Requirements Specific Modeling Language : un langage formel d'expression d'exigences

Florian Galinier, Sophie Ebersold, Jean-Michel Bruel

Requirements Specific Modeling Language : un langage formel d'expression d'exigences

Florian Galinier, Sophie Ebersold, et Jean-Michel Bruel

IRIT, Université de Toulouse, France
{firstname.name}@irit.fr

Résumé

La traçabilité et l'intégration des exigences au court du cycle de vie d'un système informatique est un enjeu majeur de l'ingénierie des exigences. En effet, cela permet de s'assurer que le système réponde bien aux attentes du client. Si les méthodes formelles permettent de prouver de telles propriétés, elles restent peu appliquées dans le cas des systèmes non-critiques, et de nombreux acteurs préfèrent exprimer les exigences en langue naturelle. Nous proposons un langage dédié à l'expression des exigences : RSML. Ce paradigme, situé dans une approche sans rupture, fournit un cadre contraint pour l'expression des exigences, basé sur une formalisation en Eiffel. Cela permet d'exprimer les exigences d'un système dans un langage proche de la langue naturelle, tout en permettant de prouver formellement la validité de ce système au regard de ces exigences.

Abstract

Traceability and integration of requirements during the lifecycle of systems is a major challenge in requirements engineering. This lead to ensure that the system is the "good system". Formal methods can be used to prove such properties, however, they are rarely applied in non-critical systems, and many actors prefer to express requirements in natural language. We propose a requirements specific language: RSML. This language, situated in a seamless approach, provides a constrained grammar for the expression of requirements, and semantically defined in Eiffel. This leads to express system requirements in a language close to the natural language, while making possible to prove formally the validity of this system with regard to these requirements.

Mots clés : Ingénierie des exigences, DSL, Développement sans rupture, Traçabilité, Vérification et Validation

Keywords: Requirements engineering, DSL, Seamless development, Traceability, Verification and Validation

1 Introduction

L'intégration des exigences tout au long du cycle de vie d'un système est un gage de qualité : elle permet de s'assurer que le système que l'on réalise est bien celui qui est attendu par le client.

En effet, comment justifier qu'un élément d'un système réponde à une réelle exigence afin d'assurer la qualité du produit, mais également de garder la possibilité de faire évoluer facilement le système si les exigences évoluent. Ce problème de traçabilité est un problème majeur de l'ingénierie des exigences.

Des outils comme IBM Rational Doors [1], Reqtify [2] ou encore SysML [3] permettent d'exprimer ces relations. Ces outils permettent de créer des liens entre des exigences textuelles et des parties de modélisation du système. Ces liens n'ont cependant pas de définitions formelles.

Des approches existantes, telles que Relax [4] et Stimulus[5], proposent d'exprimer les exigences dans des langages proches de la langue naturelle tout en leur associant une sémantique formelle. Relax a ainsi une grammaire contrainte proche de l'anglais et sémantiquement définie en FBTL [6]. Ces approches s'inscrivent cependant dans une démarche d'élicitation et ne permettent donc pas de lier les exigences avec des artefacts d'autres niveaux de spécifications, tels que du code ou des éléments de modèle.

Par ailleurs, même si les approches plus formelles telles que Event-B permettent la génération de code, celui-ci ne possède pas de liens de traçabilité explicites. Il est par conséquent difficile de faire évoluer ce code tout en prenant en compte les changements qui ont pu impacter d'autres parties du système.

La solution proposée par R. Paige et J. Ostroff dans [7] et reprise par B. Meyer dans [8] est une approche « sans rupture » du développement, exprimant dans un même modèle les différents niveaux d'abstraction du système, des exigences à son implémentation.

Dans et article, nous présentons le langage RSML, que nous proposons dans le cadre d'une approche sans rupture allant de l'expression des exigences à l'implémentation du logiciel les satisfaisant.

2 RSML : un langage de modélisation des exigences

RSML[1] (*Requirements Specific Modeling Language*) est un langage de modélisation spécifique à la modélisation des exigences qui permet d'exprimer des exigences dans une grammaire proche de la langue naturelle. L'éditeur a été défini à l'aide du studio Gemoc[2] [9] – une capture d'écran de l'outil est donnée Fig. 1. L'utilisation d'une approche basée modèles permet de le combiner avec d'autres langages, comme SysML ou Eiffel. Nous avons choisi d'en donner une définition formelle en Eiffel. Ainsi avec RSML, nous proposons une approche sans rupture, depuis l'élicitation des exigences jusqu'à la documentation d'une implémentation, en proposant un syntaxe concrète plus proche de la langue naturelle, tout en utilisant la sémantique d'Eiffel. Nous nous basons en effet sur une approche d'expression des exigences en Eiffel qui bénéficie ainsi de l'outil de preuve automatisée d'Eiffel, Autoproof [10]. RSML se base également sur le mécanisme de l'IDE EiffelStudio nommé EIS (*Eiffel Information System*[3]) qui permet d'établir des liens entre différents éléments.

2.1 Etude de cas illustrative

Afin d'illustrer notre approche, nous avons choisi d'utiliser l'étude de cas du London Ambulance Service (LAS) proposée dans [11]. Cet exemple décrit un système de traitement d'incidents survenus dans la ville de Londres, ainsi que l'envoi sur le terrain d'ambulances. Dans sa thèse, E. Letier décrit un certain nombre de buts (*goals*) afin de répondre à l'exigence :

> *An ambulance must arrive at the scene within 14 minutes for 95% of the calls.*

Par raffinements successifs, des buts opérationnels sont déduits, ce qui permet de proposer une première description du système.

Nous nous intéressons plus particulièrement à l'exigence *Every incident requiring emergency service is eventually resolved.*

[1] https://gitlab.com/fgalinier/RSML
[2] http://gemoc.org/studio.html
[3] https://www.eiffel.org/doc/eiffelstudio/Eiffel+Information+System

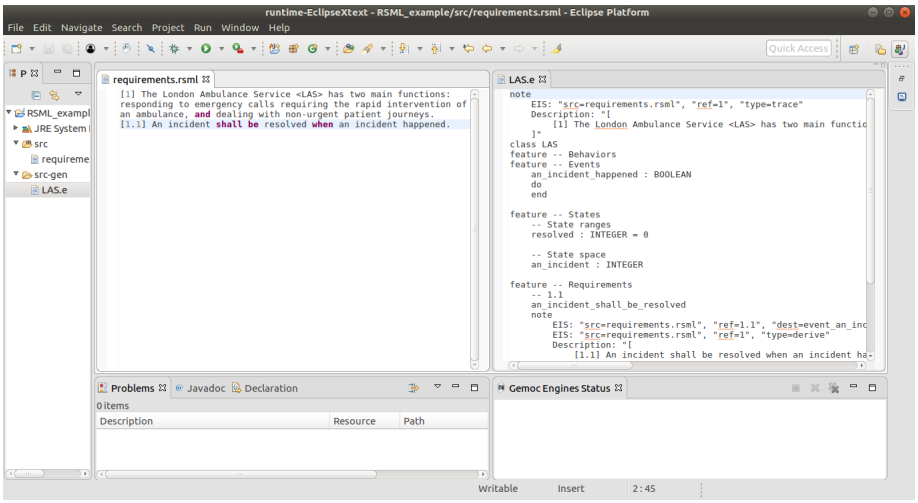

Figure 1: Interface de l'outil d'expression de RSML, avec la représentation en RSML et en Eiffel

2.2 Approche sans rupture

Dans [12], l'INCOSE définit le problème de la traçabilité comme étant un défi majeur de l'ingénierie des exigences. Ce que proposent les méthodes formelles, telles que Event-B [13] ou VDM [14], est d'exprimer les exigences à l'aide d'hypothèses et d'invariants et de définir la spécification du système de façon à vérifier ces invariants ; permettant ainsi de réaliser la preuve du système. Une génération de code est ensuite nécessaire, ce qui représente une rupture dans la chaîne de validation. De plus, ces approches restent souvent utilisées uniquement pour les parties critiques des systèmes ; celles-ci étant difficile à prendre en main pour des "non-experts".

L'idée présentée dans [7] et [8] consiste à entremêler dans un unique langage les différentes représentations des mêmes spécifications : que ce soient des exigences textuelles, des modèles ou encore du code. Ceci permet alors une meilleure traçabilité, liant fortement dans un même paradigme les exigences à leurs solutions. Les contrats d'Eiffel permettent d'exprimer les exigences de façon plus formelle et de vérifier que le code réponde bien aux exigences. Cependant, s'il est tout à fait possible d'ajouter dans les commentaires d'une méthode l'exigence satisfaite, le lien reste faible au sens où il n'a pas de sémantique définie.

Dans [15], A. Naumchev et B. Meyer proposent une méthode pour exprimer les exigences en Eiffel. L'exigence *Every incident requiring emergency service is eventually resolved* formalisée dans [11] telle que :

> **Goal** Achieve[IncidentResolved]
> **InformalDef** Every incident requiring emergency service is eventually resolved.
> **FormalDef** $\forall$ inc: Incident
> inc.Happened $\Rightarrow \Diamond$ inc.Resolved

serait ainsi traduite en Eiffel :

```
1 incident_resolved (inc: Incident)
2 require
3   inc.happened
4 do
```

```
5    inc.resolve
6  ensure
7    inc.resolved
8  end
```

Listing 1: Exemple d'exigence exprimée en Eiffel

où `inc.happened` est une hypothèse, `inc.resolve` la méthode devant satisfaire l'exigence et `inc.resolved` la condition à vérifier pour juger de la satisfaction de l'exigence. Le solveur Autoproof pourra analyser le code produit pour relever les violations d'exigences, s'il y en a.

Par ailleurs, quand il n'est pas possible d'utiliser une approche sans rupture (pour des raisons contractuelles, ou parce que des informations sont exprimées dans d'autres paradigmes), le mécanisme EIS (*Eiffel Information System*) du principal IDE du langage de programmation Eiffel, EiffelStudio, permet de lier une méthode (appelée `feature` en Eiffel) ou une `class` avec un document externe (et de manière plus précise il est possible de préciser un signet d'un document Microsoft Word ou d'un document PDF), via le mécanisme de *note* (équivalent aux annotations en Java). Nous avons étendu[4] ce mécanisme pour permettre l'établissement de liens « plus fins » (c'est à dire plus précis car permettant de lier non plus des blocs de code – classes ou méthodes – mais également des assertions du langage). De plus, nous avons ajouté la notion de type de relation afin de permettre l'ajout de sémantiques spécifiques aux relations en lien avec les exigences (décrites section 2.3). L'exemple décrit ci-dessus (Listing 1) peut alors être enrichi de la sorte :

```
1  incident_resolved (inc: INCIDENT)
2  note
3    EIS: "src=requirements.docx", "ref=req1", "dest=
         assume_inc_happened,check_inc_resolved", "type=verify"
4    EIS: "src=INCIDENT.resolve", "type=satisfy"
5    Description: "[
6      Every incident requiring emergency service is eventually
           resolved.
7    ]"
8  require
9    assume_inc_happened: inc.happened
10 do
11   inc.resolve
12 ensure
13   check_inc_resolved: inc.resolved
14 end
```

Listing 2: Exemple de liens EIS en Eiffel

Dans Listing 2, l'ajout de liens plus fins permet de lier l'exigence *req1* située dans le document *requirements.docx* non seulement à `incident_resolved`, mais directement aux assertions qui implémentent cette exigence, c'est à dire la précondition `assume_inc_happened` et la post-condition `check_inc_resolved`. De plus, les liens entre artefacts du même langage permettent également de lier l'exigence à la méthode qui la satisfait, soit la méthode `resolve` de la classe `INCIDENT`, afin d'expliciter la traçabilité au sein même du langage. Ces différents ajouts permettent ainsi de créer plus de liens de traçabilité (et des liens plus fins) pour faciliter l'analyse

[4] https://github.com/fgalinier/EiffelStudio

de l'impact d'un changement.

Afin de faciliter ces ajouts, le langage RSML permet de générer automatiquement les liens considérés. Ainsi, l'exigence exprimée en RSML :

```
[1] The London Ambulance Service <LAS> has two main functions:
responding to emergency calls requiring the rapid intervention of
an ambulance, and dealing with non-urgent patient journeys.
```

qui correspond à la grammaire (simplifiée) :

<**FreeRequirement**> ::='[' <ID> ']' <FreeText>*
'<' <ClassName> '>' <FreeText>* '.'

sera traduite en Eiffel par la génération de la classe `LAS` suivante :

```
1  note
2          EIS: "src=requirements.rsml", "ref=1", "type=trace"
3          Description: "[
4                  [1] The London Ambulance Service <LAS> has two main
                       functions: responding to emergency calls
                       requiring the rapid intervention of an ambulance,
                        and dealing with non-urgent patient journeys.
5          ]"
6  class LAS
7
8  end
```

Listing 3: Représentation en Eiffel de la documentation associée à la classe LAS

Cela permet de générer le corps des classes, déjà documenté, avec des liens de traçabilité.

En complément, en nous appuyant sur les *patterns* que nous définissons dans [16], nous proposons une syntaxe contrainte pour les exigences fonctionnelles qui va permettre de générer les représentations en Eiffel de celles-ci. L'exigence :

```
[1.1] An incident shall be resolved when an incident happened.
```

correspond à la grammaire (simplifiée) :

<**StateRequirement**> ::='[' <ID> ']' <Target> <Priority> 'be' <State>
('when' <Condition>)? '.'
<**Priority**> ::='must' | 'shall' | 'could' | 'would'
<**Condition**> ::=<StateCondition> | <EventCondition>

et sera traduite (avec la version actuelle de l'outil) en :

```
1  -- 1.1
2  an_incident_shall_be_resolved
3  note
4      EIS: "src=requirements.rsml", "ref=1.1", "dest=
           event_an_incident_happened, an_incident_is_resolved", "type=
           verify"
```

```
5       EIS: "src=requirements.rsml", "ref=1", "type=derive"
6       EIS: "src=an_incident_shall_be_resolved_impl", "type=satisfy"
7       Description: "[
8                   [1.1] An incident shall be resolved when an incident
                        happened.
9           ]"
10  require
11      event_an_incident_happened: an_incident_happened
12  do
13      an_incident_shall_be_resolved_impl
14  ensure
15      an_incident_is_resolved: an_incident = resolved
16  end
```

Listing 4: Représentation en Eiffel de l'exigence 1.1

où `an_incident_happened` est une méthode booléenne valant vrai ou faux, et `an_incident` un état représentant si l'incident est résolu ou non. Le corps de l'exigence est un appel à la méthode vide nommée `an_incident_shall_be_resolved_impl`, qui devra être modifiée pour permettre la satisfaction de cette exigence.

2.3 Relations en lien avec les exigences

Si les approches industrielles comme IBM Rational Doors [1] ou Reqtify [2] ne définissent pas de type pour les relations en lien avec les exigences, ces relations sont typées dans les approches GORE (Goal Oriented Requirement Engineering) comme KAOS [17], ou encore dans le diagramme d'exigences de SysML.

La définition de différent types de liens renforce la traçabilité, car elle permet de comprendre quel est le rôle d'un lien. Les relations liées aux exigences en KAOS ou en SysML permettent à la fois de lier les exigences entre elles, mais aussi de lier les exigences avec des parties du système. Nous avons choisi d'incorporer dans notre approche les relations d'exigences existantes en SysML pour deux raisons principales :

- ces relations couvrent quasiment toutes les relations de KAOS qui établissent des liens entre exigences et entre exigences et système (à l'exception de la décomposition OR),
- SysML étant un langage de modélisation d'intérêt croissant dans l'industrie, nous espérons pouvoir fournir des liens de transformations de modèles entre SysML et RSML.

Il convient de distinguer deux grandes catégories de relations : les relations entre exigences, et les relations entre les exigences et les autres artefacts. La relation *trace* quant à elle est une relation qui permet de définir un lien de traçabilité sans plus de sémantique. La relation *trace* est donc orthogonale à ces deux catégories de relations.

2.3.1 Relations entre exigences

Ces relations permettent de lier des exigences à d'autres exigences : ce sont *refine*, *contains*, *copy* et *derive*.

Refine: Elle est sans doute la relation la mieux connue en ingénierie des exigences. Elle est notamment utilisée dans les méthodes formelles. Ainsi, une exigence R_1 affine[5] une autre exigence R_2, si R_1 est une version plus précise de R_2. Autrement dit, elle peut relâcher les hypothèses pour traiter plus de cas, mais elle doit prendre en compte au minimum les invariants de l'exigence qu'elle affine – qu'elle peut renforcer. Elle correspond au lien *«refine»* de SysML ou globalement au *refinement* de KAOS. Quand plusieurs exigences affinent la même exigence de départ, il s'agit le plus souvent d'exprimer des choix possibles.

Contains: La relation de contenance permet de décomposer une exigence en plusieurs sous-exigences. Ainsi, si une exigence contient d'autres exigences, elle ne sera vérifiée que si toutes ses sous-exigences sont vérifiées. Par ailleurs, une exigence ne peut être contenue que dans une unique autre exigence. *Contains* correspond au lien de *containment* (ou $\oplus-$) de SysML ou à la *AND-decomposition* de KAOS.

Copy: Elle permet de créer une copie d'une exigence. Elle correspond au lien *«copy»* de SysML et n'a pas d'équivalent en KAOS. Cette relation est très peu utilisée, mais nous l'avons définie dans un soucis de complétude vis-à-vis des relations SysML existantes.

Derive: Une exigence R_1 est dérivée d'au moins une autre exigence R_2 si elle est déduite de celle-ci mais qu'elle n'en est pas un raffinement. Ainsi, R_1 doit être satisfaite pour que R_2 soit totalement satisfaite, mais elle peut également être nécessaire à la satisfaction d'autres exigences. Elle correspond au lien *«derivedReq»* de SysML et n'a pas d'équivalent direct en KAOS, qui utilise plutôt des variations de sa relation de *refinement*. Il arrive souvent qu'une même exigence dérive de plusieurs autres (e.g., une exigence d'accélération dérivant d'une exigence de puissance et d'une exigence de poids).

2.3.2 Relations entre exigences et autres artefacts

Ces relations relient les exigences aux autres artefacts de modélisation : ce sont les relations de satisfaction (*satisfy*) et de vérification (*verify*). Elles permettent une traçabilité qui pourra servir aussi bien à des objectifs de documentation ou de facilitation des évolutions futures, qu'à une éventuelle automatisation de la validation. Ainsi, il est par exemple possible de savoir s'il existe des exigences non satisfaites ou non vérifiées (notion de couverture), ce qui permet de faire le lien entre les trois premières étapes de la décomposition classique de l'ingénierie des exigences [18] (élicitation, analyse et modélisation) et l'étape de vérification et de validation, qui nécessite à la fois les exigences et la description du système.

Satisfy: Une exigence est satisfaite s'il existe au moins un élément du système – par exemple une `feature` en Eiffel – qui implémente les éléments nécessaires à la satisfaction de cette exigence. C'est la relation *«satisfy»* de SysML ou la relation *operationalization* de KAOS.

Verify: Une exigence est vérifiée par un artefact si cet artefact permet de s'assurer que l'exigence est bien satisfaite – une `assertion` en Eiffel. Elle correspond au lien *«verify»* de SysML et n'a pas d'équivalent en KAOS.

[5] Ou *raffine*.

2.4 Sémantique formelle

Notre objectif principal étant d'automatiser la vérification des liens, nous proposons une sémantique pour ces liens, définie dans la Table 1. La syntaxe utilisée dans le tableau est la suivante :

- R_i est une exigence (textuelle ou RSML) ;
- r_i est la modélisation en Eiffel de l'exigence R_i ;
- f est une méthode ou attribut (`feature`) en Eiffel ;
- a est une assertion en Eiffel (précondition, postcondition, invariant).

Relationships	**Semantics**
Trace	Link with no semantics
Refine	R_1 refines $R_2 \implies r_1$ redefine r_2
Contains	R_1 contains $R_2 \implies$ r_2 is called in $r_1 \wedge (\nexists\ r_3 : R_3 \mid r_2$ is called in $r_3)$
Copy	R_1 copies $R_2 \implies$ r_1 body is a unique call to r_2
Derive	R_1 derives from $R_2 \implies$ r_1 is called in r_2
Satisfy	f satisfies $R_1 \implies f$ is called in r_1
Verify	a verifies $R_1 \implies a$ is an assertion of r_1

Table 1: Sémantique des relations entre exigences en Eiffel

Ainsi, l'exigence 1.1 définie dans le Listing 5 ci-dessous est satisfaite par la méthode (`feature`) `an_incident_shall_be_resolved_impl`, qui est appelée dans son corps (`do`) à la ligne 6.

```
1 -- 1.1
2 an_incident_shall_be_resolved
3 require
4     event_an_incident_happened: an_incident_happened
5 do
6     an_incident_shall_be_resolved_impl
7 ensure
8     an_incident_is_resolved: an_incident = resolved
9 end
```

Listing 5: Représentation (simplifiée) en Eiffel de l'exigence 1.1

La méthode représentant l'exigence 1.1 en Eiffel n'a cependant pas vocation à être appelée dans une utilisation normale du système. L'appel au solveur statique Autoproof sur l'implémentation de l'exigence `an_incident_shall_be_resolved` va cependant simuler l'exécution du code se trouvant dans son corps pour vérifier que les postconditions (ligne 8) sont bien vérifiées, en se basant sur l'hypothèse des préconditions (ligne 4). Dans Fig. 2 est donné le retour d'un appel à Autoproof sans aucune spécification du système. Le solveur est par conséquent incapable de prouver que la postcondition est assurée.

```
====================================
LAS.an_incident_shall_be_resolved
Verification failed (0.01s).

Line: 36. Postcondition an_incident_is_resolved may be violated.
====================================
```

Figure 2: Échec de la vérification de `an_incident_shall_be_resolved`

Après spécification suffisante et correcte du système, l'appel au solveur renvoie le message donné Fig. 3, qui indique par conséquent que l'exigence 1.1 est bien satisfaite et que l'implémentation est valide.

```
====================================
LAS.an_incident_shall_be_resolved
Successfully verified (0.00s).
====================================
```

Figure 3: Succès de la vérification de `an_incident_shall_be_resolved`

L'appel à Autoproof permet donc de donner tout au long du développement du système un aperçu de la couverture des exigences.

2.5 Illustration des avantages

L'utilisation de cette approche présente plusieurs avantages. Elle permet tout d'abord d'avoir dans un unique paradigme la documentation, les exigences, la spécification, les contrats et éventuellement l'implémentation répondant à ces exigences (ceci grâce aux liens forts que nous rendons possible d'établir entre les exigences et le mécanisme EIS). Elle permet également d'intégrer les exigences de manière quasi-formelle dans le processus de Vérification et de Validation, tout en demeurant plus accessible que les méthodes formelles. Par ailleurs, l'utilisation du solveur Autoproof de manière statique permet de s'assurer tout au long du développement du projet que le système est bien celui attendu. Enfin, RSML est un langage de modélisation. Si le langage est actuellement formalisé en Eiffel, il est possible de le porter vers d'autres formalismes (tels qu'Event-B, KAOS, SysML ou même vers des documents Microsoft Word). Ceci permettrait de faire des ponts avec les outils habituels des utilisateurs.

De plus, l'application à l'exemple du LAS ou encore du Landing Gear System (LGS) défini dans [19], a permis de détecter des problèmes lors de la conception des systèmes implémentant ces études de cas. En effet, en appliquant notre méthode, nous nous sommes rendu compte qu'une exigence du LAS que nous avions traduite par :

```
[1.3] An ambulance shall not be available when an ambulance is
allocated.
```

devait en fait être traduite par :

```
[1.3] An allocated ambulance shall not be available.
```

La différence résidant dans le fait que la seconde implique que les deux états ne peuvent pas être simultanés, tandis que la première implique que quand une ambulance est allouée, elle ne peut par la suite être disponible, sans prendre en compte un changement d'état de l'ambulance entre temps. De même, des erreurs ont été trouvées dans l'étude de cas LGS [16].

L'utilisation d'outils complémentaires, tels que Stimulus ou KAOS, pour l'élicitation doit permettre de couvrir des cas que nous ne pouvons pas traiter par preuve automatique (qui seraient couverts par l'approche model-checking de Stimulus par exemple).

Bien que ne bénéficiant pas d'un environnement de preuve aussi complet et interactifs que ceux qui entourent des technologies telles que VDM ou Event-B, notre approche introduit une formalisation des exigences dans des systèmes où ce n'est habituellement pas fait, en proposant un paradigme plus simple d'accès.

3 Conclusion et perspectives

Nous avons montré dans cet article comment exprimer les exigences d'un système dans un langage proche de la langue naturelle, tout en permettant de prouver formellement la validité de ce système au regard de ces exigences. Cette approche, située dans un paradigme sans rupture, dote de sémantique les différentes relations en lien avec les exigences en se fondant sur le langage Eiffel.

Le langage RSML ainsi que l'outillage associé sont en cours de développement et toutes les fonctionnalités ne sont pas encore opérationnelles, mais devraient l'être à court terme.

L'approche présentée dans cet article fournit des résultats encourageants. Appliquée pour l'instant à de petits cas d'utilisation (en terme de nombre d'exigences), elle nécessite cependant d'être validée sur des cas d'utilisation plus conséquents et plus variés (en terme de types d'exigences).

Cette approche étant basée sur l'idée de globalisation des modèles, nous envisageons de l'étendre en créant des ponts avec d'autres formalismes d'expression des exigences (SysML entre autres), permettant ainsi aux utilisateurs de continuer à utiliser leurs outils habituels tout en bénéficiant des ajouts proposés par notre approche.

References

[1] IBM Rational Doors. https://www.ibm.com/us-en/marketplace/requirements-management. Accessed: 2018-05-23.

[2] Dassault Systems Catia Reqtify. https://www.3ds.com/products-services/catia/products/reqtify. Accessed: 2018-05-23.

[3] Object Management Group (OMG). *OMG Systems Modeling Language (OMG SysML^{TM}), V1.0.* 2007. OMG Document Number: formal/2007-09-01 Standard document URL: http://www.omg.org/spec/SysML/1.0/PDF.

[4] 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.

[5] Bertrand Jeannet and Fabien Gaucher. Debugging real-time systems requirements: simulate the "what" before the "how". In *Embedded World Conference, Nürnberg, Germany*, 2015.

[6] Seong-ick Moon, Kwang H. Lee, and Doheon Lee. Fuzzy branching temporal logic. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 34(2):1045–1055, April 2004.

[7] Richard Paige and Jonathan 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.

[8] Bertrand Meyer. Multirequirements. *Modelling and Quality in Requirements Engineering (Martin Glinz Festscrhift)*, 2013.

[9] Benoit Combemale, Olivier Barais, and Andreas Wortmann. Language engineering with the gemoc studio. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pages 189–191, April 2017.

[10] 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 Berlin Heidelberg, April 2015. DOI: 10.1007/978-3-662-46681-0_53.

[11] Emmanuel Letier. *Reasoning about agents in goal-oriented requirements engineering*. PhD thesis, PhD thesis, Université catholique de Louvain, 2001.

[12] INCOSE. *SE Vision 2025*. 2014. http://www.incose.org/docs/default-source/aboutse/se-vision-2025.pdf.

[13] Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, New York, NY, USA, 1st edition, 2010.

[14] Dines Bjørner and Cliff B. Jones, editors. *The Vienna Development Method: The Meta-Language*, volume 61 of *LNCS*. Springer-Verlag, 1978.

[15] Alexandr Naumchev and Bertrand Meyer. Seamless requirements. *Computer Languages, Systems & Structures*, 49:119–132, 2017.

[16] Alexandr Naumchev, Bertrand Meyer, Manuel Mazzara, Florian Galinier, Jean-Michel Bruel, and Sophie Ebersold. Expressing and verifying embedded software requirements. *arXiv preprint arXiv:1710.02801*, 2017.

[17] Axel van Lamsweerde. Goal-oriented requirements engineering: a guided tour. In *Proceedings Fifth IEEE International Symposium on Requirements Engineering*, pages 249–262, 2001.

[18] Ian Sommerville and Pete Sawyer. *Requirements Engineering: A good practice guide*. John Wiley & Sons, Inc., 1997.

[19] 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. DOI: 10.1007/978-3-319-07512-9_1.