mathcal{C}_{R_1} = \{C_1 \vee C_n, ..., C_m \vee C_n\}$, $\mathcal{P}_{R_1} = \{P_1, ..., P_m, ..., P_n\}$ and $\mathcal{P}_{R_0} = \{P_1, ..., P_m\}$.

$$
\begin{aligned}
sat(R_1) &\equiv hold(\mathcal{C}_{R_1}) \rightarrow hold(\mathcal{P}_{R_1}) \\
&\equiv ((C_1 \vee C_n) \wedge ... \wedge (C_m \vee C_n)) \rightarrow (P_1 \wedge ... \wedge P_m \wedge ... \wedge P_n) \\
&\equiv ((C_1 \wedge ... \wedge C_m) \vee C_n) \rightarrow (P_1 \wedge ... \wedge P_m \wedge ... \wedge P_n) \\
&\equiv ((C_1 \wedge ... \wedge C_m) \vee (C_1 \wedge ... \wedge C_m) \vee C_n) \\
&\quad \rightarrow (P_1 \wedge ... \wedge P_m \wedge P_1 \wedge ... \wedge P_m \wedge ... \wedge P_n) \\
&\equiv (hold(\mathcal{C}_{R_0}) \vee hold(\mathcal{C}_{R_1})) \rightarrow (hold(\mathcal{P}_{R_0}) \wedge hold(\mathcal{P}_{R_1})) \\
&\equiv (hold(\mathcal{C}_{R_0}) \rightarrow (hold(\mathcal{P}_{R_0}) \wedge hold(\mathcal{P}_{R_1})) \\
&\quad \wedge (hold(\mathcal{C}_{R_1}) \rightarrow (hold(\mathcal{P}_{R_0}) \wedge hold(\mathcal{P}_{R_1})) \\
&\equiv (hold(\mathcal{C}_{R_0}) \rightarrow hold(\mathcal{P}_{R_0})) \wedge (hold(\mathcal{C}_{R_0}) \rightarrow hold(\mathcal{P}_{R_1})) \\
&\quad \wedge (hold(\mathcal{C}_{R_1}) \rightarrow hold(\mathcal{P}_{R_0})) \wedge (hold(\mathcal{C}_{R_1}) \rightarrow hold(\mathcal{P}_{R_1})) \\
&\equiv sat(R_0) \wedge (hold(\mathcal{C}_{R_0}) \rightarrow hold(\mathcal{P}_{R_1})) \wedge \\
&\quad (hold(\mathcal{C}_{R_1}) \rightarrow hold(\mathcal{P}_{R_0})) \wedge sat(R_1)
\end{aligned}
$$

However, $hold(\mathcal{C}_{R_0}) \rightarrow hold(\mathcal{P}_{R_1})$ is an addition R'_1 to R_1 and $hold(\mathcal{C}_{R_1}) \rightarrow hold(\mathcal{P}_{R_0})$ is a constraint R'_0 to R_0. So we have:

$$
\begin{aligned}
sat(R_1) &\equiv sat(R_0) \wedge sat(R'_1) \wedge sat(R'_0) \wedge sat(R_1) \\
&\equiv sat(R_0) \wedge (sat(R_0) \wedge sat(R_1)) \wedge \\
&\quad (sat(R_0) \vee sat(R_1)) \wedge sat(R_1) \\
&\equiv sat(R_0) \wedge sat(R_1)
\end{aligned}
$$

Using the same properties than in addition, we can thus deduce $sat(R_1) \rightarrow sat(R_0)$.

Note: The addition relation is a particular case of the refinement (a refinement with $\mathcal{C}'_{R_0} = \emptyset$, i.e., without constraints weakening).

Contradiction

Example: "*At any time, the drone shall avoid obstacles, fly at the minimum safe height and be able to stop in less than 1 second*" contradicts "*When the battery is under 10% then the drone shall switch to recovery mode and land quickly*".

Definition: A requirement is a contradiction of another one if when one of them is satisfied, the other cannot be satisfied. Especially, when considering R_1 and R_2 with $\mathcal{C}_{R_1} \subseteq \mathcal{C}_{R_2}$, if there is a property P in $\mathcal{P}_{R_1}$ and $\neg P$ in $\mathcal{P}_{R_2}$, then R_1 and R_2 are contradictory.

Satisfaction's propagation : R_1 contradicts R_2 if $sat(R_1) \wedge sat(R_2)$ cannot be true in a given context. That can also be expressed as: $hold(\mathcal{P}_{R_1}) \rightarrow \neg hold(\mathcal{P}_{R_2})$ and $hold(\mathcal{P}_{R_2}) \rightarrow \neg hold(\mathcal{P}_{R_1})$,

Demonstration: Let's take two requirements R_1 and R_2 with $\mathcal{C}_{R_1} \subseteq \mathcal{C}_{R_2}$ and $\mathcal{P}_{R_1} = \{P_1, ..., P, ..., P_n\}$ and $\mathcal{P}_{R_2} = \{P'_1, ..., \neg P, ..., P'_n\}$.

Let's try to satisfy both requirements in the context of R_2 ($sat(R_1) \wedge sat(R_2) \wedge hold(\mathcal{C}_{R_2})$).

$$
\begin{aligned}
& sat(R_1) \wedge sat(R_2) \wedge hold(\mathcal{C}_{R_2}) \\
\equiv\ & (hold(\mathcal{C}_{R_1}) \rightarrow hold(\mathcal{P}_{R_1})) \wedge (hold(\mathcal{C}_{R_2}) \rightarrow hold(\mathcal{P}_{R_2})) \wedge hold(\mathcal{C}_{R_2}) \\
\equiv\ & ((C_1 \wedge ... \wedge C_m) \rightarrow (P_1 \wedge ... \wedge P \wedge ... \wedge P_m)) \wedge \\
& ((C_1 \wedge ... \wedge C_m \wedge ... \wedge C_n) \rightarrow (P'_1 \wedge ... \wedge \neg P \wedge ... \wedge P'_m)) \wedge \\
& (C_1 \wedge ... \wedge C_m \wedge ... \wedge C_n) \\
\equiv\ & (C_1 \wedge ... \wedge C_m \wedge ... \wedge C_n) \wedge \\
& (P_1 \wedge ... \wedge P \wedge ... \wedge P_m) \wedge (P'_1 \wedge ... \wedge \neg P \wedge ... \wedge P'_m) \\
\equiv\ & \bot
\end{aligned}
$$

Thus, we cannot have $sat(R_1) \wedge sat(R_2)$ in the context of R_2. Let us try now to satisfy both requirements in the context of R_1.

$$
\begin{aligned}
& sat(R_1) \wedge sat(R_2) \wedge hold(\mathcal{C}_{R_1}) \\
\equiv\ & (hold(\mathcal{C}_{R_1}) \rightarrow hold(\mathcal{P}_{R_1})) \wedge (hold(\mathcal{C}_{R_2}) \rightarrow hold(\mathcal{P}_{R_2})) \wedge hold(\mathcal{C}_{R_1}) \\
\equiv\ & ((C_1 \wedge ... \wedge C_m) \rightarrow (P_1 \wedge ... \wedge P \wedge ... \wedge P_m)) \wedge \\
& ((C_1 \wedge ... \wedge C_m \wedge ... \wedge C_n) \rightarrow (P'_1 \wedge ... \wedge \neg P \wedge ... \wedge P'_m)) \wedge \\
& (C_1 \wedge ... \wedge C_m) \\
\equiv\ & ((C_1 \wedge ... \wedge C_m) \wedge (P_1 \wedge ... \wedge P \wedge ... \wedge P_m) \wedge \\
& \neg(C_1 \wedge ... \wedge C_m \wedge ... \wedge C_n)) \vee ((C_1 \wedge ... \wedge C_m) \wedge \\
& (P_1 \wedge ... \wedge P \wedge ... \wedge P_m) \wedge (P'_1 \wedge ... \wedge \neg P \wedge ... \wedge P'_m)) \\
\equiv\ & ((C_1 \wedge ... \wedge C_m) \wedge (P_1 \wedge ... \wedge P \wedge ... \wedge P_m) \wedge \\
& \neg(C_1 \wedge ... \wedge C_m \wedge ... \wedge C_n)) \vee \bot \\
\equiv\ & hold(\mathcal{C}_{R_1}) \wedge \neg hold(\mathcal{C}_{R_2}) \wedge hold(\mathcal{P}_{R_1})
\end{aligned}
$$

This result means that, in the context of R_1, if we want to have $sat(R_1) \wedge sat(R_2)$, we need to be strictly in the context of R_1 ($hold(\mathcal{C}_{R_1}) \wedge \neg hold(\mathcal{C}_{R_2})$) and so, R_2 won't be satisfied (since we have $\neg hold(\mathcal{C}_{R_2})$).

Note: The given definition of *contradiction* must not be limited to the case $\mathcal{C}_{R_1} \subseteq \mathcal{C}_{R_2}$ and P in $\mathcal{P}_{R_1}$ and $\neg P$ in $\mathcal{P}_{R_2}$. This relation does not express anything about two totally unrelated requirements. For example, we can consider that both requirements *The system shall be cheap* and *The system shall be safe* are contradictory even if they does not deal with the same property. *Contradiction* can thus be used to detect simple contradictions, or to highlight contradictions in a set of requirements and to propagate the information.

5.3 How to use semantics of relationships to improve requirements?

The semantic definition of relationships between requirements has several advantages. First of all, it allows us to remove ambiguity about the meaning of these relations. Thus, the work done in the first part of this chapter (on the definition of relations between requirements in natural language) is completed by this semantic definition. As a reminder, in [8], we had defined nine relations. Of these nine relations, two can be left out of the semantic definition: the relation *disjoins* and the relation *characterizes*. The first one is that there is no semantic link between two disjoint requirements due to its nature. The second because the notion of metarequirement implies not a relation between requirements but a relation on the properties on which a requirement is based (the property is a predicate on another requirement and not on the system).

This reduces the set of relations to six. Among them, some are trivial to align with the relations we have deduced from the semantic combinations of properties and constraints: the *repeat* corresponds to the *repeats*, the *contradiction* corresponds to the *contradicts*, the *constraint* corresponds to the *constrains* and the *partition* corresponds to the *belongs*. The *extends* can be aligned with two relations whose semantics we have defined: the *addition* but also the *refinement*. That makes sense, since addition is only a special case of refinement.

Two relationships of the work done in [8] cannot be aligned with the semantic definition work done here. Indeed, the *follows* implies that one requirement is a consequence of another requirement. This consequence, as understood in [8], is not a logical implication, but rather that a R_1 requirement exists because of the existence of a R_2 requirement. The last relation, the *excepts*, implies that the contexts are different and, therefore, cannot be defined with the semantics we propose in this chapter.

That said, this work already provides additional tools. We will detail them in the following sections.

5.3.1 Propagating satisfaction

Since we defined the implication of satisfaction, it is thus possible, given a set of requirements, to propagate the satisfaction of requirements. Such propagation can lead to two benefits: on the one hand, it is possible, given a set of satisfied requirements and the relationships between the requirements, to emphasize what are the requirements already satisfied. On the other hand, it is possible to go the other way around and isolate the requirements that are needed to be satisfied and the requirements that will be satisfied by propagating.

We can thus define satisfaction links between requirements from these relationships, forming a directed graph. A example is given in Fig. 5.1 (with an emphasize on the minimum set to satisfy the whole requirements set).

A set of requirements and the relationships between them is given. Using the semantics defined in the previous sections, it is thus possible to provide satisfaction links between requirements, as given in Fig. 5.2. A simple graph traversal algorithm can do this (see algorithm 1) that will return a new directed graph with satisfaction links, given the semantics of the relationships.

For the example given in Fig. 5.1, we thus obtain the directed graph illustrated in Fig. 5.2 (with an emphasize on the minimum set to satisfy the whole requirements set).

The resulting graph can be used in (at least) two ways: firstly, it is possible to propagate a requirement's satisfaction through the graph. For instance, if the requirement $R0$ is satisfied, it

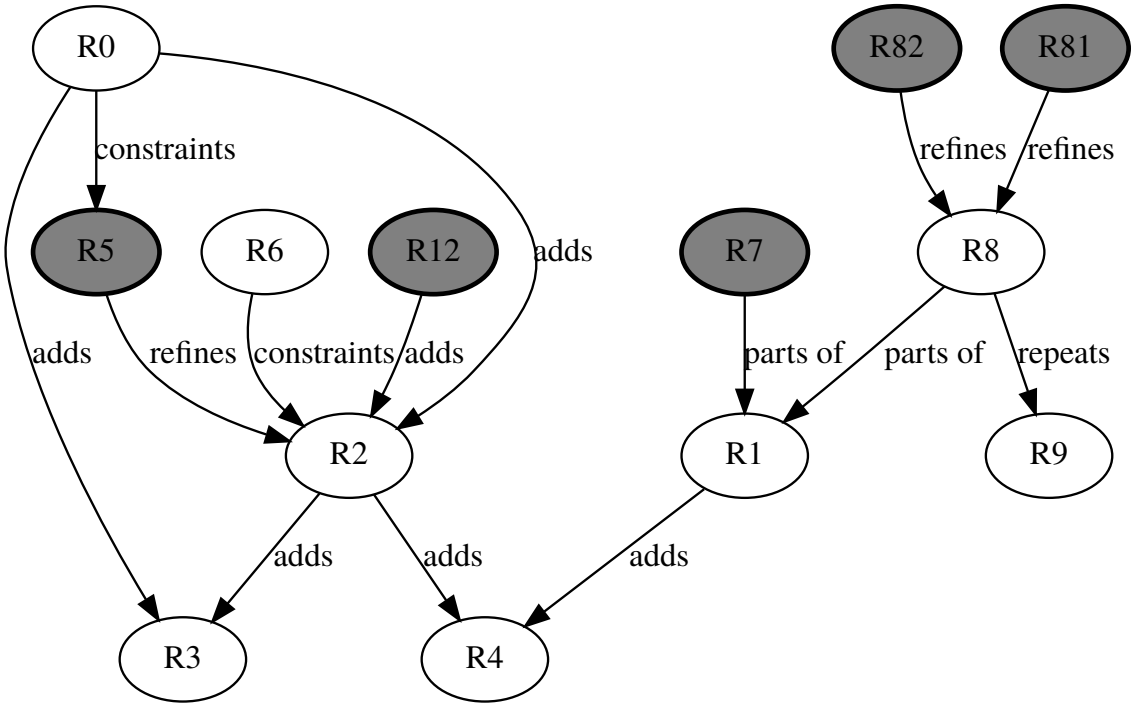


Figure 5.1: Example of requirements and relationships between them

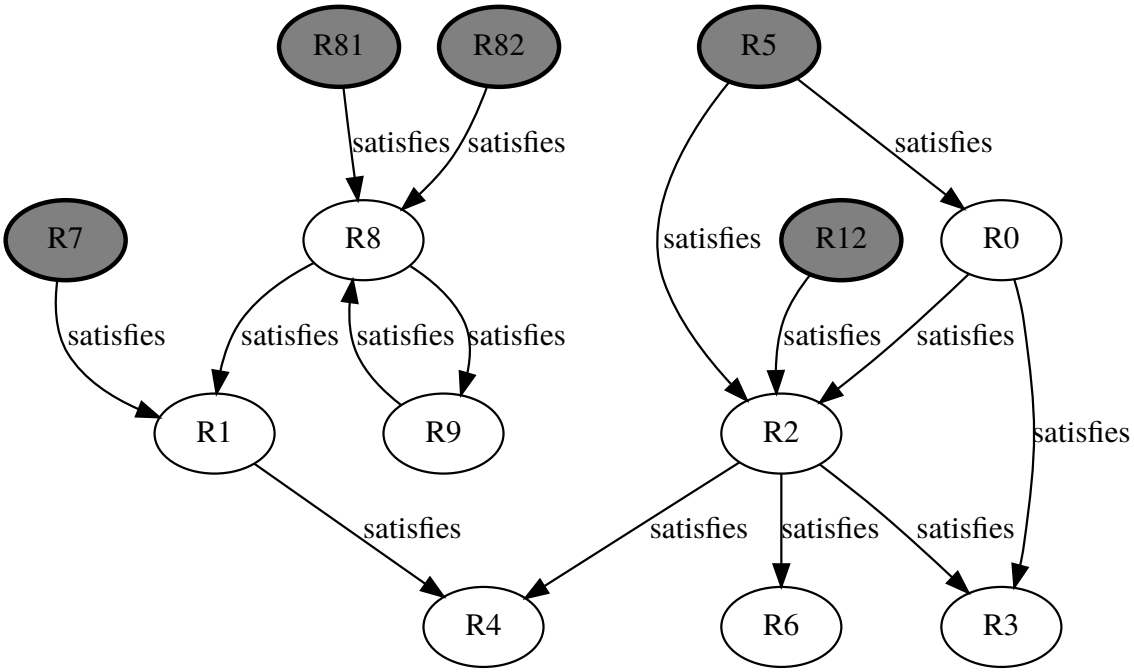


Figure 5.2: Example of satisfaction links for requirements and relationships from Fig. 5.1

Algorithm 1: Compute satisfaction links given requirements and relationships between them

Input: A directed graph $R = (V, E)$ of requirements ($r \in V$) and relationships between requirements ($(r_1, r_2, relation) \in E$; where r_1 is linked to r_2 by a relation of kind $relation$).

Output: A directed graph $O = (X, S)$ of requirements ($r \in V$) and satisfaction propagation links ($(r_1, r_2) \in S$).

$X \leftarrow V$
$S \leftarrow \{\}$
foreach $r_1 \in V$ **do**
  foreach $(r_1, r_i, rel) \in E$ **do**
    switch *rel* **do**
      case *addition* **do**
        $S \leftarrow S \cup \{(r_1, r_i)\}$
      end
      case *repetition* **do**
        $S \leftarrow S \cup \{(r_1, r_i); (r_i, r_1)\}$
      end
      case *constraint* **do**
        $S \leftarrow S \cup \{(r_i, r_1)\}$
      end
      case *partition* **do**
        $S \leftarrow S \cup \{(r_1, r_i)\}$
      end
      case *refinement* **do**
        $S \leftarrow S \cup \{(r_1, r_i)\}$
      end
    end
  end
end
return (X, S)

is possible to deduce that the requirements $R2$, $R3$, $R4$, and $R6$ are also satisfied by propagation. Finally, it is also possible to find the minimum subset of requirements that must be met to satisfy the complete set of requirements. This can be done by finding the sources in the directed set, as given in algorithm 2. However, this naive approach does not take into account the case of a cycle that can appear (especially with repetitions) – a smarter implementation is available in gitlab[1].

In Fig. 5.1 and Fig. 5.2, requirements that are in this minimal set are shaded. Since such a semantic definition can give advantages, even at a very early stage of research, we found it interesting to support this kind of relationship in SIRCOD and RSML. Moreover, such an approach, linked to the strong dependency between requirements and code implied by SIRCOD, allows to detect the impact of requirements change directly in the code. This can be compared to impact analysis algorithms [107], since it allows computing where a change in requirements can effect the code, by

[1] https://gitlab.com/fgalinier/minimumset

Algorithm 2: Find sources of the directed graph containing the requirements and satisfaction links between them

Input: A directed graph $O = (X, S)$ of requirements ($r \in X$) and satisfaction propagation links ($(r_1, r_2) \in S$).
Output: A set M of requirements $r \in X$ that are sources
$M \leftarrow \{\}$
foreach $r \in X$ **do**
  if $\nexists (r_i, r) \in S$ **then**
    $M \leftarrow M \cup \{r\}$
  end
end
return M

propagation. Moreover, combined with such algorithms, it allows developers to see exactly what are all the part of code that are impacted by a change, and not only the code directly linked to the study requirement.

5.3.2 Improving SIRCOD and RSML

Relationships in SIRCOD

In section 5.4, a possible formal representation of the relationships will be detailed in Eiffel. However, even without going into this detail level, it is interesting to introduce such relationships in the SIRCOD approach. Using the previously defined algorithms (and assuming that the relationships are well defined) can help decrease the number of requirements to analyze, decreasing the complexity of such an analysis.

We introduced this notion of relationships between requirements in the same way that we traced the source of requirements: using the notes' mechanism. In List. 5.1, the feature of List. 5.1 is enriched with a *adds* note, that gives the information that requirement_5_1_doc is an extension of requirement_1_1_3_docs. Such relationships can then be used to trace requirements and to check that all requirements are correctly satisfied. In the example of List. 5.1, since `requirement_5_1_doc` is an addition of `requirement_1_1_3_doc`, it is expected that if `requirement_5_1_doc` is satisfied, `requirement_1_1_3_doc` is satisfied too.

```
feature
  requirement_5_1_doc: STRING
        note
                EIS: "src=use-case_drone.docx", "bookmark=5.1"
                adds: "{DELIVERY_REQUIREMENTS}.requirement_1_1_3_doc"
        do
                Result := "[
                        [5.1] (is an addition of [1.1.3]) When drone
                            location is equal to destination then
                            eventually attached parcel shall be equal to
                             null and drop location status shall be
                            equal to confirmed.
```

```
                ]"
        end
```

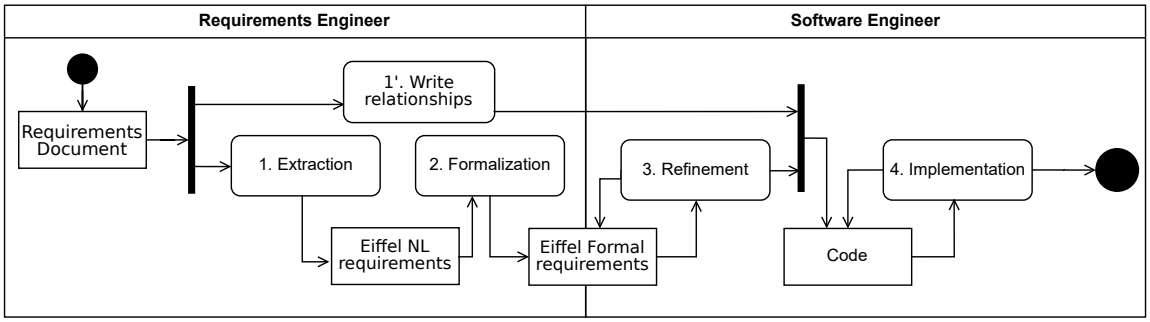
Listing 5.1: Relationships note between requirement_5_1_doc and requirement_1_1_3_doc

These notes are added by applying the pattern given in List. 5.2.

```
feature
  <identifier of the requirement>_doc: STRING
  note
    EIS: "src=<path to source document>", "bookmark=<bookmark of the
        requirement>"
                <FOREACH relation rel of the requirement DO>
                        <Type of rel>: <target of rel>
          <END FOREACH>
  do
    result :=
                <FOREACH addition a of the requirement DO>
                        <target of a> +
          <END FOREACH>
                "[
      <text of the requirement>
    ]"
        end
```

Listing 5.2: Eiffel pattern for a requirement feature with relationships

We do not add the *refinement* note since, as we already present in chapter 3, the refinement is represented in SIRCOD using the redefinition mechanism. The addition of relationships nevertheless impacts the SIRCOD process. Indeed, we have now to consider the relationships during all the steps.



This is represented in Fig. 5.3.2. The process now includes the writing of the relationships, in parallel with the three first steps. Thus, the engineer must also identify and create links between requirements.

However, this new activity is not only in parallel with the *extraction* activity. Indeed, since the *formalization* and *refinement* activities may give rise to new requirements, it becomes necessary to check if the requirements are linked to other requirements.

To support the addition of new notes for relationships, the documentation view has been updated to let appear these relationships and print the related requirement. Thus, in Fig. 5.3, `requirement_5_1_doc` includes the text of `requirement_1_1_3_doc`, since it is an addition of this requirement feature (expressed by: `adds: "{DELIVERY_REQUIREMENTS}.requirement_1_1_3_doc"`).

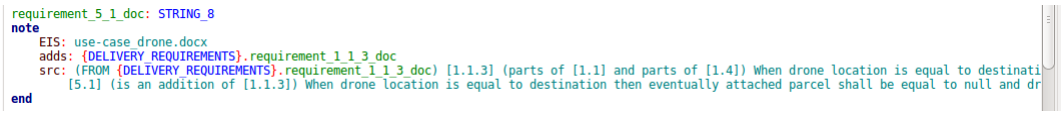


Figure 5.3: Documentation view of the class

Moreover, thanks to these relationships and the EIS mechanism, it is possible to propagate the change. If a change is detected, the developer can see in EIS what feature is potentially impacted by a change and, using links between features, find if some other features can be concerned. On the other hand, after modifying these other features, the user can check again by using the EIS links if the changes he introduced are adequate to the source document's textual requirements.

Relationships in RSML

The semantics propose in this chapter and the addition of relationships to SIRCOD led us to propose an extension to the RSML metamodel. As detailed in previous sections, it allows an easier requirement analysis and reduces the set of requirements to verify. Moreover, NL requirements are only treated as units, with no specific semantics on the requirements themselves. These relationships can thus be used to give some information about the satisfaction of these requirements too.

This horizontal traceability between requirements is expressed by the reification of the relation between two requirements. This reification is done through the *Trace* artifacts. The trace does not own any semantics as it, and shall be specialized by one of the six relationships that we defined, based on the semantics proposed in section 5:

- addition: the requirement R_1 add a property to the target requirement R_2;
- containment: the requirement R_1 contains the target requirement R_2;
- refine: the requirement R_1 refines the target requirement R_2;
- contradiction: the requirement R_1 and the target requirement R_2 cannot be satisfied in the same time;
- repetition: the requirement R_1 is a repetition of the target requirement R_2;
- constraint: the requirement R_1 constrains the context of the target requirement R_2.

We can, however, specify some *OCL* constraints on these relationships using these semantics. The constraints are limited to the case where the source and target of the relationships are *Formal Requirements*; indeed, it is not possible to analyze the semantics of *Natural Language Requirements* in this context. Of course, this rule does not consider the semantics of properties and constraints themselves, and two different properties (resp. constraints) can be semantically identical. In this case, the OCL rule will report a bad usage of a relation. While this may initially seem problematic,

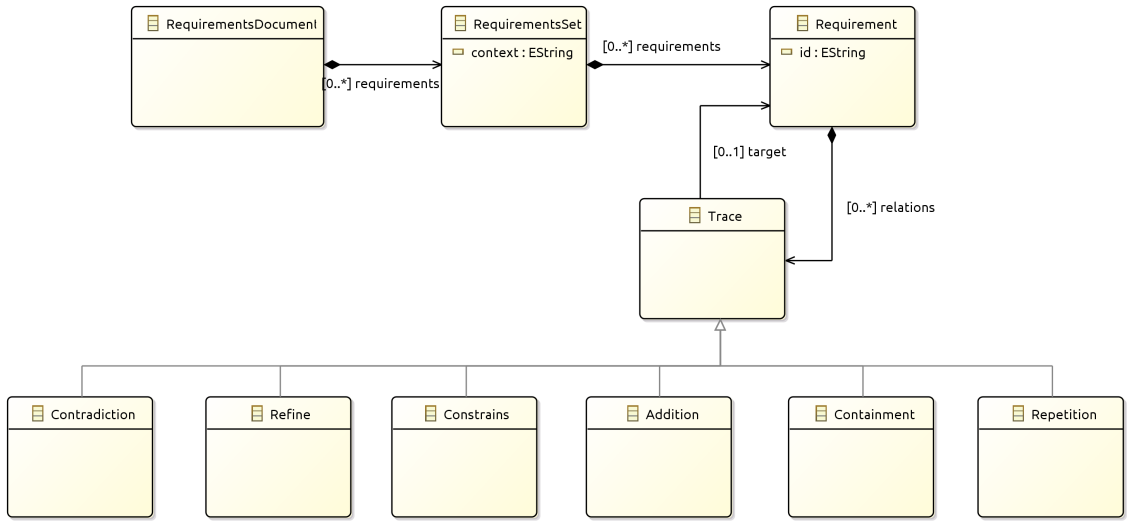


Figure 5.4: Relationships (iii) added to the RSML metamodel

it should be noted that the use of different formulations for the same predicate (the use of synonyms) is one of the biases that lead to ambiguity. This reduces ambiguity by forcing the user to use the same formulation.

Addition implies that the two *Formal Requirements* share the same context and the target requirement properties is a subset of the source requirement properties, that can be verified by:

```
context Addition inv:
  (self.source.oclIsTypeOf(FormalRequirement) and self.target.
      oclIsTypeOf(FormalRequirement)) implies (
    self.target.constraints = self.source.constraints and
    self.source.properties->includesAll(self.target.properties)
  )
```

Containment is an addition, with the exception that the union of the properties of all the sub-requirements R_i shall be equal to the set of properties of the container R_1, that can be verified by:

```
context Containment inv:
  (self.source.oclIsTypeOf(FormalRequirement) and self.target.
      oclIsTypeOf(FormalRequirement)) implies (
    self.target.constraints = self.source.constraints and
    self.source.properties->includesAll(self.target.properties) and
    self.source.properties->forAll(
        p | self.source.relations->exists(
```

```
            r | r.oclIsTypeOf(Containment) and
                r.target.properties->includes(p))
  )
```

Refine implies that the properties of the target requirement (refined) are a subset of the source requirement (refining) and the constraints of the refined requirement are more restrictive than the constraints of the refining requirements. That can be model by:

```
context Refine inv:
  (self.source.oclIsTypeOf(FormalRequirement) and self.target.
     oclIsTypeOf(FormalRequirement)) implies (
    self.source.properties->includesAll(self.target.properties) and
    self.target.constraints->forAll(
        c1 | self.source.constraints->exists(
            c2 | c2.elements->includesAll(c1.elements)))
  )
```

Constraint implies that the two *Formal Requirements* share the same properties, and the target requirement constraints are a subset of the source requirement constraints, that can be verified by:

```
context Constraint inv:
  (self.source.oclIsTypeOf(FormalRequirement) and self.target.
     oclIsTypeOf(FormalRequirement)) implies (
    self.target.properties = self.source.properties and
    self.source.constraints->includesAll(self.target.constraints)
  )
```

Alternative implies that the properties of the source requirement are a subset of the properties of the target requirement and the same for the constraints. That can be verified by:

```
context Alternative inv:
  (self.source.oclIsTypeOf(FormalRequirement) and self.target.
     oclIsTypeOf(FormalRequirement)) implies (
    self.target.properties.includesAll(self.source.properties) and
    self.target.constraints->includesAll(self.source.constraints)
  )
```

Repetition implies that the properties and the constraints of the two requirements are identical; that can be verified by:

```
context Repetition inv:
  (self.source.oclIsTypeOf(FormalRequirement) and self.target.
     oclIsTypeOf(FormalRequirement)) implies (
    self.target.properties = self.source.properties and
    self.target.constraints = self.source.constraints
  )
```

Contradiction implies that the two requirements share the same context, while the properties are incompatible. OCL rule can only check that one of the two requirements constraints set is a subset of the other (i.e., the two requirements are considered in the same context) since we do not analyze the semantics of the properties themselves in the metamodel:

```
context Contradiction inv:
  (self.source.oclIsTypeOf(FormalRequirement) and self.target.
      oclIsTypeOf(FormalRequirement)) implies (
    self.target.constraints.includesAll(self.source.constraints) or
     self.source.constraints.includesAll(self.target.constraints)
  )
```

These OCL constraints shall ensure that the relationships defined in this metamodel are semantically well defined regarding the semantics.

RSML metamodel allows the expression of relationships that can exist in requirements. These relationships can so be expressed in RSML. They are attached to the sources of the relationships and are introduced between parenthesis ('(' and ')'), with a keyword for each kind of relationship, the target being identified by the requirement identifier. All kinds of requirements can be linked using this kind of relationship.

The grammar of **requirement** is modified as follows:

⟨*requirement*⟩ ::= '`[`' ⟨*id*⟩ '`]`' ('`(`' ⟨*relationships*⟩'`)`')?
(⟨*natural language requirement*⟩ | ⟨*formal requirement*⟩) ⟨*newline*⟩

⟨*relationships*⟩ ::= ⟨*relationship kind*⟩ ⟨*id*⟩ ('`and`' ⟨*relationships*⟩)?

⟨*relationship kind*⟩ ::= '`refines`'
| '`part of`'
| '`is an addition of`'
| '`is a constrained version of`'
| '`contradicts`'
| '`repeats`'
| '`is an alternative of`'
| '`trace`'

5.4 Applying this semantics to Eiffel

In [85], a representation of requirements using routines of the Eiffel language is proposed. This allows the enactment of requirements on the one hand, and on the other hand, the proof of the requirements' satisfaction by the system, thanks to the theorem prover Autoproof [88]. Indeed, by providing a representation of requirements in Eiffel, the users can specify (and even implement) the system in a seamless way [108] in a unique language and execute the theorem prover to verify if the system is compliant with its requirements.

In this section, we propose to extend this approach by mapping the relationships defined in section 5.2 above to the concepts of object-oriented programming languages. Thanks to the formal representation of requirements and relationships section 5.2.2, we show how we can propagate satisfaction between requirements through their relationships.

Note that we do not pretend to be exhaustive in the list below (since we can use several ways to express the addition, for example), but our objective is to provide a simple way to express the relationships considered in section 5.2 formally.

To map formal representation of requirements to Eiffel concepts, we consider that a requirement R is a feature of a class, a constraint $\mathcal{C}_R$ of R a *precondition* of this feature and a property $\mathcal{P}_R$ a *postcondition*. Autoproof will use preconditions as assumptions to prove that postconditions are true or not.

We applied these relationships to the example given in section 5.2.

Repetition

In Eiffel, expressing a repetition means that we want to represent a requirement that owns the same pre and postconditions than another one. The call mechanism can be used to do that, as presented in List. 5.3.

```
Req_1
note
    req: "[
        When activated, the drone shall take over a parcel (pick it in
            the warehouse and deliver it)
    ]"
do
    Req_2
end

Req_2
note
    req: "[
        The drone shall pick up a parcel, go to the destination and
            drop it off when activated
    ]"
do
    -- ...
end
```

Listing 5.3: Representation of a repetition in Eiffel

In this listing, `Req_1` is a repetition of `Req_2` – an analysis of the system will have to prove that `Req_2` is correct (the postconditions are true when the preconditions are held) to prove that `Req_1` is correct, and nothing else.

Constraint

To constrain a requirement R, we need to add a new precondition to the Eiffel representation of R. To keep the trace that a requirement R_4 is only a constrained version of a requirement R_3, we can use the call mechanism. Thus, `Req_4` will be a routine adding some new constraints to check, just calling `Req_3`, as seen in List. 5.4.

Req_3

```
note
    req: "[
        When the battery is under 10%, the drone shall stay in charge
            and a notification of battery status shall be available for
            the user
    ]"
require
    battery_low: battery_level < 0.1
do
ensure
    charge: is_in_charge
    notification: app.notification_log.size > 0
end

Req_4
note
    req: "[
        When delivery order is given and when the battery is low (<10%)
            , the drone shall send a  notification and stay in the
            charge pad
    ]"
require
    delivery_order_received: has_delivery_order
do
    Req_3
end
```

Listing 5.4: Representation of a constraint in Eiffel

A static analysis of this code will have to prove that `Req_4` is true by proving `Req_3`, when adding its new preconditions.

If `Req_3` can be proved as itself, `Req_4` will be proved true too, but if `Req_3` cannot be proved, `Req_4` can however be proved using the introduced assumptions (that correspond to the property of constraint $sat(R_3) \rightarrow sat(R_4)$).

Addition

In a similar way than *constraint*, to handle *addition* we can add new properties by adding new postconditions. Thus, a requirement R_6 is an addition to a requirement R_5 if R_6 owns the same body than R_5 with new postconditions. In a similar way than in *constraint*, in List. 5.5 feature `Req_6` is a call to `Req_5` with new postconditions.

```
Req_5
note
    req: "[
        When a parcel is handled by the drone, the drone shall
            transport it to its destination
    ]"
```

```
require
    parcel_handled: parcel /= Void
do
ensure
    transport_to_destination: location = parcel.destination and parcel
       = Void
end

Req_6
note
    req: "[
        The drone shall retrieve destination of handled parcel from an
           existing webservice and transport it  to this destination
    ]"
do
    Req_5
ensure
    destination_from_webservice: location = webservice.destination(
       parcel)
end
```

Listing 5.5: Representation of an addition in Eiffel

The satisfaction's propagation of addition is preserved; indeed, `Req_6` can be proved only if `Req_5` is proved too.

Partition

The partition can be expressed using the same mechanism as decomposition. Thus, a requirement R_0 partitioned in several requirements R_1, ..., R_n can be seen as the composition of the preconditions and postconditions of R_1, ..., R_n. In List. 5.6, `Req_10` is partitioned into `Req_11` and `Req_12`.

```
Req_10
note
    req: "[
       When the distance to warehouse is more than 500 meters and when
          the drone battery is under 10%, then the drone shall switch
          to recovery mode, send a location message and land quickly
    ]"
require
    distance_more_than_500: distance (location, warehouse.location) >
       500 -- m
    battery_low: battery_level < 0.1
do
    Req_11
    Req_12
ensure
```

```
    message_sent: app.notification_log.size > 0
    recovery_mode: mode = recovery
    land: altitude < 0.01 -- m
end

Req_11
note
    req: "[
        When the distance to warehouse is more than 500 meters then the
            drone shall send a position message
    ]"
require
    distance_more_than_500: distance (location, warehouse.location) >
        500 -- m
do
ensure
    message_sent: app.notification_log.size > 0
end

Req_12
note
    req: "[
       When the battery is under 10% then the drone shall switch to
           recovery mode and land quickly
    ]"
require
    battery_low: battery_level < 0.1
do
ensure
    recovery_mode: mode = recovery
    land: altitude < 0.01 -- m
end
```

Listing 5.6: Representation of a partition in Eiffel

To prove `Req_10`, verifiers will have to prove that all `Req_11` and `Req_12` are correct.

Refinement

There is only one object-oriented relationship that allows for the weakening of preconditions and the strengthening of postconditions: redefinition. Indeed, since the Liskov principle (applied to redefinition in design by contracts) is an extension of the Hoare logic, the definition of refinement given by Abrial follows the same principle. Thus, in Eiffel, the refined requirement `Req_13` will be expressed as usually in a class A (see List. 5.7).

```
class A
feature
    Req_13
```

```
    note
        req: "[
            When the drone handle a parcel, then it shall transport it
                carefully
        ]"
    require
        parcel_handled: parcel /= Void
    do
    -- no ensure since the concept of "carefully" is too abstract
    end
end
```

Listing 5.7: Representation of a requirement Req_13 in Eiffel

A correct refinement of `Req_13` will be `Req_14` in List. 5.8.

```
class B
inherit A
  redefine
    Req_13 as Req_14
  end
feature
    Req_14
    note
        req: "[
            At any time, the drone shall avoid obstacles, fly at the
                minimum safe height and be able to stop in less than 1
                second.
        ]"
    require else
        any_time: True
    do
    ensure then
        avoid_obstacle: in_range(2) = Void -- No obstacles in a range
            of 2 meters
        min_safe_height: altitude = {DOMAIN_KNOWLEDGE}.
            minimum_safe_height
        stop_in_less_than_1_second: current_speed / -max_acceleration <
             1
    end
end
```

Listing 5.8: Representation of refinement of Req_13 from List. 5.7 in Eiffel

To prove `Req_14`, verifiers will also use preconditions and postconditions of `Req_13` (based on the composition mechanism of redefinition). If the verifier can prove `Req_14`, it is necessarily able to prove `Req_13`. This corresponds to the satisfaction propagation expressed in section 5.2.2.

Note: Using this representation, we can define a refined requirement with the same constraints as the original one, i.e., an addition.

Contradiction

Writing the specification of `Req_14` and `Req_12` (given in List. 5.9), we noticed that an analysis of the system could not simultaneously proved both requirements.

Indeed, on the one hand, `Req_14` expresses the need to prove that `min_safe_height: altitude = DOMAIN_KNOWLEDGE.minimum_safe_height` at any time. On the other hand, `Req_12` expresses the need to prove that `land: altitude < 0.01 -- m`. Since we can consider that `DOMAIN_KNOWLEDGE.minimum_safe_height > 0.01`, both requirements cannot be proved at the same time.

```
Req_14
note
    req: "[
        At any time, the drone shall avoid obstacles, fly at the
            minimum safe height and be able to stop in less than 1
            second.
    ]"
require else
    any_time: True
do
ensure then
    avoid_obstacle: in_range(2) = Void -- No obstacles in a range of 2
        meters
    min_safe_height: altitude = {DOMAIN_KNOWLEDGE}.minimum_safe_height
    stop_in_less_than_1_second: current_speed / -max_acceleration < 1
end

Req_12
note
    req: "[
       When the battery is under 10% then the drone shall switch to
           recovery mode and land quickly
    ]"
require
    battery_low: battery_level < 0.1
do
ensure
    recovery_mode: mode = recovery
    land: altitude < 0.01 -- m
end
```

Listing 5.9: Representation of two contradictory requirements Req_14 and Req_12 in Eiffel

The contradiction relationship between both requirements is not expressed here and is, in this case, not desired. However, if we want to express that these two requirements R_{14} and R_{12} shall

not be met at the same time, it is possible to create a verifier that will try to prove $R_{14} \wedge R_{12}$, as in List. 5.10:

```
Req_14_contradict_Req_12
do
    Req_14
    Req_12
end
```

Listing 5.10: Using the composition mechanism to express the contradiction

Proof by contradiction shall fail to assume `Req_14_contradict_Req_12`, since at least one of the two requirements can not be proved.

5.5 Conclusion

This chapter provides formal definitions of requirements and their relationships that improve the system compliance checking process by a mechanism based on traceability. This mechanism relies on the propagation of requirements' satisfaction through formal requirements relationships.

We define requirements as pairs of (i) a set of constraints, and (ii) a set of properties that have to be satisfied when constraints are held. By making combinations of requirements constraints sets and requirements properties sets on the one hand, and by considering commonly used n-ary relations of KAOS on the other hand, we identify eight different relationships that can be established between requirements. These relationships are formally defined, and we demonstrate how satisfaction is propagated from one requirement to another through the relationship that links them. According to the provided formalization, we show that requirements and their relationships can be implemented in languages that come with theorem provers (here the Eiffel language).

In the current status of our proposed formalization, some basic verifications can be done on requirements, using the semantics of their relationships. Here are some tracks on how this semantics can help:

- By adding relationships between requirements, we can find ill-formed requirements. Indeed, if the semantics of one given relationship cannot be respected, this can denote two things: (i) this relationship should not exist, or (ii) this requirement should be linked with this relationship but is not well formalized, or else (iii) this requirement is not correct.
- Considering a set of requirements, and thanks to their semantics definition, we can (i) deduce relationships and highlight traceability links that were not expressed before or even (ii) identify relationships such as repetitions or contradictions that reveal an issue.
- Analyzing the number of relationships linked to a requirement can help to find problems: (i) a requirement with no relationships, for example, can be a requirement that has not been correctly analyzed, (ii) too many or too few relationships of a given kind can reveal a bad analysis of requirements (too many repetitions, contradictions, not enough refinements, ...).

Future work in the verification process would provide a methodology and analytic tools to detect inconsistencies between requirements and support their verification.

Verifying that a requirement is satisfied is done by analyzing its relationships and spreading satisfaction to other related requirements.

Even with a non-formal representation of requirements, the semantics of relationships help analyze the coverage of requirements. For example, it can help engineers to complete matrix compliance by propagating the satisfaction. It can also help verify informal requirements by clarifying the semantics of the relationships (for example, by asking engineers: "Are you sure that satisfy R_1 will necessary satisfy R_2?). To ease the acceptance of the proposed semantics, another future work would be to integrate them into traceability tools.

We finally made this work practical by improving SIRCOD and RSML, and providing tools to analyze the trace links.

The work presented in this chapter is thus a step in the introduction of formal semantics into traceability, making possible to analyze requirements (detecting inconsistencies and possible simplifications) automatically and to use their relationships to reduce the set of requirements that need to be proven satisfied using classical validation approaches such as tests or human assessments.

Part III

Conclusion

Chapter 6

Conclusion and discussion

Contents

6.1 Summary of contributions

In this thesis, we propose an approach to better integrate the requirements in code. This thesis was split into three contributions (as seen in Fig. 6.1):

- First (1), we introduced the SIRCOD approach, a seamless development approach that integrates requirements as code artifacts, allowing a strong traceability through the inheritance mechanism;
- Then (2), we improve the first steps of the process by providing RSML, a language and a tool that shall address all the stakeholders' needs;
- Finally (3), we defined relationships and their semantics to support the analysis of requirements, thereby improving the two other contributions.

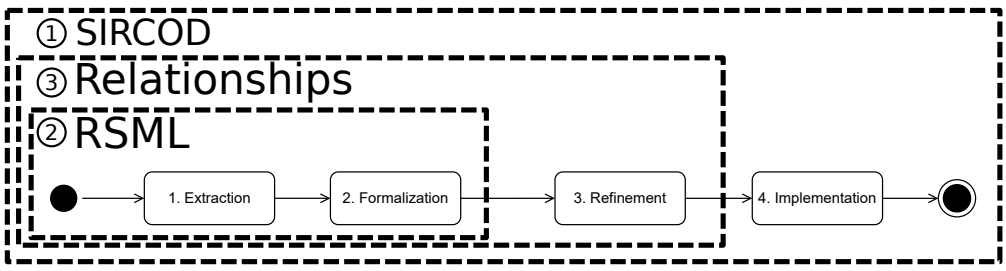


Figure 6.1: Part of each contribution to the SIRCOD process

This seamless and lean approach aims at introducing more formality in the treatment of requirements in development. Indeed, as developed in chapter 1, formality helps reduce the number of failures in software systems. In chapter 2 (and more specifically in section 2.1.1), we defined properties that shall be taken into account for good requirements approaches:

(a) **intended audience**;

(b) **traceability support**;

(c) **coverage**;

(d) **semantics definition**;

(e) and **verifiability**.

The following section will analyze each of these properties and discuss how the contributions answer to the issues related to these properties.

6.1.1 Intended audience

While most of the approaches are dedicated either for the general user, with no required skills, or expert users, we have sought to address all the stakeholders. In chapter 3, we introduced the SIRCOD approach. Based on the work made on SOOR [86], SIRCOD aims to provide an easier way to introduce formal requirements in the development life cycle. Indeed, by using of a programming language, engineers can work with formalized requirements that are expressed in a formalism they are familiar with. Besides, by keeping a link and providing tools to switch to requirements expressed in NL, we ensure that the developers can still have access to a less-formal but more affordable representation of the requirements.

However, SIRCOD does not take into account the stakeholders with no programming skills involved in RE. To overcome this issue, we propose in chapter 4 a tool and a language. RSML allows non-experts to express requirements in a constrained NL, in the same way than RELAX or Stimulus. It thus provides an interface between all the stakeholders that are involved in a project. The semantics of the language is given by SIRCOD, giving a simple way to formally express requirements (and to be sure that they are all effectively formalized).

Moreover, the use of MDE makes it possible to bridge the gap to other approaches, allowing more stakeholders to be involved in the project by providing a way to transform their usual tools in RSML (for example, we can transform SysML requirements in RSML).

6.1.2 Traceability

The traceability has been one of our main objectives throughout this thesis. In SIRCOD, a strong emphasis has been put on the need to keep the vertical traceability, from the original requirements (which we call NL requirements) to the implementation. To improve the classical traceability, we applied the well-known refinement process to the approach, providing links between formal representations of requirements. We also keep a link between NL requirements and formal requirements (with the `src` links, for example), to be able to compare, at any time, a formal requirement with its source. Some tools have been updated to improve the way to link features in Eiffel with the source, that can come from a docx document for example.

In RSML, the distinction between NL requirements and formal requirements are thinner. Indeed, an RSML formal requirement still is expressed in a kind of NL way. Traceability is, however, kept in the semantics transformation of requirements. Indeed, according to SIRCOD enactment, each RSML element is formalized in Eiffel by two features: a NL one, repeating the text as it is in RSML, and a formal one with a formal representation when it is possible, and always linked by a `src` relation to the NL one. Certainly, the traceability between Eiffel and RSML is always taken into account, using EIS, to keep a trace of the original source.

In chapter 5, we further explore the kind of relationships that can exist between requirements. This work aims to improve horizontal traceability, providing a better organization of requirements. Using these kinds of relationships and their semantics, we also ensure that if a change occurred, it will impact all the requirements that shall be considered – e.g., if a requirement is composed by another requirement that changed, the first requirement shall take the change into account. We also slightly modified SIRCOD and RSML to introduce this semantics of relationships.

6.1.3 Coverage

One of the major issues that we observed in chapter 2 is that formal approaches are often considering some kinds of requirements only. This is something understandable since it is difficult to formalize all the requirements, first because of the effort needed to introduce a formal approach to a whole system, then because some of the requirements cannot be formalized as is. E.g., *"The system must be efficient"* is difficult to formalize and, most of the time, it will be refined into other requirements that can be formalized.

This is why, to provide an approach that covers as many requirements as possible, in chapter 3, we emphasize the need to represent all requirements in Eiffel, including non-formalized ones. For those that cannot be formalized easily, we chose to represent them as deferred features, with no pre and postconditions. Even if all the developer has to do is redefine the functionality, we still require that the existence of this requirement to be taken into account during the development process.

A similar way to answer to this question is in chapter 4. Indeed, we provide a way to express NL requirements, without any control on the semantics. The user can thus express in NL requirements that he cannot express in another way. However, such requirements still are formalized, as defined in SIRCOD, with a deferred feature that needs to be redefined later, for a full implementation of the system.

One of the objectives of chapter 5 was also to provide a way to better integrate non-formalized requirements. We took the position to accept that we cannot formalize all the requirements, especially in RSML, but even if requirements are not formalized, we consider that they can be related to other requirements. So we can say that a requirement is satisfied, even if not formalized, because it is linked to several requirements that are formalized and so participate to its satisfaction. This is made thanks to the satisfaction propagation algorithm, based on the semantics of relationships, that we developed. A strong emphasis is thus put on the need to keep such relationships between requirements, not only for validation purposes but also to verify non-formal requirements.

6.1.4 Semantic definition

As we introduced in the previous section, we aimed to provide a semantic definition to each part of our contribution, to ease as much as possible the analysis of requirements. The objective of chapter 5 is to introduce semantics in requirements relationships. We do this by analyzing how two formal

requirements can be linked and thus provide a set of semantically defined relationships. Using such a semantic definition, it is easier to provide tools like the satisfaction propagation algorithm, and even they are some perspectives to explore in this work, the introduction of such semantics in the early phase of development should help to produce better software.

In chapter 3, we thus improved the approach proposed in SOOR [86] not only by providing new traceability mechanisms and tools but also by using the formal definition of the refinement and by applying it to the object-oriented development. We proposed a refinement process in SIRCOD, applied to a programming language, using the semantics of both inheritance and redefinition to provide an implementation that satisfies the original requirements. The work done in chapter 4 is also based on this semantics.

6.1.5 Verifiability

Chapter 3 aims to ease the adoption of formal approaches. We targeted this objective because of the importance, in our point of view, of the verifiability of requirements. Indeed, a formal definition of requirements, even in a programming language, with appropriate tools, shall help to avoid major failures. This is done in SIRCOD by using Autoproof, which analyzes assumptions and assertions in Eiffel to prove that the system is correct. Moreover, since it is based on the already semantically defined inheritance mechanism, the refinement process is supported by such tools. The use of redefinition to ensure that requirements are satisfied also contributes to the verifiability, at a lower level, by ensuring that the system considers the requirement (otherwise, the software could not be compiled).

As already introduced (especially in section 6.1.3), the semantic definition of relationships contributes for two points to the verifiability:

- First, even non-formalized requirements can be verified at some point using their relationships; if a requirement is the union of two other requirements, verifying the two requirements participate in verifying the first requirement.
- Then, using the satisfaction propagation algorithm, we provide a subset of requirements. Even if it is not possible to automatically verify the entire set, the amount of work needed is reduced to a subset of requirements that have to be analyzed.

Moreover, the ongoing work on the classification of NL requirements and their relationships shall lead to an easier first step of SIRCOD, providing a way to answer the question of completeness.

6.2 Perspectives

If the work done in this thesis contributes to the issues that we raised in chapter 1, there is still much more work to be done on the generalization of formal approaches.

The first perspective is on the SIRCOD approach itself. If the approach has been applied several times during this research work and is quite intuitive to developers since it uses mechanism already mastered by engineers, we need to improve the evaluation by applying it to a wide industrial use-case.

Several tools (RSML, improvement of the EIS mechanism of EiffelStudio, propagation algorithm) have been created during the research work, but it is necessary to improve them to provide a wider use of the approach. Some of them are partially implemented and still require human intervention.

At the same time, it is possible to automate some processes (e.g., when creating a traceability link between a feature and a document, user have to know the bookmark related to a part of the document – the possibility to select a specific part in a visual editor shall ease this step).

The SIRCOD approach was developed in the context of Eiffel and the EiffelStudio IDE. Besides the work needed on other IDE to integrate the developed tools, consideration should be given to how the approach fits into other programming languages. In particular, the redefinition mechanism of SIRCOD was thought for Eiffel, that allows multiple inheritance, and it would be interesting to see how to integrate it in object approaches that do not allow this multiple inheritance. This perspective would help to wider use of the approach.

Furthermore, the verification process is for now based on proof tools and design by contracts. Nevertheless, it is indisputable that the most widely used verification approaches in the industry are tests, regardless of their form. One of the most important remaining work in our perspective is analyzing how we can integrate tests to SIRCOD, including test generation.

One of the avenues we are currently exploring is integrating the approach in a DevOps approach (and, more specifically, into continuous integration). Indeed, since we provide artifacts than can be used to verify the system, it seems logical for us to explore ways to integrate the approach in existing workflows.

From the RSML point of view, the language itself can be improved to provide better abstractions. Indeed, for now, formal requirements are still very close to state expressions, while other existing languages such as Stimulus provide a better abstraction. Supplying ways to customize the language and to encapsulate the formality into expressions, shall lead to a wider acceptance of the this approach. In the context of this thesis, the work done by Antoine Gambier was to improve RSML by analyzing existing requirements set to find the habits and integrate them in RSML. However, a wider study should be done to provide more complete results.

Another improvement can be made on the bridges from and to other paradigms. If only several ones were developed in this thesis's context, more transformations (and more complete ones) should ease integrating the approach in existing tools. For instance, we plan to add transformations to other approaches, like KAOS or Event-B, to confirm our approach's scalability in a more global context. Adding new bridges is relatively simple, thanks to the tools of MDE, but the alignment of models themselves can be quite difficult. However, since the representation of requirements in the form of pairs of assumptions and assertions is quite common, we hope that such a model alignment will not be too daunting as a task.

Moreover, there are still many perspectives in the transformation from textual requirements document to more formal approaches, such as RSML. The use of NLP can (and shall) be explored for this purpose. By considering a textual document has a model, we have to define a precise syntax of requirements document to ease the generation of artifacts. This perspective could also benefit RSML itself, by extending the language throuh some kind of standard, like EARS.

Finally, the work done on relationships is still preliminary. Their semantics shall be further explored. For instance, we consider in this work that a requirement is satisfied or not. However, there may be nuances, and it can be interesting to analyze how this can be propagated through relationships.

Using the existing semantics, one of the perspectives that we will explore is detecting existing relationships. Indeed, since we defined semantics for the relationships based on the interdependencies between requirements, it is possible to deduce relationships when they are not explicit.

These perspectives, which are still numerous, highlight the work remaining in RE to propose an approach that would be perfectly integrated into users' habits. This is all the more difficult

since it is not in developers' habits to rigorously analyze requirements (except, thankfully, in the case of critical systems). Therefore, our job is to provide tools that are increasingly accessible to developers and encourage best practices, especially through education.

Appendix

Appendix A

RSML

Contents

A.1 RSML complete metamodel

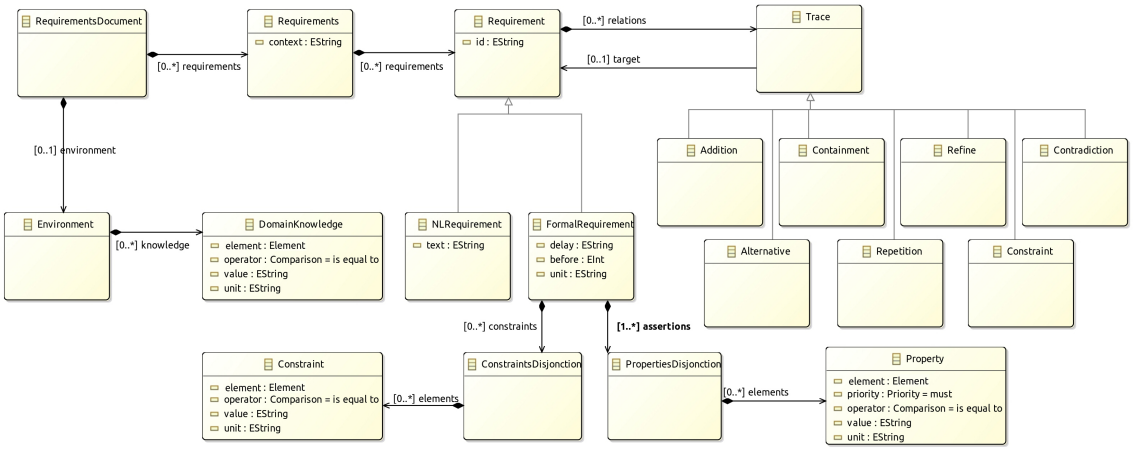


Figure A.1: RSML complete metamodel

A.2 RSML grammar

⟨*requirements document*⟩ ::= ⟨*environment*⟩? ⟨*requirements set*⟩*

⟨*environment*⟩ ::= (‘`Env`’ | ‘`Environment`’) ‘`:`’ ⟨*newline*⟩ ⟨*domain knowledge*⟩+

⟨*domain knowledge*⟩ ::= ‘`-`’ (⟨*simple definition*⟩ | ⟨*interval definition*⟩ | ⟨*range definition*⟩) ‘`.`’ ⟨*newline*⟩

⟨*simple definition*⟩ ::= ⟨*element*⟩ ⟨*comparison operator*⟩ ⟨*value*⟩ (‘`[`’ ⟨*unit*⟩ ‘`]`’)?

⟨*interval definition*⟩ ::= ⟨*element*⟩ (‘`is in`’ | ‘`is out`’) ⟨*value*⟩ ‘`and`’ ⟨*value*⟩

⟨*range definition*⟩ ::= ⟨*element*⟩ ‘`can be`’ ⟨*value*⟩ (‘`,`’ ⟨*value*⟩)*

⟨*element*⟩ ::= ⟨*word*⟩+

⟨*comparison operator*⟩ ::= ‘`is`’ (‘`equal to`’)?
| ‘`less than`’
| ‘`less or equal to`’
| ‘`greater than`’
| ‘`greater or equal to`’
| ‘`different to`’

⟨*value*⟩ ::= ⟨*number*⟩ | ⟨*state*⟩ | ⟨*element*⟩

⟨*unit*⟩ ::= ⟨*word*⟩ (‘`^`’ ⟨*number*⟩)? (‘`/`’ ⟨*unit*⟩)?

⟨*requirements set*⟩ ::= ⟨*word*⟩+ ‘`:`’ ⟨*newline*⟩ ⟨*requirement*⟩+

⟨*requirement*⟩ ::= ‘`[`’ ⟨*id*⟩ ‘`]`’ (‘`(`’ ⟨*relationships*⟩‘`)`’)?
(⟨*natural language requirement*⟩ | ⟨*formal requirement*⟩) ⟨*newline*⟩

⟨*id*⟩ ::= ⟨*digit*⟩ (‘`.`’ ⟨*id*⟩)?

⟨*natural language requirement*⟩ ::= ⟨*single line text*⟩ | ⟨*multiline text*⟩

⟨*single line text*⟩ ::= ‘`"`’ .* ‘`"`’

⟨*multiline text*⟩ ::= ‘`"[`’ (. | ⟨*newline*⟩)* ‘`]"`’

⟨*formal requirement*⟩ ::= (⟨*properties disjunction*⟩ (‘`and`’ ⟨*properties disjunction*⟩)?) ‘`.`’
| ‘`When`’ ⟨*constraints disjunction*⟩ (‘`and`’ ⟨*constraints disjunction*⟩)? ‘`then`’
‘`immediately`’ ⟨*properties disjunction*⟩ (‘`and`’ ⟨*properties disjunction*⟩)? ‘`.`’
| When’ ⟨*constraints disjunction*⟩ (‘`and`’ ⟨*constraints disjunction*⟩)? ‘`then`’
‘`eventually`’ ⟨*properties disjunction*⟩ (‘`and`’ ⟨*properties disjunction*⟩)?
(‘`within`’ ⟨*integer*⟩ ‘`[`’ ⟨*Unit*⟩ ‘`]`’)?) ‘`.`’

⟨*constraints disjunction*⟩ ::= ⟨*constraint*⟩ (‘`or`’ ⟨*constraint*⟩)?

⟨*constraint*⟩ ::= ⟨*element*⟩ ⟨*comparison operator*⟩ ⟨*value*⟩ (‘`[`’ ⟨*unit*⟩ ‘`]`’)?

⟨*properties disjunction*⟩ ::= ⟨*property*⟩ (‘`or`’ ⟨*property*⟩)?

⟨*property*⟩ ::= ⟨*element*⟩ ⟨*priority*⟩ ⟨*modal comparison operator*⟩ ⟨*value*⟩ (‘`[`’ ⟨*unit*⟩ ‘`]`’)?

⟨*modal comparison operator*⟩ ::= ‘`be equal to`’
| ‘`be less than`’
| ‘`be greater than`’
| ‘`be less or equal to`’
| ‘`be greater or equal to`’
| ‘`be not equal to`’

⟨*relationships*⟩ ::= ⟨*relationship kind*⟩ ⟨*id*⟩ (‘`and`’ ⟨*relationships*⟩)?

⟨*relationship kind*⟩ ::= ‘`refines`’
| ‘`part of`’
| ‘`is an addition of`’
| ‘`is a constrained version of`’
| ‘`contradicts`’
| ‘`repeats`’
| ‘`is an alternative of`’
| ‘`trace`’

Appendix B

Satisfaction propagation

Contents

B.1 Python implementation of the minimum set algorithm

```
#!/usr/bin/env python3
import os

class Requirement:
        def __init__(self,rid):
                self.rid = rid
                self.adds = []
                self.constraints = []
                self.repeats = []
                self.canBeDecomposedInto = []
                self.parts = []
                self.refines = []
                self.contradicts = []
                self.alternatives = []
                self.sat = []
                self.satBy = []
                self.satisfiableByOther = False

        def __str__(self):
                ret= "\tR"+str(self.rid)+";\n"
                for r in self.adds:
                        ret+= "\tR"+str(self.rid)+" -> R"+str(r.rid)+"
                            [label=\"adds\"];\n"
```

```
                for r in self.constraints:
                        ret+= "\tR"+str(self.rid)+" -> R"+str(r.rid)+"
                            [label=\"constraints\"];\n"
                for r in self.repeats:
                        ret+= "\tR"+str(self.rid)+" -> R"+str(r.rid)+"
                            [label=\"repeats\"];\n"
                for r in self.canBeDecomposedInto:
                        ret+= "\tR"+str(r.rid)+" -> R"+str(self.rid)+"
                            [label=\"parts of\"];\n"
                for r in self.parts:
                        ret+= "\tR"+str(r.rid)+" -> R"+str(self.rid)+"
                            [label=\"parts of\"];\n"
                for r in self.refines:
                        ret+= "\tR"+str(self.rid)+" -> R"+str(r.rid)+"
                            [label=\"refines\"];\n"
                for r in self.contradicts:
                        ret+= "\tR"+str(self.rid)+" -> R"+str(r.rid)+"
                            [label=\"contradicts\"];\n"
                for r in self.alternatives:
                        ret+= "\tR"+str(r.rid)+" -> R"+str(self.rid)+"
                            [label=\"alternative to\"];\n"
                return ret

        def computeSat(self,reqs):
                for r in self.adds:
                        if not r in self.sat:
                                self.addSat(r)
                for r in self.repeats:
                        if not r in self.sat:
                                self.addSat(r)
                        if not self in r.sat:
                                r.addSat(self)
                for r in self.constraints:
                        if not self in r.sat:
                                r.addSat(self)
                decomposed = []
                for r in self.canBeDecomposedInto:
                        if not r in self.sat:
                                r.addSat(self)
                        if not r in decomposed:
                                decomposed.append(r)
                rid = ""
                for r in self.parts:
                        if not r in self.sat:
                                self.addSat(r)
                for r in self.refines:
```

```
                        if not r in self.sat:
                                self.addSat(r)
                for r in self.alternatives:
                        if not self in r.sat:
                                r.addSat(self)

        def addSat(self, r):
                if not r in self.sat:
                        self.sat.append(r)
                if not self in r.satBy:
                        r.satBy.append(self)


######################## Compute minimum set ##########################

def naiveMinimumSat(reqs,verb=False):
        for r in reqs:
                if r.satBy != []:
                        r.satisfiableByOther = True
        ret = []
        for r in reqs:
                if not r.satisfiableByOther:
                        ret.append(r)
        return ret

def mininumSat(reqs,verb=False):
        ret = leafSat(reqs)
        eliminateUnnecessaryReqs(ret,reqs)
        return ret

def leafSat(reqs):
        for r in reqs:
                if not r.satisfiableByOther: # It has already been
                    visited
                        visit(r, [r])
        return [entry for entry in reqs if not entry.satisfiableByOther
            ]

def visit(r, alreadyVisited):
        for s in r.sat:
                if not s in alreadyVisited:
                        s.satisfiableByOther = True
                        visit(s, alreadyVisited + [s])

def eliminateUnnecessaryReqs(ret,reqs):
        toRemove = []
```

```
        for r in ret:
                if len(r.sat) == 1:
                        if r in r.sat[0].alternatives:
                                for rs in r.sat[0].satBy:
                                        if rs.satisfiableByOther or not
                                            rs in toRemove:
                                                toRemove.append(r)
        for r in toRemove:
                if r in ret:
                        ret.remove(r)

########################## Printing in Dot ##########################

def dotSat(reqs):
        ret = "digraph RequirementsPropagation {\n"
        for r in reqs:
                ret += "\tR"+str(r.rid)+";\n"
                for s in r.sat:
                        ret += "\tR"+str(r.rid)+" -> R"+str(s.rid)+" [
                            label=\"satisfies\"];\n"
        for r in mininumSat(reqs):
                ret += "\tR" + str(r.rid)+ " [style=\"bold, filled\",
                    fillcolor=\"gray\"];\n"
        return ret+"}"

def dot(reqs):
        ret = "digraph Requirements {\n"
        for r in reqs:
                ret += str(r)
        for r in mininumSat(reqs):
                ret += "\tR" + str(r.rid)+ " [style=\"bold, filled\",
                    fillcolor=\"gray\"];\n"
        return ret+"}"

########################## Main ##########################

r0 = Requirement(0)
r1 = Requirement(1)
r2 = Requirement(2)
r3 = Requirement(3)
r4 = Requirement(4)
r5 = Requirement(5)
r6 = Requirement(6)
r7 = Requirement(7)
r8 = Requirement(8)
r9 = Requirement(9)
```

```
r10 = Requirement(10)
r11 = Requirement(11)
r12 = Requirement(12)
r13 = Requirement(13)
r14 = Requirement(14)
r81 = Requirement(81)
r82 = Requirement(82)

reqs = [r0, r1, r2, r3, r4, r5, r6, r7, r8, r9, r10, r12, r13, r14, r81
    , r82]

r0.adds.append(r2)
r0.adds.append(r3)
r2.adds.append(r3)
r2.adds.append(r4)
r1.adds.append(r4)
r1.canBeDecomposedInto.append(r7)
r1.canBeDecomposedInto.append(r8)
r0.constraints.append(r5)
r6.constraints.append(r2)
r5.refines.append(r2)
r81.refines.append(r8)
r82.refines.append(r8)
r8.repeats.append(r9)
r7.repeats.append(r14)
r12.adds.append(r2)
r11.adds.append(r10)
r10.adds.append(r13)
r13.adds.append(r11)

for r in reqs:
        r.computeSat(reqs)

print(dot(reqs))

mfile = open('requirementsDot.dot','w')
mfile.write(dot(reqs))
mfile.close()
os.system("dot -Tps requirementsDot.dot -o requirementsDot.pdf")

mfile = open('requirementsSat.dot','w')
mfile.write(dotSat(reqs))
mfile.close()

os.system("dot -Tps requirementsSat.dot -o requirementsSat.pdf")
```

```
print ("[")
for r in mininumSat(reqs, True):
        print (" - R"+str(r.rid))
print ("]")
```

Bibliography

[1] Reiner Hähnle, Kristofer Johannisson, and Aarne Ranta. An Authoring Tool for Informal and Formal Requirements Specifications. In Ralf-Detlef Kutsche and Herbert Weber, editors, *Fundamental Approaches to Software Engineering*, number 2306 in Lecture Notes in Computer Science, pages 233–248. Springer Berlin Heidelberg, April 2002. DOI: 10.1007/3-540-45923-5_16.

[2] Jon Whittle, Pete Sawyer, Nelly Bencomo, Betty H. C. Cheng, and Jean-Michel Bruel. RELAX: Incorporating Uncertainty into the Specification of Self-Adaptive Systems. In *2009 17th IEEE International Requirements Engineering Conference*, pages 79–88, 2009.

[3] Pierre Bourque and Richard E. Fairley, editors. *Guide to the Software Engineering Body of Knowledge (SWEBOK(R)): Version 3.0*. IEEE Computer Society Press, 3rd edition, 2014.

[4] Richard E Fairley. *Managing and leading software projects*. John Wiley & Sons, 2011.

[5] IEEE Computer Society Software Engineering Standards Committee and IEEE-SA Standards Board. IEEE Standard for Software Reviews and Audits. *IEEE Std 1028-2008 (Revision of IEEE Std 1028-1998)*, 2008.

[6] ISO/IEC International Standard - Software Engineering – Software Life Cycle Processes – Maintenance. *ISO/IEC 14764:2006*, 2006.

[7] IEEE Computer Society Software Engineering Standards Committee and IEEE-SA Standards Board. ISO/IEC/IEEE International Standard - Systems and Software Engineering – Vocabulary. *IEEE Std 24765-2010*, 2010.

[8] Bertrand Meyer, Jean-Michel Bruel, Sophie Ebersold, Florian Galinier, and Alexandr Naumchev. Towards an anatomy of software requirements. In Manuel Mazzara, Jean-Michel Bruel, Bertrand Meyer, and Alexander Petrenko, editors, *Software Technology: Methods and Tools*, pages 10–40, Cham, 2019. Springer International Publishing.

[9] Mich Luisa, Franch Mariangela, and Novi Inverardi Pierluigi. Market research for requirements analysis using linguistic tools. *Requirements Engineering*, 9(1):40–56, 2004.

[10] Mohamad Kassab, Colin Neill, and Phillip Laplante. State of practice in requirements engineering: contemporary data. *Innovations in Systems and Software Engineering*, 10(4):235–241, 2014.

[11] Michael Jackson and Pamela Zave. Deriving specifications from requirements: an example. In *1995 17th Int. Conf. on Software Engineering*, pages 15–15. IEEE, 1995.

[12] M. Jackson. *Problem Frames: Analyzing and Structuring Software Development Problems.* Addison-Wesley., 2001.

[13] Ian Sommerville and Pete Sawyer. *Requirements engineering: a good practice guide.* John Wiley & Sons, Inc., 1997.

[14] Axel Van Lamsweerde. *Requirements engineering: From system goals to UML models to software*, volume 10. Chichester, UK: John Wiley & Sons, 2009.

[15] Matt Lake. Epic failures: 11 infamous software bugs — Computerworld, 09/2010.

[16] Joseph P. Elm, Dennis Goldenson, Khaled El Emam, Nichole Donitelli, and Angelica Neisa. A survey of systems engineering effectiveness. Technical Report CMU/SEI-2008-SR-034, National Defense Industrial Association, Carnegie mellon, 2008.

[17] Matthias Jarke. Requirements Tracing. *Commun. ACM*, 41(12):32–36, December 1998.

[18] Francisco AC Pinheiro. Requirements traceability. In *Perspectives on software requirements*, pages 91–113. Springer, 2004.

[19] Jean-Michel Bruel, Sophie Ebersold, Florian Galinier, Manuel Mazzara, Alexandr Naumchev, and Bertrand Meyer. The role of formalism in system requirements. *ACM Comput. Surv.*, 54(5), May 2021.

[20] IEEE Computer Society Software Engineering Standards Committee and IEEE-SA Standards Board. IEEE Recommended Practice for Software Requirements Specifications. Institute of Electrical and Electronics Engineers, 1998.

[21] Iso/iec/ieee international standard - systems and software engineering – life cycle processes – requirements engineering. *ISO/IEC/IEEE 29148:2018(E)*, pages 1–104, Nov 2018.

[22] INCOSE. *SE Vision 2025.* 2014.

[23] Michel dos Santos Soares, Jos Vrancken, and Alexander Verbraeck. User Requirements Modeling and Analysis of Software-intensive Systems. *J. Syst. Softw.*, 84(2):328–339, 2011.

[24] Richard E. Schneider and Dennis M. Buede. 6.3.1 Properties of a High Quality Informal Requirements Document. *INCOSE International Symposium*, 10(1):352–359, 2000.

[25] R. Paige and J. Ostroff. The Single Model Principle. In *Proceedings of the Fifth IEEE International Symposium on Requirements Engineering*, RE '01, pages 292–, Washington, DC, USA, 2001. IEEE Computer Society.

[26] M. Broy, M. Feilkas, M. Herrmannsdoerfer, S. Merenda, and D. Ratiu. Seamless Model-Based Development: From Isolated Tools to Integrated Model Engineering Environments. *Proceedings of the IEEE*, 98(4):526–545, April 2010.

[27] Jean-Marc Jézéquel and Bertrand Meyer. Design by contract: The lessons of ariane. *Computer*, 30(1):129–130, 1997.

[28] J. M. Carrillo de Gea, J. Nicolás, J. L. F. Alemán, A. Toval, C. Ebert, and A. Vizcaíno. Requirements Engineering Tools. *IEEE Software*, 28(4):86–91, 2011.

[29] IBM. *Rational Doors*. 2015.

[30] Dassault Systems. *Catia Reqtify*. 2016.

[31] Axel van Lamsweerde. Goal-oriented requirements engineering: a guided tour. In *Proc. 5th IEEE Int. Symposium on Requirements Engineering*, pages 249–262, 2001.

[32] Eric SK Yu. Towards modelling and reasoning support for early-phase requirements engineering. In *Requirements Engineering, 1997, Proc. of the 3rd IEEE Int. Symposium on*, pages 226–235. IEEE, 1997.

[33] Respect-it. *Objectiver V3*. 2011.

[34] Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, New York, NY, USA, 1st edition, 2010.

[35] Abderrahman Matoussi, Frédéric Gervais, and Régine Laleau. An Event-B formalization of KAOS goal refinement patterns. Technical Report Tech. Rep. TRLACL-2010-1, LACL, University of Paris-Est, 2010.

[36] Object Management Group. OMG Systems Modeling Language (OMG SysML™), V1.0, 2007. OMG Document Number: formal/2007-09-01 Standard document URL: `http://www.omg.org/spec/SysML/1.0/PDF`.

[37] Object Management Group (OMG). UML 2.5, March 2015. http://www.omg.org/spec/UML/2.5/.

[38] Jonas Helming, Maximilian Koegel, Florian Schneider, Michael Haeger, Christine Kaminski, Bernd Bruegge, and Brian Berenbach. Towards a unified Requirements Modeling Language. In *2010 5th Int. Workshop on Requirements Engineering Visualization*, pages 53–57, September 2010.

[39] B. Berenbach, F. Schneider, and H. Naughton. The use of a requirements modeling language for industrial applications. In *2012 20th IEEE International Requirements Engineering Conference (RE)*, pages 285–290, September 2012.

[40] Object Management Group (OMG). MARTE 1.1, June 2011. http://www.omg.org/spec/MARTE/1.1/.

[41] A. Albinet, S. Begoc, J. L. Boulanger, O. Casse, I. Dal, H. Dubois, F. Lakhal, D. Louar, M. A. Peraldi-Frati, Y. Sorel, and others. The MeMVaTEx methodology: from requirements to models in automotive application design. In *4th European Congress ERTS (Embedded Real Time Software), Toulouse, France*, 2008.

[42] International Telecommunication Union (ITU-T). Z.151 : User Requirements Notation (URN) - Language definition.

[43] jUCMNav, 2017.

[44] Dines Bjørner and Cliff B. Jones, editors. *The Vienna Development Method: The Meta-Language*, volume 61 of *LNCS*. Springer-Verlag, 1978.

[45] overturetool.org. Overture tool: Formal modelling in vdm.

[46] E. Durr and J. van Katwijk. Vdm++, a formal specification language for object-oriented designs. In *CompEuro 1992 Proceedings Computer Systems and Software Engineering*, pages 214–219, May 1992.

[47] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.

[48] Jean-Raymond Abrial, Stephen Schuman, and Bertrand Meyer. A specification language. In *On the Construction of Programs*, pages 343–410. Cambridge University Press, 1980.

[49] Jean-Raymond Abrial. *The B-book: assigning programs to meanings*. Cambridge University Press, 2005.

[50] Thuy Nguyen. Verification of Behavioural Requirements for Complex Systems with FORM-L, a MODELICA Extension. In *26th International Conference on Software & Systems Engineering and their Applications*, EDF R&D, 6 quai Watier, 78110 Chatou, FRANCE, 2015.

[51] Sven Erik Mattsson, Hilding Elmqvist, and Martin Otter. Physical system modeling with Modelica. *Control Engineering Practice*, 6(4):501–510, 1998.

[52] Chetan Arora, Mehrdad Sabetzadeh, Lionel Briand, and Frank Zimmer. Extracting domain models from natural-language requirements: approach and industrial evaluation. In *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, pages 250–260. ACM, 2016.

[53] Mohd Ibrahim and Rodina Ahmad. Class diagram extraction from textual requirements using natural language processing (nlp) techniques. In *2010 Second International Conference on Computer Research and Development*, pages 200–204. IEEE, 2010.

[54] MG Ilieva and Olga Ormandjieva. Automatic transition of natural language software requirements specification into formal presentation. In *Int. Conf. on Application of Natural Language to Information Systems*, pages 392–397. Springer, 2005.

[55] C. Arora, M. Sabetzadeh, L. Briand, and F. Zimmer. Automated checking of conformance to requirements templates using natural language processing. *IEEE Transactions on Software Engineering*, 41(10):944–968, 2015.

[56] S. Abualhaija, C. Arora, M. Sabetzadeh, L. C. Briand, and E. Vaz. A machine learning-based approach for demarcating requirements in textual specifications. In *2019 IEEE 27th International Requirements Engineering Conference (RE)*, pages 51–62, 2019.

[57] D. Aceituna, H. Do, G. S. Walia, and S. W. Lee. Evaluating the use of model-based requirements verification method: A feasibility study. In *Workshop on Empirical Requirements Engineering (EmpiRE 2011)*, pages 13–20, 2011.

[58] Feng-Lin Li, Jennifer Horkoff, Alexander Borgida, Giancarlo Guizzardi, Lin Liu, and John Mylopoulos. From Stakeholder Requirements to Formal Specifications Through Refinement. In Samuel A. Fricker and Kurt Schneider, editors, *Requirements Engineering: Foundation for Software Quality*, Lecture Notes in Computer Science, pages 164–180. Springer International Publishing, March 2015. DOI: 10.1007/978-3-319-16101-3_11.

[59] Sean Bechhofer. OWL: Web Ontology Language. In LING LIU and M. TAMER ÖZSU, editors, *Encyclopedia of Database Systems*, pages 2008–2009. Springer US, 2009. DOI: 10.1007/978-0-387-39940-9_1073.

[60] William Scott and S. C. Cook. *A Context-free Requirements Grammar to Facilitate Automatic Assessment*. PhD thesis, UniSA, 2004.

[61] Object Management Group (OMG). OCL 2.4, February 2014. http://www.omg.org/spec/OCL/2.4/.

[62] Jordi Cabot, Robert Clarisó, and Daniel Riera. UMLtoCSP: A Tool for the Formal Verification of UML/OCL Models Using Constraint Programming. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 547–548, New York, NY, USA, 2007. ACM.

[63] W. Ahrendt, T. Baar, B. Beckert, M. Giese, E. Habermalz, R. Hähnle, W. Menzel, and P. H. Schmitt. The Key Tool. *Software & Systems Modeling*, 4(1):32–54, 2005.

[64] Seong-ick Moon, Kwang H. Lee, and Doheon Lee. Fuzzy branching temporal logic. *IEEE Transactions on Systems, Man, and Cybernetics, Part B*, 34(2):1045–1055, April 2004.

[65] Bertrand Jeannet and Fabien Gaucher. Debugging real-time systems requirements: simulate the "what" before the "how". In *Embedded World Conf., Nürnberg, Germany*, 2015.

[66] Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. A Conservative Extension of Synchronous Data-flow with State Machines. In *Proc. of the 5th ACM Int. Conference on Embedded Software*, EMSOFT '05, pages 173–182, New York, NY, USA, 2005. ACM.

[67] Pascal Raymond, Yvan Roux, and Erwan Jahier. Specifying and Executing Reactive Scenarios With Lutin. *Electronic Notes in Theoretical Computer Science*, 203(4):19–34, 2008.

[68] Martin Fowler and Jim Highsmith. The agile manifesto. *Software Development*, 9(8):28–35, 2001.

[69] Bertrand Meyer. *Agile!: The Good, the Hype and the Ugly*. Springer Science & Business Media, 2014.

[70] Dan North and others. Introducing BDD. *Better Software, March*, 2006.

[71] Bertrand Meyer. Multirequirements. *Modelling and Quality in Requirements Engineering (Martin Glinz Festscrhift)*, 2013.

[72] Antonio Cicchetti, Federico Ciccozzi, and Alfonso Pierantonio. Multi-view approaches for software and system modelling: a systematic literature review. *Software & Systems Modeling*, pages 1–27, 2019.

[73] Hugo Bruneliere, Erik Burger, Jordi Cabot, and Manuel Wimmer. A feature-based survey of model view approaches. *Software & Systems Modeling*, 18(3):1931–1952, 2019.

[74] Eclipse Foundation. Sirius.

[75] Benoit Combemale, Julien Deantoni, Benoit Baudry, Robert B. France, Jean-Marc Jézéquel, and Jeff Gray. Globalizing Modeling Languages. *Computer*, pages 10–13, June 2014.

[76] A Wayne Wymore. *Model-based systems engineering*, volume 3. CRC press, 2018.

[77] Philip Langer, Konrad Wieland, Manuel Wimmer, Jordi Cabot, et al. EMF Profiles: A Lightweight Extension Approach for EMF Models. *Journal of Object Technology*, 11(1):1–29, 2012.

[78] Hugo Bruneliere, Jokin Garcia Perez, Manuel Wimmer, and Jordi Cabot. EMF Views: A View Mechanism for Integrating Heterogeneous Models. In *International Conference on Conceptual Modeling*, pages 317–325. Springer, 2015.

[79] Fahad R. Golra, Antoine Beugnard, Fabien Dagnat, Sylvain Guerin, and Christophe Guychard. Addressing Modularity for Heterogeneous Multi-model Systems Using Model Federation. In *Companion Proceedings of the 15th International Conference on Modularity*, MODULARITY Companion 2016, pages 206–211, New York, NY, USA, 2016. ACM.

[80] Fahad R. Golra, Antoine Beugnard, Fabien Dagnat, Sylvain Guerin, and Christophe Guychard. Continuous Requirements Engineering using Model Federation. *RE:Next! Track at 24th IEEE International Requirements Engineering Conference 2016*, 2016.

[81] Fahad Rafique Golra, Fabien Dagnat, Jeanine Souquières, Imen Sayar, and Sylvain Guerin. Bridging the gap between informal requirements and formal specifications using model federation. In *International Conference on Software Engineering and Formal Methods*, pages 54–69. Springer, 2018.

[82] Ernst Sikora, Bastian Tenbergen, and Klaus Pohl. Industry needs and research directions in requirements engineering for embedded systems. *Requirements Engineering*, 17(1):57–78, Mar 2012.

[83] Gauthier Fanmuy, Anabel Fraga, and Juan Llorens. Requirements verification in the industry. In Omar Hammami, Daniel Krob, and Jean-Luc Voirin, editors, *Complex Systems Design & Management*, pages 145–160, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[84] Jonathan S. Ostroff and Faraz Ahmadi Torshizi. Testable Requirements and Specifications. In Yuri Gurevich and Bertrand Meyer, editors, *Tests and Proofs*, number 4454 in Lecture Notes in Computer Science, pages 17–40. Springer Berlin Heidelberg, February 2007. DOI: 10.1007/978-3-540-73770-4_2.

[85] Alexandr Naumchev, Bertrand Meyer, Manuel Mazzara, Florian Galinier, Jean-Michel Bruel, and Sophie Ebersold. AutoReq: Expressing and verifying requirements for control systems. *Journal of Computer Languages*, 51:131 – 142, 2019.

[86] Alexandr Naumchev. *Exigences orientées objets dans un cycle de vie continu*. PhD thesis, Université de Toulouse, Université Toulouse III-Paul Sabatier, 2019.

[87] Matthew B Dwyer, George S Avrunin, and James C Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No. 99CB37002)*, pages 411–420. IEEE, 1999.

[88] Julian Tschannen, Carlo A. Furia, Martin Nordio, and Nadia Polikarpova. AutoProof: Auto-Active Functional Verification of Object-Oriented Programs. In Christel Baier and Cesare Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, number 9035 in Lecture Notes in Computer Science, pages 566–580. Springer, April 2015.

[89] Donald Ervin Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.

[90] Pamela Zave and Michael Jackson. Four dark corners of requirements engineering. *ACM Trans. Softw. Eng. Methodol.*, 6(1):1–30, January 1997.

[91] Barbara H Liskov and Jeannette M Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(6):1811–1841, 1994.

[92] Frédéric Boniol and Virginie Wiels. The Landing Gear System Case Study. In Frédéric Boniol, Virginie Wiels, Yamine Ait Ameur, and Klaus-Dieter Schewe, editors, *ABZ 2014: The Landing Gear Case Study*, number 433 in Communications in Computer and Information Science, pages 1–18. Springer International Publishing, June 2014.

[93] Christof Ebert and Michael Jastram. Reqif: Seamless requirements interchange format between business partners. *IEEE software*, 29(5):82–87, 2012.

[94] Emmanuel Letier. *Reasoning about agents in goal-oriented requirements engineering*. PhD thesis, PhD thesis, Université catholique de Louvain, 2001.

[95] Carlos Solis and Xiaofeng Wang. A study of the characteristics of behaviour driven development. In *Software Engineering and Advanced Applications (SEAA), 2011 37th EUROMICRO Conference*, pages 383–387. IEEE, 2011.

[96] Matt Wynne, Aslak Hellesoy, and Steve Tooke. *The cucumber book: behaviour-driven development for testers and developers*. Pragmatic Bookshelf, 2017.

[97] Krzysztof Czarnecki and Simon Helsen. Classification of model transformation approaches. In *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, volume 45, pages 1–17. USA, 2003.

[98] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. ATL: A model transformation tool. *Science of computer programming*, 72(1-2):31–39, 2008.

[99] Jean-Marc Jézéquel, Olivier Barais, and Franck Fleurey. Model driven language engineering with kermeta. In *International Summer School on Generative and Transformational Techniques in Software Engineering*, pages 201–221. Springer, 2009.

[100] Florian Galinier. A DSL for Requirements in the Context of a Seamless Approach. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, pages 932–935, New York, NY, USA, 2018. ACM.

[101] Hazeline U Asuncion, Arthur U Asuncion, and Richard N Taylor. Software traceability with topic modeling. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, volume 1, pages 95–104. IEEE, 2010.

[102] Wentao Wang, Nan Niu, Hui Liu, and Zhendong Niu. Enhancing automated requirements traceability by resolving polysemy. In *2018 IEEE 26th International Requirements Engineering Conference (RE)*, pages 40–51. IEEE, 2018.

[103] Salome Maro, Jan-Philipp Steghöfer, Jane Hayes, Jane Cleland-Huang, and Miroslaw Staron. Vetting automatically generated trace links: what information is useful to human analysts? In *2018 IEEE 26th International Requirements Engineering Conference (RE)*, pages 52–63. IEEE, 2018.

[104] Neta Aizenbud-Reshef, Brian T Nolan, Julia Rubin, and Yael Shaham-Gafni. Model traceability. *IBM Systems Journal*, 45(3):515–526, 2006.

[105] Richard F Paige, Nikolaos Drivalos, Dimitrios S Kolovos, Kiran J Fernandes, Christopher Power, Goran K Olsen, and Steffen Zschaler. Rigorous identification and encoding of trace-links in model-driven engineering. *Software & Systems Modeling*, 10(4):469–487, 2011.

[106] Stale Walderhaug, Ulrik Johansen, Erlend Stav, and Jan Aagedal. Towards a generic solution for traceability in mdd. In *ECMDA Traceability Workshop (ECMDA-TW)*, pages 41–50. Citeseer, 2006.

[107] Alessandro Orso, Taweesup Apiwattanapong, James Law, Gregg Rothermel, and Mary Jean Harrold. An empirical comparison of dynamic impact analysis algorithms. In *Proceedings. 26th International Conference on Software Engineering*, pages 491–500. IEEE, 2004.

[108] Florian Galinier, Jean-Michel Bruel, Sophie Ebersold, and Bertrand Meyer. Seamless integration of multirequirements in complex systems. In *2017 IEEE 25th International Requirements Engineering Conference Workshops (REW)*, pages 21–25. IEEE, 2017.

List of Abbreviations

BDD	Behavior-Driven Development	25, 26
CAS	Complex Adaptive System	21
DSL	Domain Specific Language	27, 60, 144
DSML	Domain Specific Modeling Language	27, 56, 60, 142, 144
EAST-ADL2	Electronic Architecture & Software Tools - Architecture Description Language	15
EIS	Eiffel Information System	37–39, 107, 139
EMF	Eclipse Modeling Framework	27, 65, 145
FBTL	Fuzzy Branching Temporal Logic	22
GORE	Goal-Oriented Requirements Engineering	12, 67, 68, 73, 145
GRL	Goal-oriented Requirement Language	15
IDE	Integrated Development Environment	63
INCOSE	International Council on Systems Engineering	10
LAS	London Ambulance Service	68
MARTE	Modeling and Analysis of Real Time and Embedded systems	15
MBSE	Model-Based Systems Engineering	27

MDE	Model-Driven Engineering	9, 27, 28, 55, 56, 60, 106, 109, 155
NL	Natural Language	5, 9, 11–13, 15, 18–26, 28, 33–36, 40, 44, 46, 47, 52, 53, 57, 58, 60, 62, 65–67, 74, 106–108, 138, 142
NLP	Natural Language Processing	18, 19, 23, 29, 53, 67, 109, 136, 155
OCL	Object Constraint Language	16, 21, 93
PLTL	Propositional Linear Temporal Logic	22
RE	Requirements Engineering	4, 9, 26, 33, 73, 106, 109, 137, 145

RSML	Requirements-Specific Modeling Language	56, 58, 60, 62, 64, 67, 87, 90, 93, 101, 106, 109, 142, 144, 151, 153, 154
SIRCOD	Seamless Integration of Requirements in CODe	34–36, 40, 44, 52, 53, 58, 87–90, 101, 105–109, 137, 138, 140–143, 151, 153, 154
STD	State Transition Diagram	19
SysML	Systems Modeling Language	13, 15, 24, 67, 68
UCM	Use Case Map	15
UML	Unified Modeling Language	13, 14, 21, 24, 27
URML	User Requirements Modeling Language	14
URN	User Requirements Notation	15
VDM	Vienna Development Method	16

Résumé long

1 Introduction

Comment s'assurer qu'un système est le "bon" système ? Les systèmes complexes, les systèmes à forte complogicielle, les systèmes embarqués nous entourent tous les jours : smartphones, voitures, avions, satellites sont autant d'exemples d'objets que nous utilisons quotidiennement, parfois inconsciemment. La qualité de ces systèmes est donc cruciale : comment avoir confiance en ces appareils ? Si la question semble mineure pour les applications des smartphones, on comprend vite qu'elle devient cruciale pour des engins qui peuvent dépasser la tonne et être lancés à plusieurs dizaines de kilomètres par heure. Par ailleurs, si tout le monde a été confronté un jour ou l'autre à un bug logiciel, nous l'avons probablement identifié parce que le système ne faisait pas ce qu'il était censé faire.

Mais que doit faire le système ? C'est la première étape de tout développement de système : déterminer ce que le client veut que le système fasse. En d'autres termes, quelles sont les exigences du client à l'égard du système ? Ces exigences sont les briques de base utilisées pour construire le système. Par conséquent, pour s'assurer que le système est le bon, il faut "simplement" vérifier que ses exigences sont satisfaites.

Par ailleurs, la criticité des exigences a souvent été abordée dans la littérature. En effet, l'ambiguïté, le manque de cohérence, et les exigences mal traitées peuvent conduire à une mauvaise conception et une mauvaise implémentation. Dans le contexte de cette thèse, nous nous concentrerons sur les systèmes à forte intensité logicielle – c'est-à-dire les systèmes où le logiciel joue un rôle important. Cependant, même dans ce contexte plus restreint, de nombreuses défaillances peuvent exister. Même si les conséquences d'un problème de code peuvent être moins évidentes qu'un défaut structurel dans l'architecture d'un pont, il y a de plus en plus de logiciels dans nos vies, dans les objets du quotidien, et la plupart du temps, les exigences sont mal traitées. De plus, même si les problèmes peuvent sembler insignifiants et mineurs, les échecs causés par des erreurs de spécification sont nombreux, coûteux et parfois, tragiques.

Pour cette raison, en nous basant sur l'état de l'art, dans le contexte de cette thèse, nous nous avons défini un certain nombre de propriétés importantes pour les approches des exigences :

(a) **Public visé** : l'approche nécessite-t-elle une expertise, ou est-elle compréhensible par toutes les parties prenantes ? Cela répond aux propriétés qui exigent que les parties prenantes prennent position sur les exigences (l'exigence est-elle *nécessaire*, *complète*, *faisable* ?) et réduit l'*ambiguïté* des exigences pour les parties prenantes.

(b) **Support de la traçabilité** : existe-t-il un moyen de lier les exigences à d'autres artefacts (tels que les exigences ou d'autres parties du système) et quel est le type de ces liens ? Cette

propriété doit aider à répondre aux besoins de *correction*, *consistance* et *absence d'ambiguïté*.

(c) **Couverture** : quels types d'exigences peuvent être exprimés avec l'approche (toutes les exigences ? seulement les exigences fonctionnelles ?) ? Cette propriété est nécessaire pour indiquer la *complétude* des exigences.

(d) **Définition sémantique** : les exigences sont-elles définies sémantiquement dans l'approche ? Une définition sémantique permet d'éviter l'*ambiguïté* et de vérifier la *consistance*, la *correction* et l'*unicité*. De plus, la puissance de la formalisation permet une gestion rigoureuse des expressions dotées de sémantique.

(e) **Verifiabilité** : une exigence peut-elle être formellement vérifiée ? Cette propriété vise à répondre au besoin de *verifiabilité* des exigences.

Deux mondes se confrontent principalement de nos jours pour l'expression des exigences :

- dans les systèmes critiques, les méthodologies formelles, basées sur une base mathématique, sont largement utilisées pour prouver la sécurité d'un système ;
- dans les autres systèmes, les exigences sont souvent exprimées sous forme de phrases décrivant les besoins des utilisateurs, sans lien réel avec le système lui-même.

Il existe ainsi un fossé entre :

- des exigences en langue naturelle, abordables et compréhensibles par toutes les parties prenantes mais, évidemment, non formelles et par conséquent, potentiellement ambiguës, incohérentes, non complètes, etc ;
- et une représentation plus formelle des exigences, dans un langage dédié, mais abordable par seulement certains experts du projet.

Les approches formelles doivent être considérées : la seule façon d'automatiser l'analyse des exigences passera par le processus de formalisation, soit par une approche formelle de bout en bout, soit par une transformation d'une approche non formelle à une approche formelle (par le biais de NLP ou d'approches qui fournissent une vue plus abstraite du système).

Cependant, les compétences requises pour travailler avec les méthodes formelles classiques ne sont pas accessibles à tous, et tous les ingénieurs ne peuvent pas comprendre ces approches. Les approches basées sur le langage de programmation ont l'avantage d'être formelles tout en restant accessibles aux ingénieurs. Dans une perspective multi-vues, ces approches peuvent être utilisées pour exprimer plusieurs niveaux d'abstraction dans le même langage. De plus, les approches agiles de plus en plus utilisées pourraient inclure ces approches puisqu'elles nécessitent des échanges fréquents avec les clients et impactent directement les modifications du code source.

Il semble donc plus réaliste de proposer un modèle dédié à l'ingénierie des exigences tout en créant des passerelles vers les outils existants. Les différents acteurs pourront ainsi continuer à utiliser leurs outils habituels tout en bénéficiant des apports de ce modèle.

2 SIRCOD

Comme mentionné dans l'introduction, les exigences sont la base d'un processus d'ingénierie. Cependant, bien qu'il y ait un réel besoin de les exprimer d'une manière qui soit abordable par toutes

les parties prenantes – après tout, tout le monde doit comprendre clairement ce qu'est le système développé –, ces exigences doivent être définies d'une manière suffisamment claire pour ne laisser aucune ambiguïté – et l'ambiguïté est une propriété de la langue naturelle. Ainsi, la recherche en RE a proposé plusieurs moyens pour tenter de surmonter ce problème : une syntaxe stricte pour le document des exigences, des outils d'analyse, des langages formels. De notre point de vue, les approches proposées souffrent de leur qualité : soit elles ne sont pas assez formelles et donc, peuvent conduire à l'introduction d'ambiguïté, soit elles sont assez formelles, mais les ingénieurs non experts sont réticents à utiliser de telles approches. Nous proposons par conséquent une voie intermédiaire : l'utilisation d'un langage de programmation. Cela permettra d'introduire un formalisme abordable par les ingénieurs logiciels qui travaillent sur un projet.

En effet, les exigences et le code sont étroitement liés au développement de systèmes logiciels. Chaque morceau de code doit répondre à un besoin exprimé dans une exigence, et chaque exigence doit être satisfaite par (au moins) un morceau de code. Cependant, la plupart du temps, on demande aux développeurs d'écrire du code dans des éditeurs dédiés, qui peuvent fournir des outils de commentaires et de documentation, mais les exigences restent des artefacts dans d'autres formalismes, déconnectés du code écrit (documents d'exigences écrits dans Microsoft Word, problèmes dans un gestionnaire de suivi des problèmes ou même user stories). Si un changement est introduit dans le code ou les exigences, il n'y a pas de mécanisme simple pour analyser et impacter ce changement.

En imbriquant les exigences et le code, les approches basées sur le principe du modèle unique tentent de surmonter ce problème. L'approche multi-exigences met l'accent sur la nécessité d'exprimer les exigences et le code dans le même formalisme : le langage de programmation. L'avantage d'une telle approche est de permettre au développeur d'accéder aux exigences lorsqu'il programme dans le même environnement.

Cependant, cette approche souffre de plusieurs problèmes : tout d'abord, les relations entre les différents niveaux d'abstraction des exigences (de l'exigence elle-même au code qui la réalise) ne sont pas clairement définies. Ensuite, même si certains liens peuvent exister entre ces différentes représentations dans l'approche multi-exigences, ils ne sont pas sémantiquement définis. Pour faciliter l'analyse du code concernant l'exigence, un lien sémantique doit exister entre eux.

L'approche SIRCOD propose d'explorer le principe de multirequirement pour surmonter les problèmes mentionnés ci-dessus. Comme mentionné dans l'introduction, les exigences sont les briques de base du système. En tant que telles, elles font partie des premiers artefacts introduits au cours du développement. Cette introduction précoce conduit à plusieurs faits : (i) les premières phases impliquent plusieurs acteurs qui ne sont pas tous experts en exigences ou en code, et (ii) les exigences peuvent exister sous plusieurs représentations mais sont, la plupart du temps, exprimées en langue naturelle.

En partant de cette hypothèse, il faut considérer que les exigences en langue naturelle sont le point d'entrée de toute approche de développement. C'est pourquoi, même si nous considérons que la formalisation précoce est une bonne pratique, nous proposons dans SIRCOD de partir des documents d'exigences.

Le processus présenté dans Fig. C.1 est donc assez classique, commençant par l'extraction des exigences et aboutissant à la mise en œuvre. Cependant, l'application de l'approche multi-exigences (et du principe du modèle unique) signifie que toutes ces étapes sont réalisées dans l'environnement du langage de programmation.

Dans Fig. C.2, nous présentons les différents artefacts qui seront impliqués dans le processus. SIRCOD partent donc d'un document d'exigences (en langue naturelle), et les introduiront progressivement dans le langage Eiffel, d'abord en tant qu'artefacts en langue naturelle puis, en tant

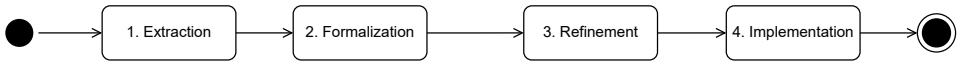


FIGURE C.1 : Les quatre étapes du processus SIRCOD

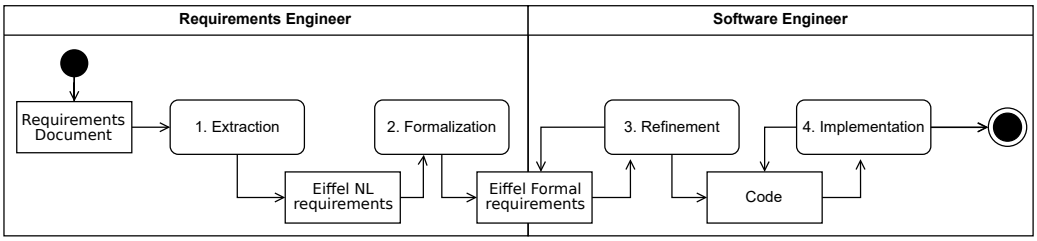


FIGURE C.2 : Étapes détaillées du processus SIRCOD (en anglais)

qu'artefacts formels. Enfin, le dernier résultat est le code lui-même.

De l'exigence dans la documentation à l'exigence dans le code

Il existe plusieurs types de documents d'exigences. Ils peuvent être appelés "document d'exigences utilisateur" (URD), "spécification d'exigences système" (SRS), ou même plus simplement "spécifications". La première étape de l'approche présentée ici (activité 1.a) consiste à extraire ces exigences et à les exprimer dans un langage de programmation. La méthodologie présentée dans cette section suppose une extraction manuelle des exigences (et de leurs relations) et leur traduction dans le langage de programmation choisi. Dans la section 3, nous présenterons la traduction automatique des exigences en code et l'outil qui la supporte.

La toute première étape de SIRCOD consiste à créer une méthode Eiffel (appelée *feature*) pour chaque exigence. Ces fonctionnalités – que nous avons appelées **requirements features** – renvoient une phrase décrivant l'exigence (par exemple, dans List. C.1, la fonctionnalité *requirement_1_1_doc* renvoie l'expression en NL de l'exigence correspondante).

```
feature
  requirement_1_1_doc: STRING
  do
    Result := "[
      The drone shall pick up a parcel, go to the destination and drop
          it off when activated on the web application.
    ]"
```

```
end
```

Listing C.1 : Fonctionnalité Eiffel décrivant une exigence

Lier l'exigence en code et l'exigence en documentation

La deuxième étape (activité 1.b) consiste à lier l'exigence dans le code et son document d'origine pour permettre la navigation de l'un à l'autre. Grâce à EIS, un mécanisme d'EiffelStudio, un élément de code (une fonctionnalité, par exemple) peut être lié à un document externe. Plus précisément, il peut être lié à un morceau de texte de ce document, grâce à des signets.

L'utilisateur peut choisir graphiquement dans EiffelStudio (voir Fig. C.3) un signet sur le document – pour faciliter l'ajout de tels signets, si certains paragraphes des documents n'ont pas de signets, des signets génériques sont ajoutés par l'outil – ou ajouter manuellement une **note** à l'élément donné. Les notes Eiffel peuvent être utilisées pour enrichir les fonctionnalités qui expriment des exigences. Ces notes peuvent être comparées à des annotations Java et n'intègrent aucune sémantique, sauf celle donnée par les outils qui les utilisent.

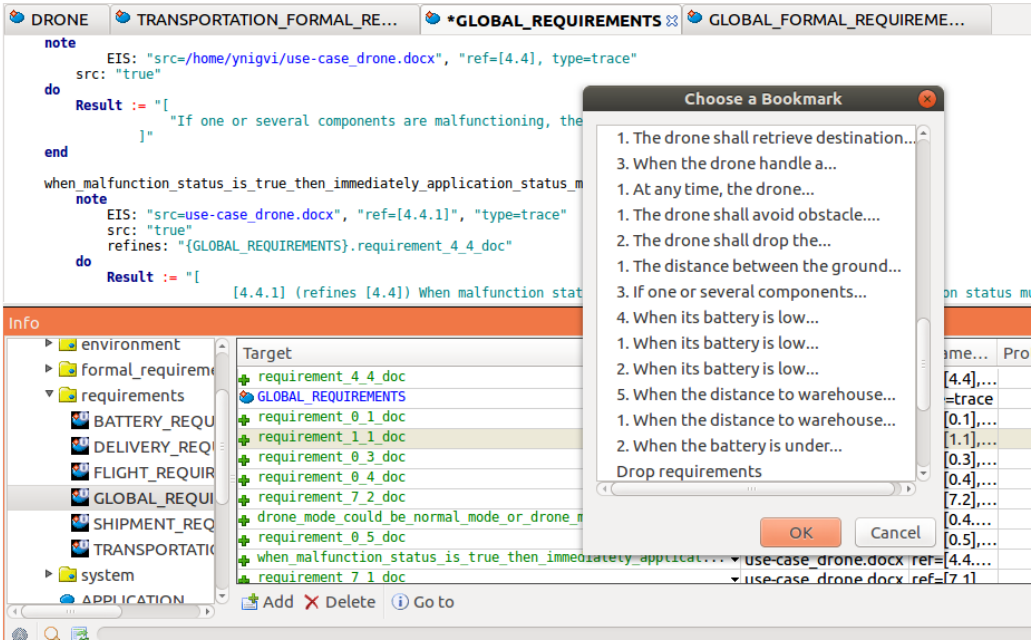


FIGURE C.3 : La vue "Info" qui permet à l'utilisateur d'ajouter graphiquement une source

Par exemple, le lien entre la source originale de l'exigence de List. 3.1 et la caractéristique d'exigence Eiffel (*requirement_1_1_doc*) est exprimé par la note :
`EIS : "src=use-case_drone.docx", "bookmark=1.1"`.

Vue documentation

L'intégration des exigences et de la documentation directement dans le code doit améliorer la traçabilité. Cependant, la lisibilité des exigences écrites dans un langage de programmation n'est pas la meilleure possible, avec les nombreux artefacts de code (tels que do, end, Result, etc.). L'objectif de la vue de documentation est de surmonter ce problème. EiffelStudio intègre déjà un mécanisme de vues. Par exemple, la vue *plain text* est la vue classique, avec les caractéristiques d'une classe, tandis que la *flat view* intègre tous les éléments des ancêtres de la classe.

Nous avons d'abord ajouté la vue documentation comme un moyen d'imprimer la documentation du code disponible dans une autre classe. L'idée était que la plupart du temps, dans une classe bien documentée (à l'aide de notes ou autres commentaires de doc), l'expression enchevêtrée du code et de la documentation ne permet pas une lecture claire et simple du code d'une part et de la documentation d'autre part. Si la lisibilité de la documentation peut être améliorée par la génération de HTML, par exemple, avec un long extrait de documentation en code, sa lecture peut être rendue difficile. Nous avons donc décidé d'extraire la documentation du code dans des classes dédiées, de la même manière, nous avons séparé les caractéristiques des exigences et les caractéristiques des exigences formelles. Cependant, pendant le développement, l'ingénieur doit toujours avoir accès à la documentation, et le forcer à changer de classe lui rendait la tâche plus difficile.

Nous avons donc obtenu une vue de la documentation qui permet d'imprimer à la fois la documentation et des extraits de code. Bien que le procédé puisse être utilisé pour documenter le code, il nous a semblé qu'il était particulièrement intéressant dans le contexte de SIRCOD. En effet, il est ainsi facile lorsqu'on est dans une fonctionnalité d'exigence formelle de passer à la représentation en langue naturelle, améliorant la lisibilité et la compréhensibilité des exigences tout en gardant la formalité de l'expression du code.

Dans Fig. C.4, une capture d'écran de la vue de documentation pour List. C.1 est donnée. Dans la fonctionnalité `requirement_1_1_doc`, seuls l'identifiant de la fonctionnalité, la source de l'exigence (note EIS) et le contenu de l'exigence apparaissent.

```
        note
            EIS: use-case_drone.docx
            src:
                    The drone is propelled by four rotors, each driven by a brushless motor. The power shall be enough to transport parce
        end

        requirement_1_1_doc: STRING_8
        note
            EIS: /home/ynigvi/use-case_drone.docx (at bookmark "1.1")
                "The drone shall pick up a parcel, go to the destination and drop it off when activated on the web application.
            src:
                    The drone shall pick up a parcel, go to the destination and drop it off when activated on the web application.
        end

        requirement_4_4_doc: STRING_8
        note
            EIS: use-case_drone.docx
            src: "If one or several components are malfunctioning, the drone send a message to an operator."
        end

        requirement_7_1_doc: STRING_8
        note
            EIS: use-case_drone.docx
```

FIGURE C.4 : Vue documentaire des caractéristiques de l'exigence

Le texte de la source, spécifié dans la note EIS, est récupéré dans le document docx et imprimé. Cela simplifie et détend ainsi le développement en empêchant l'utilisateur d'ouvrir un éditeur ex-

terne. Dans Fig. C.4, le lien EIS vers le signet 1.1 dans le document `use-case_drone.docx` est utilisé pour extraire la première phrase de la section 1.1. Cette vue permet une meilleure lisibilité en débarrassant l'éditeur de tous les éléments qui pourraient gêner la lecture (comme le corps des caractéristiques).

Pour faciliter la navigation, la vue de la documentation est également cliquable. Cela implique que toutes les références aux éléments de code, comme les fonctionnalités des autres classes, peuvent être cliquées et automatiquement ouvertes dans l'éditeur. Cela contribue à nouveau à la fluidité du développement en évitant à l'utilisateur de rechercher des classes dans le système de fichiers.

Raffiner les exigences

Après avoir exprimé les exigences en langue naturelle dans le code, grâce aux **features requirements**, la deuxième étape du processus (activité 2) vise à introduire un certain formalisme. L'objectif de cette étape est de s'assurer que le futur système répondra à ses exigences. La toute première étape est, avant toute autre considération, d'obliger le développeur à prendre en compte l'exigence pendant le développement. À cette fin, nous avons basé l'approche SIRCOD sur un processus de raffinement, en partant des exigences, au niveau d'abstraction le plus élevé, jusqu'à l'implémentation. Cette idée de partir de l'abstrait pour aller vers le concret existe également dans la programmation orientée objet, et est rendue possible grâce à la notion de méthodes et de classes *abstract* (*deferred* dans Eiffel).

La toute première étape de la formalisation consiste donc à fournir, pour chaque exigence, une *deferred feature* qui est une représentation formelle de cette exigence. Le développeur qui devra hériter de la classe où les exigences sont formalisées devra prendre en compte cette fonctionnalité. Sinon, le système ne compilera pas.

En appliquant ceci à l'exigence 1.1 nous obtenons donc la **formal requirement feature** dans List. C.2. Notez que les deux caractéristiques (*requirement_1_1_doc* et *requirement_1_1*) sont liées par la note **src** de List. 3.6.

```
feature
  requirement_1_1
  note
    src: "{GLOBAL_REQUIREMENTS}.requirement_1_1_doc"
  deferred
  end
```

Listing C.2 : Représentation formelle d'une exigence en Eiffel

Si ce modèle peut s'appliquer à toutes les exigences, il est possible d'aller plus loin dans la formalisation de certaines d'entre elles. Une exigence étant une paire composée d'un ensemble de contraintes (les hypothèses) et d'un ensemble de propriétés (les prédicats), il est possible de les exprimer à l'aide des pré et postconditions. Puisque ces features sont abstraites, chaque classe qui héritera de la classe qui contient des **formal requirements features** devra prendre en compte ces features abstraites : soit en les implémentant directement, soit en les implémentant dans leurs classes filles.

Cela se fait dans la troisième activité, appelée raffinement, qui est itérative. L'idée est de partir d'exigences très simples et abstraites et de les enrichir de manière itérative, en les rendant plus concrètes. Le paradigme orienté objet et la conception par contrats sont tout à fait adaptés à cette approche.

Conclusion

L'utilisation de l'approche SIRCOD, si elle introduit une augmentation du travail au début (avec la transformation des NL exigences en code et avec la formalisation des exigences), est assez similaire à un processus classique de développement orienté objet à partir de la troisième étape et n'introduit pas d'augmentation de l'effort de travail requis. De plus, nous abordons plusieurs des propriétés que nous avons définies :

(a) **Public visé** : si les exigences sont exprimées pour la première fois en langue naturelle, l'approche sous sa forme actuelle s'adresse davantage aux développeurs de logiciels.

(b) **Support de la traçabilité** : la traçabilité, de la source au code, en passant par les différents processus de raffinement, est assurée par l'approche.

(c) **Couverture** : Les exigences en langue naturelle permettent d'exprimer tous les types d'exigences. Les **formal requirements features** permettent uniquement à l'utilisateur d'exprimer les parties fonctionnelles du système de manière formelle. Cependant, il est toujours possible d'écrire une exigence formelle vide, liée à des exigences non fonctionnelles, pour exprimer la nécessité de les prendre en compte.

(d) **Définition sémantique** : En utilisant la sémantique du langage de programmation, les exigences sont définies sémantiquement une fois formalisées.

(e) **Vérifiabilité** : Les contrats permettent de vérifier que le système est correct vis à vis des exigences.

3 RSML

Certaines parties prenantes impliquées dans un projet peuvent préférer travailler avec des approches textuelles, tandis que d'autres peuvent préférer utiliser des approches mathématiques. Par exemple, un ingénieur électricien ne travaillera pas dans la même optique qu'un avocat, même s'il est impliqué dans le même projet. Il est pourtant crucial de tenir compte de ces différents points de vue. Nous ne pouvons pas nous attendre à ce que les utilisateurs travaillent avec une approche donnée sans qu'il n'existe de passerelle avec leurs approches habituelles existantes.

Pour surmonter cette difficulté, nous proposons d'utiliser d'ingénierie dirigée par les modèles et de définir un métamodèle basé sur la sémantique que nous avons définie dans les chapitres précédents. Les avantages d'une telle approche sont multiples : d'une part, elle nous permet de définir un DSML basé sur ce métamodèle, que nous avons nommé RSML, plus accessible aux non-experts que Eiffel, et d'autre part, en s'inscrivant dans l'idée de la globalisation des langages de modélisation, elle peut être utilisée pour faire le pont avec d'autres approches.

Un méta-modèle d'exigences

Le métamodèle RSML[1] est organisé en deux parties principales :

(i) les connaissances du domaine ;

[1] Le métamodèle complet se trouve en annexe A.1

(ii) les exigences elles-mêmes, et la distinction entre les exigences en langage naturel et les exigences formelles.

Chacun des éléments du métamodèle est précisément défini sémantiquement.

Connaissances du domaine

La partie (i) du métamodèle est utilisée pour modéliser ces propriétés de domaine. Elle permet de définir la valeur ou l'état d'un *Element*, qui peut être utilisé ultérieurement lors de la rédaction ou de la spécification des exigences.

La connaissance du domaine est formalisée sous forme d'hypothèses en utilisant la sémantique que nous avons définie dans la section 2. Ces hypothèses doivent être considérées pour toutes les exigences pendant l'analyse. Il est donc possible de les exprimer comme des préconditions requises pour toutes les fonctionnalités du système. Cependant, si une précondition est un état requis pour toutes les fonctionnalités, il est plus pratique de l'exprimer comme un invariant. Pour permettre l'utilisation de ces hypothèses dans l'ensemble du système, nous les plaçons dans une classe spécifique héritée par toutes les autres classes du système.

Exigences

Les exigences elles-mêmes sont exprimées dans la partie (ii) du métamodèle. Elles sont divisées en deux catégories : les exigences en langue naturelle et les exigences formelles. Cette distinction permet d'exprimer des exigences formelles tout en gardant la possibilité d'exprimer certaines des exigences en langue naturelle (pour celles qui ne peuvent pas être formalisées). En effet, la distinction classique est faite entre les exigences fonctionnelles et non fonctionnelles. Alors que les premières peuvent - plus ou moins - facilement être formalisées, la formalisation des exigences non fonctionnelles est plus difficile.

Les exigences en langue naturelle sont similaires aux exigences SysML ou aux objectifs KAOS : elles ont un attribut de texte libre. Elles seront formalisées comme des exigences langue naturelle : comme des features sans corps ni contrat, liées à une documentation textuelle. Sur la base de SIRCOD, l'utilisateur devra les prendre en compte, même si elles ne sont pas formalisées automatiquement.

Les exigences formelles sont utilisées pour exprimer les propriétés que le système doit posséder. Le contexte est exprimé par les *Constraints*, qui sont les hypothèses des exigences (par exemple, les préconditions). Les *Properties* traitent des *Elements* et sont comparées aux *Values*. Une *Value* peut être une valeur primitive (un nombre ou une chaîne), un état ou une référence à un autre *Element*. Dans le métamodèle, deux classes intermédiaires respectivement nommées *ConstraintsDisjunction* et *PropertiesDisjunction* sont utilisées pour représenter le *ou logique*. En effet, par défaut, nous supposons que chaque *assertion* doit se vérifier lorsque toutes les *constraints* sont satisfaites. Ces artefacts (*ConstraintsDisjunction* et *PropertiesDisjunction*) sont donc introduits pour faciliter la composition en utilisant une structure *ou*. Cela impose cependant d'exprimer ces contraintes et assertions sous forme de formes normales conjonctives. Cela simplifie le passage à la sémantique Eiffel puisque c'est la manière la plus naturelle d'exprimer des ensembles de contrats en Eiffel : un contrat peut être composé par une disjonction de prédicats, tandis que la conjonction de tous les contrats doit être satisfaite.

Le langage contraint RSML

Comme indiqué précédemment, le métamodèle prend en charge l'expression des connaissances du domaine, des exigences et des relations entre les exigences. La manière la plus naturelle d'écrire des exigences reste, dans la plupart des cas, l'écriture textuelle. Pour faciliter l'utilisation du métamodèle, l'ingénierie dirigée par les modèles fournit des outils appropriés : les DSL et plus particulièrement, les DSML.

RSML est un DSML proche de la langue naturelle. Un tel langage peut être utilisé pour permettre à des non-spécialistes d'exprimer des exigences d'une manière similaire aux outils de développement piloté par le comportement (BDD) tels que Cucumber. Cependant, contrairement à ces outils, RSML est défini sémantiquement. Tous les éléments du métamodèle peuvent ainsi être exprimé dans ce DSML, et un outil est fourni pour exprimer celui-ci.

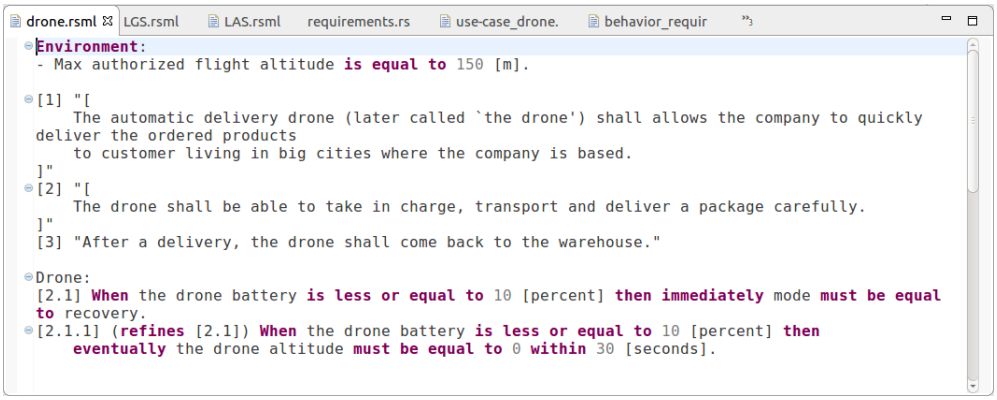


FIGURE C.5 : Exemple du DSL RSML en pratique

L'exemple Fig. C.5 illustre comment RSML peut être utilisé pour exprimer la connaissance du domaine, les exigences NL et les exigences formelles. **Les documents d'exigences** en RSML sont composés de deux éléments :

- l'**environnement**, qui contient la connaissance du domaine ;
- les **exigences**, qui contient les exigences et leurs relations.

Les éléments de l'environnement sont exprimés dans une liste introduite par le mot-clé *Environment* (ou sa version abrégée *Env*), suivi de deux points (*:*) et d'une nouvelle ligne. Chaque élément est introduit par un tiret (-). Les exigences quand à elles sont introduites par un identifiant, entre crochets (*[]*), suivi soit d'une exigence en langue naturelle, entre guillemets (*"*), soit d'une exigences formalisées, dont la syntaxe est plus contraintes. La grammaire complète peut être trouvée chapter A.2.

Transformer vers d'autres formalismes

L'utilisation d'un langage de modélisation permet de passer facilement d'un modèle à un autre. Cela peut être fait en utilisant des langages dédiés, comme ATL ou Kermeta, ou même de manière

programmatique, en utilisant EMF[2]. Dans le contexte des exigences, cela peut aider à fournir un pont vers différentes approches existantes, souvent utilisées par les différentes parties prenantes impliquées dans un projet.

Dans le contexte de RSML, nous proposons plusieurs transformations, depuis et vers d'autres langages, pour faciliter l'adoption d'une telle approche. Comme le métamodèle est construit en utilisant EMF, il est possible d'utiliser les différentes approches présentées précédemment (ATL, Kermeta, etc.) pour réaliser ces transformations.

Nous avons ainsi fourni des transformations depuis et vers SysML, mais aussi vers des représentations parmi les plus répandues pour les exigences : Microsoft Word et Excel.

Conclusion

Dans cette section, nous présentons RSML, un langage dédié aux exigences qui est proche de la langue naturelle tout en supportant la sémantique formelle. RSML est basé sur un méta-modèle d'exigences permettant leur expression à la fois en langue naturelle et en langage formel. Puisque RSML est sémantiquement défini sur SIRCOD, les expressions RSML peuvent être utilisées et analysées, comme introduit dans la section 2. Par conséquent, les non-experts peuvent bénéficier de la formalité inhérente au RSML.

En utilisant les techniques et les outils du MBSE, nous proposons également des passerelles vers des langages existants tels que SysML ou MSWord/MSExcel, pour permettre aux ingénieurs de travailler avec leurs outils habituels. Nous traitons plusieurs questions d'ingénierie des exigences, en fournissant un moyen de transformer les expressions d'exigences habituelles vers et depuis le RSML. Le RSML peut alors être considéré comme un pivot entre divers espaces d'expression. C'est également un pivot entre les espaces formels et non formels. RSML vise à s'intégrer dans une approche sans rupture, à réduire l'écart entre les exigences NL et les représentations formelles, et à fournir des moyens de s'adresser à plusieurs types de parties prenantes (qu'il s'agisse d'utilisateurs d'outils de modélisation, d'utilisateurs de NL, ou même d'utilisateurs d'approches plus formelles). Nous avons pour ambition d'ajouter de nouvelles transformations à d'autres approches, comme KAOS, pour confirmer l'évolutivité de notre approche dans un contexte plus global.

Les premières expériences nous ont donné de bons résultats. L'utilisation de RSML sur des études de cas s'est avérée utile pour plusieurs raisons :

1. Il est facilement utilisable par les intervenants habitués à travailler avec la langue naturelle.
2. Il permet d'obtenir une expression formelle pour les exigences et leur validation à travers un outil de preuve.
3. Il permet la vérification des systèmes par rapport à leurs exigences.

4 Relations entre exigences

Les relations entre les exigences et d'autres artefacts (y compris d'autres exigences) sont assez courantes en RE. Les méthodes formelles comme Event-B soulignent la nécessité de raffiner les exigences de haut niveau en exigences de plus bas niveau, plus complexes mais plus précises et plus proches de la solution. Les approches GORE sont également basées sur les relations entre ces artefacts, et les outils les plus utilisés dans l'industrie sont des outils fournissant une traçabilité.

[2] `http://www.eclipse.org/modeling/emf/`

Dans un processus de développement classique, il est nécessaire de distinguer la traçabilité des autres relations. Alors que la traçabilité est utilisée entre les artefacts de plusieurs étapes de développement (par exemple, entre une exigence et un morceau de code), les relations sont adressées aux artefacts de la même étape. La plupart des efforts de recherche ont été faits pour fournir des outils et une méthodologie pour la traçabilité, avec quelques outils automatiques mais la plupart d'entre eux nécessitent toujours une expertise humaine pour vérifier la traçabilité. De plus, dans une approche sans rupture, cette distinction ne peut pas être utilisée de la même manière. Puisque nous préconisons une approche qui considère tous les éléments comme des parties d'un modèle unique (dans les limites des contraintes d'un projet réel), toutes les relations sont entre les artefacts de ce modèle unique et doivent être analysées, pas seulement les liens de traçabilité.

Relations existantes liées aux artefacts d'exigences

Comme déjà mentionné, le langage le plus courant pour les exigences est la langue naturelle. Il est donc important de faire des efforts pour l'analyse de ce type d'exigences.

Pour essayer de fournir un ensemble complet de relations d'exigences possibles, nous avons analysé un certain nombre de documents d'exigences. Au cours de cette analyse, nous avons défini plusieurs catégories pour les exigences et les relations entre les exigences. Le but ici n'est pas de critiquer la qualité du document d'exigences. Une telle critique peut être intéressante ; en effet, si des arguments objectifs peuvent être utilisés pour juger un document d'exigences, ils peuvent être utilisés pour améliorer ce document. Cependant, il est nécessaire d'analyser les documents d'exigences existants sans prescription pour dire comment ils sont. Sur la base de cette analyse, la deuxième étape du travail proposera des outils ou une méthodologie sur la façon d'utiliser une telle classification pour améliorer les documents d'exigences existants ou nouveaux. Nous avons présenté ce travail à plusieurs reprises lors d'ateliers afin de le valider, ce qui nous a permis de modifier progressivement la classification jusqu'à la liste actuelle.

Nous avons finalement défini dix catégories pour les exigences (et 12 catégories dérivées). Notez que nous considérons chaque partie d'un document d'exigences comme une exigence dans le contexte de ce travail. Nous ne développerons pas ici cette classification (le but de cette section est principalement de présenter les relations), mais voici un aperçu rapide des catégories (pour une version plus détaillée, veuillez vous référer à [8]) :

1. **Composant** : une propriété qui décrit une partie du système, de l'environnement ou du projet ;
2. **But** : un objectif du projet ou du système ;
3. **Comportement** : une propriété d'une opération d'un composant (ou de l'ensemble du système) ;
4. **Tâche** : une activité du projet ;
5. **Produit** : un artefact créé par une tâche du projet ;
6. **Contrainte** : une propriété de l'environnement qui affecte une partie du système (de ou son comportement) ou du projet ;
7. **Role** : une responsabilité pour une tâche ou un comportement ;

8. **Limite** : une propriété que le système, le projet ou l'environnement n'inclut pas ;
9. **Manque** : une propriété non définie dans le cahier des charges, mais qui devrait l'être ;
10. **Meta-exigence** : une propriété d'une exigence.

En utilisant ces catégories, nous avons défini plusieurs relations qui peuvent exister entre les exigences. De manière similaire aux catégories pour les exigences, nous avons obtenu progressivement neuf relations, énumérées ici :

1. **Disjonction** : deux exigences R_1 et R_2 sont disjointes si elles ne sont pas liées ;
2. **Appartenance** : une exigence R_1 appartient à une autre exigence R_2 si R_1 est une sous-exigence de R_2 ;
3. **Répétition** : a requirement R_1 repeats a requirement R_2 iff R_1 specifies the same property as R_2 ;
4. **Contradiction** : une exigence R_1 contredit une exigence R_2 si les propriétés spécifiées par les exigences ne sont pas compatibles ;
5. **Conséquence** : une exigence R_1 suit une exigence R_2 si R_1 est une conséquence de R_2 ;
6. **Extension** : une exigence R_1 étend une exigence R_2 si R_1 assume R_2 et spécifie une nouvelle propriété ;
7. **Exception** : une exigence R_1 excepte une exigence R_2 si R_1 modifie une propriété spécifique de R_2 dans un contexte spécifique ;
8. **Contrainte** : une exigence R_1 contraint une exigence R_2 si R_1 spécifie une contrainte sur une propriété spécifiée dans R_2 ;
9. **Caracterisation** : une exigence R_1 caractérise une exigence R_2 si R_1 est une méta-exigence de R_2.

Pour compléter cette analyse, nous avons également analysé comment les exigences formelles peuvent être reliées entre elles.

Définitions formelles de ces relations

Nous avons précédement introduit une définition formelle des exigences elles-mêmes. Nous définissons également la satisfaction des exigences puisqu'il s'agit de la propriété que nous voulons tracer pour assurer la conformité des systèmes. Ces définitions sont utilisées dans cette section pour définir formellement les relations entre les exigences. Pour rappel, nous avons défini une exigence comme une paire d'un ensemble de contraintes C_R et d'un ensemble de propriétés P_R avec $C_R \vdash P_R$. Nous avons ainsi obtenu l'équivalence :

$$sat(R) \equiv hold(\mathcal{C}_R) \rightarrow hold(\mathcal{P}_R)$$

Nous donnons ici deux nouvelles définitions que nous utiliserons dans les sections suivantes pour compléter cette définition.

Définition 1

Deux exigences R_1 et R_2 partagent un même contexte si $\mathcal{C}_{R_1} = \mathcal{C}_{R_2}$.

Définition 2

Une exigence atomique R est une exigence où $|\mathcal{P}_R| = 1$ (une exigence avec une propriété unique).

En utilisant ces définitions, nous pouvons définir les relations entre les exigences. Pour construire l'ensemble des relations entre exigences, nous avons considéré deux exigences R_1 et R_2, leurs contraintes ($\mathcal{C}_{R_1}$ et $\mathcal{C}_{R_2}$) et leurs propriétés ($\mathcal{P}_{R_1}$ et $\mathcal{P}_{R_2}$). Nous avons analysé quelles pouvaient être les relations entre les contraintes et entre les propriétés. Grâce à cette analyse, nous avons défini six relations pour lesquelles nous avons défini une sémantique précise :

- *répétition*
- *addition*
- *contradiction*
- *contrainte*
- *partition*
- *raffinement*

La définition sémantique des relations entre les exigences présente plusieurs avantages. Tout d'abord, elle permet de lever toute ambiguïté sur la signification de ces relations. Par ailleurs, puisque nous avons défini l'implication de la satisfaction, il est donc possible, étant donné un ensemble d'exigences, de propager la satisfaction des exigences. Une telle propagation peut conduire à deux avantages : d'une part, il est possible, étant donné un ensemble d'exigences satisfaites et les relations entre les exigences, de mettre en évidence quelles sont les exigences déjà satisfaites. D'autre part, il est possible de faire l'inverse et d'isoler les exigences qu'il est nécessaire de satisfaire et les exigences qui seront satisfaites par la propagation.

On peut ainsi définir des liens de satisfaction entre exigences à partir de ces relations, formant un graphe dirigé. Un exemple est donné dans Fig. C.6 (une emphase est mise sur l'ensemble minimum pour satisfaire l'ensemble des exigences).

Un ensemble d'exigences et de relations entre elles est donné. En utilisant la sémantique définie, il est donc possible de fournir des liens de satisfaction entre les exigences, comme indiqué dans Fig. C.7. Un simple algorithme de traversée de graphe peut le faire et retournera un nouveau graphe dirigé avec des liens de satisfaction, étant donné la sémantique des relations.

Amélioration de SIRCOD et RSML

Relations dans SIRCOD

Nous avons introduit cette notion de relations entre les exigences de la même manière que nous avons retracé la source des exigences : en utilisant le mécanisme des notes. Dans List. C.3, la feature de List. C.3 est enrichie d'une note *adds*, qui donne l'information que requirement_5_1_doc est une extension de requirement_1_1_3_docs. De telles relations peuvent ensuite être utilisées pour tracer les exigences et vérifier que toutes les exigences sont correctement satisfaites. Dans l'exemple

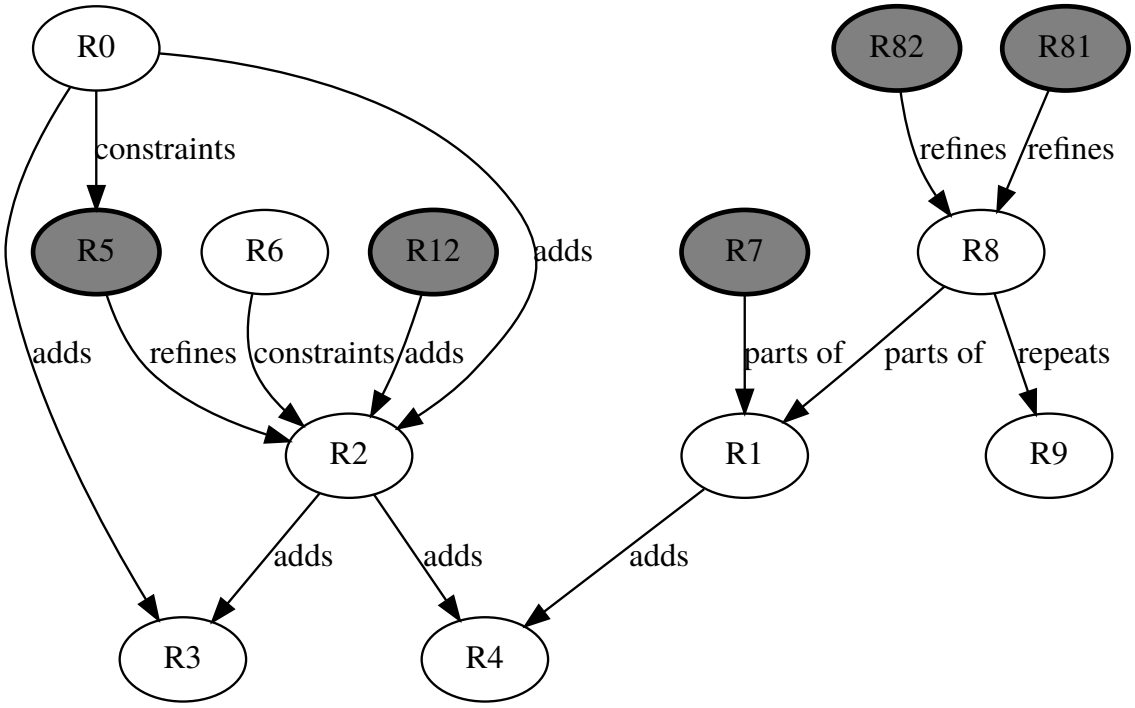


FIGURE C.6 : Exemple d'exigences et de relations entre elles

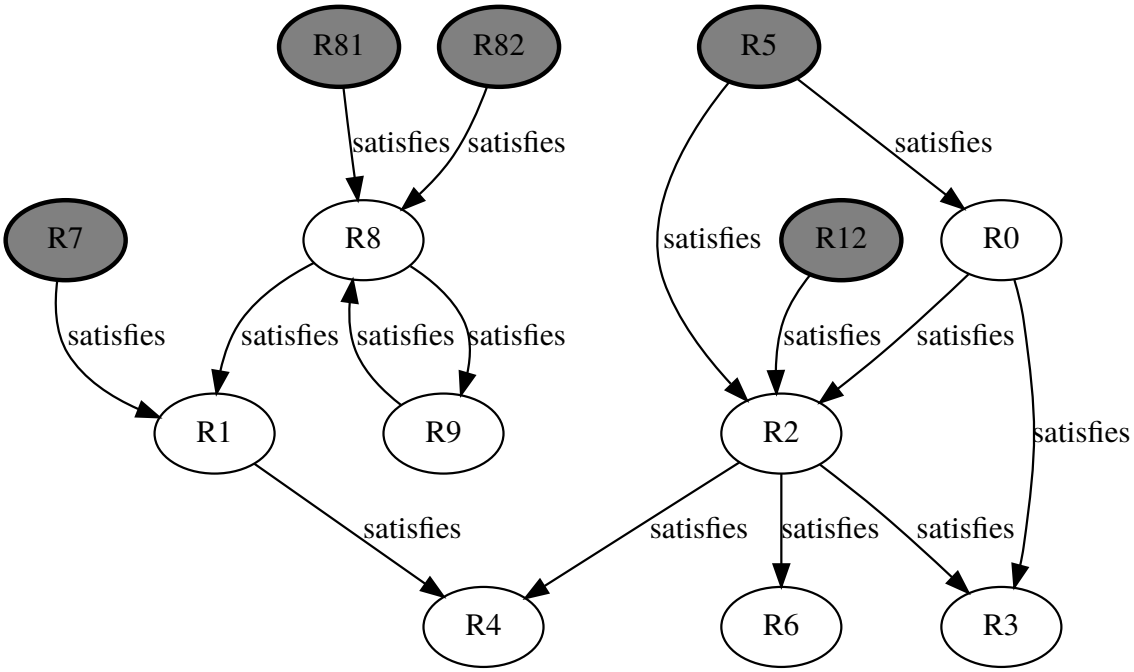


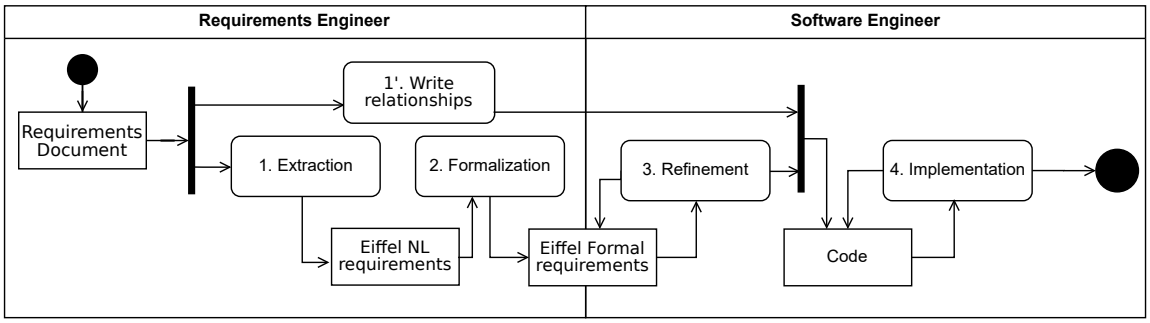
FIGURE C.7 : Exemple de liens de satisfaction pour les exigences et les relations de Fig. C.6

de List. C.3, puisque `requirement_5_1_doc` est un ajout de `requirement_1_3_doc`, on s'attend à ce que si `requirement_5_1_doc` est satisfaite, `requirement_1_3_doc` l'est aussi.

```
feature
  requirement_5_1_doc: STRING
        note
                EIS: "src=use-case_drone.docx", "bookmark=5.1"
                adds: "{DELIVERY_REQUIREMENTS}.requirement_1_1_3_doc"
        do
                Result := "[
                        [5.1] (is an addition of [1.1.3]) When drone
                            location is equal to destination then
                            eventually attached parcel shall be equal to
                             null and drop location status shall be
                            equal to confirmed.
                ]"
        end
```

Listing C.3 : Relations note entre l'exigence_5_1_doc et l'exigence_1_1_3_doc

Nous n'ajoutons pas la relation de *raffinement* puisque le raffinement est représenté dans SIRCOD à l'aide du mécanisme de redéfinition. L'ajout de relations a néanmoins un impact sur le processus SIRCOD. En effet, nous devons maintenant considérer les relations pendant toutes les étapes.



Ceci est représenté dans Fig. 4. Le processus comprend maintenant l'écriture des relations, en parallèle avec les trois premières étapes. Ainsi, l'ingénieur doit également identifier et créer des liens entre les exigences.

Cependant, cette nouvelle activité n'est pas seulement parallèle à l'activité d'*extraction*. En effet, puisque les activités *formalisation* et *raffinement* peuvent donner lieu à de nouvelles exigences, il devient nécessaire de vérifier si les exigences sont liées à d'autres exigences.

Relations dans RSML

La sémantique proposée dans cette section et l'ajout des relations à SIRCOD nous ont conduit à proposer une extension au métamodèle RSML. Comme détaillé dans les sections précédentes, cela

permet une analyse plus facile des exigences et réduit l'ensemble des exigences à vérifier. De plus, les exigences en langue naturelle ne sont traitées que comme des unités, sans sémantique spécifique sur les exigences elles-mêmes. Ces relations peuvent donc être utilisées pour donner des informations sur la satisfaction de ces exigences également.

Cette traçabilité horizontale entre les exigences est exprimée par la réification de la relation entre deux exigences. Cette réification est effectuée par le biais des artefacts *Trace*. La trace ne possède pas de sémantique en tant que telle, et doit être spécialisée par l'une des six relations que nous avons définies, sur la base de la sémantique proposée dans cette section :

- addition : l'exigence R_1 ajoute une propriété à l'exigence cible R_2 ;
- contenance : l'exigence R_1 contient l'exigence cible R_2 ;
- raffinement : l'exigence R_1 raffine l'exigence cible R_2 ;
- contradiction : l'exigence R_1 et l'exigence cible R_2 ne peuvent pas être satisfaites en même temps ;
- répétition : l'exigence R_1 est une répétition de l'exigence cible R_2 ;
- contrainte : l'exigence R_1 contraint le contexte de l'exigence cible R_2.

Conclusion

Ce chapitre fournit des définitions formelles des exigences et de leurs relations qui améliorent le processus de vérification de la conformité du système par un mécanisme basé sur la traçabilité. Ce mécanisme repose sur la propagation de la satisfaction des exigences à travers les relations formelles des exigences.

Nous définissons les exigences comme des paires (i) d'un ensemble de contraintes, et (ii) d'un ensemble de propriétés qui doivent être satisfaites lorsque les contraintes sont tenues. En faisant des combinaisons d'ensembles de contraintes et d'ensembles de propriétés des exigences d'une part, nous identifions six relations différentes qui peuvent être établies entre les exigences. Ces relations sont formellement définies, et nous démontrons comment la satisfaction est propagée d'une exigence à une autre à travers la relation qui les lie.

Dans l'état actuel de la formalisation que nous proposons, certaines vérifications de base peuvent être effectuées sur les exigences, en utilisant la sémantique de leurs relations. Voici quelques pistes sur la façon dont cette sémantique peut aider :

- En ajoutant des relations entre les exigences, nous pouvons trouver des exigences mal formées. En effet, si la sémantique d'une relation donnée ne peut être respectée, cela peut indiquer deux choses : (i) cette relation ne devrait pas exister, ou (ii) cette exigence devrait être liée à cette relation mais n'est pas bien formalisée, ou encore (iii) cette exigence n'est pas correcte.

- En considérant un ensemble d'exigences, et grâce à leur définition sémantique, nous pouvons (i) déduire des relations et mettre en évidence des liens de traçabilité qui n'étaient pas exprimés auparavant ou encore (ii) identifier des relations telles que des répétitions ou des contradictions qui révèlent un problème.

- l'analyse du nombre de relations liées à une exigence peut aider à trouver des problèmes : (i) une exigence sans relations, par exemple, peut être une exigence qui n'a pas été correctement analysée, (ii) trop ou trop peu de relations d'un type donné peuvent révéler une mauvaise analyse des exigences (trop de répétitions, de contradictions, pas assez de raffinements, ...).

Les travaux futurs dans le processus de vérification fourniraient une méthodologie et des outils analytiques pour détecter les incohérences entre les exigences et soutenir leur vérification.

La vérification de la satisfaction d'une exigence se fait par l'analyse de ses relations et la propagation de la satisfaction à d'autres exigences connexes.

Même avec une représentation non formelle des exigences, la sémantique des relations aide à analyser la couverture des exigences. Par exemple, elle peut aider les ingénieurs à compléter la conformité de la matrice en propageant la satisfaction. Elle peut également aider à vérifier des exigences informelles en clarifiant la sémantique des relations (par exemple, en demandant aux ingénieurs : " Êtes-vous sûr que satisfaire R_1 nécessitera de satisfaire R_2 ? "). Pour faciliter l'acceptation de la sémantique proposée, un autre travail futur serait de l'intégrer dans les outils de traçabilité.

Nous avons finalement rendu ce travail pratique en améliorant SIRCOD et RSML, et en fournissant des outils pour analyser les liens de traçabilité.

Le travail présenté dans cette section est donc une étape dans l'introduction de la sémantique formelle dans la traçabilité, permettant d'analyser les exigences (en détectant les incohérences et les simplifications possibles) automatiquement et d'utiliser leurs relations pour réduire l'ensemble des exigences qui doivent être prouvées satisfaites en utilisant des approches de validation classiques comme les tests ou les évaluations humaines.

5 Conclusion

Dans cette thèse, nous proposons une approche pour mieux intégrer les exigences dans le code. Cette thèse a été divisée en trois contributions (comme on le voit dans Fig. C.8) :

- Tout d'abord (1), nous avons présenté l'approche SIRCOD, une approche de développement sans rupture qui intègre les exigences en tant qu'artefacts de code, permettant une forte traçabilité grâce au mécanisme d'héritage ;
- Ensuite (2), nous améliorons les premières étapes du processus en fournissant RSML, un langage et un outil qui répondent à tous les besoins des parties prenantes ;
- Enfin (3), nous avons défini les relations et leur sémantique pour soutenir l'analyse des exigences, améliorant ainsi les deux autres contributions.

Cette approche transparente et *lean* vise à introduire plus de formalité dans le traitement des exigences dans le développement. En effet, comme développé dans la section 1, la formalité permet de réduire le nombre de défaillances des systèmes logiciels. Dans la section 1, nous avons défini les propriétés qui doivent être prises en compte pour les bonnes approches des exigences :

(a) **public visé** ;

(b) **support de traçabilité** ;

(c) **couverture** ;

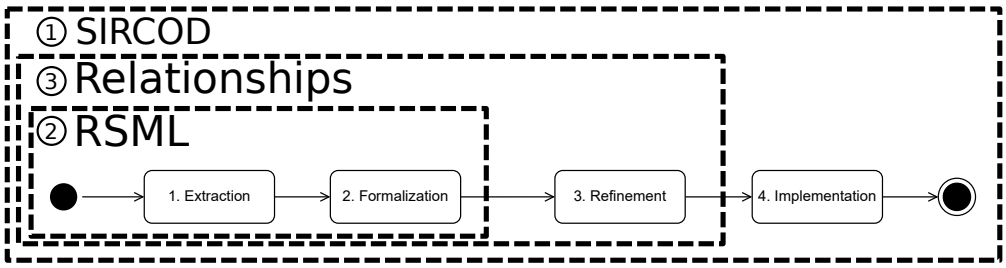


FIGURE C.8 : Partie de chaque contribution au processus SIRCOD

(d) **définition de la sémantique** ;

(e) et **verifiabilité**.

Si le travail effectué dans cette thèse contribue aux questions que nous avons soulevées dans la section 1, il reste encore beaucoup de travail à faire sur la généralisation des approches formelles.

La première perspective concerne l'approche SIRCOD elle-même. Si l'approche a été appliquée à plusieurs reprises au cours de ce travail de recherche et qu'elle est assez intuitive pour les développeurs puisqu'elle utilise des mécanismes déjà maîtrisés par les ingénieurs, nous devons améliorer son évaluation en l'appliquant à un large cas d'utilisation industriel.

Plusieurs outils (RSML, amélioration du mécanisme EIS d'EiffelStudio, algorithme de propagation) ont été créés au cours de ce travail de recherche, mais il est nécessaire de les améliorer pour permettre une utilisation plus large de l'approche. Certaines d'entre elles sont partiellement mises en œuvre et nécessitent encore une intervention humaine. En même temps, il est possible d'automatiser certains processus (par exemple, lors de la création d'un lien de traçabilité entre une fonctionnalité et un document, l'utilisateur doit connaître le signet lié à une partie du document - la possibilité de sélectionner une partie spécifique dans un éditeur visuel devrait faciliter cette étape).

L'approche SIRCOD a été développée dans le contexte d'Eiffel et de l'IDE EiffelStudio. Outre le travail nécessaire sur d'autres IDE pour intégrer les outils développés, il convient d'examiner comment l'approche s'adapte à d'autres langages de programmation. En particulier, le mécanisme de redéfinition de SIRCOD a été pensé pour Eiffel, qui permet l'héritage multiple, et il serait intéressant de voir comment l'intégrer dans des approches objets qui ne permettent pas cet héritage multiple. Cette perspective aiderait à une utilisation plus large de l'approche.

Par ailleurs, le processus de vérification est pour l'instant basé sur des outils de preuve et de conception par contrats. Néanmoins, il est indiscutable que les approches de vérification les plus utilisées dans l'industrie sont les tests, quelle que soit leur forme. L'un des travaux les plus importants qui restent à faire dans notre perspective est d'analyser comment nous pouvons intégrer les tests à SIRCOD, y compris la génération de tests.

L'une des pistes que nous explorons actuellement est l'intégration de l'approche dans une démarche DevOps (et, plus précisément, dans l'intégration continue). En effet, puisque nous fournissons des artefacts qui peuvent être utilisés pour vérifier le système, il semble logique pour nous d'explorer les moyens d'intégrer l'approche dans les flux de travail existants.

Du point de vue du RSML, le langage lui-même peut être amélioré pour fournir de meilleures abstractions. En effet, pour l'instant, les exigences formelles sont encore très proches des expressions d'état, alors que d'autres langages existants tels que Stimulus fournissent une meilleure abstraction.

Fournir des moyens de personnaliser le langage et d'encapsuler la formalité dans des expressions, conduira à une acceptation plus large de cette approche. Dans le contexte de cette thèse, le travail effectué par Antoine Gambier était d'améliorer RSML en analysant les exigences existantes pour trouver les habitudes et les intégrer dans RSML. Cependant, une étude plus large devrait être faite pour fournir des résultats plus complets.

Une autre amélioration peut être faite sur les ponts depuis et vers d'autres paradigmes. Si seulement quelques unes ont été développées dans le cadre de cette thèse, plus de transformations (et plus complètes) devraient faciliter l'intégration de l'approche dans les outils existants. Par exemple, nous envisageons d'ajouter des transformations à d'autres approches, comme KAOS ou Event-B, pour confirmer l'évolutivité de notre approche dans un contexte plus global. L'ajout de nouveaux ponts est relativement simple, grâce aux outils de MDE, mais l'alignement des modèles eux-mêmes peut être assez difficile. Cependant, étant donné que la représentation des exigences sous la forme de paires d'hypothèses et d'assertions est assez courante, nous espérons qu'un tel alignement de modèles ne sera pas une tâche trop ardue.

De plus, il existe encore de nombreuses perspectives dans la transformation d'un document d'exigences textuel en approches plus formelles, telles que RSML. L'utilisation de NLP peut (et doit) être explorée à cette fin. En considérant qu'un document textuel a un modèle, nous devons définir une syntaxe précise du document d'exigences pour faciliter la génération d'artefacts. Cette perspective pourrait également bénéficier au RSML lui-même, en étendant le langage par le biais d'une sorte de norme, comme EARS.

Enfin, le travail effectué sur les relations est encore préliminaire. Leur sémantique doit être explorée plus avant. Par exemple, nous considérons dans ce travail qu'une exigence est satisfaite ou non. Cependant, il peut y avoir des nuances, et il peut être intéressant d'analyser comment cela peut être propagé par les relations.

En utilisant la sémantique existante, une des perspectives que nous allons explorer est la détection des relations existantes. En effet, puisque nous avons défini une sémantique pour les relations basée sur les interdépendances entre les exigences, il est possible de déduire des relations lorsqu'elles ne sont pas explicites.