Une méthode de formalisation progressive des exigences basée sur un modèle simulable[1]

Clémentine Nebut * — **Franck Fleurey**[**]

* *IUT de Bayonne*
Château Neuf – Place Paul Bert
F-64100 Bayonne
nebut@iutbayonne.univ-pau.fr

** *IRISA*
Campus universitaire de Beaulieu
F-35042 Rennes cedex
Franck.Fleurey@irisa.fr

RÉSUMÉ. *Les exigences d'un logiciel, souvent rédigées en langage naturel, sont à la base des phases de conception et de test fonctionnel. Le langage naturel est par nature ambigu, et les exigences peuvent donc être différemment interprétées lors de la construction et de la validation du logiciel. C'est pourquoi nous proposons une méthode de raffinement progressif des exigences vers des modèles dont la sémantique est connue. À partir d'exigences en langage naturel contrôlé, nous proposons tout d'abord de générer un modèle fonctionnel simulable basé sur les cas d'utilisation UML ; la simulation permet l'amélioration des exigences et peut aussi servir de base à un mécanisme de génération de tests. À partir des exigences ainsi améliorées, nous proposons également la génération d'une ébauche de modèle statique.*

ABSTRACT. *Software requirements, usually wrutten in natural language, are the basis for design and functionnal testing phases. Due to natural language ambiguity, the requirements can thus be interpretyed differently while building and validating the software. We thus propose an incremental refinement method from the requirements to models with well-known semantics. From requirements in controlled natural language, we first generate a simulatable functional model based on UML use cases, then the simulation is used to enhance the requirements and to generate tests. We also generate a draft of static model*

MOTS-CLÉS : *modèles d'exigences, cas d'utilisation, UML.*

KEYWORDS: *requirement models, use cases, UML.*

1. Ces travaux ont été partiellement supportés par le projet MUTATION, faisant partie du programme de recherche Carroll. Mutation repose sur la collaboration entre Thalès Research and Technology, Thalès Airborne Systems, le CEA et l'INRIA. Voir www.carroll-research.org

1. Introduction

La continuité dans le processus de raffinement des spécifications d'un système vers sa modélisation, son implémentation et sa validation, est un problème crucial, en particulier pour les systèmes de grande taille. En effet, les premières étapes de spécification des exigences, généralement écrites en langage naturel, doivent être intensivement utilisées pour la conception et le test du produit final. Ce problème de continuité concerne tant les aspects d'analyse et traitement du langage naturel, que les aspects de complétude et de cohérence des exigences. La plupart des solutions proposées dans la littérature sont basées sur l'utilisation de méthodes formelles ; soit elles préconisent de rédiger les exigences directement sous forme de langage formel, soit elles proposent des mécanismes de traduction du langage naturel vers des langages formels :

– La spécification directe en langage formel offre de nombreux moyens de s'assurer de la cohérence des exigences ; cependant il n'est pas envisageable dans tous les contextes industriels de disposer de personnel compétent dans les langages formels, et les spécialistes du domaine ne sont pas toujours des informaticiens. De plus, résoudre le problème par l'utilisation de langages comme Z pour pallier les ambiguïtés du langage naturel reste un peu utopique, dans la mesure où comme le soulignent *Finkelstein et Emmerich* [FIN 00], la plupart des canevas formels comme Z sont accompagnés de larges parts de langage naturel, et sont inutilisables sans elles.

– Plusieurs travaux abordent la traduction des exigences en langage naturel vers un langage formel. *Fantechi et al* proposent ainsi dans [FAN 94] une traduction automatique du langage naturel vers le langage de logique temporelle ACTL. La traduction s'appuie sur un dictionnaire rempli par l'utilisateur. À partir de la formalisation en ACTL, des ambiguïtés peuvent être détectées : un manque d'information, ou la portée des conjonctions qui ne peuvent pas être résolues par le parenthésage du langage naturel. Les auteurs de [AMB 97] proposent un analyseur du langage naturel qui permet de produire différents types de modèles comme des modèles de flot de données, des diagrammes OMT, etc. L'analyse des différents modèles produits permet alors de détecter un certain nombre d'anomalies dans les exigences : l'inconsistance des diagrammes de flots de données, l'ambiguïté, et la redondance. *Fuchs et al* se basent sur le fait qu'un langage contrôlé peut remplacer la logique du premier ordre [FUC 99b], et proposent un tel langage, et le moyen de le transformer en logique du premier ordre [FUC 99a].

L'approche que nous proposons est basée sur un langage naturel contrôlé, qui sert de langage de surface pour la construction de modèles d'exigences avec une sémantique non ambiguë. Un mécanisme est proposé pour associer les éléments de syntaxe du langage avec des éléments d'un modèle de cas d'utilisation simulable. Les exigences sont ainsi rendues simulables. Être capable de simuler les exigences permet de les valider et de s'assurer de leur cohérence, et nous paraît être une caractéristique importante de cette approche : si la littérature propose pléthore de méthodes de validation des exigences exprimées avec un langage formel, aucune ne garantit l'équivalence entre ce qui est écrit dans les exigences formalisées et les exigences fournies par le client,

ou plus largement les souhaits du client. Nous proposons également de générer une ébauche de modèle statique à partir des exigences.

Ce papier est organisé comme suit. La section 2 donne un aperçu de la méthode proposée, la section 3 donne une brève description du langage d'exigences, la section 4 explique la construction des modèles d'exigences par interprétation des exigenecs, et la section 5 décrit la construction de modèles statiques et de modèles de test. La section 6 conclut.

2. Vue globale et méthodologique de l'approche

L'approche que nous proposons vise à compléter les exigences pour les rendre non-ambiguës afin qu'elles puissent servir à la génération d'autres artéfacts de conception. Le point d'entrée de notre approche est un ensemble d'exigences textuelles écrites dans notre Langage de Description des Exigences (LDE), qui, même s'il restreint l'usage du langage naturel, autorise l'écriture de phrases ambiguës, incomplètes, voir incorrectes. Nous proposons ensuite un processus d'amélioration incrémental de ces exigences, de manière à ce qu'elles atteignent un niveau de précision équivalent à celui d'une approche formelle. Ainsi, après une première étape d'écriture d'exigences, celles-ci sont progressivement modifiées de manière à les rendre complètes, non ambiguës et cohérentes, ce qui est difficile à atteindre en une seule étape. Cette approche incrémentale est présentée Figure 1, et se compose de trois étapes principales : l'analyse syntaxique, l'interprétation, et la simulation des exigences.

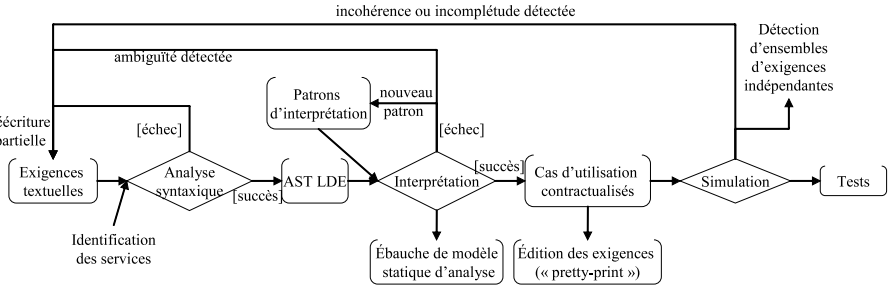

Figure 1. *Approche incrémentale d'amélioration des exigences*

La phase d'analyse syntaxique

L'analyse syntaxique produit un arbre syntaxique abstrait (AST, Abstract Syntax Tree) représentant les exigences. Si une exigence ne peut pas être analysée syntaxiquement, elle doit être modifiée jusqu'à ce qu'elle soit correcte vis-à-vis de la syntaxe LDE.

La phase d'interprétation

La phase d'interprétation a pour but de déterminer l'interprétation de l'AST LDE issue de l'analyse syntaxique. Le principe est d'appliquer des *patrons d'interprétation* qui fixent la sémantique associée au LDE. Les patrons d'interprétation lient les structures de l'AST LDE à une sémantique formelle, ce sont des règles de traduction du modèle syntaxique au modèle sémantique. Si une exigence ne peut être interprétée à l'aide des patrons d'interprétation, alors :

– soit cette exigence n'a pas de signification telle qu'elle est écrite, et doit être réécrite,

– soit un nouveau patron d'interprétation doit être défini et ajouté à l'ensemble des patrons d'interprétation existants,

– soit l'exigence n'est pas d'un niveau correct d'abstraction (par exemple si l'exigence décrit le *comment* plutôt que le *quoi* : si elle décrit comment le système fonctionne et non pas ce que fait le système).

L'exigence peut aussi avoir de multiples interprétations. Le rédacteur des exigences doit alors choisir l'interprétation correcte, et si possible modifier l'exigence textuelle pour supprimer l'ambiguïté de la phrase. Cette ambiguïté est classiquement due à la combinaison d'opérateurs logiques et temporels, sans parenthésage explicite.
L'interprétation peut déboucher sur deux types de modèles : un modèle fonctionnel, et une ébauche de modèle statique.

– Le modèle fonctionnel se compose de cas d'utilisation contractualisés, i.e augmentés de pré et post conditions (voir Section 4). Ce modèle reflète en fait la sémantique associée aux exigences. Il est à noter qu'à ce stade, les cas d'utilisation déduits peuvent être reformatés sous forme textuelle, par rétro-ingénierie, de manière à vérifier la compatibilité entre l'interprétation formelle des exigences, et ce que le rédacteur de l'exigence avait en tête (sortie « pretty printing »).

– L'ébauche de modèle statique est une synthèse des différentes composantes du système telles qu'elles peuvent être déduites des exigences, comme cela sera illustré Section 5.

La phase de simulation

Le modèle de cas d'utilisation qui est produit par la phase d'interprétation est simulable. Pratiquée exhaustivement, la simulation permet tout d'abord de détecter les exigences totalement déconnectées des autres exigences. De telles exigences doivent être étudiées pour s'assurer qu'elles ont bien lieu d'être dans le système décrit, s'il n'y a pas lieu de scinder le système en sous-systèmes plus ou moins indépendants, ou si des liens existant entre les exigences n'ont pas été omis ou sous-spécifiés. La simulation est également et surtout un moyen de valider les exigences, ou d'y détecter des erreurs ou des manques. Ainsi, la simulation permet l'amélioration et la complétion des exigences. Elle permet également de guider la génération de tests.

La traçabilité au centre du processus

La traçabilité est une problématique centrale de notre processus : tout au long des différentes transformations à partir des exigences textuelles, des informations de traçabilité sont conservées ; ainsi, à tout moment, on peut déterminer pour chaque élément construit par nos transformations quelle est l'exigence ou quel est l'ensemble d'exigences qui lui a donné naissance. Nous assurons bien sûr la traçabilité entre cas d'utilisation et exigences, ce qui est un minimum. Nous permettons également la traçabilité entre le modèle statique et les exigences. En effet, l'ébauche de modèle statique que nous générons contient toutes les informations de traçabilité nécessaires : elles peuvent être réutilisées lors de la conception des modèles statiques. Enfin, puisque nous générons des tests à partir des cas d'utilisation comme cela sera détaillé dans le chapitre suivant, chaque test généré se réfère à la liste d'exigences concernées.

Exemple illustratif

Le reste de ce document sera illustré en utilisant un exemple de serveur de réunions virtuelles déjà décrit dans [NEB 03], dans lequel des participants peuvent entrer et sortir de réunions planifiées par des organisateurs, y parler, etc.

3. Le LDE, langage de description des exigences

Le LDE a été originellement défini pour les systèmes d'aviation THALÈS dans le cadre du projet CARROLL (voir note de bas de page 1) ; cependant il n'est pas spécifique à ce domaine d'application. Nous en définissons ici la syntaxe, la sémantique étant donnée par l'application de patrons d'interprétation. Le LDE ne définit ainsi qu'une formalisation structurelle des exigences, c'est-à-dire une syntaxe : il n'y a pas de sémantique particulière associée à un AST LDE. Autrement dit, le LDE est juste un langage de surface permettant la construction aisée de modèles d'exigences.

Le LDE est indépendant du domaine d'application, et cela influence directement sa grammaire, qui ne doit contenir aucun mot clef spécifique à un domaine particulier. Les terminaux du langage sont de deux types : les terminaux de la grammaire, et les termes du domaine. Nous n'utilisons pas de dictionnaire contenant les termes du domaine, et ceux-ci doivent donc être identifiés par un autre biais. Ce problème est résolu dans le LDE en marquant (ici en mettant entre guillemets) tous les mots du domaine. Ces mots spécifiques au domaine ne font pas partie de la grammaire du langage, c'est le rédacteur des exigences qui les définit comme tels en les mettant entre guillemets, comme cela est illustré dans l'exemple suivant.

Before a "manager" does "plan" a "meeting", this "manager" must be "connected".

Dans cet exemple, chaque mot spécifique à l'exemple de réunion virtuelle est placé entre guillemets. Les autres mots de cette phrase exemple font partie du LDE, et n'ont qu'une signification grammaticale. Les mots placés entre guillemets sont considérés comme des terminaux du langage et ne sont donc pas analysés. Cette approche implique que les structures fournies par le LDE sont des squelettes de structures gram-

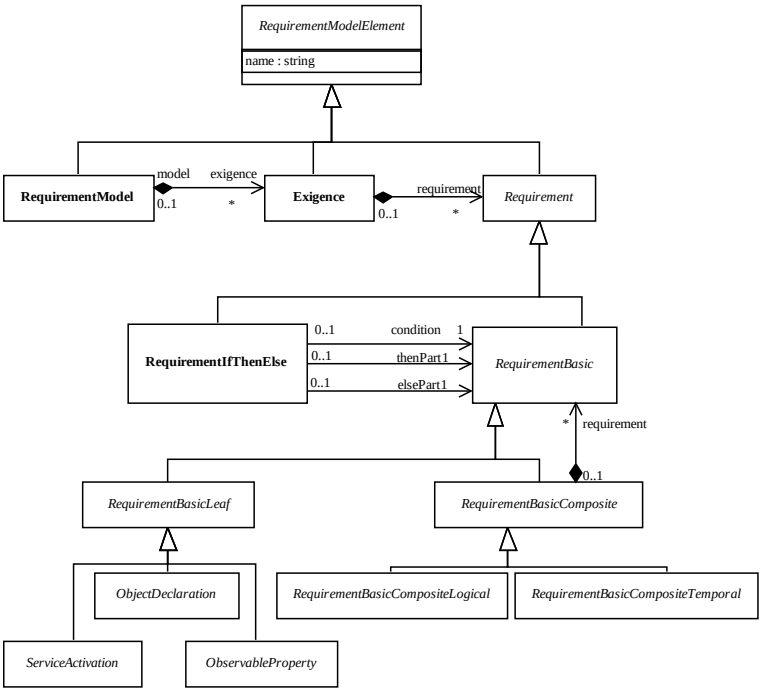

Figure 2. *Modèle de l'*AST LDE

maticales extraites du langage naturel (ici de l'anglais). De ce point de vue, analyser une phrase LDE va générer une formalisation des exigences reflétant leur structure grammaticale (dans notre cas, un AST dont un aperçu est donné Figure 2 sous forme d'un diagramme de classe UML). Dans ce modèle, un modèle d'exigences peut contenir des modèles d'exigences et des exigences. Une exigence peut être de différents types : exigence conditionnelle (possédant une condition, une exigence qui doit être vérifiée si la condition est vérifiée, et éventuellement une exigence qui doit être vérifiée dans le cas contraire), exigence contenant de la logique temporelle, déclaration d'entités, activation de service, etc.

Le LDE a été construit dans le but de rester aussi proche que possible du langage naturel, et le langage s'appuie donc sur des constructions de phrase de type sujet/verbe. Dans la mesure où un autre but du LDE est de décrire un système à un haut niveau, le couple sujet/verbe doit être naturellement compris comme un couple acteur/action. Le rédacteur des exigences peut définir une propriété sur un objet du domaine, et le langage lui permet d'utiliser des liens logiques entre ces propriétés tels que la négation, la conjonction, et la disjonction, comme cela est illustré dans les exemples suivants.

– The "participant" is "connected".

– The "status" of a "meeting" is "open".

– The "participant" is "connected" and the "meeting" is not "closed".

Le LDE permet de plus de décrire un système avec des structures causales et causales temporelles. Les structures temporelles permettent de définir les conditions qui doivent être vérifiées avant ou après l'exécution d'une action. Ces structures sont illustrées dans les exemples suivants.

– If a "participant" does "close" a "meeting" then each "participant" does "leave" this "meeting".

– Before a "participant" does "enter" a "meeting", this "meeting" must be "opened".

– After a "participant" did "open" a meeting, each "participant" being "connected" can "enter" this "meeting".

Pour décrire le système, il est possible de déclarer les objets du domaine présents dans le système, et leur statut à l'état initial du système, comme dans l'exemple suivant.

There is a "participant" named "Emma". The "status" of "Emma" is "connected".

Une part importante du langage est l'usage de quantificateurs. Pour le moment, cinq quantificateurs peuvent être utilisés : a/an, this, the, each, one.

– Le quantificateur "a" ou "an" suivi d'un identificateur représente n'importe quel objet de la catégorie référencée par l'identificateur. Par exemple, a "meeting" représente n'importe quelle réunion définie dans le système.

– Le quantificateur "this" référence un objet non défini précisément (c'est-à-dire quantifié par un "a /an") qui apparaît dans la même phrase.

– Le quantificateur "the" souligne que l'objet référencé doit être un singleton, c'est-à-dire qu'il ne doit y avoir qu'un seul objet de cette catégorie dans le système.

– Le quantificateur "each" décrit l'universalité et le quantificateur "one" l'existence.

4. Analyse sémantique et construction d'un modèle fonctionnel

Les principales structures du LDE ont été décrites dans la section précédente à un niveau syntaxique. Nous introduisons ici un formalisme pour définir une sémantique non ambiguë à chaque structure du LDE. La sémantique est donnée en termes de cas d'utilisation UML contractualisés, qui sont décrits dans la sous-section suivante.

4.1. *Un modèle fonctionnel basé sur les cas d'utilisation*

Le modèle de cas d'utilisation que nous proposons fait intervenir pour chaque cas d'utilisation des paramètres et des contrats. Le but d'un tel modèle est de formaliser

suffisamment la notion de cas d'utilisation de manière à rendre leur évaluation possible sans ambiguïté. Ce modèle est simulable, les principes de simulation ne sont pas détaillés ici, les principes étant exposés dans [NEB 03].

Les paramètres d'un cas d'utilisation représentent les concepts qui y sont impliqués. Ainsi, les acteurs d'un cas d'utilisation sont un type particulier de paramètres. Les paramètres sont souvent des concepts métiers manipulés ou utilisés par le cas d'utilisation. Ces concepts métiers pourront être réifiés dans la conception du système ; dans la phase d'analyse des exigences, ils sont juste identifiés comme des concepts métiers auxquels les cas d'utilisation sont intrinsèquement liés. À titre d'exemple, si on s'intéresse au cas d'utilisation *planifier* de la réunion virtuelle, on lui associe deux paramètres : la réunion qui est planifiée, et le participant qui planifie.

Chaque cas d'utilisation possède une pré-condition et une post-condition. La pré-condition d'un cas d'utilisation exprime les conditions nécessaires pour l'exécution du cas d'utilisation. La pré-condition peut refléter des contraintes sur les paramètres, ou sur l'état du système du point de vue métier. La post-condition exprime la manière dont le système est modifié par l'exécution du cas d'utilisation. Dans les différentes approches [COC 97, COC 01, D'S 99] préconisant d'ajouter de tels contrats aux cas d'utilisation, les contrats sont écrits sous forme textuelle. En effet, ni le canevas de Cockburn ni la méthode Catalysis ne proposent de langage d'action ou de contraintes pour les cas d'utilisation. En proposant un langage de contrats pour les cas d'utilisation, nous appliquons à un niveau d'abstraction supérieur l'approche de conception par contrats de Bertrand Meyer [MEY 92].

Un langage de contrats pour les cas d'utilisation

Le langage de contrats pour les cas d'utilisation est nommé UCL (*Use case Contracts Language*), et est très proche des expressions logiques du premier ordre : UCL inclut les expressions logiques du premier ordre et y ajoute un certain nombre de constructions. Les expressions logiques que nous proposons manipulent à l'aide d'opérateurs booléens et de quantifieurs des propriétés booléennes sur le système.

Propriétés manipulées. Les propriétés manipulées portent sur les instances d'entités métier qui ont été définies dans le système. Les propriétés booléennes peuvent par exemple exprimer des informations sur le statut d'un acteur, sur les différents rôles, ou sur l'état d'une ressource. La sémantique associée aux propriétés booléennes pourra être implicitement déduite de son nom, ou explicitée par un dictionnaire attaché au système de cas d'utilisation. Ces propriétés sont donc des prédicats d'arité quelconque. Par exemple :

– *created(m : Meeting)* est une propriété vraie si la réunion m est créée et fausse sinon

– *manager(u : Participant,m : Meeting)* est une propriété vraie si et seulement si le participant *u* est l'organisateur de la réunion *m*.

Les propriétés énumérées sont du sucre syntaxique rajouté à l'UCL pour en faciliter l'usage. Elles n'augmentent pas l'expressivité d'UCL : une transformation bijective

existe entre les propriétés énumérées et les propriétés booléennes. Une propriété énumérée permet par exemple d'associer à une réunion un type parmi les valeurs *{democratic, private, standard}*. On exprimera alors par exemple le fait qu'une réunion m est privée par l'expression : $type(m) = "private"$.

Opérateurs. Les opérateurs pouvant être utilisés pour combiner des propriétés sont les opérateurs de la logique du premier ordre : la conjonction, la disjonction, la négation, l'implication. Les quantifieurs universel et existentiel peuvent aussi être utilisés.

La figure 3 fournit un exemple de cas d'utilisation contractualisés en UCL pour la réunion virtuelle. Un participant peut ouvrir une réunion s'il en est l'animateur, et que la réunion est créée tout en étant ni ouverte ni fermée. La réunion devient alors ouverte. Similairement, un participant peut fermer une réunion s'il en est l'animateur et que la réunion est ouverte. La réunion devient alors fermée, non ouverte, et ne contient plus de participants.

```
UC  open(u : participant ; m : meeting)
pre  created(m)  and  moderator(u, m)  and not  closed(m)  and not  opened(m)
   and  connected(u)
post  opened(m)
UC  close(u : participant ; m : meeting)
pre  opened(m)  and  moderator(u, m)
post not  opened(m)  and  closed(m)  and
   forall(v : participant) {not  entered(v, m)  and
   not  asked(v, m)  and not  speaker(v, m) }
```

Figure 3. *Exemples de cas d'utilisation contractualisés*

4.2. *Définition de la sémantique du* LDE

Conceptuellement, la traduction des exigences textuelles en un modèle de cas d'utilisation est une transformation de modèles, comme suggéré dans l'approche Model-Driven Engineering [BéZ 03]. Les méta-modèles des modèles d'entrée et de sortie sont exprimés de manière à ce qu'ils soient compatibles avec le méta-méta modèle MOF (Meta Object Facility, [OMG]). La transformation de modèles que nous avons ainsi définie peut être vue comme une sémantique du LDE. Techniquement, cette transformation fonctionne par application d'une technique de recherche de motifs (*pattern matching*).

4.2.1. *Structure d'un patron d'interprétation*

Un patron d'interprétation est un couple [motif, production], où le motif est une structure du modèle d'entrée, et où la production représente ce qui est produit dans le modèle de sortie à la rencontre du motif. Nous avons défini un ensemble de tels patrons d'interprétation, dans lesquels les motifs sont des structures dans l'AST LDE

des exigences LDE analysées, et une production est un cas d'utilisation contractualisé (ou au moins une partie d'un cas d'utilisation).

Pour illustrer la notion de patron d'interprétation, considérons un patron d'interprétation PI_1 décrivant le fait qu'une propriété d'un objet change à l'activation d'un cas d'utilisation. Le motif M_1 de ce patron d'interprétation PI_1 est donné tableau 1. Un patron d'interprétation s'appuie fortement sur la structure de l'AST LDE donnée partiellement Figure 2. La première ligne de ce tableau décrit la forme générale du motif, et les lignes suivantes décrivent des contraintes additionnelles nécessaires à l'application du patron d'interprétation correspondant (i.e. des contraintes sur les éléments lexicaux provenant de l'analyse lexicale des exigences LDE). Ainsi, ce motif détecte les structures *IF ... THEN* dont la conséquence est une propriété observable (*observable property*) de type *BECOMES* (c'est-à-dire la modification d'un attribut) et s'applique à un objet non explicitement identifié (l'objet référencé est quantifié par *A*), et dont la cause est une action de type *DOES* (c'est-à-dire une activation simple de service).

IF S1=Action THEN O1=ObservableProperty	
O1.type	BECOMES
O1.reference	A
S1.type	DOES

Tableau 1. *Un exemple de motif de patron d'interprétation*

Type	USE CASE
Titre	S1.title
Paramètres	x1 : S1.activator.type x2 : O1.reference.owner.type
Précondition	not O1.reference.observable = O1.reference.value
Postcondition	O1.reference.observable = O1.reference.value

où S1.title (resp. S1.activator.type) est le nom (resp. le type de l'activateur) du service S1, et où O1.reference.owner.type est le type de l'observable O1, O1.reference.observable est 1'observable O1, et O1.reference.value sa valeur.

Tableau 2. *Exemple de production de patron d'interprétation*

Pour définir le patron d'interprétation PI_1, il faut associer le motif M_1 avec une production P_1. Dans notre exemple, P_1 doit produire un cas d'utilisation indiquant que la propriété observable O1 doit changer de valeur à l'activation de l'action S1. La production P_1 est donnée tableau 2. Cette production définit une transformation (et donc la sémantique) d'une phrase LDE donnée en un cas d'utilisation décrit en termes d'objets et d'attributs LDE. Par exemple, l'application du patron d'interprétation PI_1

à la phrase : if a participant does plan a meeting then this meeting becomes planned produit le cas d'utilisation suivant :

UC Plan(p :Participant,m :Meeting)
pre not planed(m)
post planed(p,m)

En d'autres termes, le patron d'interprétation PI_1 exprime le fait que l'utilisation du mot-clef *becomes* permet de spécifier qu'une propriété booléenne d'un objet change de valeur au cours d'une action spécifique.

4.2.2. *Agrégation de cas d'utilisation*

Pendant l'interprétation d'une phrase LDE, il peut arriver (et c'est même le cas général) que plus d'une phrase décrive le même service du système. La transformation du modèle LDE vers le modèle de cas d'utilisation va alors générer plusieurs cas d'utilisation pour le même service. Il faut alors agréger ces cas d'utilisation en un seul. Ainsi, considérons deux cas d'utilisation UC_1 et UC_2 décrivant le même service. Ces deux cas d'utilisation sont agrégés en un cas d'utilisation UC_a décrit tableau 3.

Cas d'utilisation	UC_1	UC_2	UC_a
Précondition	pre_1	pre_2	$pre_1 \, or \, pre_2$
Postcondition	$post_1$	$post_2$	$(pre_1 \, implies \, post_1) \, and \, (pre_2 \, implies \, post_2)$

Tableau 3. *Agrégation de cas d'utilisation*

Le processus d'agrégation permet au rédacteur des exigences de décrire un même service complexe en plusieurs phrases simples. Un autre bénéfice de l'agrégation est qu'il est possible de décrire un même service à différents endroits des exigences, ce qui peut être utile pour l'organisation des exigences. L'agrégation est enfin un moyen de détecter des incohérences entre plusieurs exigences : par exemple si on génère un cas d'utilisation nécessitant une chose ou son contraire, on doit étudier si les exigences sont bien cohérentes.

5. Modèles générés à partir des exigences

À l'issue du processus incrémental d'amélioration des exigences, celles-ci sont suffisamment détaillées, cohérentes et complètes pour servir à la génération d'autres modèles : ébauche de modèle statique, et modèle fonctionnel permettant la génération d'objectifs de tests.

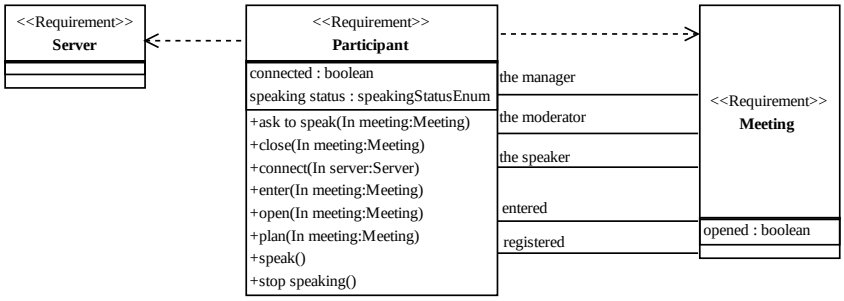

Figure 4. *Modèle statique d'analyse généré pour le système de réunion virtuelle*

5.1. *Génération d'une ébauche de modèle statique*

À partir des exigences LDE, une ébauche de modèle statique peut être générée. Cette ébauche est en fait une synthèse de toutes les entités décrites dans les exigences : leurs propriétés et leur liens avec les autres entités. Ce modèle est un bon support pour conception d'un premier modèle d'analyse. Pour rester dans la notation UML, nous présentons le modèle statique généré sous forme de diagramme de classes, mais les classes et les attributs sont stéréotypés de manière à les distinguer de vraies classes.

Le principe de génération de cette ébauche est très simple, et est illustré par la figure 4 qui donne l'ébauche de modèle statique construite pour la réunion virtuelle. Les propriétés observables de chaque entité du système sont collectées. Les entités sont représentées par des classes stéréotypées *Requirement.* Dans l'exemple, 3 entités ont été identifiées, et sont représentées par les 3 classes *Meeting*, *Participant* et *Server* de la figure 4. Les propriétés n'impliquant qu'une entité sont représentées par des attributs stéréotypés *Requirement* de l'entité. On détermine pour cela si les propriétés sont de type booléen ou énuméré. Quand elles sont de type énuméré, on crée le type énuméré correspondant. Par exemple, la propriété *connected* n'implique que l'entité *Participant*, elle est représentée par l'attribut *connected* dans la classe *Participant*, et est de type booléen. Les propriétés impliquant deux entités sont transformées en associations entre ces deux entités. Par exemple, la propriété *registered* implique une réunion et un participant, elle est représentée par une association entre les classes *Participant* et *Meeting*. Tous les cas d'utilisation sont représentés par des méthodes de la classe représentant leur acteur principal. Dans l'exemple, les cas d'utilisation ont tous été représentés par des méthodes de la classe *Participant*. Les paramètres des méthodes sont ceux du cas d'utilisation représenté, excepté celui représentant l'acteur principal du cas d'utilisation.

5.2. *Modèle fonctionnel et modèles de test*

Dans la mesure où la sémantique du LDE est donnée en termes des cas d'utilisation, les exigences permettent d'obtenir un modèle fonctionnel suffisamment détaillé pour servir à la génération automatique de test. Le mécanisme de génération de test a été publié dans [NEB 03], nous en rappelons ici les principes. Notre modèle de simulation comprend un système de cas d'utilisation contractualisés, la déclaration des différentes entités utilisées dans le système, et la définition d'un état initial. À tout moment de la simulation, on mémorise une abstraction de l'état du système sous forme d'une valuation des prédicats instanciés définis dans le système de cas d'utilisation (un prédicat instancié étant un prédicat dont les paramètres formels ont été remplacés par des paramètres effectifs). La simulation consiste à déterminer quels sont les cas d'utilisation applicables avec quels paramètres. Une simulation exhaustive du système mène à la construction d'un graphe comportemental du système sur lequel on peut facilement raisonner : il permet de vérifier des propriétés sur les cas d'utilisation par des techniques de model checking, ainsi que de générer des objectifs de test en extrayant des chemins pertinents du graphe comportemental (en utilisant des critères de test adéquats). Les détails sur le graphe comportemental et son utilisation pour le test peuvent être trouvés dans [NEB 03].

6. Conclusion

Dans ce papier nous avons présenté une approche permettant l'amélioration itérative des exigences, ainsi que la génération de premiers modèles de conception. Les contributions sont plus d'ordre méthodologique que technique : nous ne prétendons pas proposer une technique performante d'analyse du langage naturel. En revanche, le processus et les techniques présentées permettent une écriture non laborieuse des exigences par des équipes coopérantes, et la détection des incohérences ou des imprécisions via une analyse sémantique ou via la simulation. De plus, notre méthode permet de capitaliser des règles d'écriture des exigences et leur sémantique exacte, sous forme de patrons d'interprétation. Enfin, nous générons un modèle fonctionnel et une ébauche de modèle statique synthétisant les connaissances extraites de l'ensemble des exigences pour chaque entité du système. Les transformations successives entre les différents modèles conservent des informations de traçabilité, et cette approche permet donc d'améliorer la traçabilité entre les exigences, les modèles d'analyse et de conception, et les tests. Cette approche a été expérimentée sur certains composants développés à Thalès Airborne Systems, dans le cadre du projet CARROLL/MUTATION. Pour les composants étudiés, environ 70 % des exigences initiales ont pu être traduites en LDE, et l'amélioration de nos outils prototypes permettraient d'en couvrir 80 %. Les 20 % restant concernent soit des aspects temps-réels que nous ne pouvant actuellement pas traiter avec notre modèle, soit des exigences de trop bas niveau, et que nous ne souhaitons pas couvrir, notre approche visant des exigences de plus haut niveau. Des travaux futurs viseront à enrichir le LDE et le modèle de cas d'utilisation de manière à pouvoir spécifier des exigences temps-réel. Les expériences menées avec

THALÈS Airborne System ont montré que l'usage de notre approche permet l'unification du vocabulaire utilisé, l'explicitation d'exigences implicites, ainsi que la réécriture d'exigences ambiguës.

7. Bibliographie

[AMB 97] AMBRIOLA V., GERVASI V., « Processing natural language requirements », *Proc of the International Conference on Automated Software Engineering 1997*, 1997.

[BéZ 03] BÉZIVIN J., FARCET N., JÉZÉQUEL J.-M., LANGLOIS B., POLLET D., « Reflective Model Driven Engineering », *Proceedings of UML 2003, San Francisco*, LNCS, Springer, octobre 2003.

[COC 97] COCKBURN A., « Structuring Use Cases with goals », *Journal of Object-oriented Programming*, vol. 10, n° 5/7, 1997, p. 35–40 and 56–62.

[COC 01] COCKBURN A., *Writing Effective Use Cases*, Addison Wesley, 2001.

[D'S 99] D'SOUZA D., WILLS A., « *Objects, Component, ans Frameworks with UML, The Catalysis approach* », chapitre Interaction Models : Uses cases, Actions, and collaborations, Addison-Wesley, 1999.

[FAN 94] FANTECHI A., GNESI S., RISTORI G., CARENINI M., VANOCCHI M., MORESCHINI P., « Assisting Requirement Formalization by Means of Natural Language Translation », *Formal Methods in System Design*, vol. 4, n° 3, 1994, p. 243–263.

[FIN 00] FINKELSTEIN A., EMMERICH W., « The Future of Requirements Management Tools », position paper in Information Systems in Public Administration and Law, 2000.

[FUC 99a] FUCHS N., SCHWERTEL U., SCHWITTER R., « Attempto Controlled English – Not Just Another Logic Specification Language », *Proceedings of the 8th international workshop on Logic-Based Program Synthesis and Transformation (Selected Papers)*, vol. 1559, 1999, p. 1–20.

[FUC 99b] FUCHS N., SCHWERTEL U., TORGE S., « Controlled Natural Language Can Replace First-Order Logic », *Proc. of Automated Software Engineering 1999*, 1999, p. 295–298.

[MEY 92] MEYER B., « Applying design by contract », *Computer*, vol. 25, n° 10, 1992, p. 40–51.

[NEB 03] NEBUT C., FLEUREY F., LE TRAON Y., JÉZÉQUEL J.-M., « Requirements by Contracts allow Automated System Testing », *Proc. of the 14th. IEEE International Symposium on Software Reliability Engineering (ISSRE'03)*, 2003.

[OMG] OMG, http ://www.omg.org/technology/documents/formal/mof.htm.