Articulation entre activités formelles et activités semi-formelles dans le développement de logiciels

Imen Sayar

UNIVERSITÉ DE LORRAINE

École doctorale IAEM Lorraine

Articulation entre activités formelles et activités semi-formelles dans le développement de logiciels

THÈSE

présentée et soutenue publiquement le 28 mars 2019

pour l'obtention du

Doctorat de l'Université de Lorraine

(mention informatique)

par

Imen SAYAR

Composition du jury

Président :	M. Vincent CHEVRIER	Professeur, Université de Lorraine
Rapporteurs :	Mme Maritta HEISEL	Professeure, Université de Duisburg-Essen
	M. Jean-Michel BRUEL	Professeur, Université de Toulouse
Examinateurs :	M. Yves LEDRU	Professeur, Université de Grenoble Alpes
	M. Mohamed Tahar BHIRI	Maître de conférences (HDR), Université de Sfax
Directrice :	Mme Jeanine SOUQUIÈRES	Professeure, Université de Lorraine

Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503

Mis en page avec la classe thesul.

Remerciements

Je voudrais tout d'abord adresser mes immenses remerciements à ma directrice de thèse, Madame *Jeanine SOUQUIÈRES*, Professeure à l'Université de Lorraine, qui m'a accueillie dans son équipe Dedale et m'a encadrée tout au long de cette thèse. Qu'elle soit aussi remerciée pour sa gentillesse, sa disponibilité permanente et pour ses nombreuses qualités scientifiques et humaines. Grâce à son suivi, j'ai pu mener cette thèse à son terme.

J'exprime ma gratitude à Monsieur *Vincent CHEVRIER*, Professeur à l'Université de Lorraine, qui a bien voulu être président de mon jury de thèse.

J'adresse tous mes remerciements à Madame *Maritta HEISEL*, Professeure à l'Université de Duisburg-Essen, ainsi qu'à Monsieur *Jean-Michel BRUEL*, Professeur à l'Université de Toulouse, pour l'honneur qu'ils m'ont fait en acceptant de rapporter sur ma thèse et de participer au jury.

Je remercie également Monsieur *Yves LEDRU*, Professeur à l'Université de Grenoble Alpes pour avoir accepté d'être un examinateur de ma thèse.

Je tiens à remercier particulièrement Monsieur *Mohamed Tahar BHIRI*, Maître de Conférences HDR à l'Université de Sfax pour avoir accepté d'être membre de mon jury et pour ses conseils pertinents. J'apprécie ses qualités scientifiques et humaines exceptionnelles.

Merci au programme *EGOV-TN* d'*Erasmus-Mundus* qui a financée trente quatre mois de ma thèse. Merci également aux personnels de ce programme qui ont été présents en permanence pour répondre efficacement à mes questions et gérer mes documents durant la durée de ma bourse Erasmus.

Il m'est toujours impossible d'oublier *Mourad KMIMECH* pour m'avoir aidée à trouver le financement des trois ans de ma thèse, je suis très reconnaissante.

Enfin, je remercie tous mes amis et collègues du laboratoire *LORIA*. Il me sera très difficile de citer toutes ces personnes. Un merci particulier à *Émilie*, *Justine*, *Thomas*, *Christophe*, *Julien*, *Régis*, *Fahad*, *Jérémy*, *Yannick*, *Damien*, *Guy* et *Thierry* pour leur soutient et leurs conseils très précieux.

Imen SAYAR

Aux membres de ma famille, grands et petits

À ma sœur, Asma

À l'âme de mon défunt frère, Faouzi

Sommaire

Table des figures

Liste des tableaux

Chapitre 1

Introduction

1.1 Cadre de la thèse

De nos jours, les systèmes deviennent complexes et la technologie s'intègre de plus en plus dans notre quotidien. Une incorporation convenable invite à l'utilisation du formel surtout quand il s'agit des systèmes critiques où l'erreur n'est pas tolérée et conduit à des pertes du matériel, de l'argent et des vies. Ces dernières années, l'utilisation des méthodes formelles connaît une hausse importante. Par exemple, des projets dans le domaine du transport tels que celui de la ligne 14 du métro parisien appelé METEOR (METro Est-Ouest Rapide) [Su et al., 2011] et CRISTAL [Colin et al., 2008], [Yang and Jacquot, 2011b] pour le convoi de voitures autonomes ainsi que des travaux en médecine comme le pacemaker [Méry and Singh, 2011c], [Méry and Singh, 2011b] ont adopté les méthodes formelles telles que B et B événementiel dans leur développement. Ces méthodes assurent un fonctionnement correct des systèmes obtenus.

Les cahiers des charges décrivant ces systèmes présentent souvent des lacunes et des incohérences. Une mauvaise qualité de ces documents mène à des échecs et des coûts élevés de correction des systèmes voire leur rejet [4] [Johnson et al., 1994]. L'adoption du formel est importante mais elle doit être complétée par une bonne qualité du cahier des charges. Afin d'aborder ce problème, il est important d'étudier le rapport entre l'informel (ou semi-formel) représenté par le cahier des charges et le formel.

1.2 Contexte et aperçu de notre approche

Cette thèse expose un travail interactif entre les deux mondes de l'informel et du formel. Nous travaillons avec un système composé de trois éléments, voir figure 1.1 :

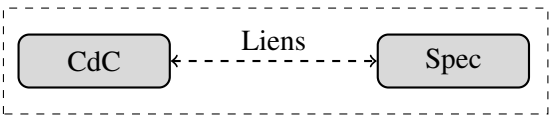


FIGURE 1.1 – Une étape de développement

- le cahier des charges réécrit par le développeur et noté *CdC*,

4. https://www.standishgroup.com/store/services/10-chaos-report-decision-latency-theory-2018-package.html

- sa spécification formelle correspondante notée *Spec* et
- les liens entre le cahier des charges *CdC* et sa spécification formelle *Spec* appelés *Liens*.

L'objectif de ce travail est d'établir, expliciter et décrire des liens et des interactions entre le CdC et la Spec correspondante en prenant en compte les échanges avec le client. Chacun de ces documents influe l'état de l'autre en y apportant des modifications. La place des outils dans notre approche est importante et leur utilisation est présente dans toutes les étapes du développement. Le système < CdC, Liens, Spec > évolue en permanence à tous les niveaux du processus de développement, voir figure 1.2, avec le passage d'un état à un autre suite à un choix de développement. Ce choix peut être :

- la prise en compte des besoins du CdC pour développer la Spec,
- l'ajout ou la suppression d'un besoin,
- l'introduction de termes formels dans le CdC,
- l'ajout ou la suppression d'éléments formels dans la Spec en cours de construction ou
- la mise à jour du système < CdC, Liens, Spec > suite à l'analyse des retours des activités de vérification et/ou de validation.

L'évolution du système engendre un nouvel état décrit par le système < CdC $'$, Liens $'$, Spec $'$ > :

- CdC$'$ représente le nouvel état du CdC,
- Spec$'$ est la Spec mise à jour et
- Liens$'$ sont les Liens mis à jour automatiquement grâce aux outils disponibles.

Durant ce travail, nous renforçons l'existence du CdC dans un processus formel de développement, les besoins étant décrits en langage naturel.

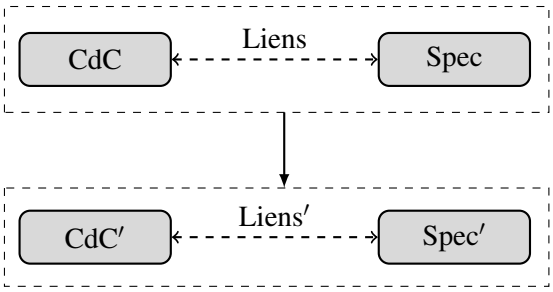


FIGURE 1.2 – Évolution du système

1.2.1 CdC

Nous réécrivons le cahier des charges du client sous forme d'une suite de phrases courtes, étiquetées et hiérarchisées. Nous appelons le document obtenu *CdC*. La hiérarchie dans ce document désigne une relation entre les besoins qui le constituent. Elle peut être de type :

- *frère-frère* c'est-à-dire les besoins sont de même niveau ou
- *parent-enfant* : dans ce cas le besoin enfant détaille le besoin parent.

Nous sauvegardons la trace du CdC et suivons son évolution tout au long du processus formel. Au fur et à mesure du développement, des termes venant du monde formel sont introduits dans le texte des exigences. Une exigence représente la demande du client pour un futur système ou logiciel. Une description complète du CdC, tel qu'il est utilisé dans notre approche, est présentée dans le chapitre 3.

1.2.2 Spec

Elle représente une formalisation des exigences du CdC. La construction de la spécification formelle est réalisée pas-à-pas via le raffinement. Cette spécification doit être vérifiée et validée au fur et à mesure de sa construction. Grâce aux outils disponibles et à certains concepts offerts par B événementiel tels que le raffinement et la preuve au sens mathématique, nous montrons tout au long de ce mémoire l'application de nos contributions en utilisant ce langage doté de la plateforme Rodin [5]. Une description détaillée de la Spec ainsi que des outils de vérification et de validation est fournie dans la section 2.2 du chapitre 2. Les activités de vérification et de validation apportent des retours sur le CdC permettant d'améliorer la qualité du système < CdC, Liens, Spec >.

Bien que nous ayons effectué le travail de thèse en grande partie en utilisant la méthode B événementiel, nous nous intéressons également au langage TLA$^+$. Celui-ci est un langage formel permettant la description des enchaînements des actions dans un système. Notre objectif est de montrer que notre approche ne dépend pas d'un langage particulier ; nos contributions sont généralisables à d'autres langages.

1.2.3 Liens

Les liens sont établis entre le CdC et sa Spec correspondante. Ils sont exprimés dans les deux sens dans notre approche. Ils sont gérés par l'outil ProR sous Rodin et définis via le contenu d'un glossaire rassemblant des termes formels issus de la Spec et leurs définitions informelles venant du CdC. La notion de liens est importante. Outre le pont qu'ils établissent entre le CdC et la Spec, ces liens jouent un rôle essentiel dans les activités de vérification et de validation. Ce point est détaillé dans le chapitre 4.

1.2.4 Outils

Le développement et l'évolution du système < CdC, Liens, Spec > se réalise grâce à l'utilisation des outils. Notre choix de la méthode formelle B événementiel est justifié par les facilités offertes par ce langage exprimées par la présence des outils permettant d'appliquer notre approche. Un atout de cette méthode formelle est la plateforme Rodin qui utilise Eclipse. Cette plateforme est extensible par des plugins favorisant la facilité d'éditer, prouver, animer, simuler les modèles formels et de les lier avec les besoins du CdC. Il s'agit de :

- ProR pour la gestion des exigences et l'établissement des liens avec la Spec,
- générateur des obligations de preuve (OPs),
- prouveurs automatiques et interactifs pour prouver les OPs,
- animateur et model-checker ProB pour la validation de la Spec,
- Project Diagram pour une visualisation globale des composants (Context et Machine) de la Spec et des liens (Sees, Extends et Refines) entre eux et
- EventB StateMachines pour une vision de la Spec sous forme de machines d'états-transitions.

Nous utilisons ces outils dans toutes les étapes du développement. Nous les exploitons pour améliorer la qualité du système < CdC, Liens, Spec >. Un apport de Rodin est son interface utilisateur englobant plusieurs vues. Ceci favorise une meilleure compréhension du système en cours de développement et aide à gérer plus facilement les liens entre les deux mondes.

5. http://wiki.event-b.org/

1.3 Motivations

En génie logiciel, le développement d'un logiciel ou d'un système commence par la compréhension de son cahier des charges. Ce dernier représente la demande du client et est consulté par toutes les parties prenantes à tous les niveaux du développement. Il souffre souvent d'une pauvreté dans sa description et des lacunes affectant la qualité du logiciel final. La nécessité de prendre soin de ce document, de renforcer son existence et de mémoriser sa trace tout au long du développement se présente. Sa formalisation est une des approches proposées dans cette finalité. Ces approches ont pour objectif l'obtention des systèmes et logiciels corrects. Ceci est favorisé grâce à leur capacité de raisonnement rigoureux assuré par la technique de raffinement et promettant la qualité de correction. Telle qualité est essentielle en génie logiciel. Elle englobe deux aspects qui doivent être respectés par le logiciel final : il doit être vérifié et validé. La vérification concerne la cohérence du logiciel en soi. La validation s'intéresse à sa cohérence par rapport au document d'exigences qui représente un contrat entre le développeur et le client. D'une part, le client doit fournir et décrire ses besoins au développeur. D'autre part, ce client a le droit d'avoir un logiciel qui répond à sa demande en retour.

A tout moment de la construction des modèles formels, leur vérification et validation, le développeur peut rencontrer des difficultés telles que des ambiguïtés et détecter des anomalies venant des besoins. Dans ce cas, il peut avoir des échanges avec le client sous forme de discussions afin de clarifier les ambiguïtés. Le problème qui se pose là est que ces deux acteurs n'utilisent pas le même langage. Le client parle un langage naturel ou un langage venant de son métier. Le développeur se sert des langages de spécification, de conception et de programmation pour réaliser le système demandé. Ce qui crée un écart entre les deux mondes. Il est important de se faire comprendre et de faire participer le client dans le processus de développement pour garantir une meilleure qualité du logiciel final. Cette qualité dépend également de la coopération et des échanges entre les différentes parties prenantes du document d'exigences : entre équipes ou développeurs de la même équipe. Les besoins sont souvent exprimés en langage naturel. Par conséquent, le problème de multi-compréhension s'impose : un besoin peut être compris de différentes manières par les développeurs. D'où l'importance de simplifier la description des besoins afin d'aider à leur compréhension.

L'activité de validation du modèle formel nécessite la consultation des besoins du client. Cette tâche devient difficile quand il s'agit d'un modèle complexe. Ceci nous invite à gérer les exigences dans le processus formel de développement pour assurer un accès facile à l'information lors de la validation.

Pour ces raisons, il s'avère important de réduire l'écart entre les besoins et leur spécification formelle associée. C'est la problématique que nous étudions tout au long de cette thèse. Nous nous focalisons sur les interactions entre ces documents afin d'améliorer la qualité du système < CdC, Liens, Spec >.

1.4 Contributions

Nos travaux sont fondés sur deux bases : raisonner et enrichir. Nous raisonnons sur et gérons les deux mondes informel et formel tout en enrichissant leurs documents tout au long du développement. L'objectif est de les faire évoluer en permanence et de bâtir un pont entre eux.

Nos principales contributions se focalisent sur trois axes :

- *La validation*. Dans les approches usuelles, cette activité est considérée comme une étape du processus formel de développement. Dans notre approche, cette étape est un processus rigoureux commençant dès les phases en amont du développement c'est-à-dire dès le traitement des exigences. Ce processus démarre par une étape de préparation à la validation du futur modèle formel en extrayant et mémorisant, depuis le CdC, des éléments typés tels que des données, fonctionnalités, obligations et comportements. L'apport de cette contribution est que nous n'attendons pas que le modèle formel soit terminé pour le valider, nous le validons au fur et à mesure de sa construction. Le processus de validation prend en compte la nature des exigences. Une exigence est de type :
  - — *"donnée"* si elle s'intéresse aux données existantes dans le futur système,
  - — *"fonctionnalité"* si elle définit un ou plusieurs services du futur système,
  - — *"obligation"* si elle décrit des contraintes sur le fonctionnement du système et
  - — *"comportement"* si elle explique le comportement attendu du futur système.

  Nous ne traitons pas le cas des exigences de type *"hypothèse"*. Nous étudions le cas des systèmes fermés, c'est-à-dire des systèmes dans lesquels l'environnement n'agit pas sur leur fonctionnement.
- *La vérification*. Vérifier un modèle formel consiste à assurer qu'il est défini correctement, c'est-à-dire qu'il est mathématiquement correct. L'activité de vérification concerne le monde du formel. Dans notre approche, cette activité concerne aussi bien le formel que l'informel. Les retours de cette activité peuvent donner des indications sur des lacunes dans le CdC telles que des oublis ou des contradictions et contribue à l'amélioration de sa qualité. La vérification via des preuves au sens de B événementiel, invite à se poser des questions sur la source du non déchargement des obligations de preuve, OPs. Selon Abrial [Abrial, 2003], quatre cas liés au résultat de la preuve d'un énoncé d'un modèle formel se présentent :
  - — L'énoncé est vrai, le modèle formel est prouvé correct.
  - — L'énoncé est faux, le modèle doit être mis à jour.
  - — La preuve ne réussit pas mais l'énoncé est néanmoins *probablement prouvable ou réfutable*. Le modèle est pauvre et ne contient pas les détails suffisants pour prouver des propriétés et décharger des OPs. Dans ce cas, ce modèle doit être revu et enrichi afin de le rendre prouvable.
  - — La preuve ne réussit pas et l'énoncé n'est *probablement ni prouvable ni réfutable*. Dans ce cas, le modèle est trop pauvre et doit être revu.

  La localisation de la source du non-déchargement des OPs n'est pas triviale. Des efforts importants sont effectués pour y parvenir. Ces efforts concernent l'analyse des OPs non-déchargées afin de comprendre d'où vient l'erreur. Dans certains cas, l'analyse s'avère difficile et demande des connaissances en matière de preuves formelles. Nous utilisons le contre-prouveur ProB sous Rodin pour résoudre ce problème. Il permet de donner un contre-exemple sur l'OP non déchargée. Ceci donne une indication qui nous guide pour localiser facilement la source d'erreur.
- *La prise en compte de nouveaux besoins dans un développement existant*. Nous traitons le problème de l'intégration de ces besoins et leur communication avec les autres besoins existants dans le CdC : seront-ils frères ou enfants ? Quel sera l'effet de leur ajout sur la Spec et sur les Liens ? Nous appliquons nos idées sur l'étude de cas du distributeur automatique de billets. La prise en

compte d'un nouveau besoin révèle l'existence des défaillances dans l'ancien système < CdC, Liens, Spec > existant. Par exemple, nous montrons que l'arrivée du besoin *"le système doit servir des clients malvoyants"* a permis de détecter l'importance de l'affichage pour les autres clients n'ayant pas de difficultés visuelles, ce qui n'a pas été pris en compte dans le système développé précédemment. Ceci entre dans la catégorie des oublis de besoins évidents selon Abrial.

- *La définition des patrons de développement*. Ces patrons facilitent le développement du système < CdC, Liens, Spec >. Ils sont paramétrés et utilisés dans les études de cas présentées avec un cahier des charges. Ce document est décrit avec une partie technique venant du client et une partie informatique décrite par l'informaticien. Nous nous intéressons à la partie informatique pour définir nos patrons. Utilisant le raffinement, ces patrons sont constitués :
  — d'une partie automatique et
  — d'une partie qui reste à compléter par le développeur. Elle concerne essentiellement la description de la gestion des paramètres des patrons ainsi que le *collage*. Ce concept est défini dans le formel. Nous traitons son sens dans le CdC : présence de plusieurs cas au même instant, un état peut exclure un autre état, etc.

1.5 Présentation des études de cas

Nous avons appliqué notre approche sur trois études de cas dont deux sont de taille importante.

1.5.1 Train d'atterrissage d'un avion

Il s'agit de la description d'un système de contrôle assurant le fonctionnement correct d'un train d'atterrissage d'un avion. Le cahier des charges de ce système a été proposé comme étude de cas de la conférence ABZ 2014 [Boniol and Wiels, 2014]. Il est réparti en :

- une partie *technique* expliquée en dix-sept pages et
- une partie *besoins et propriétés* décrite en deux pages à la fin du document.

Le système fonctionne selon deux modes, nominal et urgence. Ce système hybride est décrit par :

- une partie hydro-mécanique regroupant les composants matériels du système à savoir les portes, les boîtes contenant les trains d'atterrissage, les cylindres et les électro-vannes assurant les mouvements,
- une partie digitale transformant des actions mécaniques en données numériques pour les transmettre au logiciel de contrôle et
- une interface installée dans le tableau de bord de la cabine de pilotage de l'avion. Elle contient :
  — des voyants lumineux donnant une idée sur la position des trains et des portes et
  — une manette (handle) pour gérer la montée et la descente des trains selon deux positions possibles : en haut ou en bas. Depuis cette interface, le pilote effectue des actions notamment le commande de la manette et l'activation du mode d'urgence.

La description de ce système s'intéresse à ces trois parties, à leurs communications ainsi qu'aux processus d'extension et de rétraction des trains. Ces processus commencent par une description globale via des instructions :

- ouverture de portes,
- pliage/dépliage des trains et
- fermeture des portes.

Ces instructions seront détaillées via des actions effectuées par des cylindres et des électro-vannes. A la fin, chaque action est définie sous forme de données gérées par des modules de calcul appartenant à la partie digitale.

Le train d'atterrissage est un système critique : toute défaillance de son fonctionnement causera des pertes matérielles et mettra en danger la vie des passagers à bord de l'avion. Nous montrons dans l'annexe A une partie du CdC associé à ce système, c'est-à-dire des besoins réécrits de ce cahier des charges d'ABZ 2014. Quelques évolutions du système < CdC, Liens, Spec > associé sont également fournies dans les chapitres 3, 4 et 7.

1.5.2 Machine d'hémodialyse

Une spécificité de son cahier des charges, proposé dans la conférence ABZ 2016 [Mashkoor, 2016], est qu'il décrit les cas anormaux du fonctionnement de la machine d'hémodialyse. L'objectif est de définir, en utilisant les méthodes formelles, un logiciel qui gère ses failles. Le cahier des charges décrit ces cas de défaillance en termes de trente-six besoins.

La machine d'hémodialyse est hybride. Elle est constituée d'une :

- partie matérielle décrite par la machine physique classique de dialyse et contenant des circuits de circulation du sang, un dialyseur, une pompe à sang et des solutions chimiques pour nettoyer le sang du patient,
- partie logicielle contrôlant le bon déroulement de la machine ainsi que le processus de dialyse et
- interface utilisateur depuis laquelle le médecin, l'infirmière ou le patient commandent la machine d'hémodialyse.

Le processus de dialyse est décrit selon trois phases :

1. *préparation* au cours de laquelle les composantes de la machine doivent être vérifiées et préparées pour l'opération de dialyse,
2. *thérapie* pendant laquelle le patient est connecté à la machine et l'opération de filtrage du sang ou de dialyse se déroule et
3. *fin de thérapie* durant laquelle la machine est déconnectée du patient et les opérations de nettoyage de cette machine sont effectuées.

L'annexe B présente le travail effectué pour cette étude de cas.

1.5.3 Distributeur automatique de billets

Le cahier des charges de ce système n'existe pas. Nous le construisons par nous-mêmes. Le distributeur automatique assure au client d'une banque le retrait d'argents moyennant une carte et son code pin associé. Certaines conditions doivent être satisfaites afin d'autoriser le retrait. Parmi ces conditions,

- la carte introduite dans le lecteur du distributeur doit être valide, c'est-à-dire qu'il s'agit d'une carte bancaire et qu'elle n'est pas bloquée par la banque,

- le code saisi par le client doit être valide et
- le montant demandé par le client ne doit pas dépasser le plafond autorisé par la banque.

L'objectif de cette étude de cas est de montrer l'évolution d'un système existant < CdC, Liens, Spec > par la prise en compte d'un nouveau besoin. Cette prise en compte résulte par un nouveau système. Elle est présentée en détail dans le chapitre 5.

1.6 Plan du mémoire

Ce mémoire est organisé en six chapitres et deux annexes :

- Le chapitre 2 décrit l'état de l'art de la thèse. Il est réparti en quatre sections représentant le cadre de nos travaux. La section 2.2 donne une vue détaillée du monde du formel via les spécifications formelles. Les approches et travaux reliant le monde des exigences avec le monde formel sont détaillés dans la section 2.3. La section 2.4 présente l'état de l'art en lien avec deux types de patrons existants, les patrons de conception et les patrons de spécification formelle.
- Les chapitres allant de 3 jusqu'à 7 détaillent nos travaux et notre approche de gestion et d'évolution des besoins liés avec leurs spécifications en tenant en compte de l'amélioration de leur qualité. Nous détaillons notre approche et décrivons les différents choix de développement, leurs traitements associés ainsi que leurs effets sur le système < CdC, Liens, Spec > dans le chapitre 3. Le chapitre 4 montre notre contribution relativement à l'activité de validation. Nous considérons cette activité comme un processus rigoureux qui commence dès le traitement des exigences et se réalise tout au long du développement du logiciel/système. Le chapitre 5 décrit la prise en compte d'un nouveau besoin dans un système < CdC, Liens, Spec > existant. Nous explicitons les questions sous-jacentes pour l'intégration de ce besoin et le rôle de cette intégration pour détecter des lacunes dans le système existant. Les chapitres 6 et 7 décrivent nos deux patrons de développement *Dev-if* et *Dev-seq* et leur intégration dans un processus de développement logiciel.
- Une conclusion générale et deux annexes, détaillant les évolutions des deux systèmes de trains d'atterrissage et de la machine d'hémodialyse, terminent ce document.

1.7 Publications

Nous avons diffusé notre approche par le biais des publications suivantes :

1- Imen Sayar and Jeanine Souquières, *La Validation dans le Processus de Développement*, Actes du XXXIVème Congrès INFORSID, Grenoble, France, pages 67–82, 2016.

2- Imen Sayar and Jeanine Souquières, *Du cahier des charges à sa spécification*, 16 ème journées AFADL, Montpellier, France, 2017.

3- Imen Sayar and Jeanine Souquières, *La validation dans les premières étapes du processus de dévelopement*, Revue ISI-DAT, numéro spécial "Décisions, argumentation et traçabilité dans l'Ingénierie des Systèmes d'Information", volume 22, numéro 4, pages 11–41, 2017.

4- Fahad Rafique Golra and Fabien Dagnat and Jeanine Souquières and Imen Sayar and Sylvain Guérin, *Bridging the Gap Between Informal Requirements and Formal Specifications Using Model Federation*, Software Engineering and Formal Methods - 16th International Conference, SEFM 2018, Toulouse, France, pages 54–69, 2018.

5- Poster : Imen Sayar, *Du cahier des charges à sa spécification : patrons de développement* , 16 ème journées AFADL, Montpellier, France, 2017.

Présentations orales

1- *Du Cahier des Charges à la Spécification Formelle ?*, Journée du groupe de travail "Ingénierie des Exigences" du GDR GPL, 9 octobre 2015, Paris, 45 mn.

2- *From requirements document to a formal specification ?*, Journée des doctorants D2 du département Méthodes Formelles (FM), 20 octobre 2015, Loria, Nancy, 30 mn.

3- *Articulation between formal and semi-formal activities in the development of software*, Journée des doctorants D2 du département Méthodes Formelles (FM), 11 octobre 2016, Loria, Nancy, 30 mn.

Chapitre 2

État de l'art

Sommaire

Ce chapitre présente l'état de l'art utilisé tout au long de notre thèse. Nous nous intéressons au monde informel via les besoins, aux travaux autour du monde du formel à travers les spécifications formelles et aux interactions entre ces deux mondes permettant de réduire l'écart entre eux. Nous décrivons également les travaux existants dans le cadre de réutilisation par le biais des patrons.

2.1 Besoins

Le développement d'un système ou logiciel commence par la compréhension des besoins du client. Ceux-ci, s'ils existent, sont rassemblés dans un document appelé *cahier des charges*. La qualité du système développé dépend considérablement de ce document.

2.1.1 Définitions

2.1.1.1 Exigence et besoin

Le concept d'*exigence* englobe plusieurs sens. De multiples travaux dans la littérature ont proposé des définitions pour ce terme :

- Selon le standard [ISO, 2011], une exigence est un *"énoncé qui traduit ou exprime un besoin et ses contraintes et conditions associées"*.
- Quant au standard [IEE, 1990], il propose trois définitions :

  (1) Une condition ou capacité exigée par l'utilisateur pour résoudre un problème ou atteindre un objectif.

  (2) Une condition ou capacité qui doit être accueillie ou possédée par un système ou un composant du système pour répondre à un contrat, un standard, une spécification ou d'autres documents imposés formellement.

  (3) Une représentation documentée d'une condition ou d'une capacité comme dans (1) et (2).

Dans notre approche, mis à part les trois définitions fournies par ce standard, nous considérons que la notion d'exigence englobe toutes formes d'information fournies par le client demandeur du système. Celle-ci peut être exprimée sous forme d'une description en langage naturel, un dessin, un tableau ou une formule.

Une exigence peut être fonctionnelle ou non-fonctionnelle.

- Une exigence *fonctionnelle* spécifie une fonction pouvant être accomplie par un système ou un composant du système [IEE, 1990].
- Une exigence *non-fonctionnelle* est composée de contraintes et de qualités. Les qualités désignent les propriétés ou caractéristiques du système. Les contraintes précisent comment le système doit satisfaire les exigences fonctionnelles en termes de sécurité, performance, maintenance, robustesse, etc.

Un *besoin* correspond à l'ensemble des exigences fonctionnelles et non-fonctionnelles que le système doit prendre en compte pour répondre à la demande des utilisateurs.

Dans la suite de ce mémoire, nous ne faisons pas la différence entre les termes *exigence* et *besoin*. Nous utilisons ces deux termes comme la traduction du terme anglais *requirement*. Nous exploitons la définition donnée par [Chemuturi, 2012] : *"une exigence est un besoin, une attente, une contrainte ou une interface de toutes les parties prenantes, qui doit être satisfaite par le système ou logiciel proposé durant son développement."*

2.1.1.2 Cahier des charges

Un *cahier des charges* pour un système ou logiciel est un document rassemblant les besoins du client. Il sert à expliquer les détails du futur système ou logiciel aux différents acteurs. Il permet notamment de cadrer les missions des acteurs impliqués, dont celles du directeur de projet (côté maîtrise d'ouvrage) et/ou du chef de projet (côté maîtrise d'oeuvre) [6].

6. Définition Wikipédia à l'adresse : `https://fr.wikipedia.org/wiki/Cahier_des_charges`

Dans ce mémoire, le cahier des charges désigne l'ensemble des besoins tels qu'ils sont décrits par le client, demandeur du système. Nous ne modifions pas ce document, mais nous le réécrivons sous une nouvelle version appelée *CdC* et que nous faisons évoluer (voir section 3.1.1 du chapitre 3).

2.1.1.3 Ingénierie des exigences

[Zave, 1997] définit l'*ingénierie des exigences IE* comme *"une discipline du génie logiciel qui concerne les buts du monde réel pour les fonctions et les contraintes sur un système logiciel. Elle concerne aussi la relation entre ces facteurs pour préciser les spécifications du comportement logiciel et leur évolution dans le temps et à travers les familles de logiciels"*.
L'ingénierie des exigences couvre plusieurs activités telles que :

- l'élicitation des besoins qui représente le *"processus par lequel l'acquéreur et les fournisseurs d'un système découvrent, examinent, articulent, comprennent et documentent les exigences du système et des processus du cycle de vie"*, selon le standard [ISO, 2011],
- l'analyse du domaine pour comprendre la demande du client,
- l'analyse et la modélisation des besoins définies comme le processus dans lequel les besoins du client sont étudiés,
- la communication entre besoins et
- l'évolution des besoins relativement aux changements de l'environnement, des objectifs ou via des corrections.

2.1.1.4 Hypothèse

Il s'agit des exigences contraignant l'environnement du système en cours de développement. Selon [Jackson, 1997], la notion d'hypothèses ou assertions sur l'environnement fait partie intégrale de l'ingénierie des besoins. Dans notre travail, nous supposons que les hypothèses sur l'environnement du système sont satisfaites. Par exemple, dans le cadre du développement d'un système de contrôle de train d'atterrissage d'un avion, nous supposons que la pression de l'air, à l'extérieur du système, est adéquate pour un bon déroulement du système.

2.1.1.5 Hiérarchie

Dans notre approche, les exigences sont hiérarchisées, en utilisant une relation *parent-enfant* dans laquelle le besoin enfant détaille le besoin parent. Le standard [IEE, 1990] définit cette notion comme *"une structure dans laquelle les composants sont classés en niveaux de subordination : chaque composant a zéro, un ou plusieurs subordonnés et il n'y a pas de composant ayant plus qu'un composant subordonné"*.

2.1.1.6 Partie prenante

C'est *"un individu ou une organisation ayant un droit, une action, une revendication, ou un intérêt, pour un système ou en sa possession des caractéristiques qui répondent à leurs besoins et attentes"* [ISO, 2011]. Une partie prenante est impliquée dans le projet du futur système. Elle agit sur le déroulement des étapes de développement - tel est le cas du chef de projet, des développeurs, programmeurs, spécifieurs, etc. - et peut être affectée par les résultats, comme c'est le cas du client et de l'utilisateur final du système.

2.1.2 Place des besoins dans le développement

Fixer les différents aspects du cahier des charges tout au long du processus de développement est essentiel pour montrer l'importance de sa prise en compte. Le terme *"aspect"* signifie comment nous considérons ce document : est-ce un contrat ? une référence ? un document informel utile uniquement au départ ?
De même, localiser les endroits et les moments de consultation et d'utilisation du cahier des charges nous guide à identifier quelques problèmes liés à ce document et à renforcer son existence dans le développement. Ceci favorise notamment la participation du client dans les différentes étapes du processus.

2.1.2.1 Aspect contractuel

Les besoins sont décrits par le client. L'objectif est de développer un système répondant à cette demande. Dans ce sens, le cahier des charges représente un *contrat* :

- entre le client et le développeur : le développeur doit respecter les termes de ce contrat lors du développement de son système. Ces termes sont représentés par les exigences du cahier des charges et
- entre les développeurs eux-mêmes (spécifieurs, concepteurs, programmeurs, testeurs, chef de projet, etc.) : il s'agit de suivre les besoins du client pour favoriser une communication entre les différents membres de l'équipe. Si ses termes sont clairs, celui-ci permet d'éviter les problèmes de conflits de points de vue.

2.1.2.2 Aspect référentiel

Il s'agit d'un document à suivre et à consulter à tout moment du processus de développement du système. Son importance se manifeste :

- lorsqu'il y a des ambiguïtés : nous revenons sur ce document pour clarifier ces ambiguïtés et vérifier que nous respectons la demande du client et
- au moment de la validation ou test du système en cours de développement : c'est relativement aux besoins du cahier des charges que ces opérations s'effectuent.

2.1.2.3 Aspect explicatif

Un cahier des charges est, avant tout, utilisé pour comprendre la demande du client. Dans ce document, nous trouvons des détails et des explications relatives au futur système. Sa consultation doit être possible à tout moment du développement.

2.2 Spécifications formelles

Dans cette section, nous présentons les concepts relatifs au développement de spécifications formelles. Les structures proposées par la méthode formelle B événementiel permettent la construction graduelle de logiciels et systèmes corrects en utilisant la technique de raffinement. Les activités de vérification et de validation ont une place importante dans le développement. Elles sont accomplies grâce aux outils présents sous la plateforme Rodin. Nous nous intéressons spécifiquement à la méthode B événementiel et montrons un aperçu sur le langage TLA$^+$.

2.2.1 Méthode formelle

Une *spécification formelle* est un formalisme qui utilise les mathématiques et la logique du premier ordre pour définir rigoureusement des systèmes corrects par construction. Cette spécification est aussi appelée *modèle formel*. Un système est dit *correct par construction* s'il remplit les critères suivants :

1. il est construit pas-à-pas,
2. il est vérifié et
3. il est validé.

Le développement de spécifications formelles se réalise en adoptant une *méthode formelle*. Celle-ci est utilisée dans les phases en amont du cycle de développement logiciel. Il s'agit de décrire *"le quoi"* et non pas *"le comment"*. En d'autres termes, nous décrivons ce que le système est censé réaliser. [Abrial, 2010] définit les méthodes formelles par :

"Des techniques utilisées pour construire des plans adaptés à notre discipline. De tels plans sont nommés modèles formels."

Durant ces dernières années, l'utilisation des méthodes formelles telles que B [Abrial, 2005] et B événementiel [Abrial, 2010] a évolué. Elle est observée dans plusieurs domaines critiques tels que :

- *la médecine* : à travers les travaux de [Méry and Singh, 2011b] et [Singh et al., 2015] sur l'élaboration d'un système de pacemaker et les travaux de [Banach, 2016] pour la machine d'hémodialyse [Mashkoor, 2016] ;
- *le transport* : les travaux de [Colin et al., 2008] dans le cadre du projet CRISTAL et [Yang, 2013] ont été réalisés pour le développement d'un système de convoi de véhicules autonomes en B événementiel. Les auteurs de [Su et al., 2011] et [Abrial, 2006b] ont utilisé la même méthode formelle pour développer un système de contrôle de train ;
- *l'avionique* : les travaux de [Su and Abrial, 2017] et [Ladenberger et al., 2017] se sont intéressés au développement et à la validation d'un système de contrôle de train d'atterrissage d'un avion [Boniol and Wiels, 2014] et
- *la gestion de mémoire* : le travail [Su et al., 2015] traite le problème de gestion des allocations et désallocations des zones mémoires.

2.2.2 Activités

Pendant la construction d'un modèle formel, il existe principalement trois activités distinctes : raffinement, vérification et validation. La première aide au développement progressif du modèle, la deuxième aide à garantir sa correction au sens mathématique et la dernière assure sa correction relativement aux besoins du client.

2.2.2.1 Raffinement

Cette notion a été introduite au début des années 70 [Dijkstra, 1976]. C'est la technique de transformation d'un modèle dit *abstrait* en un modèle plus détaillé dit *concret*. Ce dernier doit préserver les propriétés du modèle abstrait ainsi que son comportement. On peut parler du raffinement dans différentes étapes de développement d'un logiciel ou système notamment en amont à travers la spécification et en aval lors de l'implémentation ou le codage. [Wirth, 1971] utilise le raffinement dans l'étape de programmation. Il le considère comme une séquence de décisions de conception qui consiste à

décomposer les tâches du programme en sous-tâches et les données en structures de données. Le raffinement des spécifications formelles permet l'élaboration incrémentale d'une spécification complexe en partant d'un modèle abstrait cohérent et en le concrétisant pas-à-pas. Cette cohérence est assurée via des obligations de preuves de raffinement. L'auteur de [Abrial, 2009] distingue deux types de raffinement :

- *Raffinement horizontal.* Appelé aussi *raffinement par superposition*, il consiste à commencer par un modèle formel abstrait et à le rendre de plus en plus complexe en introduisant des détails issus du cahier des charges. Une fois que tous les détails ont été pris en compte, le raffinement horizontal prend fin. En adoptant ce type de raffinement, le développeur ou spécifieur ne s'intéresse pas à l'implémentabilité du futur système ou logiciel car l'objectif est de décrire pas-à-pas le "quoi".
- *Raffinement vertical.* Il est nommé *raffinement de données*. Il prend lieu quand tous les pas du raffinement horizontal sont terminés. Il s'agit d'aller graduellement vers l'implémentation en décrivant le "comment". Pour y parvenir, on n'ajoute pas de nouveaux détails du cahier des charges mais on transforme les états existants dans le modèle formel en nouveaux états implémentables en donnant plus de précision. Par exemple, on transforme une variable de type ensemble en une variable de type tableau ou liste chaînée. Quand on adopte un raffinement vertical, un concept important émerge : l'*invariant de collage*. Il s'agit d'un prédicat liant les deux espaces d'états concret et abstrait. Le paragraphe *Machine* de la sous-section 2.2.3.2 donne plus de détails sur cet invariant.

2.2.2.2 Vérification

Cette activité a pour objectif de répondre à la question *"est-on en train de développer le logiciel ou système correctement ? "*. Elle concerne la cohérence mathématique du modèle en cours de construction. Après avoir formalisé les propriétés du système, il est fondamental de prouver leur correction via des preuves au sens mathématique : le modèle est correct, sans erreur et bien conçu ; ses propriétés sont explicitées. La preuve est pratiquée tout au long du développement [Abrial, 2003]. Les outils de preuve sont décrits et détaillés ultérieurement dans cette section (sous-section 2.2.3.3). La preuve d'un modèle formel peut être automatique ou interactive c'est-à-dire avec une intervention du développeur.

2.2.2.3 Validation

Elle vise d'apporter une réponse à la question *"est-on en train de développer le logiciel ou système correct ?"*. En d'autres termes, on veut s'assurer que le logiciel ou système développé répond aux besoins du client. Cette activité, utilisée tout au long du processus formel de développement, permet de détecter un ensemble de problèmes comme l'inter-blocage ou les comportements non autorisés. Il existe diverses techniques de validation telles que :

- *l'animation* qui permet d'explorer les différents états que peut prendre un modèle,
- *la traduction de modèles* qui consiste à transformer les modèles en programmes exécutables,
- *la simulation* à travers laquelle on peut exécuter un modèle et
- *le model-checking* dont l'idée a émergé en 1980 [Emerson and Clarke, 1980]. Elle a pour objectif de vérifier si un modèle donné satisfait une propriété. A défaut, le model-checker fournit un contre-exemple.

2.2.3 B événementiel

B événementiel est une méthode formelle qui permet la construction graduelle des spécifications formelles correctes moyennant une activité de preuve [Abrial et al., 2010] [Abrial, 2007]. Cette méthode est une extension du langage B classique [Abrial, 2005]. Elle est utilisée pour l'élaboration des modèles formels pour les systèmes réactifs, les algorithmes séquentiels et distribués. Elle permet également le développement des systèmes dits *hybrides* regroupant des parties logicielles, matérielles et des interfaces utilisateurs.

2.2.3.1 Langage

La méthode B événementiel utilise un langage basé sur la théorie des ensembles et la logique des prédicats. Ce langage gère [7] :

- des prédicats logiques : un prédicat est un énoncé contenant une ou plusieurs variables et ayant une valeur *vrai* ($\top$) ou *faux* ($\bot$). Pour deux prédicats P et Q, nous citons :
  - la conjonction $P \wedge Q$ et la disjonction $P \vee Q$,
  - la négation $\neg P$,
  - l'implication $P \Rightarrow Q$ et l'équivalence $P \Leftrightarrow Q$ et
  - les quantificateurs universel $\forall$ et existentiel $\exists$,
- des notations ensemblistes qui concernent les opérations sur les ensembles comme l'intersection $\cap$ et l'union $\cup$,
- des relations mathématiques comme les relations totales notées par $\leftrightarrow\!\!\!\!\!\rightarrow$ et des fonctions mathématiques comme les fonctions partielles $\nrightarrow$ et les bijections $\rightarrowtail$ et
- des opérateurs sur les relations mathématiques tels que le domaine *dom*, le co-domaine *ran*, la restriction sur le domaine $\lhd$ et la soustraction du domaine $\hookleftarrow$.

2.2.3.2 Constituants

Un modèle en B événementiel est caractérisé par deux constituants importants : le contexte dénoté par le mot clé *CONTEXT* et la machine dénotée par le mot clé *MACHINE*.

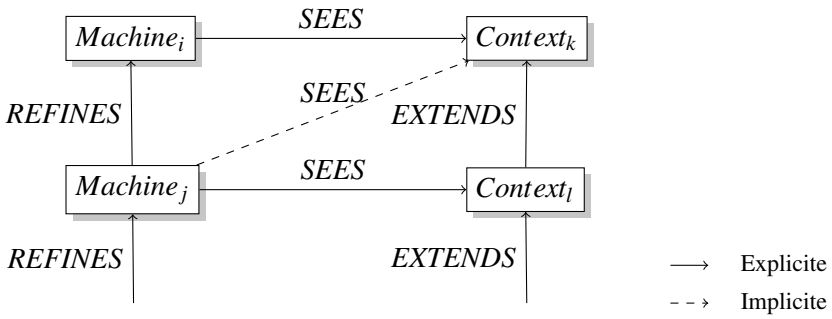


FIGURE 2.1 – Relations entre machines et contextes

7. La liste complète des éléments du langage de B événementiel est définie dans `https://lfm.iti.kit.edu/download/EventB_Summary.pdf`

Une machine peut raffiner, à travers la relation *REFINES*, une seule autre machine. Elle peut utiliser, via la relation *SEES*, explicitement ou implicitement un ou plusieurs contextes. Un contexte peut étendre, par le mot clé *EXTENDS*, un ou plusieurs contextes. La structure générale montrant ces différentes relations est décrite dans la figure 2.1.

2.2.3.2.1 Contexte. Il décrit la partie statique du modèle en cours de développement. Le terme *"statique"* rassemble les ensembles, constantes, axiomes et éventuellement des théorèmes. Un contexte est constitué de plusieurs clauses. Chaque clause est introduite par un mot clé. La figure 2.2 décrit chacune de ces clauses. Il est à noter que les champs avec "*" peuvent être vides.

CONTEXT *Identifiant_Contexte*
EXTENDS *
Liste_Identifiants_Contextes_Abstraits
SETS *
Liste_identifiants_sets
CONSTANTS *
Liste_identifiants_constantes
AXIOMS *
<label> : <predicat>
...
THEOREMS *
<label> : <predicat>
...
END

FIGURE 2.2 – Structure d'un contexte

- *CONTEXT :* permet d'attribuer un identifiant *Identifiant_Contexte* unique à ce contexte.
- *EXTENDS :* contient la liste des contextes abstraits hérités par ce contexte.
- *SETS :* définit la liste des ensembles porteurs qui vont servir pour le typage des constantes.
- *CONSTANTS :* contient la liste des constantes du modèle.
- *AXIOMS :* définit les axiomes du contexte. Un axiome est un prédicat décrivant une propriété sur les constantes. Il est identifié par un *label* pour le distinguer des autres axiomes.
- *THEOREMS :* rassemble les théorèmes du contexte. Un théorème doit être prouvé en utilisant les axiomes. De même, chaque théorème est identifié par un *label*.

Exemple.

Prenons l'application de distributeur automatique de billets (DAB). L'objectif est de développer un système de contrôle du DAB pour assurer son déroulement en évitant le risque de perte de carte bancaire. Une description brève de ce système est la suivante :

DAB 1. Le client, identifié par sa carte bancaire et le code associé, retire de l'argent de son compte géré par la banque, à partir d'un distributeur. Le client peut retirer un certain montant chaque semaine.
NB. Nous supposons que la carte est valide ainsi que son code.
Le distributeur a assez d'argent dans sa caisse.

La figure 2.3 montre un contexte nommé *DAB1_Ctx*. Il intègre des ensembles porteurs *CARTES*, *DECIDE* et *MSG* modélisant respectivement l'ensemble de toutes les cartes valides pour la banque concernée, les décisions de la banque et le message de retour au client. Les termes qui suivent les "//" représentent un commentaire sur la constante *pin*. La relation notée par le mot clé *partition* et décrite dans *axm*1 signifie que :

- les constantes $c1$, $c2$ $c3$ et *pas_de_carte* appartiennent à l'ensemble $CARTES$ et
- ces constantes sont distinctes.

CONTEXT *DAB1_Ctx*
SETS
CARTES, DECIDE, MSG
CONSTANTS
c1, c2, c3, pas_de_carte
OK, NotOK
solde_insuffisant, montant_depasse_plafond, code_invalide
pin // ensemble des codes pin affectés à toutes les cartes
AXIOMS
axm1 : partition(CARTES, {c1}, {c2}, {c3}, {pas_de_carte})
axm2 : partition(DECIDE, {OK}, {NotOK})
axm3 : partition(MSG, {solde_insuffisant }, {montant_depasse_plafond}, {code_invalide })
axm4 : pin $\in$ *CARTES* $\rightarrowtail\!\!\!\!\!\rightarrow \mathbb{N}$
END

FIGURE 2.3 – Contexte pour le DAB

2.2.3.2.2 Machine. Elle définit la partie dynamique du modèle formel. Elle est constituée de variables, invariants, théorèmes et événements. Sa structure est décrite dans la figure 2.4 dans laquelle les champs avec "*" peuvent être vides.

- *MACHINE :* définit un identifiant unique *Identifiant_Machine* de cette machine.
- *REFINES :* précise la machine abstraite raffinée par cette machine.
- *SEES :* contient la liste des contextes "vus" ou utilisés explicitement par cette machine. Cette relation permet à la machine d'utiliser les constantes et les ensembles porteurs définis dans la liste des contextes vus explicitement ou implicitement (via la relation *extends*).
- *VARIABLES :* précise la liste des variables introduites dans cette machine.
- *INVARIANTS :* l'état d'une machine est décrit par la valeur de ses variables. Sa sémantique est précisée via des invariants. Ceux-ci décrivent des propriétés données sous forme de contraintes sur les variables. Il existe plusieurs sortes d'invariants :
  - *Invariant de typage*. Toute variable déclarée dans la machine doit avoir un type. Ce dernier peut être soit prédéfini, comme les types $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{R}$ et *BOOL*, soit un type défini par le développeur dans la clause SETS des contextes vus par cette machine.
  - *Invariant sur les propriétés des variables*. Il décrit les contraintes que les variables doivent respecter en permanence comme les propriétés de vivacité, de terminaison et d'accessibilité.

- *Invariant de collage*. Dans le cas de raffinement, ce type d'invariant définit la relation entre l'espace d'états de la machine raffinée avec celui de la machine abstraite. Il met en relation les variables abstraites et les variables concrètes. La correction de l'invariant de collage est primordiale pour assurer la correction du raffinement.

MACHINE *Identifiant_Machine*
REFINES *
Identifiant_Machine_Abstraite
SEES *
Liste_Identifiants_Contextes
VARIABLES
Liste_identifiants_variables
INVARIANTS
<label> : <predicat>
...
THEOREMS *
<label> : <predicat>
...
VARIANT *
<variant>
EVENTS
INITIALISATION
Liste_autres_evenements
END

FIGURE 2.4 – Structure d'une machine

- *THEOREMS :* décrit la liste des théorèmes à prouver dans cette machine. Leur preuve se réalise en utilisant :
  - — les théorèmes des machines abstraites et
  - — les axiomes et les théorèmes des contextes vus explicitement ou implicitement par cette machine.

  Par exemple, un des théorèmes les plus courants est celui de l'absence de blocage dans la machine [Yang and Jacquot, 2011a].
- *VARIANT :* un variant apparaît dans une machine raffinée contenant des événements convergents (voir la description d'un événement convergent dans le paragraphe *Événement* suivant). Il est décrit sous forme d'une expression de type entier naturel qui est diminué par les événements convergents ou de type ensemble ayant une cardinalité décroissante.
- *EVENTS :* définit la liste des événements de la machine. Il existe un événement particulier appelé *INITIALISATION* présent dans toutes les machines. Celui-ci est destiné à initialiser l'espace d'états de cette machine en donnant une valeur à chacune de ses variables. Cet événement n'a pas de gardes.

Exemple.

Revenons à l'application du DAB. La figure 2.5 montre un aperçu de la machine abstraite nommée *DAB1_Mch* associée à cette application. Elle utilise le contexte *DAB1_Ctx* de la figure 2.3 et donc ses ensembles et ses constantes. Son état est décrit par cinq variables. Les cinq premiers invariants de cette machine sont consacrés au typage de ses variables. Le dernier invariant nommé *nbEssaiPin_inv* décrit une propriété sur le nombre d'essais de saisie de code pin qui ne doit pas dépasser 3 fois.

MACHINE *DAB1_Mch*
SEES
DAB_Ctx1
VARIABLES
carte // carte actuelle présente dans le lecteur du DAB
code // code pin saisi dans le DAB par le client
decision // décision de la banque
message // message affiché au client
nb_essai_pin // nombre d'essais pour la saisie du code pin par le client
INVARIANTS
carte_Typ : carte $\in$ *CARTES*
code_Typ : code $\in \mathbb{N}$
decision_Typ : decision $\in$ *DECIDE*
message_Typ : message $\in$ *MSG*
nbEssaiPin_Typ : nb_essai_pin $\in \mathbb{N}$
nbEssaiPin_inv : nb_essai_pin $\leq$ *3*
EVENTS
INITIALISATION
insere_carte // le client insère sa carte
entre_pin // le client saisit son code pin
choisit_montant // le client choisit le montant souhaité
ejecte_carte // le DAB éjecte la carte de son lecteur
...
END

FIGURE 2.5 – Machine abstraite du DAB

2.2.3.2.3 Événement. Un événement est l'élément dynamique permettant de naviguer entre les différents états d'une machine. En d'autres termes, il permet d'aller d'un état à un autre en changeant les valeurs des variables. Il est composé de deux parties : les *gardes* et les *actions* qui modélisent une relation "*avant-après*". Celle-ci définit une relation entre les états des variables avant et après l'exécution de cet événement. La représentation formelle d'un événement est décrite par la figure 2.6. Dans cette figure, les champs avec "*" peuvent être vides.

- *identifiant_evenement :* désigne le nom de l'événement.
- *STATUS :* définit le statut d'un événement, soit :
  - *ordinary*,

— *convergent* c'est-à-dire qu'il diminue la valeur numérique du variant si ce dernier est un entier naturel ou il diminue le nombre d'éléments de ce variant si c'est un ensemble ou

— *anticipated* dans le sens que cet événement sera convergent dans un raffinement ultérieur.

identifiant_evenement
STATUS
{*ordinary, convergent, anticipated*}
REFINES *
Identifiant_Evenement_Abstrait
ANY *
$identif\_parametre_1$
...
$identif\_parametre_n$
WHERE *
<*label*> : <*predicat*>
...
WITH *
<*label*> : <*witness*>
...
THEN *
<*label*> : <*action*>
...
END

FIGURE 2.6 – Structure d'un événement

- *REFINES :* désigne l'événement abstrait raffiné par cet événement.
- *ANY :* donne la liste des variables locales, appelées *paramètres*, accessibles et utilisées uniquement dans cet événement.
- *WHERE :* décrit les gardes de cet événement. Une garde désigne un prédicat qui doit être évalué à *vrai* ($\top$) afin de pouvoir déclencher un événement. Elle inclut des conditions sur les variables de la machine et/ou sur les paramètres de l'événement. Le mot clé WHERE est remplacé par le mot *WHEN* dans le cas où la clause ANY est vide.
- *WITH :* l'absence d'un paramètre de l'événement abstrait dans la définition concrète de cet événement est exprimée par l'introduction d'un *"témoin"* appelé *witness*.
- *THEN :* définit les actions réalisées par cet événement. Pour une variable x, un ensemble S, une expression E et un prédicat P, une action est :
  — *déterministe :* de la forme $x := E$. Elle affecte à la variable x une valeur suivant l'expression E. Le symbole $:=$ correspond à *"devient égal à"* ou
  — *non déterministe :* elle a deux formes possibles :
    – $x :\in S$ pour dire qu'on associe une valeur quelconque de S à x. Le symbole $:\in$ signifie *"devient un élément de"* ou
    – $x :| P$ avec laquelle on effectue un choix arbitraire de valeur pour x. Ce choix doit satisfaire le prédicat P. Le signe $:|$ signifie *"sachant que"*.

Un événement peut avoir une partie vide des actions. Il s'agit de l'action *SKIP* qui ne fait rien.

Exemple.

L'événement *entre_pin* de la figure 2.7 modélise le traitement effectué par le DAB suite à la saisie d'un code pin dans le DAB après insertion de la carte bancaire. Il est caractérisé par deux paramètres *c* et *crt* modélisant respectivement le code saisi par le client et la carte insérée. Ces paramètres doivent obéir à la contrainte décrite par les gardes. Une fois ces gardes évaluées vraies, la partie *THEN* est examinée.

entre_pin
STATUS
ordinary
ANY
c // code pin saisi par le client
crt // carte insérée par le client
WHERE
grd1 : $c \in \mathbb{N}$
grd2 : $crt \in CARTES$
grd3 : $nb\_essai\_pin < 3$
grd4 : $c \neq pin(crt)$ // le code saisi ne correspond pas au pin de la carte
THEN
act1 : $code := c$
act2 : $nb\_essai\_pin := nb\_essai\_pin + 1$
END

FIGURE 2.7 – Un événement du DAB

2.2.3.3 Outils

L'élaboration des modèles B événementiel pour des logiciels et systèmes complexes se réalise sous Rodin (Rigorous Open Development Environment for Complex Systems) [Abrial et al., 2006]. C'est une plateforme ou environnement à accès libre qui offre un support pour le raffinement et pour la preuve mathématique. Le développement de cet environnement, basé sur Eclipse [8], a été partiellement financé par la Commission Européenne à travers trois projets : projet RODIN [9] entre 2004 et 2007, projet DEPLOY [10] entre 2008 et 2012 et projet ADVANCE [11] de 2011 à 2014. Il est pris en charge par l'Agence Nationale Française de Recherche dans le cadre du projet IMPEX [12] à partir de décembre 2013. Durant 10 ans, entre 2004 et 2014, cette plateforme a évolué [Voisin and Abrial, 2014]. Son évolution est passée par deux étapes principales :

- l'utilisation de cet outil par les membres développeurs eux-mêmes avant de le mettre en disposition des utilisateurs,

8. http://www.eclipse.org/
9. http://rodin.cs.ncl.ac.uk/
10. http://www.deploy-project.eu/index.html
11. http://www.advance-ict.eu/
12. http://www.agence-nationale-recherche.fr/Project-ANR-13-INSE-0001

- l'autorisation, par l'aspect "Open Source", aux utilisateurs et aux autres développeurs d'améliorer l'environnement Rodin via des retours sous forme de suggestions, ajouts, suppressions et de modifications du code interne.

Rodin intègre une multitude de composants logiciels dont :

- un éditeur de spécifications formelles,
- des analyseurs lexico-syntaxique et de typage,
- un outil ProR de gestion des exigences depuis lesquelles sont issus les modèles formels. Cet outil peut également être indépendant de Rodin,
- des générateurs d'obligations de preuve,
- des prouveurs automatiques et interactifs,
- des outils de validation comme ProB,
- un plugin Decomposition [13] pour la composition et la décomposition des spécifications et
- d'autres outils pour la visualisation des modèles.

2.2.3.3.1 Vérification. Chaque composant, machine ou contexte, en cours de développement doit être mathématiquement correct. Ceci se réalise par le biais des *obligations de preuve* notées *OPs*. Une OP est un prédicat qui définit une propriété qui doit être prouvée pour le modèle B événementiel. Les OPs sont générées automatiquement par les *générateurs d'obligations de preuve* sous forme de *séquents*. Pour un but G et un ensemble d'hypothèses H, un séquent de la forme $H \vdash G$ signifie que G est à démontrer à partir de H. Les OPs sont par la suite démontrées par les prouveurs tels que les solveurs SMT [Déharbe et al., 2012] et SAT [Plagge and Leuschel, 2012] [Eén and Sörensson, 2003]. Ces prouveurs sont composés d'un ensemble de "raisonneurs" écrits en Java. Chaque raisonneur prend en entrée un séquent, il le vérifie, et s'il réussit, il fournit une *règle de preuve*. Cette dernière correspond à un *arbre des preuves* généré par un outil appelé *calculateur des séquents*. Un arbre des preuves est associé à chaque OP montrant le chemin utilisé pour la prouver.

[Abrial, 2003] distingue quatre cas liés au résultat de preuve d'un énoncé :

Cas 1. L'énoncé est vrai, ses OPs associées sont dites *déchargées*.

Cas 2. L'énoncé est faux, une révision du modèle est nécessaire.

Cas 3. La preuve ne réussit pas mais l'énoncé est néanmoins *probablement prouvable ou réfutable*. Dans ce cas, le modèle doit être revu afin de le rendre prouvable.

Cas 4. La preuve ne réussit pas et l'énoncé n'est *probablement ni prouvable ni réfutable*. Dans ce cas, le modèle est trop pauvre et doit être revu.

Dans les trois derniers cas, une ou plusieurs étapes supplémentaires sont nécessaires pour assurer la correction du modèle. Ces étapes consistent soit à une intervention du spécifieur dans le cadre de la preuve *interactive*, soit à relancer des prouveurs dans une nouvelle tentative de preuve. Une difficulté importante concerne la distinction entre les cas 3 et 4. En effet, même avec l'existence des outils de preuve assez puissants, la localisation de la source du non-déchargement des OPs reste une tâche difficile à accomplir et nécessite des efforts importants de la part du spécifieur.

L'activité de preuve en B événementiel [Abrial et al., 2010] se focalise sur trois propriétés fondamentales :

13. `http://wiki.event-b.org/index.php/Decomposition_Plug-in_User_Guide`

- la bonne définition des expressions,
- la préservation de l'invariant et
- la correction du raffinement.

A partir de ces trois propriétés émergent plusieurs catégories d'OPs. La table 2.1 présente un aperçu général de certaines d'entre elles.

Nom OP	***Signification***
INV	l'invariant est préservé par les événements de la machine
GRD	les gardes des événements abstraits sont renforcés dans les événements raffinés
THM	chaque théorème est prouvable
FIS	une action non déterministe est faisable
WD	les axiomes, invariants, théorèmes, invariants, gardes, actions, variants et witness sont bien définis

TABLE 2.1 – Quelques obligations de preuve en B événementiel

2.2.3.3.2 Validation. Dans ce paragraphe, nous donnons une description des outils de validation disponibles sous Rodin. Il existe plusieurs techniques :

- *Animation.* Les outils comme ProB [Leuschel and Butler, 2008] [Bendisposto et al., 2008], AnimB [14] et Brama [Servat, 2007] assurent cette tâche. Les auteurs de [Hallerstede et al., 2011] proposent un algorithme en ProB pour l'animation du raffinement de plusieurs niveaux simultanés. Cette animation permet de détecter une variété d'erreurs introduites avec le raffinement. Un avantage de l'utilisation de ces outils est leur aspect graphique. Cependant, leurs limites majeures concernent :
  - l'explosion combinatoire de l'espace d'états : on ne peut pas parcourir tous les états du système pour l'animer et
  - le non-déterminisme : on ne peut pas fixer des valeurs pour démarrer l'animation si les variables sont abstraites.
- *Traduction de modèles.* Les outils B2ALL [Méry and Singh, 2011a] et B2C [Wright, 2009] génèrent des programmes écrits en plusieurs langages tels que C, C++ et Java à partir des modèles écrits en B ou B événementiel. Ces programmes sont précis et prêts à être compilés et exécutés. La limite de cette méthode est le fait que cette transformation nécessite un modèle concret c'est-à-dire ne contenant pas du non-déterminisme. Lorsqu'il s'agit d'un modèle abstrait contenant du non-déterminisme, cette technique n'est pas utile.
- *Simulation.* Cette technique consiste à déclencher une séquence d'événements et à contrôler le comportement du modèle et les changements de son état. Les outils ProB et JeB [Yang et al., 2014] [Yang, 2013] assurent cette activité. Ils fournissent également des options d'automatisation de la simulation permettant d'explorer un nombre important de chemins reliant les différents états du modèle. JeB permet d'éviter l'utilisation de l'animation pour laquelle l'espace d'états d'un modèle B événementiel concerné peut être exhaustif.

14. `http://www.animb.org/index.xml`

- *Model-checking*. ProB [Leuschel and Butler, 2003], à travers son contre-prouveur, se base sur l'idée suivante : l'outil parcourt les états possibles pour un modèle donné afin de vérifier leur cohérence par rapport à une propriété précise. S'il trouve un contre-exemple, il arrête le processus de parcours. Une spécificité du contre-prouveur de ProB est son aide à l'activité de preuve interactive via un accès direct aux OPs [Krings et al., 2015].

Les auteurs de [Mashkoor and Jacquot, 2016] et [Mashkoor et al., 2016] ont décrit d'autres idées d'aide à la validation de modèles B événementiel. Ils ont mis en évidence la possibilité de définir des sémantiques mathématiques qui garantissent la correction de modèles exécutables. Ils ont esquissé une extension de la notion de raffinement en tant qu'étape dans le processus de développement.

2.2.3.3.3 Visualisation. Avoir une vue d'ensemble des modèles développés est important pour mieux les gérer. La plateforme Rodin fournit un aspect visuel de ces modèles grâce à :

- un plugin Project Diagram [15] permettant de décrire une vue d'ensemble de l'organisation des machines et des contextes. Il s'agit d'un diagramme similaire à celui décrit dans la figure 2.1 et
- un outil Event-B Statemachines [16] permettant de décrire une spécification sous forme d'une machine d'états-transitions. Cet outil offre une nouvelle vision des machines B événementiel en cours de construction et de voir les changements de ses états suite au déclenchement d'un ou de plusieurs événements.

2.2.4 Aperçu sur TLA$^+$

Notre travail a essentiellement porté sur l'utilisation du langage B événementiel. Notre approche est applicable pour les langages de spécification formelle ; nous avons en particulier utilisé TLA$^+$ dont une vue d'ensemble est présentée dans ce paragraphe. TLA$^+$ [Lamport, 2002] (Logique Temporelle des Actions) est une variante du langage formel LTL (Logique Temporelle Linéaire) [Pnueli, 1977]. Une spécification en TLA$^+$ est partitionnée en *modules*. Il existe des modules prédéfinis comme *Naturals* représentant l'ensemble des entiers naturels. Une description dans ce langage permet de spécifier le comportement d'un système. Son utilisation est pratique pour décrire les propriétés en une seule formule. Bien que la quasi-totalité de nos travaux sont réalisés en B événementiel, nous choisissons également TLA$^+$ pour montrer que notre approche est applicable à d'autres langages. Le choix de ce langage est justifié par sa capacité d'exprimer explicitement des propriétés d'enchaînement d'opérations ou d'actions via des *théorèmes*. Il s'agit d'une formule temporelle qui doit être satisfaite par chaque comportement ou état du système. Ce langage est doté de l'outil *TLA Toolbox* [17], un environnement de développement intégré (IDE). Cet environnement intègre les outils suivants :

- un analyseur syntaxique,
- un model-checker TLC [18],
- un traducteur de modèles appelé PlusCal [19],
- un outil TLATeX pour édition des modèles et
- un système de preuve TLAPS [20]

15. `http://wiki.event-b.org/index.php/Project_Diagram`
16. `http://wiki.event-b.org/index.php/Event-B_Statemachines`
17. `https://lamport.azurewebsites.net/tla/toolbox.html`
18. `https://lamport.azurewebsites.net/tla/tlc.html`
19. `https://lamport.azurewebsites.net/tla/pluscal.html`
20. `http://tla.msr-inria.inria.fr/tlaps/content/Home.html`

2.3 Besoins et spécifications formelles

L'écart entre le monde des besoins et le monde des spécifications formelles correspondantes est un problème d'actualité. Celui-ci devient de plus en plus important en avançant dans les étapes de développement mettant en évidence la présence de failles et souvent l'échec des projets. Après avoir donné quelques définitions des termes - utilisés en ingénierie des exigences et - exploités tout au long de ce mémoire, cette section décrit comment et où s'exprime cet écart, les approches remédiant à ce problème ainsi que les impacts des liens établis entre ces deux mondes sur l'avancement de la construction d'un système ou logiciel.

2.3.1 Écart entre besoins et spécification formelle

2.3.1.1 Pourquoi établir un pont entre les besoins et la spécification formelle ?

Une réponse intuitive à cette question est le fait qu'il existe un écart entre ces deux mondes. Ceci nous incite à poser les questions suivantes :

- en quoi consiste cet écart ?
- comment se matérialise-t-il ?
- quels problèmes engendre-t-il ?
- comment le réduire ?

Afin d'apporter des réponses à ces questions, nous nous basons sur des études menées par le groupe Standish [21] afin d'identifier les champs d'échecs des projets, les facteurs majeurs contribuant à ces échecs ainsi que des alternatives pour les réduire. Dans un rapport *CHAOS* paru en 1995 [22], les auteurs classifient les projets en trois catégories :

- *Projet réussi.* Le projet est terminé à temps et respecte le budget, en prenant en compte toutes les fonctionnalités spécifiées.
- *Projet contesté.* Le projet est terminé et opérationnel mais dépasse le budget prévu, excède les délais et offre moins de caractéristiques et fonctionnalités que celles spécifiées au départ.
- *Projet échoué.* Le projet est annulé pendant le cycle de développement.

Les auteurs de ce rapport évoquent, selon un sondage réalisé sur plusieurs projets de différents domaines, les facteurs de succès et d'échec selon la catégorie du projet. Quelques exemples sont donnés dans la table 2.2. Le pourcentage associé à un facteur décrit l'importance de celui-ci relativement aux autres. Par exemple, pour un projet réussi, *"l'implication de l'utilisateur"* occupe 15.9% dans l'ensemble des autres facteurs de réussite du projet, la *"clarté de l'énoncé des exigences"* en occupe 13%.

Selon cette table, l'implication de l'utilisateur ou client, l'incomplétude des besoins et leurs changements font partie des facteurs importants à l'origine des échecs des projets.

21. https://www.standishgroup.com/
22. voir la version réimprimée et publiée en 2014 par le *Project Smart* : *The Standish Group Report* https://www.projectsmart.co.uk/
23. Données issues du résumé du rapport CHAOS, Projet Smart https://www.projectsmart.co.uk/white-papers/chaos-report.pdf

Projet réussi	*Projet contesté*	*Projet échoué*
implication de l'utilisateur (15.9%)	manque d'implication de l'utilisateur (12.8%)	incomplétude des besoins (13.1%)
clarté de l'énoncé des exigences (13%)	incomplétude des besoins et des spécifications (12.3%)	manque d'implication de l'utilisateur (12.4%)
–	changements des besoins et des spécifications (11.8%)	changements des besoins et des spécifications (8.7%)

TABLE 2.2 – Facteurs de réussite et d'échec des projets selon le groupe Standish [23]

2.3.1.2 Problématique

Le problème de l'ingénierie des exigences n'est pas récent. Il a été abordé dès les années 70 par [Bell and Thayer, 1976]. Il concerne essentiellement l'analyse du domaine, la définition des besoins, leur élicitation, leur écriture, leur documentation et leur évolution dans un processus de développement. Des études basées sur des statistiques concernant les échecs des projets réalisées par l'European Software Institute (ESI) [24] en 1996 montrent que la pauvreté des cahiers des charges, l'incomplétude des besoins, leur ambiguité, l'imprécision des objectifs et le manque d'implication du client voire son exclusion dans le processus de développement sont à l'origine de ces échecs. Des études publiées dans des rapports CHAOS du groupe Standish montrent qu'une mauvaise qualité des besoins a un impact négatif sur la qualité du logiciel résultant. Plus que la moitié des projets coûtent deux fois plus cher que le budget initial estimé ; d'autres projets ont été abandonnés avant d'être achevés. Les erreurs les plus courantes se produisent lors des phases en amont du projet. Une correction tardive de ces erreurs coûte deux cents fois plus cher qu'une correction effectuée dans la phase d'analyse des besoins [Boehm, 1981]. Ces échecs deviennent cruciaux dans le cas des systèmes critiques. Selon la définition de [Knight, 2002], un système critique est "*un système dont l'échec peut entraîner une perte de vie, des dégâts matériels importants ou des conséquences graves pour l'environnement. Il existe de nombreux exemples bien connus dans les domaines d'application tels que les dispositifs médicaux, le contrôle d'un avion, les armes et les systèmes nucléaires*". Dans ce cas, il s'agit d'un échec critique qui, selon [Neumann, 1995], est *"un échec pouvant mener à des conséquences sérieuses comme les dangers pour les individus ou violer les exigences critiques du système"*.

Jean-Raymond Abrial a parlé de la catastrophe de Vasa lors d'une présentation en septembre 2006 intitulée "*Have we learned from the Vasa Disaster ?*" [25]. Il s'agit d'un navire qui avait coulé en 1628 après une longue période de travail de construction. Comme bilan de ce naufrage, 53 personnes ont perdu la vie et le navire a été totalement détruit. Abrial a evoqué les erreurs à l'origine de ce désastre et qui ont été présentes dans toutes les étapes de construction du bateau. Ces erreurs concernent principalement : les changements fréquents des besoins du cahier des charges, le manque de spécification du modèle de navire, l'absence d'une architecture explicite du modèle et le non-suivi des retours des tests effectués. Le bateau construit était mal conçu. La conclusion retenue de cette catastrophe s'adresse aux développeurs et aux parties prenantes du cahier des charges. L'orateur de cette présentation insiste sur l'importance de comprendre les exigences, prendre en compte leurs changements et de mettre en place des échanges entre les développeurs au sein du projet. Dans ce contexte, l'absence de

24. renommée actuellement Tecnalia `https://www.tecnalia.com`
25. `https://niif.videotorium.hu/en/recordings/1674/have-we-learned-from-the-wasa-disaster`

coopération entre les différentes parties prenantes du cahier des charges représente un aspect potentiel montrant l'écart entre ses besoins et les autres documents. En effet, il manque un lien entre ce que veut dire le client, ce qu'a compris le spécifieur et ce qu'a compris le programmeur. Ceci est principalement dû à l'absence d'un langage qui unifie la lecture des différents documents : ce qui est écrit, modélisé ou spécifié par une équipe reste dans le domaine de cette équipe. Par exemple, pour comprendre une spécification formelle il faut avoir des compétences en mathématiques et en logique et pour comprendre un programme en C, il faut avoir les bases en C. Le résultat de cette absence de coopération est un développement qui diverge dans plusieurs directions ; le projet n'avance pas et risque d'échouer.

Un autre point montrant l'écart entre le cahier des charges et la spécification formelle est l'absence de la trace des besoins dans le développement. Revenons sur quelques définitions de la notion de *traçabilité*. Tout d'abord, elle permet d'établir une relation entre deux ou plusieurs produits du processus de développement. Par exemple, on peut établir une relation entre une exigence donnée et l'élément conceptuel qui implémente cette exigence [IEE, 1990]. Plusieurs standards définissent la traçabilité :

- Pour le standard [ISO, 2010], il s'agit de :
  (1) *L'identification et la documentation du chemin de dérivation (vers le haut) et du chemin d'allocation / de descente (vers le bas) des exigences dans la hiérarchie des exigences.*
  (2) *L'association discernable entre une exigence et les exigences connexes, les implémentations et les vérifications.*
- Quant au standard IEEE Std 610.12-1990 [IEE, 1990], il la définit dans le cas général comme : *"la mesure dans laquelle une relation peut être établie entre deux ou plusieurs produits du processus de développement, en particulier les produits ayant entre eux une relation prédécesseur-successeur ou maître-subordonné : par exemple, le degré auquel correspondent les exigences et la conception d'un élément du système donné."*
- Le standard ANSI/IEEE Standard 830-1984 [IEE, 1984] opte pour la définition suivante : *"une spécification des exigences logicielles est traçable si (i) l'origine de chacune de ses exigences est claire et si (ii) elle facilite le référencement de chaque exigence dans la documentation du développement ou d'amélioration future."*

Selon ces trois définitions, nous déduisons que :

(1) tracer les besoins signifie suivre leurs changements tout au long du développement et
(2) tracer les besoins dans d'autres documents signifie trouver leur trace dans ces documents issus des étapes de développement. Par exemple, la présence de cette trace dans les modèles formels permet de justifier la présence des éléments constituant ces modèles.

La sauvegarde de la trace des besoins sert à montrer les évolutions du cahier des charges et facilite le retour vers les exigences pour les mettre à jour. Elle donne aux parties prenantes du projet un accès permanent à ses besoins à tout moment du développement. Cette notion n'est toujours pas assurée. Autrement dit, une fois ces besoins compris, ils sont mis à l'écart des autres documents issus des différentes étapes de développement. Dans le papier [Gotel and Finkelstein, 1994], les auteurs évoquent le problème d'absence de la trace des besoins dans le développement et le manque des outils favorisant sa gestion. L'importance de cette trace est de définir des liens entre l'informel, le formel et leurs propriétés. Dans ce cadre, les auteurs de [Jones et al., 1998] assument que *"les langages formels sont utilisés pour spécifier précisément et sans ambiguïté ce dont on a besoin, tandis que les traces aident à répondre aux questions sur la nécessité de tels composants et sur leur lien avec les exigences exprimées de manière informelle"*. En son absence, le développement sera fastidieux et difficilement

adaptable aux changements.

Un dernier point depuis lequel se manifeste l'écart entre le formel et l'informel est l'exclusion du client. Tous les membres du projet doivent participer aux étapes de construction du futur logiciel ou système. Parmi ces membres, le client joue un rôle principal dans l'avancement du développement. Sa participation consiste essentiellement à clarifier les points d'ambiguïté dans sa demande et à valider le logiciel final c'est-à-dire en indiquant s'il correspond à ce qu'il veut obtenir. D'après la table 2.2, l'absence de communication et des échanges avec le client représente un facteur important causant l'échec des projets ou leur contestation.

2.3.2 Approches de gestion de l'écart

Vue la complexité des systèmes et la diversité des problèmes liés au cahier des charges, il s'avère indispensable de donner une importance à ce document et de renforcer son existence, sa trace et sa communication avec les autres documents dans le même processus de développement tels que les spécifications, les modèles de conception et les programmes. La qualité de ce document est déterminante pour mener à terme les étapes du développement et pour l'obtention d'un système final de qualité, cohérent et correct. Dans la suite, nous exposons des travaux autour des deux mondes, formel et informel, permettant d'établir des interactions et des échanges entre eux.

2.3.2.1 WRSPM

Les auteurs de [Gunter et al., 2000] décrivent un modèle de référence pour le développement des exigences client en appliquant des méthodes formelles. Ce modèle, appelé WRSPM, est basé sur cinq artefacts :

- la connaissance du domaine, notée *W* (World assumptions),
- les exigences du client, notées *R* (Requirements),
- la spécification faisant le lien entre le système et son environnement, notée *S* (Specification),
- le programme qui implante la spécification, noté *P* (Program) et
- la plateforme de programmation fournissant un environnement d'exécution, notée *M* (Machine).

Les auteurs introduisent la spécification S comme interface entre le système et l'environnement. Leur modèle WRSPM est indépendant du choix du langage. Chaque artefact est exprimé dans un langage. Ils décrivent la relation entre le système et l'environnement via un ensemble de propriétés d'adéquation, de consistance et de consistance relative. Cette approche est formelle. Elle engendre des difficultés pour les développeurs n'ayant pas de connaissances notamment en logique d'ordre supérieur HOL [Leivant, 1994].

2.3.2.2 ProR

Cette approche est basée sur des concepts issus de l'approche WRSPM. Dans [Jastram et al., 2011], les auteurs proposent une approche de gestion des besoins appelée *ProR*. Cette approche commence par l'élicitation des exigences initiales et les hypothèses du domaine. Les auteurs ont également développé un outil ProR [Jastram, 2010]. Cet outil a été initialement développé à l'institut d'Informatique de l'Université Heinrich-Heine à Düsseldorf et actuellement fait partie d'Eclipse Foun-

dation [26]. Il est basé sur le standard ReqIF [27] et sur le Requirements Modeling Framework (RMF), fournissant le noyau pour ProR. Cet outil est également intégré comme plug-in de la plateforme Rodin [Abrial et al., 2010].

Un besoin sous ProR est décrit à l'aide de plusieurs *paramètres* ou champs. La figure 2.8 présente quelques uns de ces paramètres :

- *ID* décrit un identifiant unique de chaque besoin du cahier des charges.
- *Description* contient une description initialement informelle du besoin. Celle-ci inclura des termes formels lorsqu'ils ont été introduits dans la spécification formelle.
- *Link* exprime le lien établi entre les besoins écrits sous ProR et les éléments de la spécification formelle correspondante. Concrètement sous l'outil ProR, ce champ permet de localiser - dans la spécification formelle - les éléments formels liés avec une ou plusieurs exigences concernées.

ID	*Description*	*Link*
Req1	Description d'un besoin parent	
Req1-1	Description d'un besoin enfant	
Req1-2	Description d'un autre besoin enfant	

FIGURE 2.8 – Structure des besoins sous ProR

ProR permet d'exprimer une structure *hiérarchique* des exigences et supporte le raffinement : les besoins sont détaillés en utilisant cette notion de hiérarchie. Celle-ci est exprimée via deux types de liens entre exigences :

- lien *parent-enfant* dans lequel le besoin enfant donne des détails sur le besoin parent. Dans la figure 2.8, *Req1* est un parent des besoins *Req1-1* et *Req1-2* et
- lien *frère-frère* où les deux besoins sont de même niveau tel que *Req1-1* et *Req1-2*.

Outre cette structure hiérarchique proposée par ProR, cet outil offre un environnement pour décrire la notion de *lien*. Un lien concerne aussi bien les exigences entre elles-mêmes (*lien inter-exigences*) que les exigences avec les autres composants du processus de développement tels que les spécifications formelles (*lien exigences-spécification*).

- Pour le premier type de liens, les besoins d'un cahier des charges peuvent communiquer entre eux via des liens notés par les symboles $\triangleright$ et/ou $\triangleleft$. Soient R1, R2 et R3 trois besoins décrits sous ProR :
  - un lien R1 $\triangleright$ R2 exprime que R1 *est suivi de* R2. Ce lien décrit un enchaînement entre les besoins,
  - un lien R3 $\triangleleft$ R2 indique que R3 *est précédé par* R2 et
  - un lien R1 $\triangleright$ R2 $\triangleright$ R3 exprime que R2 *est précédé par* R1 et *suivi de* R3.
- Pour le deuxième type de liens, l'approche ProR le réalise en permettant d'introduire des termes formels dans la description informelle des besoins. Ceci commence une fois le développement de la spécification formelle démarre. Ces termes sont mis entre *crochets [ ]* et introduits avec leurs descriptions informelles correspondantes dans un glossaire. Les liens fournis avec cette approche permettent d'aller des besoins vers la spécification formelle mais pas l'inverse.

26. https ://www.eclipse.org/
27. http ://www.eclipse.org/rmf/

2.3.2.3 KAOS

[van Lamsweerde, 2000] insiste sur l'importance de la spécification du domaine et des besoins pour faciliter leur compréhension, leur communication et leur évolution. Il présente un ensemble de techniques décrites dans l'approche KAOS (Keep All Objectives Satisfied) orientée buts pour la gestion des exigences [van Lamsweerde, 2008] [van Lamsweerde, 2009]. Cette approche est basée sur une identification et un raffinement des objectifs progressivement jusqu'à l'obtention des contraintes. Il décrit un modèle d'exigences selon cinq vues complémentaires :

(1) Une vue *intentionnelle* assurée par un modèle d'objectif. Elle définit les objectifs du futur système.

(2) Une vue *structurelle* assurée via un modèle d'objet. Cette vue détaille les composants prévus.

(3) Une vue de *responsabilité* décrite par un modèle d'agent et associe à chaque agent un ensemble de services et de responsabilités à accomplir.

(4) Une vue *fonctionnelle* réalisée par un modèle d'opération. Elle modélise les exigences fonctionnelles du système.

(5) Une vue *comportementale* présentée par un modèle de comportement. Cette vue décrit le comportement prévu du système.

Un outil appelé Objectiver [28] [Delor et al., 2003] a été conçu dans ce contexte. Il permet la réalisation de toutes les vues du modèle en KAOS et de les rassembler avec différents niveaux de raffinement.

Selon ces différents types de modèles, la description des exigences dans cette approche est basée sur des objectifs, des agents, des objets, des opérations et des comportements. La communication entre les modèles est assurée via différents types de liens. Ces derniers sont définis via un ensemble de règles vérifiées automatiquement. Le modèle d'objectif (goal model) pour les exigences inclut des agents pour accomplir l'objectif (goal satisfaction). L'auteur fait la différence entre la description des propriétés du domaine qui doivent être stables et non modifiables et celle des objectifs qui sont variables et susceptibles à des modifications. Le modèle d'exigences combinant les différentes vues (multi-view model) aide à montrer clairement les différents aspects concernés par le futur système à savoir ses objectifs, ses opérations, ses acteurs, ses objets et ses comportements. Cependant, l'élaboration de ce modèle s'avère difficile lorsqu'il s'agit d'un système complexe. L'approche KAOS définit une méthodologie pour simplifier cette tâche. Elle propose un ensemble d'heuristiques et de règles permettant de définir chaque modèle. Dans cette approche, un modèle d'exigences est composé de plusieurs niveaux d'abstraction qui concernent les objectifs et les sous-objectifs. L'auteur exploite différents outils et techniques comme la réutilisation pour la construction des modèles, la logigue temporelle linéaire LTL [Pnueli, 1977], le solveur SAT [Moskewicz et al., 2001] et les patrons de raffinement, d'opérationalisation, d'obstruction et de divergence. Les modèles orientés-objectifs sont utilisés pour générer un cahier des charges complet et enrichi avec plus de détails décrivant tous les cas prévus ainsi que les risques contournant le futur système.

L'approche KAOS utilise les méthodes et techniques formelles telles que LTL et solveur SAT dans quelques étapes de la construction du modèle d'exigences. L'évolution des exigences est décrite via deux points dans KAOS : le premier consiste à la description initiale des exigences en langage naturel, et le deuxième est présenté par le cahier des charges enrichi obtenu suite à l'application de l'ensemble

28. `http://www.objectiver.com/index.php?id=23`

d'étapes de construction du modèle d'exigences.

Les auteurs de [Ponsard et al., 2015] complètent cette approche en considérant les interactions entre les artefacts des besoins. Ils utilisent des outils comme *Objectiver* et des diagrammes tels qu'UML [29], SysML [30] et BPMN [31] pour aider à la compréhension des besoins et offrir une vue d'ensemble sur les documents au sein des projets de grandes tailles.

2.3.2.4 Structures semi-formelles

Les auteurs de [Alkhammash et al., 2015] proposent une approche pour gérer la traçabilité entre les besoins et les modèles en B événementiel. Cette approche, représentée par la figure 2.9, utilise des structures semi-formelles à l'instar d'UML-B [Snook and Butler, 2006] et d'Event Refinement Structure (ERS) [Fathabadi et al., 2012] pour construire un pont entre les exigences et la spécification formelle en B événementiel. La structure UML-B est une notation qui offre un environnement graphique de modélisation permettant le développement des modèles formels en utilisant une notation graphique d'UML. Un avantage de cette structure UML-B est sa capacité à supporter le raffinement. ERS est une approche basée elle aussi sur une notation graphique. Son apport majeur consiste à augmenter le raffinement en B événementiel avec une notation graphique capable de représenter explicitement les relations entre les événements abstraits et les événements concrets.

Les auteurs décrivent leur méthodologie en trois étapes. Tout d'abord, ils classifient les exigences - selon les structures dans les modèles B événementiel - en cinq classes : *orienté-données*, *orienté-contrainte*, *orienté-événement*, *orienté-flux* et *autres*. Par la suite, ils utilisent UML-B et ERS pour fournir une vue graphique des exigences classifiées et développer une stratégie de raffinement du futur modèle. Ils génèrent la spécification formelle en B événementiel en utilisant les outils appropriés.

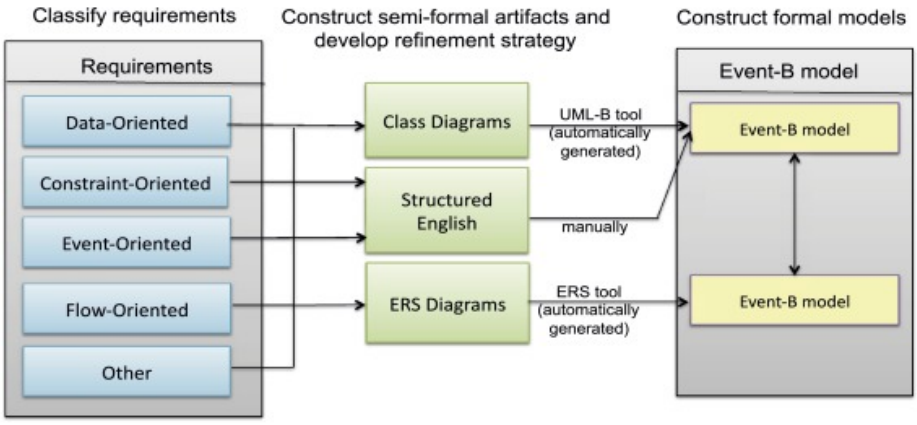


FIGURE 2.9 – Étapes de construction des modèles formels traçables [32]

29. Unified Modelling Languages : `http://www.omg.org/spec/UML`
30. System Modeling Language : `http://www.omgsysml.org/`
31. Business Process Model and Notation : `http://www.omg.org/spec/BPMN/2.0`
32. Figure présentée dans le papier [Alkhammash et al., 2015]

2.3.2.5 Approche par réécriture

Abrial a traité le problème de l'écart entre les deux mondes en commençant par le document d'exigences du client. Dans l'introduction du livre [Abrial, 2010], l'auteur exhibe les problèmes liés au cahier des charges. Ce document est souvent mal rédigé, pauvre voire inexistant. Tel qu'il est présenté par le client, le cahier des charges ne se prête pas bien à l'utilisation dans les phases suivantes du développement. L'auteur recommande, dans ce livre ainsi que dans [Su et al., 2011], de réécrire les besoins du client dans une étape de *restructuration*. En s'inspirant de l'organisation des documents en mathématiques tel est le cas des théorèmes (voir figure 2.10), l'idée de cette étape consiste à réorganiser le cahier des charges en deux parties distinctes :

- *Un texte explicatif.* Il englobe une description détaillée du futur système. Cette description est réalisée en utilisant différentes formes : des descriptions techniques, des formules, des paragraphes en langues naturelles, des images explicatives, etc. Un texte explicatif sert à expliquer le "pourquoi ?" et le "comment ?" et est destiné à la compréhension du système. Il ne servira pas dans la suite du processus de développement. Une fois compris et maîtrisé, ce texte devient moins important et ne sera consulté, dans la suite des étapes, qu'en lecture pour comprendre ou vérifier des détails.
- *Un texte référentiel.* Comme son nom l'indique, ce texte est la référence pour le reste des étapes du développement. Il s'agit d'une description concise des besoins et des propriétés essentielles du système à développer à travers laquelle on exprime le "quoi ?". Abrial propose de réorganiser ce texte sous forme de phrases courtes, lisibles et étiquetées. Les étiquettes servent comme identifiants des phrases dans le cahier des charges (texte référentiel) et favorisent la traçabilité des besoins dans les autres étapes du développement. Selon l'objectif du besoin, l'étiquette qui lui est associée appartient à un des ensembles suivants :
  - *FUN* pour les exigences décrivant des fonctions du système,
  - *ENV* pour les exigences externes au système et appartenant à son environnement,
  - *SAF* pour les besoins détaillant des contraintes de sécurité,
  - *EQU* pour les équipements physiques,
  - *DEL* pour les exigences décrivant des délais ou le temps,
  - ...

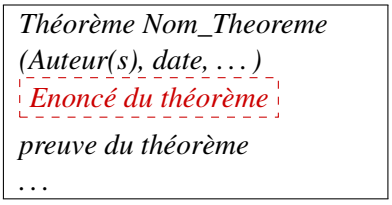


FIGURE 2.10 – Description d'un théorème en mathématiques

En revenant à la figure 2.10, on peut établir l'analogie suivante [Abrial et al., 2010] :

- le texte référentiel est analogique à l'énoncé du théorème,
- le texte explicatif est analogique aux détails cités dans l'entête du théorème - à savoir le nom du théorème, son ou ses auteur(s), sa date - et à sa preuve.

L'approche d'Abrial vise la simplification de la prise en compte des besoins et leur traçabilité dans un processus de développement. Le document référentiel résultant sera fréquemment consulté, utilisé et évolué via des modifications tout au long du développement. Il ne doit pas contenir des détails techniques ni des phrases complexes et difficiles à comprendre. Cette méthode de réécriture des besoins est utile pour les systèmes actuels devenant de plus en plus hybrides [Su et al., 2014] et qui sont généralement difficiles à comprendre [Abrial, 2006a]. L'accès aux besoins ainsi qu'à leur trace tout au long du processus de développement est assuré vu que chaque besoin réécrit est référencé par une étiquette ou identifiant unique. Un autre apport de la réécriture des exigences est de résoudre le problème de communication entre les différentes parties prenantes du cahier des charges : les besoins réécrits sont plus lisibles et ne nécessitent pas de compétence particulière pour les comprendre.

Afin de rédiger un texte référentiel de qualité, il est important d'avoir une définition claire de ce document. L'Association Française d'Ingénierie Système (AFIS) [33] définit un référentiel d'exigences comme suit :
"Véritable carte d'identité du système (futur ou existant), le référentiel d'exigences est la structure d'informations pivot de l'ingénierie des systèmes. C'est la référence (qui peut évoluer dans le temps) des activités de conception, vérification, validation et soutien. Il doit cependant être accompagné d'une structure de traçabilité pour permettre de relier les exigences aux éléments de conception, les exigences entre elles et les exigences avec leur origine."

L'AFIS décrit, dans un de ces rapports datant de 2001, des recommandations et des bonnes pratiques pour la rédaction d'un référentiel d'exigences. Ces bonnes pratiques concernent l'élaboration et l'analyse des exigences, l'identification des parties prenantes, l'identification des sources des besoins (venant du client, venant de l'étape de conception, etc.) et leurs modifications.

2.3.2.6 Agendas

L'approche décrite dans [Heisel and Souquières, 1999] propose un guide méthodologique pour l'élicitation des besoins et de spécification. Cette approche n'introduit pas de nouveaux langages ni de formalismes. Le processus d'élicitation des exigences est indépendant du langage de spécification choisi. Un lien de traçabilité entre les exigences et leur spécification est maintenu via les *agendas* qui expriment le sommaire de la méthode. Les agendas décrivent et guident le processus de développement, assurant une certaine garantie de la qualité du produit obtenu à travers des conditions de validation associée aux différentes étapes.

2.3.2.7 Approche agile

Apparue dans les années 80, cette approche a eu un succès et plusieurs méthodes et approches ont émergé en s'y référant. Parmi ces méthodes, les plus connues sont *XP (Extreme Programming)* décrite en 1999 par Kent Beck [Beck, 1999] et *Scrum* présentée en 1995 par Ken Schwaber [Schwaber, 1995]. L'approche agile est itérative et collaborative : elle encourage la collaboration entre les différentes parties prenantes dans le projet et assure la révision des différents documents du projet en faisant des retours pour les faire évoluer. Ces documents sont ceux issus des étapes de développement tels que les besoins issus de l'étape de collecte et analyse des besoins, les modèles formels venant de l'étape de spécification, les modèles issus de l'étape de conception et les programmes résultant du codage ou

33. https://www.afis.fr

de l'implémentation.

Dans le site officiel [34], les auteurs encouragent l'utilisation de la méthode agile pour développer des logiciels par la pratique. Ils ont rédigé, en 2001, un manifeste constitué des quatre points suivants dans lesquels ils favorisent :

- les individus et leurs interactions plus que les processus et les outils,
- des logiciels opérationnels plus qu'une documentation exhaustive,
- la collaboration avec les clients plus que la négociation contractuelle et
- l'adaptation au changement plus que le suivi d'un plan.

Ce manifeste suit douze principes. La table 2.3 en propose quatre [35] avec l'aspect à prendre en compte par chacun d'eux.

Principe	***Aspect***
1. Notre plus haute priorité est de satisfaire le client en livrant rapidement et régulièrement des fonctionnalités à grande valeur ajoutée	validation
2. Accueillez positivement les changements de besoins, même tard dans le projet. Les processus Agiles exploitent le changement pour donner un avantage compétitif au client	prise en compte des changements des besoins
3. Les utilisateurs ou leurs représentants et les développeurs doivent travailler ensemble quotidiennement tout au long du projet	coopération avec le client
4. La méthode la plus simple et la plus efficace pour transmettre de l'information à l'équipe de développement et à l'intérieur de celle-ci est le dialogue en face à face	coopération entre équipes du projet

TABLE 2.3 – Quelques principes du manifeste agile

Ces principes apportent un élément de réponse au problème d'écart entre les différents mondes notamment les besoins et la spécification formelle. Ils favorisent les échanges entre les parties prenantes du document d'exigences et éventuellement la participation du client dans les différentes étapes de développement.

2.3.2.8 Autres approches

Parmi ces approches, l'utilisation des ontologies occupe une place importante. Dans ce cadre, [Driss, 2014] propose l'utilisation d'un modèle pivot basé sur les ontologies pour relier les exigences et la spécification formelle. De leur part, les auteurs de [Ameur and Méry, 2016] utilisent les ontologies afin d'expliciter les connaissances du domaine dans le développement formel des systèmes complexes.
Afin de résoudre le problème d'exclusion du client et de coopération entre les différentes équipes d'un projet, [Idani, 2006] propose la traduction des modèles B vers des modèles UML.

Une autre forme de réduction de l'écart entre les deux mondes consiste à réutiliser l'existant pour faire évoluer l'ensemble des documents. La réutilisation de composants logiciels dans les premières phases

34. `http://agilemanifesto.org/iso/fr/manifesto.html`
35. Ces manifestes sont cités à l'adresse : `http://agilemanifesto.org/iso/fr/principles.html`

du cycle de développement logiciel a un impact positif sur les projets logiciels [Cybulski et al., 1998]. Dans leur papier, les auteurs décrivent une méthode de classification des différents artefacts des besoins. Ils ont identifié une centaine de types d'artefacts et esquissent des techniques et méthodes permettant la réutilisation de ces artefacts en se basant sur la catégorie de chacun d'eux. Quant à [Wang, 1998], il a présenté une expérience pratique de la réutilisation précoce c'est-à-dire la réutilisation des besoins ou de la spécification. L'auteur propose une approche pour cette réutilisation dans les premières étapes de développement appelée FORE (Family Of REquirements). Son objectif est de développer une famille d'exigences génériques réutilisables. Cette approche permet d'enchaîner facilement les différentes étapes de développement en exploitant ce qui a été développé dans d'autres applications. Ceci réduit le risque des oublis et des erreurs.

Dans [Cimatti et al., 2008], les auteurs posent la problématique : comment peut-on valider les exigences et vérifier leur consistance ? Ils proposent une méthodologie de validation des besoins basée sur l'utilisation des méthodes (semi-)formelles telles que UML et le langage naturel contrôlé, Controlled Natural Language (CNL). Leur démarche est basée sur une :

- Phase d'*analyse informelle*, dans laquelle le développeur :
  - — classifie les exigences en divers types notamment : glossaire, architecture, exigences fonctionnelles, exigences de communication, comportement, exigences environnementales, scenario, propriétés et annotation,
  - — réécrit des exigences en phrases simples et
  - — analyse informellement les exigences réécrites pour détecter les inconsistances.
- Phase de *formalisation* qui consiste à :
  - — transformer des besoins en diagrammes (de classes, de séquences, state machines, etc.),
  - — formaliser quelques catégories d'exigences (comportementales, scénarios et de propriétés) avec CNL et
  - — ajouter des liens entre exigences et diagrammes en utilisant des outils automatiques tels que IBM Rational RequisitePro [36] et IBM Rational Software Architect [37].
- Phase de *validation formelle* : il s'agit de valider les exigences relativement à diverses propriétés telles que la consistance et l'absence de redondance via les outils de LTL et les solveurs SMT.

Les auteurs de [Aceituna et al., 2014] proposent une méthode pour traduire incrémentalement des besoins décrits en langage naturel en un diagramme d'états-transitions. Leur approche offre, aux ingénieurs en IE ainsi qu'aux autres parties prenantes, la possibilité de participer à cette traduction. Ils insistent sur l'importance de l'inspection du logiciel durant son développement. Ceci se réalise par l'intervention d'une équipe d'individus spécialistes pour revoir le produit logiciel afin d'identifier les erreurs comme les incohérences et les incomplétudes. Ils ont utilisé leur approche pour la détection des exigences fonctionnelles manquantes et des ambiguïtés.

2.3.3 Avantages de la réduction d'écart

La qualité d'un système ou logiciel ainsi que le coût de son développement sont affectés par la présence des liens entre les différents documents du même projet ou leur absence. Les effets touchent les différents niveaux de développement.

36. https://www.ibm.com/support/knowledgecenter/en/SSSHCT
37. https://www-03.ibm.com/software/products/fr/ratsadesigner

2.3.3.1 Compréhension des besoins

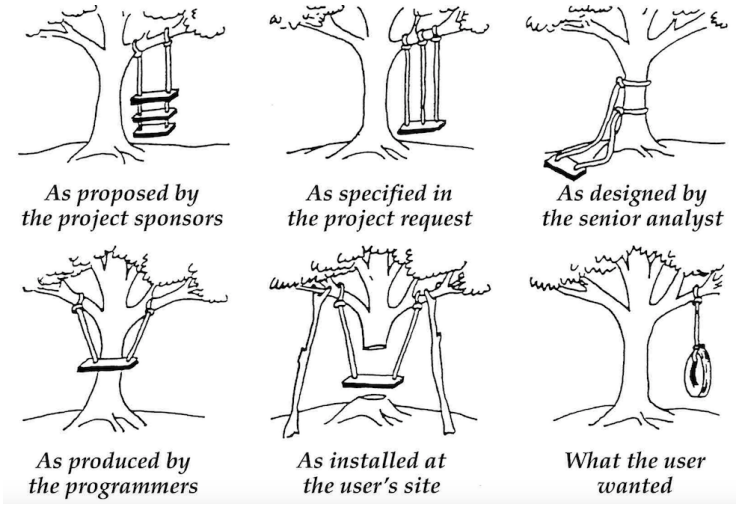


FIGURE 2.11 – Différentes compréhensions du cahier des charges [38]

Il s'agit d'une étape importante avant de commencer et au cours du développement. Elle aide à définir et savoir ce qu'on veut du futur système [van Lamsweerde, 2008]. Le cahier des charges du client est écrit sous une multitude de catégories des besoins : phrases en langue naturelle, schémas, dessins, formules venant du domaine, ou des tables. Bien que cette diversité est censée favoriser une meilleure lecture de la demande du client, elle peut engendrer des difficultés de compréhension et rend les besoins sujets à différentes interprétations. Prenons le fameux exemple de la demande du client pour une balançoire. Cette demande est interprétée sous multiples manières par les parties prenantes du même projet, voir figure 2.11. La diversité de ces interprétations invite à trouver un consensus entre ces parties prenantes pour se mettre d'accord. La présence de la trace des besoins dans toutes les étapes de développement aide à la détection et la résolution de ces incompréhensions. Dans son livre [Lamport, 2002], Leslie Lamport définit la spécification comme : *"une description écrite de ce que le système est censé faire. Spécifier un système nous aide à le comprendre. C'est une bonne idée de comprendre un système avant de le construire, c'est donc une bonne idée d'écrire une spécification de système avant de l'implémenter"*. Dans cette définition, l'auteur insiste sur l'importance de comprendre le futur système à partir de sa spécification. Comprendre le futur système signifie comprendre les besoins du client. Cette tâche est primordiale vu qu'elle constitue la pierre fondatrice du processus de développement : une bonne compréhension des besoins favorise l'obtention d'un logiciel ou système de qualité. Cette compréhension évolue et s'affine au fur et à mesure du processus de développement.

2.3.3.2 Détection des lacunes

L'importance de communication ou de liaison entre les besoins et leurs spécifications formelles a été discutée dans un rapport NASA datant de 1996 [Ben, 1996]. Il s'agit d'une formalisation du *système de positionnement global (GPS)* en utilisant PVS (Prototype Verification System), un environnement de spécification et de vérification développé au sein du laboratoire SRI International's Computer

38. Figure issue d'une illustration par S. Høgh in Teaching Design (1993), Source : `www.businessballs.com`

Science [Owre et al., 1995]. Ce projet fait partie des projets de démonstration pour les applications dans l'espace proposé par la NASA Formal Methods [39]. L'objectif de ce projet est de déceler les apports de l'utilisation des méthodes formelles dans l'étape d'analyse des besoins ainsi que dans les autres étapes. Dans le rapport [Ben, 1996], l'auteur montre que le développement des spécifications formelles a aidé à découvrir des problèmes dans *86 besoins* du cahier des charges du système. Ces anomalies, réparties entre mineures et substantielles, n'ont pas été détectées lors de la phase d'analyse des besoins.

Revenons au papier [Abrial, 2003] dans lequel Abrial a cité les causes d'échec de la preuve (voir section 2.2.3 paragraphe *Vérification*). Parmi ces causes, les besoins peuvent être coupables : ils présentent des lacunes et des incohérences empêchant la correction de la spécification. C'est grâce aux liens établis entre le document d'exigences et la spécification formelle que ces défauts peuvent être détectés facilement et corrigés. En leur absence, la correction d'une erreur peut prendre du temps voire ne pas être prise en compte.

2.3.3.3 Prise en compte des changements

Ils peuvent venir du client ou du changement de l'environnement du système à développer. La présence de liens entre les besoins et les différents documents facilite la mise à jour de l'ensemble. Ainsi, chaque monde évolue en harmonie avec les autres et le système final converge pas-à-pas vers la réalisation de la demande du client. Dans le cas contraire, si les besoins ne sont pas tracés dans le processus de développement, le coût de de la prise en compte des changements sera très élevé. Le terme *coût* signifie temps, efforts de développement et argent.

2.3.3.4 Justification des éléments logiciels

La présence des liens entre exigences et spécification formelle aide à justifier la présence des éléments de cette spécification : chaque élément est licitement présent vu qu'il est issu d'un besoin défini dans le cahier des charges. L'absence de ces liens entraîne un doute sur l'origine des parties ou éléments de la spécification ce qui rend la tâche de développement de plus en plus difficile. Ceci est essentiellement lié à l'aspect contractuel du document d'exigences. Rappelons que l'objectif d'un développement logiciel en utilisant les approches formelles est d'obtenir un logiciel vérifié et validé.La justification de ces éléments est examinée lors de l'activité de validation. Cette activité nécessite des retours aux besoins pour être accomplie. Elle devient de plus en plus fastidieuse en l'absence de pont entre les documents : on ne sait pas qui (partie spécification) fait quoi (côté besoins) ; le risque d'oublier de prendre en compte certains besoins se présente.

2.4 Patrons

Les deux mondes des besoins et de spécification formelle doivent évoluer en même temps. Afin de garantir un passage aisé entre les différentes étapes du développement, l'automatisation par réutilisation de l'existant se présente comme une des alternatives dans ce contexte. Cette section présente un aperçu des patrons et leur utilisation en génie logiciel. Servant à résoudre une classe de problèmes, un patron aide au développement grâce au concept d'instanciation en s'adaptant au problème en question. Nous nous intéressons spécifiquement aux patrons de conception et à ceux de spécification formelle.

39. `http://www.fmeurope.org/symposia/`

2.4.1 Généralités

2.4.1.1 Réutilisation en génie logiciel

Partons du principe *"ne pas réinventer la roue"*. Celui-ci introduit un concept fondamental en informatique : la *réutilisation de l'existant* permet aux projets informatiques d'optimiser le coût de développement en termes de temps, d'argent et d'efforts. La réutilisation permet l'évolution des systèmes existants. Cependant, une mauvaise réutilisation peut mener à de lourdes pertes. Prenons l'exemple du crash du vol 501 d'Ariane 5. Il s'agit d'un vol inaugural du lanceur Ariane 5 ayant lieu le 4 juin 1996. La fusée s'est auto-détruite environ 37 secondes après son lancement à une altitude de 3700 mètres. Aucune perte humaine n'a été déplorée, mais plusieurs centaines de millions de dollars ont été perdus. Après cet accident, un groupe de chercheurs comprenant Gilles Kahn et Jacques-Louis Lions a été désigné pour faire une enquête et préciser la ou les causes de cet échec. Dans leur rapport publié le 19 juillet 1996 [Lions et al., 1996] [40], les auteurs ont précisé que la cause principale concerne une défaillance du logiciel de la référence inertielle du système, système de guidage de la fusée entre autres. Il s'agit d'un dépassement de capacité dans la mémoire pour une variable d'accélération horizontale. Cette variable devait être codée sur 9 bits au lieu de 8 bits. Une différence d'un seul bit a causé ce lourd échec. En réalité, le logiciel de navigation de la fusée Ariane 5 était celui qui avait été conçu pour son prédécesseur, Ariane 4. Ceci a induit une incompatibilité entre le logiciel et le matériel vu que la fusée Ariane 5 est plus puissante et nécessite plus d'espace mémoire pour effectuer les calculs. En revenant sur ce détail, nous découvrons l'importance et le danger qui contournent la notion de réutilisation. Elle favorise un gain de temps et d'argent, mais sa mauvaise adaptation mène à de lourdes pertes.

D'une manière générale, le développement des systèmes et logiciels corrects critiques et complexes est une tâche fastidieuse. D'une part, elle nécessite des connaissances des langages utilisés et des efforts importants en termes de temps et d'investissement des développeurs pour l'élaboration des systèmes. D'autre part, elle est guidée par l'utilisation des outils disponibles dans le développement. Bien qu'il existe des aides et des méthodes, telles que la méthode agile, qui permettent l'obtention de systèmes et logiciels pertinents, il n'existe quasiment pas de guides méthodologiques qui facilitent le développement. Un problème donné peut être perçu dans plusieurs projets informatiques. Il peut également être développé pour plusieurs langages et formats mais il n'est pas généralisé dans une optique de réutilisation. Par conséquent, le travail pour un problème similaire est refait plusieurs fois de différentes manières. Parmi les idées proposées dans le cadre de réutilisation, nous identifions :

- *la programmation orientée composants* : comme en électronique, un composant en informatique est un élément logiciel pouvant être présenté sous forme d'un code compilé. Il est développé, testé et mis à disposition des développeurs pour le réutiliser, l'adapter et l'intégrer dans un projet. Cet aspect de réutilisation du composant n'est assuré que lorsque ce composant a un comportement générique. D'où la notion de *généricité*. Celle-ci consiste à abstraire des comportements ou des éléments logiciels de manière à les rendre facilement réutilisables dans différentes applications indépendamment des types de données traitées.
- *L'utilisation des patrons* : ce concept découle de la notion de généricité et favorise la réutilisation.

2.4.1.2 Qu'est-ce qu'un patron ?

L'idée des *patrons* a été initialement introduite par l'architecte Christopher Alexander dans le livre [Alexander et al., 1977], portant sur l'architecture et l'habitation. Dans leur ouvrage, les auteurs

40. Ce rapport est disponible sur le site `http://esamultimedia.esa.int/docs/esa-x-1819eng.pdf`

ont décrit un nouveau langage appelé *"langage de schéma"*. Ce langage est caractérisé par des *"modèles"* appelés *"patrons"* permettant de décrire une méthode de travail dans le domaine d'architecture urbaine. D'une manière générale, un patron décrit un savoir-faire du domaine. Il présente et identifie un sous-problème "déjà vu" et pour lequel une solution réutilisable acquise par l'expérience a été développée. En informatique, cette solution est paramétrée et provient de l'expertise des développeurs ayant travaillé sur ces problèmes récurrents. Le patron est instancié pour un problème particulier en donnant des valeurs effectives à ses paramètres génériques. Dans le cycle de développement d'un logiciel, les patrons existent et s'appliquent dans différentes phases. Ils concernent la phase de codage, de conception et de spécification, voir figure 2.12.

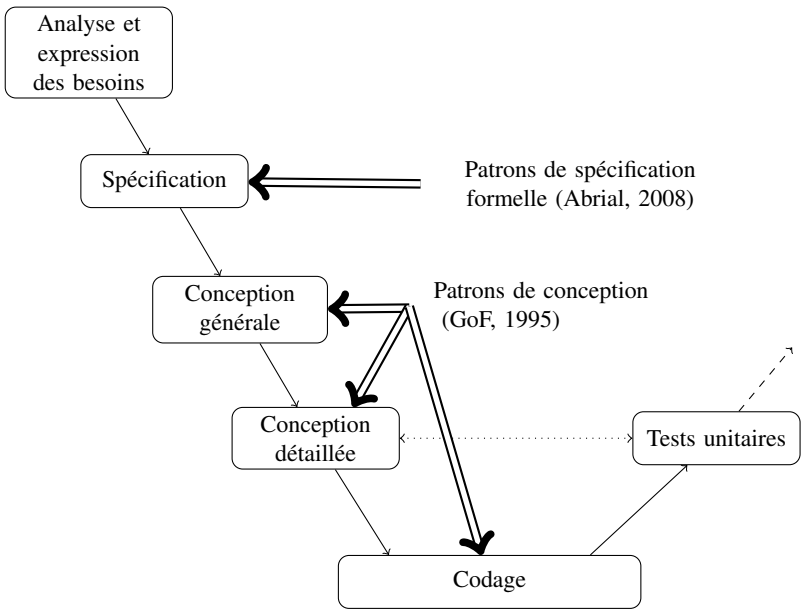


FIGURE 2.12 – Place des patrons dans un processus en V

2.4.2 Patrons de conception

Les plus connus sont ceux introduits dans les années 95 par la *"Bande des quatre, GoF (Gang of Four)"* et les patrons *GRASP (General Responsibility Assignment Software Patterns)* dans un cadre orienté objet présentés dans la suite.

2.4.2.1 Organisation

Un patron ou modèle de conception, selon les auteurs de [Gamma et al., 1996], fait "*la description d'objets coopératifs et de classes que l'on a spécialisés pour résoudre un problème général de conception dans un contexte particulier*". Dans ce cadre, les auteurs ont développé un catalogue comprenant 23 patrons. Leurs descriptions sont fournies sous forme :

- de graphiques à l'aide des diagrammes en UML et
- de prototypes ou exemples de codes implémentés en langage C++ [Delannoy, 2000].

Le catalogue des patrons de GoF est organisé selon deux critères :

- Le *rôle* qui traduit ce que fait le modèle. Selon ce critère, les patrons de conception de GoF s'organisent autour de trois types [41] :
  - — *créateur* s'intéressant au processus de création des objets,
  - — *structurel* définissant comment organiser les classes ou les objets dans une structure plus large séparant l'interface de l'implémentation et
  - — *comportemental* décrivant la manière d'organiser et d'interagir des classes ou des objets et de se répartir les responsabilités.
- Le *domaine* qui précise si le modèle s'applique à des classes ou à des objets. Selon ce critère, un patron peut concerner :
  - — *des modèles de classes* qui traitent des relations entre les classes et leurs sous-classes ou
  - — *des modèles d'objets* qui s'intéressent aux relations entre les objets. Ces relations peuvent être modifiées lors de l'exécution et sont considérées dynamiques.

2.4.2.2 Formalisme

Outre les notations graphiques, les patrons de conception de GoF sont décrits via un *canevas*. Celui-ci désigne un format unique permettant l'apprentissage facile des modèles, leur compréhension et leur utilisation. Pour chaque patron, un canevas est décrit selon les éléments suivants :

- *Nom* : décrit le principe du patron. Les auteurs définissent, dans la section 1.1 du chapitre 1 du livre [Gamma et al., 1996], un nom comme *"un moyen de décrire en un ou deux mots un problème de conception, ses solutions et leurs conséquences"*. Ils assument aussi que *"trouver de bons noms a été une des tâches les plus difficiles de la constitution de notre catalogue"*.
- *Problème* : décrit la raison d'être du patron ainsi que le problème qu'il résout.
- *Solution* : définit les éléments de la solution proposée par ce patron. Cette solution est décrite en donnant une structure du patron, ses constituants et la collaboration entre ses constituants.
- *Conséquences* : présente les effets résultants de l'application du patron concerné ainsi que les compromis qui émergent lors de cette application.

2.4.2.3 Utilisation

Le canevas décrit précédemment fournit un guide permettant d'aider à décider lequel, parmi les patrons définis, est plus adapté à un problème donné. Les noms des patrons donnent une indication précieuse au développeur pour sa décision.
Prenons l'exemple du patron *Singleton*. Il s'agit d'un modèle dont le rôle est *créateur* et le domaine est *objet*. Comme son nom l'indique, celui-ci permet la création d'une instance unique d'une classe. Il est utile dans les applications nécessitant la création d'un seul objet d'une classe comme celles demandant la création d'un seul administrateur.

2.4.3 Patrons de spécification formelle

Ils ont été décrits en 2008 par [Abrial and Hoang, 2008]. Ce sont des modèles en B événementiel consacrés à la formalisation d'un sous-problème typique tel que celui des protocoles de communication [Hoang et al., 2013]. De tels modèles sont prouvés mathématiquement sous la plateforme Rodin.

41. Description issue de [Gamma et al., 1996] avec quelques adaptations

Pour un problème issu du monde réel, l'idée consiste à instancier le patron de spécification concerné afin d'aider à la construction du modèle formel et de réduire l'effort de preuve. Cette idée est inspirée des patrons de conception. Elle est rentable lorsqu'il s'agit de spécifier un système critique complexe. Les auteurs de ces patrons soulignent les atouts de leur approche, notamment :

- La réduction de l'effort de développement : les composants - machines et contextes - du patron sont développés et prêts à la réutilisation. L'élaboration de certaines parties de la spécification pour une application réelle est guidée par une instanciation du patron concerné. Ceci implique un effort et un temps réduits de développement mais aussi un risque affaibli d'oubli de certains éléments formels de la spécification.
- L'automatisation de la preuve : le patron est vérifié correct avant son instanciation. Ceci se réalise grâce aux outils de preuves de la plateforme Rodin. Par conséquent, les parties de la spécification formelle élaborées à partir de l'instanciation des patrons sont automatiquement prouvées correctes.
- L'existence d'un outil *Pattern* [42] développé comme plugin de la plateforme Rodin. Cet outil permet une adaptation systématique des patrons décrits en B événementiel et leur incorporation ou intégration dans des spécifications formelles complexes.
- Le support du raffinement : un patron de spécification formelle peut être identifié, instancié et appliqué graduellement dans chaque pas de raffinement. Il est adapté et intégré dans une spécification formelle de taille plus large.

2.5 Conclusion

Le problème de construction d'un pont entre le monde des besoins et le monde des spécifications formelles n'est pas récent. Il a été étudié par plusieurs approches comme l'approche d'Abrial, KAOS, ProR et l'approche agile qui contribuent à la réduction de l'écart entre ces deux mondes via des interactions et des échanges. Le pont établi est important pour assurer la communication entre deux documents de différentes natures : un qui est écrit en langage naturel et qui peut contenir tout type d'imperfection et un autre très rigoureux écrit en langage de spécification formelle comme B événementiel, LTL ou TLA$^+$. L'importance des interactions réside dans la détection des incohérences et lacunes des besoins et contribue à l'obtention d'un système ou logiciel de qualité.

Les deux mondes des besoins et des spécifications formelles doivent évoluer en même temps. Une manière de faciliter cette évolution est de recourir à la réutilisation, une notion clé en génie logiciel. Parmi les idées proposées dans ce contexte, les patrons émergent comme un concept important fournissant un guide et une aide aux développeurs. Ils permettent d'éviter de refaire le travail pour des problèmes similaires. Les patrons de GoF se présentent lors de la phase de conception et d'implémentation pour favoriser la réutilisation. Bien qu'ils soient rigoureusement décrits selon un canevas, ceux-ci souffrent d'un manque de sémantique associée au code et aux diagrammes UML fournis. Ce qui complique la tâche de leur réutilisation. Une telle tâche englobe le choix du patron adapté au problème en question et son intégration. Quant aux patrons de spécification formelle, ils sont proposés dans une phase amont dans le développement. Des outils permettant l'automatisation de certaines parties du développement des spécifications grâce à ces patrons ont été mis en place. Ces modèles sont très spécifiques au domaine et ne seront pas réutilisés en dehors de ce domaine particulier.

42. Outil disponible à l'adresse : `http://wiki.event-b.org/index.php/Pattern`

Chapitre 3

Notre approche

Sommaire

Dans ce chapitre, nous présentons notre approche de gestion des exigences, de leur spécification formelle et des liens entre eux. Nous définissons le système < CdC, Liens, Spec > et décrivons son évolution tout au long du processus de développement. Notre démarche utilise les concepts du génie logiciel tels que la généricité et les approches de réutilisation de l'existant.

3.1 Vue d'ensemble

La construction des systèmes et logiciels dans un cycle en V [Forsberg and Mooz, 1991] passe par les phases suivantes :

1. *Analyse et expression des besoins* durant laquelle les besoins du client sont recueillis et gérés dans un cahier des charges.
2. *Spécification* au cours de laquelle ce que fait le futur système est décrit. Il s'agit de décrire "le quoi ?"
3. *Conception générale*, appelée aussi conception architecturale, qui consiste à élaborer une vue d'ensemble du système en le spécifiant sous forme de modules et de communications entre eux.
4. *Conception détaillée* durant laquelle les détails de chaque module sont introduits en précisant les types de données utilisées ainsi que les opérations appliquées sur ces données.

5. *Codage* ou implantation des modules en utilisant un langage de programmation.
6. *Tests unitaires* ayant pour objectif de tester chaque unité du programme développé.
7. *Test d'intégration* au cours duquel l'assemblage des modules ou des programmes développés est testé pour vérifier leur coopération et leur fonctionnement global.
8. *Test de validation*, appelé encore test fonctionnel, permettant de tester le système développé.
9. *Test d'acceptation*, ou recette, visant à assurer que le système final est conforme aux besoins du client.

Notre travail s'organise autour des deux premières phases du processus en V, voir figure 3.1. Nous nous intéressons à la gestion et l'évolution en permanence des exigences et de leur spécification formelle.

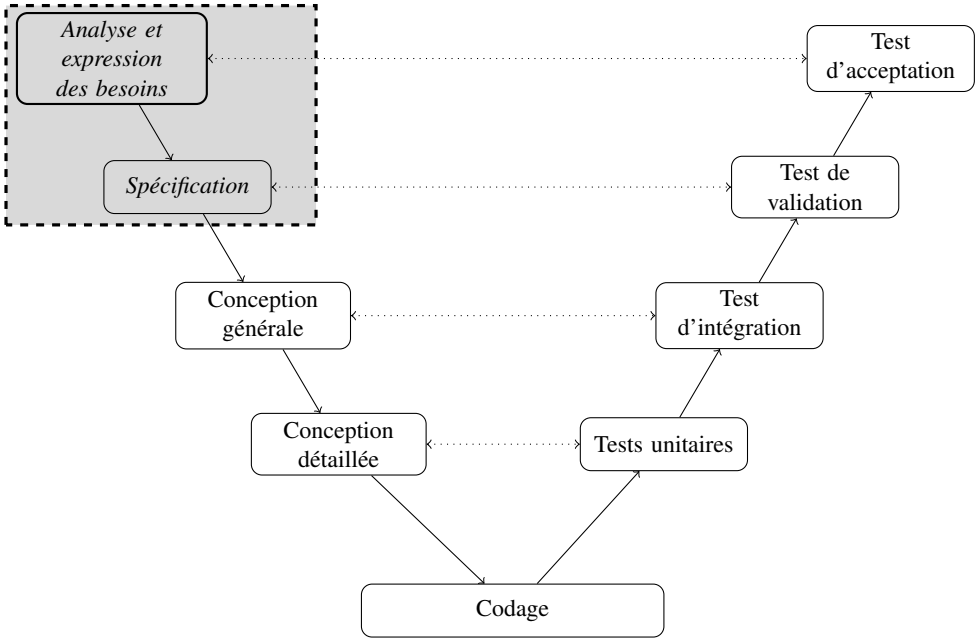


FIGURE 3.1 – Place de notre approche dans un processus en V

Notre approche, présentée dans la figure 1.2 du chapitre introductif de ce mémoire, fait intervenir deux mondes : celui du semi-formel - décrit par un CdC regroupant les besoins réécrits du client - et le formel présenté par la spécification appelée Spec dans la suite. La différence majeure entre notre approche et les deux premières phases du cycle en V est que nous considérons que ces deux étapes (ainsi que leurs documents correspondants) évoluent en même temps. Notre approche est décrite selon deux parties de nature différente qui se complètent :

- une partie *statique* représentant l'état actuel du système et composée des éléments CdC, Liens et Spec et
- une partie *dynamique* qui concerne les décisions de développement et les traitements. Celle-ci s'intéresse à l'évolution du système pour l'obtention d'un nouvel état décrit par un système < CdC′, Liens′, Spec′ >.

3.1.1 Constituants

3.1.1.1 CdC

Nous faisons la différence entre le cahier des charges du client, lorsqu'il existe, et celui que nous construisons et appelons *CdC* :

- le premier document concerne les besoins décrits par le client. Il n'est pas modifiable dans notre approche. Nous l'utilisons en consultation pour comprendre le futur système. En se référant à l'approche de Jean-Raymond Abrial [Abrial, 2010], ce document est considéré comme un texte *explicatif* et
- le deuxième document, CdC, est construit d'après notre compréhension des besoins du client. C'est le document de référence pour toutes les étapes de notre approche. Il résulte d'une étape de réécriture du cahier des charges du client sous forme d'une suite de phrases courtes, hiérarchisées, typées et étiquetées. Une phrase peut contenir des termes formels intégrés dans son texte informel. Ces termes proviennent de la Spec associée au CdC. Ce dernier évolue tout au long du processus de développement. La forme de ce document adoptée dans notre approche est décrite dans le paragraphe 3.2.2.

3.1.1.2 Spec

Désignant la spécification formelle, elle constitue le deuxième composant de notre approche. Nous avons choisi de travailler en grande partie de notre approche en utilisant la méthode formelle B événementiel pour développer cette Spec. Ce choix est justifié par :

- la facilité du développement des modèles et systèmes utilisant la technique de raffinement : un système complexe est construit progressivement en introduisant ses détails pas à pas,
- l'aspect rigoureux du développement avec cette méthode : chaque modèle développé doit être vérifié. Le passage de ce modèle au modèle raffiné nécessite la vérification de la correction du raffinement,
- l'existence des outils sous Rodin permettant l'automatisation de certaines étapes comme la vérification, la validation et la gestion des Liens entre le CdC et la Spec et
- la possibilité de communiquer entre les besoins et la Spec via les plugins de Rodin tels que ProR et ProB. Ceci réduit l'écart entre les deux mondes. L'évolution de la Spec est couplée avec celle du CdC et des Liens entre eux.

Nous montrons dans le chapitre 7 que notre approche est applicable en utilisant la méthode formelle TLA^+.

3.1.1.3 Liens

Ils sont établis lorsque nous commençons la construction de la Spec. Ils sont mis en place dans les deux sens : de la Spec vers le CdC et inversement. Grâce aux outils existants comme ProR, ces liens sont gérés, automatiquement mis à jour et propagés tout au long du développement. Ils sont présentés par un *glossaire* décrit comme suit :

Formal term	*Informal description*
...	...

Ce glossaire est constitué de couples *(Formal term, Informal description)* dans lesquels le premier élément représente le terme formel issu de la Spec et le second décrit la définition informelle correspondante à ce terme dans le CdC. Sur le plan pratique, sous l'outil ProR, ces liens se matérialisent par :

- Des termes formels venant de la Spec et introduits entre crochets [ ] dans les phrases du CdC. Un [terme_formel] exprime une relation de traçabilité entre les deux mondes formel et semi-formel. Son introduction se présente comme suit :

ID	*Description*	*Link*
Req_ID	Ceci est un *[terme_formel]* introduit dans la description informelle d'une phrase du CdC	...

FIGURE 3.2 – Introduction d'un terme formel dans le CdC sous ProR

- Le contenu de l'élément *Link*, décrit dans le chapitre 2 section 2.3.2.2. L'introduction d'un terme formel dans le CdC se réalise par la création d'un lien sous forme d'une adresse reliant ce terme formel de la Spec avec la phrase correspondante du CdC.

3.1.2 Outils

Notre approche commence par une étape de compréhension de la demande du client exprimée à partir de son cahier des charges. Si ce document n'est pas assez clair, nous enrichissons sa documentation par l'utilisation des images et des vidéos explicatives. Rodin, plateforme utilisant Eclipse, est dotée de plusieurs outils et plugins favorisant la réécriture des besoins, leur formalisation en B événementiel, la vérification, la validation et la visualisation du système < CdC, Liens, Spec >. Ces outils, utilisés à tout moment du développement, sont :

- ProR pour la gestion du CdC et des Liens. Outre la structure hiérarchique des exigences qu'il offre, cet outil permet d'établir des liens entre les exigences du CdC et les éléments formels de la Spec.
- Des générateurs d'obligations de preuve OPs permettant de générer l'ensemble des preuves relatives à la Spec et présentées sous forme de séquents. La structure d'un séquent est décrite dans le paragraphe *Vérification* de la section 2.2.3.3 du chapitre 2.
- Des prouveurs automatiques et interactifs pour vérifier la correction mathématique de la Spec en cours de construction en prouvant les OPs générées.
- Des outils comme ProB, JeB et AnimB pour la validation de la Spec relativement au CdC.
- Des outils de visualisation graphique donnant une vue d'ensemble des contextes et machines de la Spec comme l'outil Project Diagram [43] ou décrivant l'état de la Spec comme l'outil EventB StateMachines [44].

3.2 Description détaillée

Notre démarche est basée sur deux concepts fondamentaux :

43. http://wiki.event-b.org/index.php/Project_Diagram
44. http://wiki.event-b.org/index.php/Event-B_Statemachines

- le *raisonnement* via des prises de décisions et des choix de développement et
- l'*enrichissement* de l'état du système à travers des traitements qui tiennent en compte des décisions.

Nous initialisons le développement du système < CdC, Liens, Spec > par une étape de structuration des besoins du client dans un CdC. Au fur et à mesure de la réalisation de cette étape, nous enrichissons le CdC par l'introduction de paramètres. Ceci prépare à l'évolution du système dans les étapes de développement.

3.2.1 Structuration du cahier des charges du client

La structuration ou réécriture [Abrial, 2010], [Su et al., 2011] des exigences du client a pour objectif l'obtention d'un document, CdC, lisible et accessible par toutes ses parties prenantes. Les besoins de ce document sont représentés par des phrases courtes, hiérarchisées et accessibles via leurs identifiants uniques. Un identifiant est attribué à chaque phrase selon son objectif, voir les recommandations d'Abrial dans le chapitre 2 :

- FUN pour les phrases décrivant un aspect fonctionnel,
- EQU pour celles décrivant des équipements du futur système,
- ...

En utilisant l'outil ProR, nous utilisons la forme suivante du CdC :

ID	*Description*	*Link*
...	...	...

La diversité des identifiants attribués aux phrases du CdC est un premier pas vers la prise en compte de la nature *hybride* des systèmes complexes étudiés. Ce CdC rassemble des besoins décrivant la partie matérielle, ceux présentant la partie logicielle, ceux exprimant la communication entre les deux parties et ceux décrivant les échanges avec le client.

Suite à l'étape de réécriture des besoins, nous obtenons l'état initial de notre système décrit dans la figure 3.3 par :

- CdC. Il est constitué des besoins réécrits du client.
- Spec. Elle n'existe pas à cette étape.
- Liens. Les liens ne sont pas initialisés entre le CdC et la Spec.

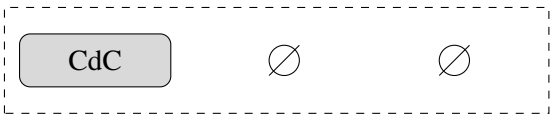


FIGURE 3.3 – Première étape dans le développement

Étude de cas

Prenons le système de train d'atterrissage d'un avion, voir annexe A. Commençons par la partie suivante de son cahier des charges décrite dans l'introduction de [Boniol and Wiels, 2014], voir figure 3.4 :

> *"The landing system is in charge of maneuvering landing gears and associated doors. The landing system is composed of 3 landing sets : front, left and right. Each landing set contains a door, a landing-gear and associated hydraulic cylinders...*
> *In nominal mode, the landing sequence is : open the doors of the landing gear boxes, extend the landing gears and close the doors."*

FIGURE 3.4 – Une partie du cahier des charges du train d'atterrissage

Avant la réécriture des besoins, notre système < CdC, Liens, Spec > est décrit par :

- un CdC vide vu que nous n'avons pas commencé l'étape de réécriture. Rappelons que ce document regroupe les besoins réécrits du cahier des charges du client,
- pas de Spec et
- pas de Liens.

Nous commençons la réécriture progressive des besoins sous ProR :

(a) Nous réécrivons la première phrase de la figure 3.4 et l'introduisons dans la colonne *Description* comme suit :

ID	***Description***	*Link*
	The landing system goal is maneuvering gears and their associated doors	

(b) Nous affectons l'identifiant *FUN-G* à la phrase réécrite. Le choix de la nature "FUN" de cet identifiant est justifié par le fait que cette phrase décrit un aspect fonctionnel du système : la manœuvre des trains et de leurs portes associées. La lettre "G" est choisie pour dire "Globale".

ID	*Description*	*Link*
FUN-G	The landing system goal is maneuvering gears and their associated doors	

Jusqu'à présent, l'élément *Link* est vide. Notre CdC contient uniquement l'exigence FUN-G et il n'y a pas de lien inter-exigences. Le développement de la Spec n'a pas commencé et il n'y a pas de liens entre CdC et Spec.

(c) Nous continuons l'étape de structuration des besoins de la figure 3.4. Nous obtenons les exigences réécrites *EQU-1*, *EQU-1-1* et *LS* de la figure 3.5. Au fur et à mesure de la réalisation de cette étape, nous déduisons de nouvelles exigences. Celles-ci proviennent de notre propre compréhension du cahier des charges. Elles concernent les exigences *FUN-1*, *FUN-2* et *FUN-3* de la même figure. Des liens de parenté ou de hiérarchisation entre les besoins sont créés. Par exemple, le besoin EQU-1-1 est un enfant d'EQU-1, il entre dans les détails des trois équipements "landing set". Les besoins structurés pour cette étude de cas sont décrits dans l'annexe A.

ID	*Description*	*Link*
FUN-G	The landing system goal is maneuvering gears and their associated doors	
EQU-1	The landing system is composed of three sets : front, left and right	
EQU-1-1	Each landing set contains : a door, a landing gear and their associated hydraulic cylinders	
FUN-1	Maneuvering gears consists on extending or retracting them or reversing their movement	
FUN-2	Maneuvering doors consists on opening or closing them	
FUN-3	The doors must be open when extending or retracting gears	
LS	In nominal mode, the landing sequence is : open the doors –> extend the landing gears –> close the doors	

FIGURE 3.5 – Des besoins structurés du CdC

3.2.2 Préparation du développement

Notre objectif est de préparer le CdC au fur et à mesure de sa structuration pour la suite des étapes de développement. Pour y parvenir, nous enrichissons la structure de ce document sous ProR avec de nouveaux paramètres. Nous souhaitons :

- faciliter le passage entre le CdC et sa Spec au cours du développement,
- préparer à la validation du futur modèle formel et valider ce modèle au fur et à mesure de sa construction et
- aider à l'évolution du CdC relativement à sa Spec. En effet, nous exploitons les retours des outils de vérification et de validation pour détecter des lacunes dans les besoins. Ces lacunes concernent des oublis, des imprécisions et des incohérences. Les détails de ce point sont donnés dans le chapitre 4. Grâce à la structure enrichie du CdC, la localisation des exigences à mettre à jour est plus facile.

La structure enrichie du CdC est présentée dans la figure 3.6. Elle est décrite par les paramètres :

ID	*Description*	*Link*	***Term_Type***	***Terms***	***Event-B Model***
...	...	...	...	...	...

FIGURE 3.6 – Structure du CdC avec ses paramètres

Term_Type.

Les phrases du CdC sont typées. Nous nous basons sur la nature des éléments de validation que peuvent contenir ces phrases, notamment :

- les données du futur système typées en Fact,
- les fonctionnalités attendues typées Functionality,
- les conditions, de type Obligation, avec lesquelles le système fonctionnera sous forme de préconditions et post-conditions [Hoare, 1969] et
- les comportements, de type Behavior, définis à l'aide des fonctions existantes.

Terms.

Selon le *Term_Type* de chaque besoin, nous précisons et extrayons les éléments destinés à la validation du futur modèle formel depuis ce besoin. Ces éléments sont mémorisés dans la colonne *Terms*.

Event-B Model.

Un lien entre le besoin et le nom du modèle B événementiel correspondant est introduit via des machines et des contextes. Il est présenté physiquement dans le champ *Event-B Model*. Dans un système composé de plusieurs modèles formels, ce paramètre facilitera l'accès aux éléments de validation dans le CdC liés au modèle correspondant.

Le CdC est décrit formellement, voir figure 3.7. Il est défini par une séquence de phrases. Une phrase est un produit cartésien, CP, des champs décrits dans la figure 3.6. Les opérations appliquées sur cette forme sont celles des types utilisés, suites, ensembles et produit cartésien. Les champs entre [ ] tels que "Order" sont optionnels. Le glossaire fait le lien entre un terme formel et sa/ses présentation(s) informelle(s) dans les besoins. Il est composé de deux parties : la première est appelée *Formal term* et contient les termes formels et la deuxième contient la définition informelle de chaque terme formel et nommée *Informel description*.

```
CdC is Sequence (Sentence)
Sentence is CP [ "ID :" TEXT,
                 ["Order :" INTEGER,]
                 "Description :" Set (Term),
                 "Link :" Set(Term),
                 [Sequence (Sentence),]
                 [Set (Term_Type)],
                 "Terms :" Set(Term),
                 "Event-B Model :" Set(Term) ]

Term is Informal description | Formal term
Term_Type is Fact | Functionality | Behavior | Obligation
```

FIGURE 3.7 – Une représentation du CdC

3.2.3 Évolution du système

Elle concerne le système < CdC, Liens, Spec > et est guidée par :

- *les besoins,* en ajoutant, supprimant ou modifiant des besoins,
- *les éléments formels,* en ajoutant, renommant, supprimant ou raffinant des éléments de la Spec et
- *les liens,* en mettant à jour le glossaire.

Une étape du développement est décrite par les quatre éléments suivants :

- *État de départ*, représentant l'état actuel du système décrit par le système < CdC, Liens, Spec >.
- *Décision*, désignant les choix pour avancer dans le développement. Ceux-ci concernent le CdC, la Spec ou les deux à la fois.
- *Traitement*, décrivant les actions exécutées suite à la prise de décision. Le traitement concerne et modifie l'état de départ.
- *État résultant*, montrant le nouvel état du système après traitement. Cet état est noté < CdC′, Liens′, Spec′ >.

NB. La prise de décisions pour un besoin se traduit par la mise à jour de ce besoin et celle des autres besoins, voir l'exemple fourni dans le paragraphe 3.2.3.1.2.

3.2.3.1 Développement guidé par les besoins

3.2.3.1.1 Présentation des cas. Il s'agit de décrire l'évolution du système < CdC, Liens, Spec > en partant des modifications apportées au CdC. Nous allons traiter deux cas de développement :

- Aucune Spec n'existe. Nous choisissons un ou plusieurs besoins à partir du CdC en vue de les formaliser en B événementiel.

- Une Spec existe. Nous mettons à jour notre système existant en partant du CdC.

3.2.3.1.2 Initialisation du développement de la Spec. Nous partons des exigences de la figure 3.5. Nous choisissons le besoin FUN-G à formaliser en B événementiel.

- *État de départ :* le système <CdC, $\varnothing$, $\varnothing$>.

- *Décision :* formaliser le besoin FUN-G.

- *Traitement :* le développement est initialisé par la création d'une machine en B événementiel appelée *Landing_System* qui utilise (Sees) un contexte *Landing_Ctx*. Cette machine est vide.

- *État résultant :* le système < CdC′, Liens′, Spec′ > est décrit par la figure 3.8 :

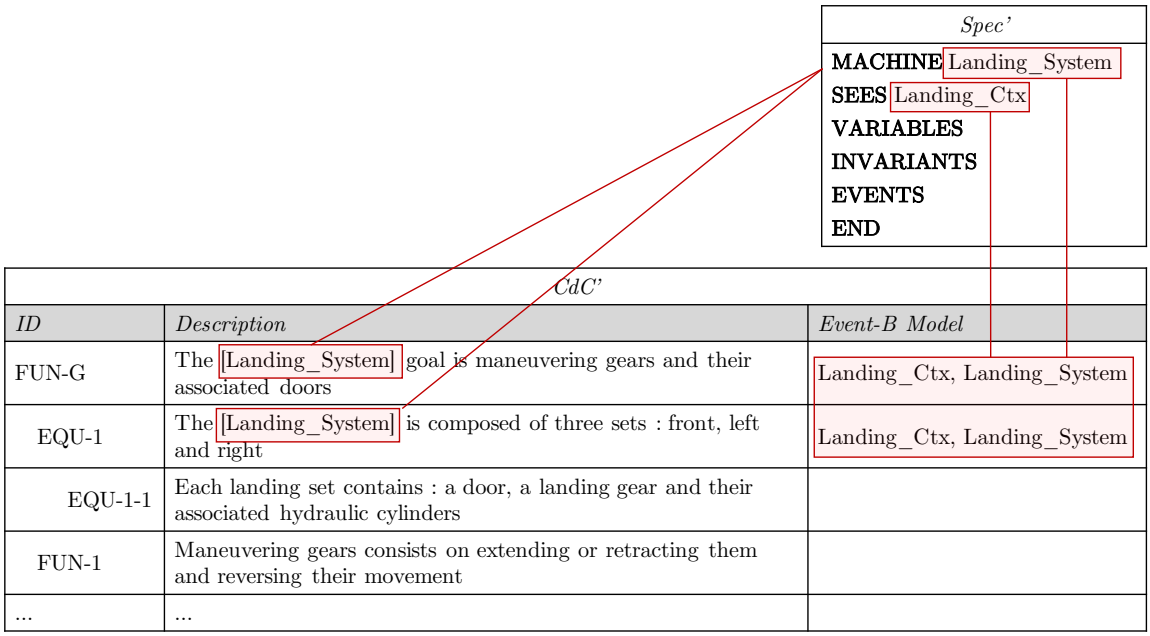
  — *Spec′*. Elle est caractérisée par sa machine et son contexte.

  — *CdC′*. Le terme formel Landing_System est introduit entre [ ] dans le texte informel du besoin FUN-G. Le besoin EQU-1 est aussi affecté par cet ajout. Le champ *Event-B_Model* est enrichi par les deux termes Landing_System et Landing_Ctx, noms respectifs de la machine et de son contexte.

  — *Liens′*. Le glossaire est initialisé par le couple *(Landing_System, landing system)* :

Formal term	*Informal description*
Landing_System	landing system

  Avec l'outil ProR, deux liens sont automatiquement créés entre les besoins FUN-G et EQU-1 et la machine Landing_System. L'élément *Link* est mis à jour par l'ajout de ces deux liens sous forme d'adresses ou chemins d'accès à la machine Landing_System et à son contexte dans l'espace de travail sous Rodin. Pour des raisons de lisibilité, nous ne représentons pas ce champ dans les figures de ce chapitre.



CdC'

ID	*Description*	*Event-B Model*
FUN-G	The [Landing_System] goal is maneuvering gears and their associated doors	Landing_Ctx, Landing_System
EQU-1	The [Landing_System] is composed of three sets : front, left and right	Landing_Ctx, Landing_System
EQU-1-1	Each landing set contains : a door, a landing gear and their associated hydraulic cylinders	
FUN-1	Maneuvering gears consists on extending or retracting them and reversing their movement	
...	...	

FIGURE 3.8 – Introduction d'une machine et son contexte

3.2.3.1.3 Mise à jour du système. Notre choix concerne la formalisation d'un terme de l'exigence FUN-G.

- *État de départ :* le système < CdC′, Liens′, Spec′ > résultant décrit dans le paragraphe précédent. Nous le renommons en < CdC, Liens, Spec >.
- *Décision :* formaliser le terme *gears* du besoin FUN-G.
- *Traitement :* le développement de la Spec est mis à jour par la création d'une variable appelée *gears_pos*. Le type *G_Positions* de cette variable est déclaré dans le contexte *Landing_Ctx* de la figure 3.9. Cette variable modélise la position du train d'atterrissage. Elle a deux valeurs distinctes :
  - — train étendu, décrit par la constante *g_extended* ou
  - — train plié, exprimé par la constante *g_retracted*.

CONTEXT *Landing_Ctx*
SETS
G_Positions
CONSTANTS
g_extended
g_retracted
AXIOMS
axm_2 : partition (G_Positions, {*g_extended*}, {*g_retracted*}*)*
END

FIGURE 3.9 – Le contexte *Landing_Ctx*

La type de ces deux constantes est exprimé par la relation *partition* décrite dans l'axiome *axm_2* du contexte Landing_Ctx. Cette relation exprime le fait que les deux constantes constituent l'ensemble des positions (*G_Positions*).

- *État résultant :* le nouveau système < CdC′, Liens′, Spec′ > est décrit par la figure 3.10 :

Spec'

MACHINE Landing_System
SEES Landing_Ctx
VARIABLES
gears_pos
INVARIANTS
inv_Typ_gears_pos: gears_pos ∈ G_Positions
EVENTS
END

CdC'

ID	*Description*	*Event-B Model*
FUN-G	The [Landing_System] goal is maneuvering [gears_pos] and their associated doors	Landing_Ctx, Landing_System
EQU-1	The [Landing_System] is composed of three sets : front, left and right	Landing_Ctx, Landing_System
EQU-1-1	Each landing set contains : a door, a landing gear and their associated hydraulic cylinders	
FUN-1	Maneuvering [gears_pos] consists on extending or retracting them and reversing their movement	Landing_Ctx, Landing_System
...	...	

FIGURE 3.10 – Formalisation du terme *gears*

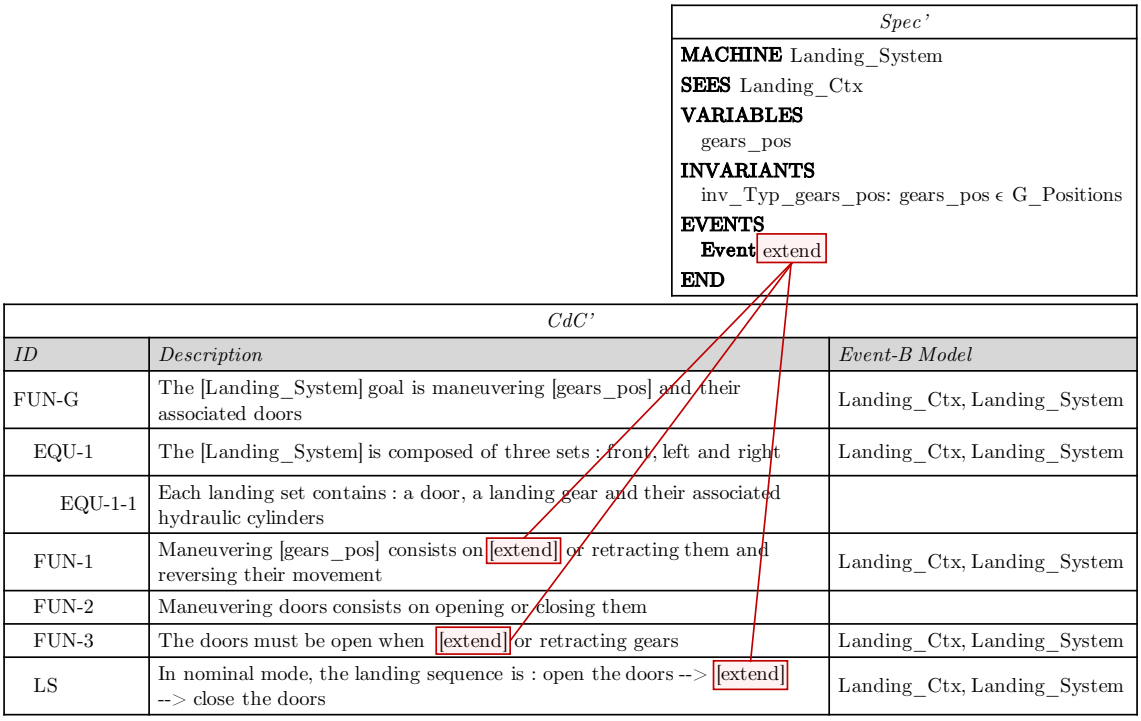
— *Spec′*. Elle est caractérisée par le contexte *Landing_Ctx* et la machine *Landing_System* augmentée par la nouvelle variable *gears_pos* et son typage décrit dans un invariant appelé *inv_Typ_gears_pos*.

— *CdC′*. Le terme formel gears_pos est introduit dans le texte informel des besoins FUN-G et FUN-1. Il remplace le terme informel *gears*. La colonne *Event-B Model* associée à l'exigence FUN-1 est enrichie par les termes Landing_System et Landing_Ctx, noms respectifs de la machine et de son contexte.

— *Liens′*. Le glossaire est mis à jour par introduction du couple *(gears_pos, gears)* comme suit :

Formal term	*Informal description*
Landing_System	landing system
gears_pos	gears

Nous avançons dans le développement en effectuant d'autres choix à partir du CdC notamment un choix de formaliser en B événementiel une fonctionnalité d'extension des trains. Nous obtenons un système décrit dans la figure 3.11 et par le glossaire de la figure 3.12.



Spec'

MACHINE Landing_System
SEES Landing_Ctx
VARIABLES
gears_pos
INVARIANTS
inv_Typ_gears_pos: gears_pos ∈ G_Positions
EVENTS
Event extend
END

CdC'

ID	*Description*	*Event-B Model*
FUN-G	The [Landing_System] goal is maneuvering [gears_pos] and their associated doors	Landing_Ctx, Landing_System
EQU-1	The [Landing_System] is composed of three sets : front, left and right	Landing_Ctx, Landing_System
EQU-1-1	Each landing set contains : a door, a landing gear and their associated hydraulic cylinders	
FUN-1	Maneuvering [gears_pos] consists on [extend] or retracting them and reversing their movement	Landing_Ctx, Landing_System
FUN-2	Maneuvering doors consists on opening or closing them	
FUN-3	The doors must be open when [extend] or retracting gears	Landing_Ctx, Landing_System
LS	In nominal mode, the landing sequence is : open the doors --> [extend] --> close the doors	Landing_Ctx, Landing_System

FIGURE 3.11 – Un état du système < CdC, Liens, Spec >

Formal term	*Informal description*
Landing_System	landing system
gears_pos	gears
extend	extending
extend	extend the landing gears

FIGURE 3.12 – Glossaire mis à jour

Dans cet état du système, l'événement *extend* provient du choix des exigences FUN-1, FUN-3 et LS. Les détails de cet événement sont définis selon notre propre compréhension. Ils ne sont pas explicitement décrits dans le texte de nos exigences du CdC.

3.2.3.2 Développement guidé par la Spec

Ce type de développement s'intéresse à l'effet des modifications de la Spec dans le système < CdC, Liens, Spec >. Nous distinguons trois cas :

Cas 1. Introduction d'un nouvel élément formel. Nous partons de la Spec existante dans laquelle nous introduisons un élément formel tel qu'une variable, une constante, un ensemble porteur, un invariant, un théorème, un axiome ou un événement. L'introduction de cet élément entraîne une évolution du système < CdC, Liens, Spec >. Nous distinguons deux sous-cas :

Cas1.1. Introduction d'un élément en provenance du CdC. Le nouvel état du système est décrit par :

- *Spec′*. C'est la Spec augmentée par le nouvel élément formel. Si cet élément est une variable ou une constante, Spec' contiendra un ou plusieurs invariants et/ou axiomes pour le typage de cet élément.
- *CdC′*. Le CdC est mis à jour par introduction du nouveau [terme_formel] venant de la Spec' dans le texte du ou des besoins concernés. Ce terme remplace des termes informels.
- *Liens′*. Le glossaire est mis à jour par le couple *(terme_formel, description informelle correspondante)*.

Cas 1.2. Introduction d'un élément relatif à l'évolution de la Spec. Le système < CdC, Liens, Spec > évolue par la mise à jour de la Spec. Les deux autres constituants restent inchangés. Par exemple, nous introduisons de nouveaux ensembles porteurs dans un contexte en B événementiel pour définir de nouveaux types des variables.

Cas 2. Suppression ou renommage d'un élément formel. Le renommage s'effectue via une option automatique appelée *refactory* sous Rodin. L'évolution est guidée par les deux sous-cas :

Cas 2.1. Élément de la Spec dont la définition existe dans le glossaire. Il s'agit d'un élément issu de la Spec. L'état résultant de ce choix de développement est décrit par :

- *CdC′*. Le [terme_formel] concerné est supprimé du CdC. Il est remplacé par son ancienne définition formelle existante dans le glossaire. Ceci se réalise en exploitant les Liens permettant de naviguer facilement et d'effectuer les changements souhaités. S'il s'agit d'un renommage, ce [terme_formel] est renommé dans le CdC. La colonne *Event-B Model* est mise à jour par suppression du nom de la Spec s'il s'agit d'une suppression d'un élément formel. Autrement, elle reste inchangée.
- *Spec′*. Elle est mise à jour par suppression ou renommage de l'élément concerné.
- *Liens′*. Le glossaire est mis à jour par :
  - — suppression du couple (terme formel, description informelle) en cas de suppression de l'élément formel de la Spec,
  - — renommage du couple (terme formel, définition informelle) en remplaçant *terme formel* par son *nouveau nom* attribué.

Cas 2.2. Élément de la Spec non existant dans le glossaire. Il s'agit d'un élément relatif à l'évolution du développement de la Spec. Sa suppression ou sa modification n'apporte pas de changement ni dans le CdC ni dans les Liens. Le système < CdC, Liens, Spec > évolue uniquement par modification de la Spec.

Cas 3. Raffinement d'une Spec existante. L'idée consiste à partir de la Spec, la raffiner et noter l'apport de ce raffinement au système < CdC, Liens, Spec >.

Exemple. Nous utilisons le **Cas 2**. Nous choisissons de renommer l'événement *extend* de la figure 3.11 :

- *État de départ :* le système décrit dans la figure 3.11 et le glossaire de la figure 3.12.

- *Décision :* renommer l'événement *extend* de la machine *Landing_System* en *extend_gears*.
- *Traitement :* la Spec est mise à jour par modification du nom de l'événement *extend*.
- *État résultant :* le nouveau système < CdC′, Liens′, Spec′ > est décrit par la figure 3.13 dans lequel :

Spec'

MACHINE Landing_System
SEES Landing_Ctx
VARIABLES
gears_pos
INVARIANTS
inv_Typ_gears_pos: gears_pos ∈ G_Positions
EVENTS
Event extend_gears
END

CdC'

ID	*Description*	*Event-B Model*
FUN-G	The [Landing_System] goal is maneuvering [gears_pos] and their associated doors	Landing_Ctx, Landing_System
EQU-1	The [Landing_System] is composed of three sets : front, left and right	Landing_Ctx, Landing_System
EQU-1-1	Each landing set contains : a door, a landing gear and their associated hydraulic cylinders	
FUN-1	Maneuvering [gears_pos] consists on [extend_gears] or retracting them and reversing their movement	Landing_Ctx, Landing_System
FUN-2	Maneuvering doors consists on opening or closing them	
FUN-3	The doors must be open when [extend_gears] or retracting gears	Landing_Ctx, Landing_System
LS	In nominal mode, the landing sequence is : open the doors --> [extend_gears] --> close the doors	Landing_Ctx, Landing_System

FIGURE 3.13 – Renommage d'un événement

— *Spec′*. Elle est décrite par le contexte *Landing_Ctx* et la machine *Landing_System*. Cette machine est caractérisée par l'événement *extend_gears*, nouveau nom de l'événement *extend*.

— *CdC′*. Le terme formel [extend] des besoins FUN-1, FUN-3 et LS est remplacé par le terme [extend_gears].

— *Liens′*. Le glossaire est mis à jour par modification des couples *(extend, extend the landing gears)* et *(extend, extending)*, voir figure 3.14.

Formal term	*Informal description*
Landing_System	landing system
gears_pos	gears
extend_gears	extending
extend_gears	extend the landing gears

FIGURE 3.14 – Renommage des éléments du glossaire

3.2.3.3 Développement guidé par les Liens

Ce type de développement se répercute par l'ajout, la suppression ou la mise à jour automatique des Liens. Un atout majeur de ce type est son aspect automatisable grâce à l'utilisation des outils : au cours de l'évolution des composantes CdC et Spec, des Liens se créent automatiquement entre eux en permanence permettant la liaison et l'accès dans les deux sens aux éléments des deux mondes. Ceux-ci sont présentés par le glossaire.

Afin d'assurer la trace des besoins dans le formel, nous introduisons :

- des commentaires dans la Spec. Ces commentaires font référence aux exigences du CdC depuis lesquelles les éléments formels sont élaborés et
- des noms pour les éléments formels de type invariant, garde, axiomes ou théorème. Ces noms modélisent les identifiants des besoins formalisés.

Exemple.

Spec

MACHINE Landing_System
VARIABLES
gears_pos *// FUN-G*
...
EVENTS
Event extend_gears
when
grd-FUN-3 : doors_pos = open
then
...
end
END

CdC

ID	*Description*	*Event-B Model*
FUN-G	The [Landing_System] goal is maneuvering [gears_pos] and their associated doors	Landing_Ctx, Landing_System
...	...	...
FUN-3	The [doors_pos] must be [open] when [extend_gears] or retracting gears	Landing_Ctx, Landing_System

FIGURE 3.15 – Introduction d'un commentaire et nomination d'une garde dans la Spec

Dans la figure 3.15, nous avons introduit le commentaire *"//FUN-G"* pour la variable gears_pos indiquant que cette variable provient du besoin FUN-G. Nous avons également attribué le nom *"grd-FUN-3"* à la garde de l'événement extend_gears pour mentionner que cette garde provient de l'exigence FUN-3.

3.3 Système < CdC, Liens, Spec >

Notre approche a pour objectif de réduire l'écart entre le monde des exigences et celui du formel. Cet objectif est guidé par la nécessité de comprendre les besoins du client : nous commençons par

une première compréhension des besoins. Les ambiguïtés se précisent au fur et à mesure du développement de la Spec. Le logiciel résultant devra répondre à la demande du client. Avoir compris ce document et le laisser de côté ne suffira pas, à notre avis, pour continuer la suite des étapes de développement car : (i) nous pouvons oublier des besoins ou des détails importants, (ii) notre compréhension s'affine et peut être différente de celle de départ et (iii) nous avons besoin de savoir quelles exigences ont été prises en compte relativement au CdC. De même, la trace des exigences sur la Spec en cours de construction demeure importante pour les parties prenantes de ces documents au sein du projet.

Notre expérience durant cette thèse montre que les deux mondes, une fois liés l'un à l'autre, interagissent entre eux : l'évolution d'un monde induit celle de l'autre et nous ne parlons plus de l'évolution de chaque monde tout seul. Nous montrons dans les chapitres ultérieurs comment le formel aide, via ses retours, à détecter des lacunes dans les besoins du CdC. Ceci peut se produire à tout moment du développement de la Spec : dans sa construction, pendant sa vérification et durant sa validation. Pour explorer cette idée, la présence et l'utilisation des outils est indispensable.

3.3.1 Utilisation des outils

La construction progressive de la Spec permet de détecter des lacunes dans le CdC. Celles-ci concernent des oublis, des exigences implicites, des besoins évidents non mentionnés ou des contradictions. Le problème des exigences implicites ou évidentes et non décrites a été mentionné dans [Abrial, 2010]. Il concerne en général les objectifs globaux attendus dans le futur système. Par exemple, un système de contrôle d'accès aux bâtiments gère les autorisations d'accès aux bâtiments, mais cette contrainte paraît tellement évidente pour le client qu'il ne la cite pas dans son cahier des charges. Les outils disponibles dans la plateforme Rodin jouent un rôle important dans la gestion, l'évolution et la détection des lacunes dans les éléments du système < CdC, Liens, Spec >. Comme mentionné dans le paragraphe précédent, la détection de ces lacunes se déroule en permanence et à tout moment du développement :

3.3.1.1 Construction de la Spec

Au fur et à mesure de cette construction, des ambiguïtés ou des imprécisions peuvent être détectées dans les besoins du CdC. Par exemple, les actions décrivant la navigation entre les états du système ne sont pas souvent citées dans les besoins du client.

3.3.1.2 Vérification

Elle permet de détecter des anomalies aussi bien dans la Spec que dans le CdC. Ces anomalies se répercutent par des contradictions dans les besoins. Ce sont les obligations de preuve non déchargées qui donnent une indication sur un problème et son origine. Cette indication mène à une révision de la Spec ou du CdC. Nous nous basons sur l'analyse de ces deux cas pour développer notre contribution concernant la détection des imperfections dans les besoins à partir de l'activité de vérification. Notre raisonnement se base sur le fait que l'origine de la Spec concernée par la vérification est le CdC. Ce document peut contenir des incohérences ou anomalies qui mènent à l'échec de l'activité de preuve. Nous détaillons cette contribution dans le chapitre 4.

3.3.1.3 Validation

Nous considérons cette activité comme un processus rigoureux qui commence par une étape de préparation du CdC en y rajoutant de nouveaux paramètres. Ces paramètres permettent de recueillir des informations relatives à la validation du futur système. Grâce à la validation, nous détectons des lacunes dans le CdC. Par exemple, l'état initial est souvent non cité dans les énoncés des besoins. Nous détaillons cette contribution dans le chapitre 4.

3.3.2 Patrons de développement

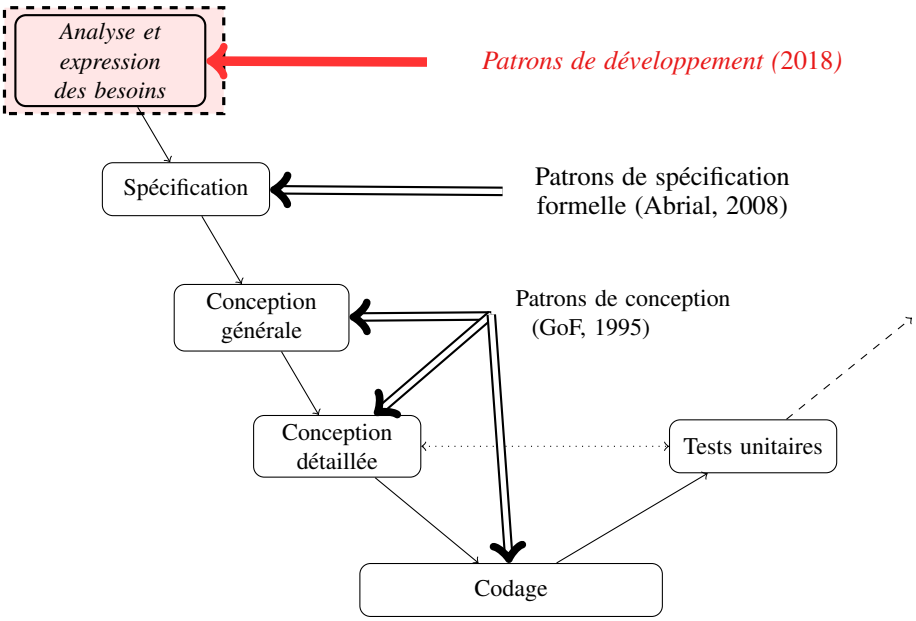


FIGURE 3.16 – Place de nos patrons dans un processus en V

Notre approche est basée sur le principe *"ne pas réinventer la roue"*. Les cahiers des charges utilisés dans nos études de cas ont une forme composée d'une partie *décrite par le client* et d'une autre partie *décrite par un informaticien*. La première partie contient une description détaillée du futur système sous forme de détails techniques, des images, tableaux ou paragraphes écrits en langage naturel. La deuxième partie est une vision informatique du système. Elle est constituée de phrases décrites en langage naturel sous une forme similaire pour plusieurs besoins. En se basant sur cette forme des besoins et en s'inspirant des idées des patrons de conception de GoF [Gamma et al., 1996] et des patrons de spécifications formelles [Abrial and Hoang, 2008], nous avons développé des *patrons de développement*. Un patron de développement utilise une fonction équivalente aux opérateurs mathématiques comme l'opérateur d'addition "$+$" ou de multiplication "$*$". Cette fonction agit sur des entrées dites *opérandes* et fournit un résultat. Dans le cas de nos patrons, les entrées sont des besoins et éventuellement un système existant et le résultat est un système < CdC, Liens, Spec >. Nous utilisons le terme "*développement*" conjointement avec le terme "*patron*" pour préciser que ce patron concerne aussi bien les besoins que leur spécification formelle. Il occupe une place en amont dans le processus de développement : celle de l'analyse et l'expression des besoins tel qu'il est montré par la figure

3.16. Durant cette phase, nous localisons des besoins ayant une ressemblance de forme. Nous distinguons trois formes possibles : *conditionnelle*, *séquentielle* et *ensembliste*. Les deux premières formes sont abordées respectivement dans les chapitres 6 et 7. La dernière forme est une perspective de cette thèse. Nos patrons génèrent automatiquement une partie de la spécification en cours de construction et précisent les éléments restant à définir par le développeur. Ils soulignent la notion de collage aussi bien dans la Spec que dans le CdC. Nous avons appliqué ces patrons sur deux études de cas : train d'atterrissage et machine d'hémodialyse.

3.3.3 Détection des oublis

L'évolution d'un système < CdC, Liens, Spec > par introduction d'un nouveau besoin donne un nouveau système < CdC', Liens', Spec' >. Ce choix donne une idée sur des lacunes pouvant exister dans le système de départ. Par exemple, nous partons d'un développement existant d'un distributeur automatique de billets pour rajouter un nouveau besoin décrivant la prise en compte des clients malvoyants. Cet ajout souligne un oubli pour la gestion d'affichage dans un système de départ développé antérieurement. Nous abordons cette idée dans le chapitre 5.

3.4 Conclusion

Ce chapitre expose notre approche de gestion du système < CdC, Liens, Spec >. Nous adoptons un développement basé sur la réutilisation de l'existant et l'utilisation des outils disponibles sous la plateforme Rodin. Le pont entre ces deux documents est établi et renforcé pas à pas avec la mise à jour des éléments du système. Nos contributions concernent la définition et l'utilisation des patrons de développement, la définition d'un processus de validation dès l'analyse des besoins et l'emploi des retours des activités de vérification et de validation pour la détection des lacunes dans le système < CdC, Liens, Spec >.

Un point important dans notre travail concerne l'étape d'analyse et de compréhension des besoins. Nous réalisons cette étape en se posant des questions et en utilisant des moyens à l'instar des images, des vidéos explicatives issues du domaine d'étude et des diagrammes d'états-transitions. Ces moyens offrent des visions différentes du problème en cours et enrichissent sa documentation.

Les limites de notre approche concernent les points suivants :

- le travail fastidieux réalisé lors de la réécriture des besoins du client. Il nécessite la compréhension de ces besoins et leur clarification,
- la gestion lourde du CdC, des Liens et de la Spec vue la complexité et les détails techniques de chaque composant et
- la localisation difficile de la source de l'échec de la preuve pendant l'activité de vérification : est-ce dans la Spec ? Est-ce dans le CdC ? Ou les deux ? Une automatisation des Liens sera efficace pour faciliter ce point.

Chapitre 4

Validation dans le développement

Sommaire

Le monde du formel, représenté par la Spec, interagit avec celui du semi-formel, décrit par les besoins. Cette interaction est rendue effective moyennant des activités permettant l'échange d'informations entre eux. La validation est établie comme une de ces activités permettant de répondre à la question "*la Spec répond-elle aux besoins du client ?* "

Dans ce chapitre, nous abordons cette question de la validation des modèles formels en B événementiel. Nous considérons cette activité comme un processus rigoureux, commençant dès la gestion des exigences, et la détaillons avec l'étude de cas d'un système de contrôle de train d'atterrissage d'un avion décrit en annexe A.

Les résultats décrits dans ce chapitre ont fait l'objet des publications [Sayar and Souquières, 2016] et [Sayar and Souquières, 2017].

4.1 Généralités

Le développement d'un système correct par construction est basé sur les trois points principaux :

(1) La construction est *progressive*. Ce point concerne l'utilisation de la technique de raffinement. Il est difficile de décrire un système complexe en une seule étape. L'idée consiste à partir d'un modèle abstrait et de lui ajouter pas-à-pas des détails jusqu'à ce que nous obtenons un modèle complexe. La description graduelle de la Spec aide à gérer sa complexité pendant le développement mais elle n'est pas suffisante pour garantir sa correction. Elle doit être accompagnée des preuves de consistance au fur et à mesure du développement. Ceci se réalise en abordant le point (2).

(2) Le modèle en cours de construction est *vérifié*. Ce point s'intéresse à la vérification formelle. Cette activité est réalisée via des preuves s'effectuant à chaque pas de raffinement. L'objectif est de démontrer que le modèle raffiné ne contredit pas le modèle abstrait. Afin de vérifier un modèle formel en B événementiel, un ensemble d'outils associés à la plateforme Rodin est mis à la disposition du développeur. Il s'agit des générateurs d'obligations de preuves et des prouveurs automatiques et interactifs. À nouveau, la preuve de consistance du modèle formel n'est pas suffisante. Elle doit être complétée par la validation : prouver qu'un système fonctionne sans erreur n'implique pas forcément que ce système correspond à celui demandé par le client dans son cahier des charges. C'est l'objectif du point (3).

(3) Le modèle en cours de construction est *validé*. La validation est accomplie au fur et à mesure de la construction du modèle formel et au plus tôt possible dans les premiers pas du processus de développement. Cette tâche concerne les deux mondes : le formel représenté par le modèle B événementiel est validé par rapport à l'informel représenté par les besoins. Plusieurs techniques et outils sont fournis au développeur afin de mener cette tâche. Ils ont été détaillés dans le chapitre 2 section 2.2.

Nous abordons le déroulement de la validation à tout moment dans le développement du système < CdC, Liens, Spec >. Nous l'étudions selon deux visions, voir figure 4.1 :

- dans un niveau d'abstraction pour montrer les détails de la démarche pas à pas et
- dans un niveau évolué du développement dans lequel nous montrons comment nous exploitons l'historique de validation des modèles précédents abstraits pour valider les modèles raffinés.

4.2 Structuration des besoins

Nous effectuons la validation à tout moment et tout au long du processus de développement. Cette activité est prise en compte dès la réécriture des besoins et concerne les futurs modèles formels en B événementiel par rapport aux exigences du client. Elle est guidée par l'évolution du système < CdC, Liens, Spec > et par des *éléments de validation*. Ces derniers sont initialement extraits du CdC par le développeur qui réécrit les besoins. Ils sont traduits progressivement en *éléments formels* dans la Spec et sont introduits dans le CdC.

Comme mentionné dans le chapitre 3, la structure du CdC est enrichie par de nouveaux paramètres au fur et à mesure de la réécriture des besoins du client. Ceci permet d'adapter ce document pour préparer à la validation de son futur modèle formel associé.

Notons que le document obtenu ainsi que les éléments de validation extraits doivent être accessibles par toutes les parties prenantes au sein du projet.

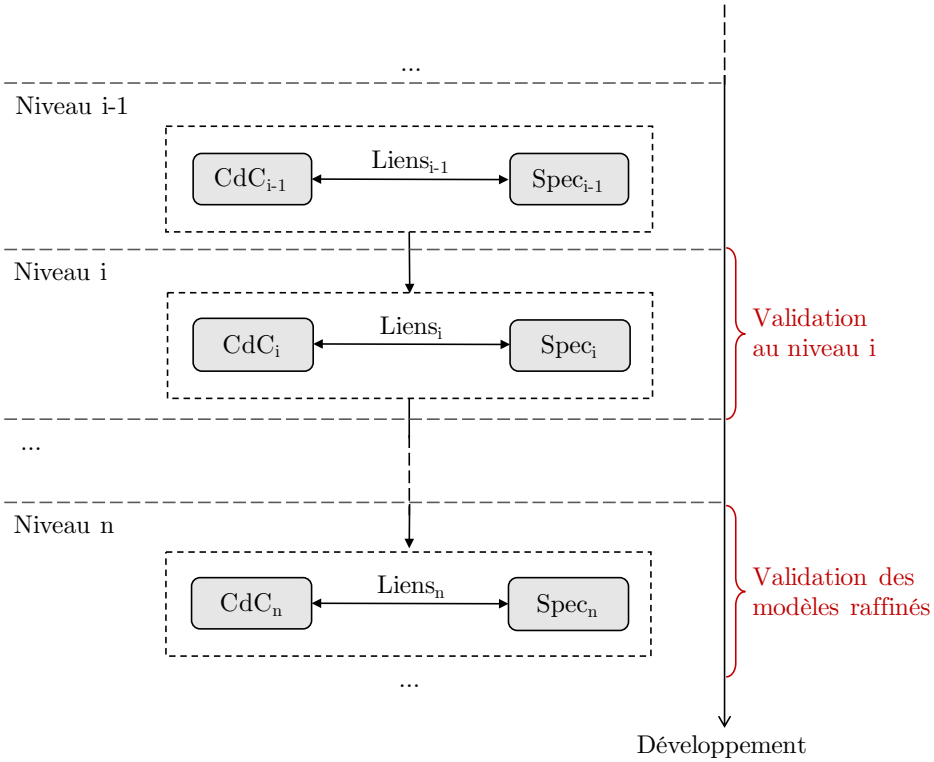


FIGURE 4.1 – Validation au cours du développement

4.2.1 Définition

Un *élément de validation* représente des termes formels ou informels selon lesquels la Spec va être validée. Cet élément de validation est typé et extrait du CdC. Il peut être :

- une donnée devant être présente dans le futur système, typée par un *Fact*,
- une fonctionnalité ou un service attendu typé *Functionality*,
- des conditions de fonctionnement du système de type *Obligation* ou
- un comportement décrit en termes de scénario de validation. Un scénario de validation décrit un enchaînement d'actions. Un comportement est typé *Behavior*.

Des exemples de ces éléments de validation sont fournis dans la section 4.2.2 suivante.

4.2.2 Paramètres du CdC sous ProR

La structure de base du CdC sous ProR est augmentée par de nouveaux paramètres, voir partie 3.2.2 du chapitre 3. Ceux-ci concernent :

Term_Type.

Il désigne le type d'un élément de validation contenu dans une phrase du CdC. Cet élément de validation ne peut avoir qu'un seul type à la fois parmi :

- Fact,

- Functionality,
- Obligation ou
- Behavior.

Une phrase du CdC peut contenir plusieurs éléments de validation de différents types. Ces éléments sont mémorisés dans *Terms*.

Terms.
Ce paramètre contient les éléments destinés à la validation du futur modèle formel depuis les besoins du CdC selon leurs *Term_Type*. Ces éléments seront mis à jour au fur et à mesure de l'évolution du système < CdC, Liens, Spec > par introduction de termes formels.

Event-B Model.
Il mémorise le nom du modèle formel correspondant à un besoin donné, c'est-à-dire la Spec prenant en compte ce besoin ou une partie de ce besoin. Cette Spec est exprimée sous forme de machines et de contextes en B événementiel.

Pour chaque besoin du CdC, les éléments de validation sont extraits et mémorisés selon leurs *Term_Type*. La sémantique de ces types est décrite par deux aspects complémentaires : *statique* et *dynamique*.

- Les deux premiers types d'éléments de validation, Fact et Functionality, concernent l'aspect *statique* de la Spec : valider la Spec relativement à ces deux types permet d'assurer que les données et services du futur système sont pris en compte et présents dans le développement.
- Les éléments de types Obligation et Behavior décrivent l'aspect *dynamique* de la Spec :
  - valider relativement aux obligations garantit que les éléments du futur système respectent les contraintes de fonctionnement. Dans le cas des systèmes fermés, c'est-à-dire les systèmes qui n'ont pas des contraintes externes à respecter, il s'agit d'assurer la cohabitation interne de ses constituants afin de garantir sa consistance en termes de correction mathématique. Dans les autres cas, il faut en plus vérifier que le système respecte les contraintes de son environnement,
  - valider relativement aux comportements permet de montrer que le système respecte les enchaînements d'actions décrits dans le CdC.

4.2.2.1 Fact

Une exigence du CdC peut décrire des données qui doivent exister dans le futur système. Une donnée est souvent décrite par le nom d'un objet dans le texte informel de l'exigence. Par exemple, le besoin FUN-G de la figure 3.5 du chapitre 3 contient des éléments de type Fact :

ID	*Description*
FUN-G	The landing system goal is maneuvering gears and their associated doors

données

Ce type correspond aux *ensembles porteurs (Sets)*, *constantes*, *variables* et *paramètres des événements* dans la Spec en B événementiel.

4.2.2.2 Functionality

Une fonctionnalité est une action ou service offert par le système. Elle est souvent décrite par un ensemble de mots contenant un verbe ou un nom extrait d'un verbe dans la phrase du besoin. Une fonctionnalité est traduite par un *événement* ou une *action dans un événement* dans la Spec. Dans l'exemple précédent, FUN-G inclut une fonctionnalité : *maneuvering gears and their associated doors*. Nous la notons *maneuvering* dans la suite (voir figure 4.2).

4.2.2.3 Obligation

Une obligation est l'expression, en logique de Hoare [Hoare, 1969], des contraintes de fonctionnement du futur système sous forme de pré-conditions et de post-conditions. Elle correspond aux *axiomes*, *invariants* et *gardes des événements* dans la Spec. Prenons le besoin FUN-3 de la figure 3.5 du chapitre 3. La lecture de son texte nous conduit à extraire l'élément de validation de type Obligation suivant :

ID	*Description*
FUN-3	The doors must be open when extending or retracting gears

pré-condition *post-condition*

4.2.2.4 Behavior

Un comportement, Behavior, est décrit sous forme d'un enchaînement d'éléments de type Functionnality. Cet enchaînement sera traduit par une séquence d'*événements* de la Spec. Un exemple de comportement est fourni dans la figure 4.2.

4.3 Préparation à la validation

Cette étape est accompagnée des trois tâches suivantes :

- la compréhension des besoins,
- leur structuration en utilisant la méthode d'Abrial et en l'appliquant sous l'outil ProR et
- leur analyse.

Rappelons qu'à cette étape de préparation, nous n'avons pas de Spec à valider ni de Liens ; nous gérons le système < CdC, ∅, ∅ >.

4.3.1 Extraction des éléments de validation

Le CdC, décrit sous l'outil ProR, est enrichi avec les nouveaux paramètres. Initialement, tous les champs - contenant les valeurs de ces paramètres - sont vides vu que nous n'avons pas commencé l'étape de structuration du cahier des charges. Une fois cette étape abordée, nous typons les phrases, extrayons les éléments de validation et les mémorisons dans *Terms*. *Event-B Model* reste vide car nous n'avons pas commencé le développement de la Spec.

4.3.2 Étude de cas

Reprenons l'exemple traité dans la section 4.2.

(a) Cette phrase contient des données et une fonctionnalité. Son *Term_Type* est à la fois *Fact* et *Functionality*.

ID	*Description*	***Term_Type***	*Terms*	*Event-B Model*
FUN-G	The landing system goal is maneuvering gears and their associated doors	**Functionality** **Fact**		

(b) Par la suite, nous extrayons les éléments informels de validation :

- *maneuvering* de type Functionality et
- *gears* et *doors* de type Fact.

Nous les mémorisons dans *Terms*.

ID	*Description*	*Term_Type*	***Terms***	*Event-B Model*
FUN-G	The landing system goal is **maneuvering gears** and their associated **doors**	Functionality	**- maneuvering**	
		Fact	**- gears** **- doors**	

Nous itérons les étapes (a) et (b) pour les autres phrases structurées de l'annexe A. Les besoins du CdC résultant sont présentés dans la figure 4.2. Cette figure montre également les informations de validation associées à plusieurs exigences du CdC. Pour des raisons de lisibilité, nous ne montrons pas le champ *Description* dans cette figure. Le résultat global de cette étape de restructuration est caractérisé par le système < CdC, $\varnothing$, $\varnothing$ > où :

- *CdC* représente le document référentiel issu de l'étape de réécriture des besoins. Ses phrases sont étiquetées, hiérarchisées et typées.
- *Spec* n'existe pas.
- *Liens* n'existent pas car la partie du formel est vide.

ID	*Term_Type*	*Terms*	*Event-B Model*
FUN-G	Functionality	- maneuvering	
	Fact	- gears - doors	
EQU-1	Fact	three landing sets - front - left - right	
EQU-1-1	Fact	a landing set : a door, a landing gear, hydraulic cylinders	
FUN-1	Functionality	- extend gears - retract gears - reverse gears movement	
FUN-2	Functionality	- open doors - close doors	
FUN-3	Obligation	*pre-condition :* - doors must be open *post-condition :* - extend gears - retract gears	
LS	Behavior	landing sequence : open doors $\rightarrow$ extend gears $\rightarrow$ close doors	

FIGURE 4.2 – Éléments de validation

4.4 Validation dans un niveau d'abstraction

Dans ce paragraphe, nous abordons la validation de la Spec au cours de sa construction dans un même niveau d'abstraction, c'est-à-dire sans raffinement. Par "niveau d'abstraction", nous voulons dire lors de l'ajout des éléments de la Spec tels que les variables, constantes, invariants, événements, gardes et actions. La figure 4.3 décrit une vision détaillée du déroulement de cette activité pour le système < CdC_i, $Liens_i$, $Spec_i$ > de la figure 4.1 où :

- CdC_i désigne le CdC au niveau i du développement,
- $Liens_i$ sont les Liens du niveau i et
- $Spec_i$ est la Spec en cours de construction de ce même niveau.

Dans un niveau i du développement, nous nous référons aux éléments de validation extraits depuis les besoins pour valider la $Spec_i$ au fur et à mesure de sa construction. Quand cette $Spec_i$ est complètement définie dans ce niveau, nous faisons le bilan de sa validation. Par l'expression "complètement définie", nous désignons la spécification formelle construite contenant tous ses composants : variables, invariants et événements.

4.4.1 Validation au cours du développement

Au fur et à mesure de l'évolution du système < CdC, Liens, Spec >, nous mettons à jour les éléments de validation et vérifions la prise en compte de ces éléments par le modèle formel en cours de construction. Ceci se réalise relativement aux Term_Type associés aux besoins et en se référant aux Terms.

Dire qu'une Spec est valide par rapport au CdC revient à dire qu'elle l'est relativement à toutes les données, les fonctionnalités, les obligations et les comportements décrits dans ce document. Au fil

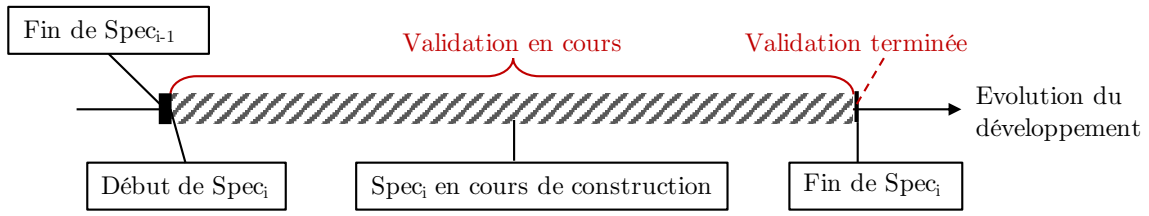


FIGURE 4.3 – Validation dans un niveau i

du développement, il y a des éléments de types Fact et Functionality qui sont pris en compte par cette Spec en B événementiel. Nous pouvons nous en assurer relativement au CdC et son glossaire :

Cas 1. Si cet élément est remplacé par un [terme_formel] dans le CdC, ceci implique qu'il existe dans le glossaire. La Spec est donc valide relativement à cet élément.

Cas 2. Si cet élément n'existe pas dans le glossaire, il n'est pas encore pris en compte dans la Spec. Dans ce cas, il se peut qu'un de ses raffinements le prendra en compte. Il faut passer à d'autres raffinements pour vérifier sa prise en compte. C'est seulement à la fin du développement et que cet élément n'a pas été pris en compte dans aucune des spécifications qu'on peut conclure que :

Cas 2.1. soit la Spec n'est pas valide par rapport à cet élément ;

Cas 2.2. soit la non-prise de cet élément dans la Spec est dûe à notre choix de développement. Ceci est applicable dans les cas où l'élément de validation concerné est abstrait et est précisé par d'autres éléments de validation. Par exemple, la fonctionnalité *maneuvering* du besoin FUN-G de la figure 4.2 n'a pas été prise en compte par toutes les spécifications : c'est notre choix de développement. Cette fonctionnalité est détaillée par d'autres fonctionnalités telles que *extend gears*, *retract gears* et *reverse gears movement* exprimées dans le besoin FUN-1 de la même figure.

Notons que les cas cités ci-dessus ne sont pas suffisants pour dire que la Spec est valide relativement aux éléments de types Obligation et Behavior. Il faut ajouter d'autres conditions que nous introduirons ultérieurement dans ce chapitre. Dans la suite de cette partie, nous expliquerons notre démarche pour la validation d'un élément de chaque Term_Type.

4.4.1.1 Validation relativement à une donnée

Revenons au système < CdC, Liens, Spec > de la figure 3.10 du chapitre 3. Une mise à jour du CdC par introduction du terme formel [gears_pos] est simultanément accompagnée par celle du contenu du champ Terms. La figure 4.4 montre une vue partielle du système.

La validation du modèle *Landing_System* en cours de construction commence à cette étape. Elle est assurée via la transformation de la donnée en une variable et la mise à jour du contenu du champ *Event-B Model* . En d'autres termes, *gears*, de type Fact, est formalisé par une variable *gears_pos* appartenant au modèle *Landing_System* présent dans *Event-B Model*. Cette donnée est prise en compte par la Spec. Par conséquent, cette Spec est valide relativement à cette donnée.

Prenons maintenant la donnée *doors*. Dans ce niveau de développement, cette donnée n'est pas encore prise en compte. Nous concluons que notre Spec, dans son état actuel présenté dans la figure 4.4, est valide par rapport à la donnée *gears* mais pas encore par rapport à la donnée *doors*.

Spec

MACHINE Landing_System
SEES Landing_Ctx
VARIABLES
gears_pos
INVARIANTS
inv_Typ_gears_pos: gears_pos ∈ G_Positions
EVENTS
END

Glossaire

Formal term	*Informal description*
Landing_System	landing system
gears_pos	gears

CdC

ID	*Description*	*Term_Type*	*Terms*	*Event-B Model*
FUN-G	The [Landing_System] goal is maneuvering [gears_pos] and their associated doors	Functionality	maneuvering	Landing_Ctx, Landing_System
		Fact	- [gears_pos] - doors	
...	...	...	...	...

FIGURE 4.4 – Donnée de validation formalisée

4.4.1.2 Validation relativement à une fonctionnalité

Spec

MACHINE Landing_System
SEES Landing_Ctx
VARIABLES
gears_pos
INVARIANTS
inv_Typ_gears_pos: gears_pos ∈ G_Positions
EVENTS
Event extend_gears
END

Glossaire

Formal term	*Informal description*
Landing_System	landing system
gears_pos	gears
extend_gears	extending, extend the landing gears, extend gears

CdC

ID	*Description*	*Term_Type*	*Terms*	*Event-B Model*
FUN-G	The [Landing_System] goal is maneuvering [gears_pos] and their associated doors	Functionality	maneuvering	Landing_Ctx, Landing_System
		Fact	- [gears_pos] - doors	
...	...	...	...	...
FUN-1	Maneuvering [gears_pos] consists on [extend_gears] or retracting them or reversing their movement	Functionality	- [extend_gears] - retract gears - reverse gears movement	Landing_Ctx, Landing_System

FIGURE 4.5 – Élément de validation de type Functionality formalisé

Partons de la figure 4.5, évolution de la figure 4.4. Prenons l'élément *extend_gears* typé Functionality et extrait du besoin FUN-1. La Spec prend en compte cette fonctionnalité en la formalisant par un événement.

4.4.1.3 Validation relativement à une obligation

Une obligation est une condition devant être respectée par le futur système. Le système < CdC, Liens, Spec > de la figure 4.5 a évolué :

- Spec. La variable *doors_pos* et la constante *open* sont introduites.
- CdC. Les deux éléments formels [doors_pos] et [open] sont introduits dans le texte informel des besoins. Le champ *Event-B Model* est mis à jour par le nom de la machine *Landing_System* et celui de son contexte *Landing_Ctx* dans les lignes d'exigences FUN-3 et LS.
- Liens. Le glossaire est mis à jour par l'ajout des couples *(doors_pos, doors)* et *(open, open)*.

Ce système évolué est décrit dans la figure 4.6. L'obligation citée dans le champ Terms et extraite du besoin FUN-3 est prise en compte sous forme d'une garde dans l'événement *extend_gears*.

Spec

MACHINE Landing_System
...
EVENTS
Event extend_gears
when
grd-FUN-3 : doors_pos = open
then
...
end
END

Glossaire

Formal term	*Informal description*
Landing_System	landing system
gears_pos	gears
extend_gears	extending, extend the landing gears, extend gears
doors_pos	doors
open	open

CdC

ID	*Description*	*Term_Type*	*Terms*	*Event-B Model*
...	...	...	...	...
FUN-3	The [doors_pos] must be [open] when [extend_gears] or retracting gears	Obligation	*pre-condition :* - [doors_pos] = [open] *post-condition :* - [extend_gears] - retract gears	Landing_Ctx, Landing_System
LS	In nominal mode, the landing sequence is : open the doors --> [extend_gears] --> close the doors	Behavior	- landing sequence : open doors --> [extend_gears] --> close doors	Landing_Ctx, Landing_System

FIGURE 4.6 – Élément de validation de type Obligation formalisé

4.4.1.4 Validation relativement à un comportement

Les scénarios de validation extraits du CdC sont mémorisés dans le champ Terms et ont Behavior comme Term_Type. Nous les utilisons pour animer notre Spec. Un scénario de validation est un

enchaînement d'actions ou d'événements et décrivant un comportement du futur système. Ce scénario peut exprimer un comportement que le système devra respecter ou un comportement que le système ne doit pas être autorisé à suivre.

Le système < CdC, Liens, Spec > de la figure 4.6 a évolué pour obtenir la figure 4.7 décrite par :

- Spec. De nouveaux événements *retract_gears*, *open_doors* et *close_doors* sont introduits.
- CdC. Des éléments formels [retract_gears], [open_doors] et [close_doors] sont introduits dans le texte informel des besoins.
- Liens. Le glossaire est mis à jour par l'ajout de nouveaux couples.

L'animation de la Spec via des scénarios de validation nous invite à utiliser les scénarios existants et à définir de nouveaux scénarios. Les scénarios de validation décrits dans le CdC ne sont toujours pas suffisants pour couvrir tous les comportements possibles de la Spec. Pour combler ce manque, nous construisons de nouveaux scénarios à partir des scénarios existants. Ces nouveaux scénarios doivent être proposés au client et aux différentes parties prenantes au sein du projet pour confirmer leur utilisation ou les rejeter. Nous validons notre Spec relativement aux comportements existants et créons de nouveaux scénarios :

- à partir de la machine à états, state machine, associée à la Spec ou
- en introduisant de nouvelles actions dans les scénarios existants.

Spec

MACHINE Landing_System
...
EVENTS
Event extend_gears
Event retract_gears
Event open_doors
Event close_doors
...
END

Glossaire

Formal term	*Informal description*
Landing_System	landing system
gears_pos	gears
extend_gears	extending, extend the landing gears, extend gears
doors_pos	doors
open	open
retract_gears	retracting gears, retract gears
open_doors	open the doors, open doors
close_doors	close the doors, close doors

CdC

ID	*Description*	*Term_Type*	*Terms*	*Event-B Model*
...	...	...	...	...
FUN-3	The [doors_pos] must be [open] when [extend_gears] or [retract_gears]	Obligation	*pre-condition :* - [doors_pos] = [open] *post-condition :* - [extend_gears] - [retract_gears]	Landing_Ctx, Landing_System
LS	In nominal mode, the landing sequence is : [open_doors] --> [extend_gears] --> [close_doors]	Behavior	landing sequence : [open_doors] --> [extend_gears] --> [close_doors]	Landing_Ctx, Landing_System

FIGURE 4.7 – Expression de la validation relativement au comportement

4.4.1.4.1 Simulation d'un scénario existant. Le comportement décrit dans le champ Terms associée à l'exigence LS de la figure 4.7 contient des [termes_formels] enchaînés dans un ordre précis. L'existence de tels termes ne suffit pas pour dire que la Spec est valide relativement à ce comportement. Il est important d'animer la machine *Landing_System* pour contrôler cet enchaînement. Ceci s'accomplit en utilisant l'outil ProB [Bendisposto et al., 2008] et en déclenchant, dans l'ordre, les événements :
(1) *open_doors* (2) *extend_gears* (3) *close_doors*. Notre Spec respecte cet ordre. Elle est valide relativement à ce comportement.

4.4.1.4.2 Construction de nouveaux scénarios à partir de la machine à états. Nous utilisons un plugin de la plateforme Rodin, Event-B Statemachines [45] pour construire de nouveaux scénarios. Une machine à états, *SM*, est associée à chaque modèle en B événementiel :

- chaque état de la *SM* représente une valeur des variables B événementiel et
- chaque transition entre deux états représente un événement.

Nous prenons un état initial dans lequel les portes sont fermées et le train d'atterrissage est étendu.

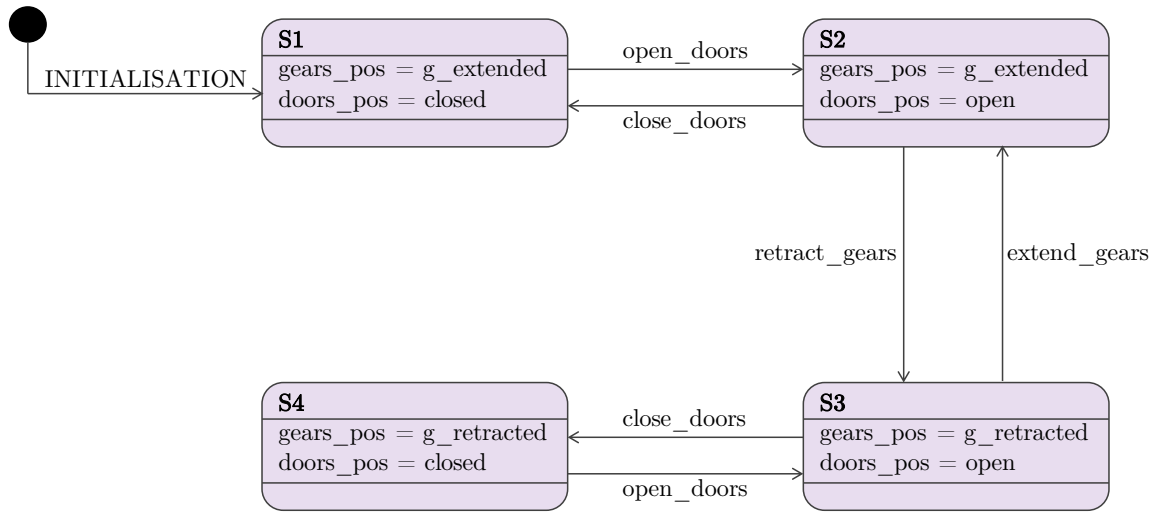


FIGURE 4.8 – Une sous-machine à états de *Landing_System*

La figure 4.8 présente la *SM* correspondante à la machine B événementiel *Landing_System*. Cette *SM* décrit la séquence de rétraction et d'extension des trains. Ses états sont décrits par les valeurs de ses variables *gears_pos* et *doors_pos*. Le passage de l'état *S1* à l'état *S2* s'effectue par le déclenchement de l'événement *open_doors* ; la variable *doors_pos* passe de *closed* à *open*.

A partir de cette *SM*, de nouveaux scénarios sont proposés dans lesquels des enchaînements d'états ne sont pas autorisés :

- *INITIALISATION* → *retract_gears*
  Ce scénario passe de *S1* à *S3*. Il est interdit car il n'est pas possible de rétracter les trains sans ouvrir les portes ;

45. http ://wiki.event-b.org/index.php/Event-B_Statemachines

- *INITIALISATION* → *close_doors*
  Il n'y a pas de passage immédiat entre un état de trains étendus (état *S1*) à un autre état de trains pliés (état *S4*) sans passer par la séquence de rétraction. Comme autre interprétation, il n'est pas possible de fermer les portes car elles sont déjà fermées ;
- *INITIALISATION* → *open_doors* → *retract_gears* → *retract_gears* → *close_doors*
  Il n'est pas possible de rétracter les trains plus d'une fois dans cette séquence.

Ces scénarios ne sont pas permis par la machine *Landing_System*. Ils sont vérifiés par l'animation de la Spec avec ProB.

4.4.1.4.3 Construction de scénarios non autorisés. Nous exploitons les techniques issues du test de logiciels pour définir de nouveaux scénarios. Par exemple, nous introduisons une ou plusieurs actions dans un scénario existant.

A partir de la compréhension des exigences, nous introduisons un événement décrivant un comportement non souhaité dans le scénario de validation existant et décrit dans la figure 4.7 :

$$open\_doors \rightarrow extend\_gears \rightarrow close\_doors$$

L'action *open_doors* est introduite dans ce scénario avant que la porte soit fermée. L'objectif du scénario obtenu est de vérifier si l'ouverture des portes des trains d'atterrissage se réalise une deuxième fois dans une séquence d'extension des trains. Ce scénario a la forme suivante :

$$open\ \_doors \rightarrow extend\_gears \rightarrow \mathbf{open\_doors} \rightarrow close\_doors$$

La machine *Landing_System* a interdit l'animation de ce scénario, à l'aide de ProB. La garde de l'événement *open_doors* interdit sa simulation lorsque les portes sont déjà ouvertes.

Tout comme la rétraction des trains, la session de leur extension est critique. Il ne faut pas autoriser la fermeture des portes tant que les trains sont en mouvement. De la même manière, nous pouvons construire et simuler un nouveau scénario pour la séquence de rétraction des trains.

4.4.2 Bilan de la validation

La construction de la $Spec_i$ du niveau i d'abstraction est terminée. Nous pouvons vérifier et prouver sa correction mathématique via les outils de preuve sous Rodin. En adoptant notre approche, la validation de cette spécification formelle est effectuée pas-à-pas au cours de son développement. La validation à cette étape est considérée comme une *récapitulation* des validations faites partiellement sur chaque donnée, fonctionnalité, obligation et comportement au fur et à mesure de la construction de la $Spec_i$.
Notons qu'un modèle qui n'est pas valide relativement à quelques éléments de validation dans un niveau i pourra être valide dans un niveau raffiné.

4.5 Validation des modèles raffinés

Par analogie au test, la validation peut être comparée au test boîte noire, appelé aussi test fonctionnel. Les informations de validation extraites du CdC ne sont pas suffisantes ni exhaustives pour

dire qu'un modèle est valide. Quand les modèles deviennent de plus en plus complexes, l'espace d'états associé à ces modèles croît de façon exponentielle. Ceci rend difficile la tâche de validation. Le principe "*les tests peuvent servir à montrer la présence d'erreurs, pas leur absence" (program testing can be used to show the presence of bugs, but never to show their absence)* de Edsger Dijkstra [Dijkstra, 1970] pour le test de logiciels devient applicable dans ce cas de spécifications formelles ; la validation d'une Spec sert à détecter des défaillances et des erreurs dans cette Spec relativement à son CdC. Cette tâche sert également à assurer que la Spec est valide seulement relativement aux éléments de validation fournis dans les besoins. Ces éléments ne sont toujours pas suffisants. Par exemple, les besoins du train d'atterrissage ne décrivent pas tous les comportements possibles du système ; il peut y avoir des comportements imprévisibles non-cités dans le cahier des charges.

Dans la suite de cette section, nous abordons le problème de validation des modèles raffinés :

- en mémorisant l'historique de la validation des modèles abstraits et
- via le concept du collage.

4.5.1 Historique

Le développement évolue par des raffinements successifs de la Spec et par l'introduction de nouveaux besoins dans le CdC. Au fur et à mesure de cette évolution, nous mémorisons l'historique concernant la validation des modèles relativement à chaque Term_Type. Nous abordons la problématique de l'espace d'états de la Spec en termes de l'explosion combinatoire : cet espace devient de grande taille au moment du raffinement. La validation de cette Spec devient difficile à gérer.
La mémorisation de l'historique et de la trace des modèles validés aide à réduire l'effort de validation des modèles complexes dans les différents raffinements. Nous distinguons deux cas :

Cas 1. Pour les éléments de type Fact et Functionality, il suffit de voir si ces éléments sont mis entre "[ ]" dans le CdC. Si c'est le cas, ces éléments sont pris en compte par la Spec et mémorisés dans l'historique.

Cas 2. Pour les éléments de types Obligation et Behavior, nous utilisons la notion de *hiérarchie* pour mémoriser cet historique :

Cas 2.1 Si un comportement ou une obligation est introduit sans avoir un lien de hiérarchie avec les autre besoins, cet élément n'a pas d'historique mémorisé.

Cas 2.2 Si ce besoin est introduit comme enfant d'un comportement ou d'une obligation validée, nous pourrons exploiter l'historique de validation du besoin parent.

4.5.1.1 Évolution du système par raffinement

Au fur et à mesure de l'évolution du système < CdC, Liens, Spec >, nous mettons à jour le CdC et les Liens et vérifions la correction mathématique de la Spec via des preuves.

Étude de cas

A partir de la figure 4.7, nous prenons en compte l'introduction :

- des cylindres appartenant à la partie hydro-mécanique du système hybride et
- de la manette et des voyants lumineux présentés dans l'interface pilote.

Les besoins concernés sont présentés dans la figure 4.9 dans laquelle :

- le besoin *LS* provient de la figure 4.7. Pour des contraintes d'espace, nous présentons les autres besoins de cette figure par des "...",
- le besoin *LS-GCy* est un enfant du besoin *LS* et
- le besoin *FUN-P-light1* est un enfant du besoin *FUN-P-I*.

En l'état de notre compréhension des besoins, nous ne pouvons pas préciser la hiérarchie du besoin *FUN-P-I* relativement au besoin *LS*.

ID	*Description*	*Term_Type*	*Event-B Model*
...	...	...	...
LS	In nominal mode, the landing sequence is : [open_doors] → [extend_gears] → [close_doors]	Behavior	Landing_System Landing_Ctx
LS-GCy	Using cylinders movement, the extending sequence is : [open_doors] → unlock the gear cylinder from the high position → move down gears cylinders → lock down gears cylinders → [close_doors]	Behavior	
FUN-P-I	The pilot interface is composed of green, orange and red lights and an Up/Down handle	Fact	
FUN-P-light1	When gears cylinders are locked down, the green light is on	Obligation Behavior	

FIGURE 4.9 – Besoins relatifs aux cylindres et à l'interface pilote

4.5.1.2 Réutilisation de l'historique mémorisé

Le besoin LS-GCy est de type Behavior. Il décrit le déroulement interne de la séquence d'extension des trains à l'aide des cylindres. Nous avons :

- le besoin *LS*. Il contient l'événement *extend_gears* provenant du modèle abstrait *Landing_System*. Ce modèle a été validé relativement au comportement dans des étapes précédentes de la section 4.4.1 ;
- le besoin *LS-GCy*. Il s'agit d'un comportement décrit par la suite d'éléments suivante, certains sont formels et certains sont informels :

[open_doors] → unlock the gear cylinder from the high position → move down gears cylinders → lock down gears cylinders → [close_doors]

Nous choisissons de formaliser le besoin LS-GCy en B événementiel.

- *État de départ :* il est représenté par le système < CdC, Liens, Spec > décrit dans la figure 4.10.
- *Décision :* formaliser le besoin LS-GCy.
- *Traitement :* raffiner la machine *Landing_System* et étendre le contexte *Landing_Ctx*.
- *État résultant :* le système < CdC', Liens', Spec' > :

Spec

MACHINE Landing_System
...
EVENTS
Event extend_gears
Event retract_gears
Event open_doors
...
END

Glossaire

Formal term	*Informal description*
Landing_System	landing system
gears_pos	gears
extend_gears	extending, extend the landing gears, extend gears
doors_pos	doors
open	open
retract_gears	retracting gears, retract gears
open_doors	open the doors, open doors
close_doors	close the doors, close doors

CdC

ID	*Description*	*Term_Type*	*Event-B Model*
...	...	...	...
LS	In nominal mode, the landing sequence is : [open_doors] --> [extend_gears] --> [close_doors]	Behavior	Landing_Ctx, Landing_System
LS-GCy	Using cylinders movement, the extending sequence is : [open_doors] --> unlock the gear cylinder from the high position --> move down gears cylinders --> lock down gears cylinders --> [close_doors]	Behavior	
...	...		

FIGURE 4.10 – État du système avec les cylindres

— *Spec'*. Le modèle *GCy_mch* spécifie le besoin *LS-GCy*. Il est défini par un raffinement du modèle abstrait *Landing_System* comme présenté dans la figure 4.11. De même, le contexte *GCy_ctx* étend (extends) le contexte *Landing_Ctx*.

Le raffinement de l'événement *extend_gears* est présenté dans la figure 4.12 créée avec le plugin Flows [46] sous Rodin. Ce raffinement est présenté en termes des trois événements *unlock_up_gears_cylinders*, *move_down_gears_cylinders* et *lock_down_gears_cylinders*. Ces trois événements décrivent un enchaînement d'actions atomiques pour effectuer l'action de dépliage des trains en utilisant les cylindres. Un tel enchaînement est exprimé par les trois conditions suivantes :

1. la garde du premier événement de la chaîne, *unlock_up_gears_cylinders*, doit contenir la garde de l'événement *extend_gears*,

2. l'action de chaque événement doit servir comme garde pour l'événement qui le suit dans la chaîne et

3. une partie de l'action du dernier événement de la chaîne, *lock_down_gears_cylinders*, doit correspondre à l'action de l'événement *extend_gears*.

46. http ://wiki.event-b.org/index.php/Flows

CONTEXT *GCy_ctx*
EXTENDS *Landing_Ctx*
SETS
GCylinders_Positions
CONSTANTS
locked_up
locked_down
moving_up
moving_down
unlocked_up
unlocked_down
AXIOMS
axm_1 : partition (GCylinders_Positions,
$\{$*locked_up*$\}$, $\{$*locked_down*$\}$,
$\{$*moving_up*$\}$, $\{$*moving_down*$\}$,
$\{$*unlocked_up*$\}$, $\{$*unlocked_down*$\}$*)*
END

MACHINE *GCy_mch*
REFINES *Landing_System*
SEES *GCy_ctx*
VARIABLES
gears_pos
gears_moving
doors_pos
gears_cylinders
INVARIANTS
inv_cyln : gears_cyln $\in$ *GCylinders_Positions*
EVENTS
INITIALISATION
// Event extend_gears est defini par
// les 3 events suivants :
Event unlock_up_gears_cylinders
Event move_down_gears_cylinders
Event lock_down_gears_cylinders
refines *extend_gears*
Event retract_gears
Event reverse
Event open_doors
Event close_doors
...
END

FIGURE 4.11 – Machine *GCy_mch* et son contexte

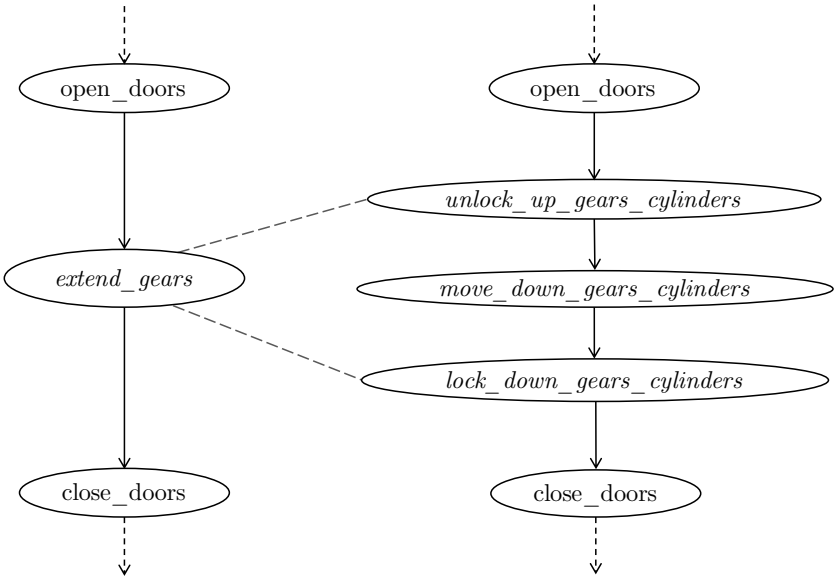


FIGURE 4.12 – Vision abstraite et raffinée de l'événement *extend_gears*

— *CdC'*. Les termes formels introduits dans *Spec'* sont introduits dans le *CdC'* :

ID	*Description*
LS-GCy	Using cylinders movement, the extending sequence is : *[open_doors]* → *[unlock_up_gears_cylinders]* → *[move_down_gears_cylinders]* → *[lock_down_gears_cylinders]* → *[close_doors]*

— *Liens'*. Le glossaire est mis à jour automatiquement par de nouveaux couples. Il a la forme suivante :

Formal_Term	*Informal_Definition*
...	...
open_doors	opening
open_doors	open doors
unlock_up_gears_cylinders	unlock the gear cylinder from the high position
move_down_gears_cylinders	move down gears cylinders
lock_down_gears_cylinders	lock down gears cylinders

Le comportement abstrait du besoin *LS* a été validé dans le paragraphe "*Validation relativement à un comportement*" de la section 4.4.1. Les trois événements raffinent l'événement *extend_gears* appartenant au scénario abstrait, situé dans la partie gauche de la figure 4.12. Ce choix de raffinement nous amène à simuler le sous-scénario décrit dans la partie droite de la même figure par l'enchaînement des trois événements. Ceci a pour objectif de garantir que le comportement décrit par cet enchaînement ne contredit pas le comportement de l'événement abstrait *extend_gears*.

L'animation de la machine *GCy_mch* simulant l'ordre des trois événements

1. *unlock_up_gears_cylinders*,
2. *move_down_gears_cylinders* et
3. *lock_down_gears_cylinders*

montre que cette Spec respecte cet enchaînement. Elle prend en compte ce sous-scénario.

4.5.2 Collage

Nous nous intéressons au caractère hybride des systèmes. Notre objectif est de valider une Spec rassemblant des composants de natures différentes comme des composants matériels, logiciels ou d'interface homme-machine. Supposons que, dans un niveau avancé du développement, nous avons deux spécifications formelles Spec 1 et Spec 2 désignant respectivement la Spec d'un composant 1 et celle d'un composant 2. Si les deux Specs sont validées séparément, alors il nous reste à valider leur assemblage. Ceci se réalise à l'aide du concept de *collage*.

4.5.2.1 Collage entre Specs validées

Regardons un nouveau raffinement dans lequel la communication s'exprime par la notion de collage. Après formalisation des besoins *FUN-P-I* et *FUN-P-light1*, nous obtenons le système suivant :

- *Spec'*. Elle représente le raffinement de la Spec de la figure 4.11. Nous introduisons :

  — les deux variables *handle* et *light_state*,

  — les constantes *green*, *orange*, *red*, *on*, *up* et *down* et

  — les ensembles porteurs (Sets) *LIGHTS* et *HANDLE_STE*.

  Cette Spec' contient les invariants de collage suivants :

  $$glu\_inv1 : gears\_pos = g\_extended \implies gears\_cyln = locked\_down \wedge doors\_pos = closed$$
  $$glu\_inv2 : light\_state(green) = on \implies gears\_cyln = locked\_down \wedge handle = down$$

- *CdC'*. Les textes informels des deux besoins *FUN-P-I* et *FUN-P-light1* évoluent par introduction de termes formels comme suit :

ID	*Description*
FUN-P-I	The pilot interface is composed of [green], [orange] and [red] [LIGHTS] and an [HANDLE_STE] [handle]
FUN-P-light1	When [gears_cylinders] are [locked_ down], the [light_state] [green] is [on]

- *Liens'*. Ils sont automatiquement mis à jour.

La Spec de la figure 4.11 est la spécification formelle du composant *hydro-mécanique* comportant les cylindres, tandis que Spec' concerne le composant *interface pilote*. Le collage entre ces deux composants est exprimé via le *raffinement*.

4.5.2.2 Validation via le collage

La mise ensemble de deux spécifications Spec 1 et Spec 2 peut être assurée via le collage en adoptant une des deux approches suivantes :

- *Approche ascendante* dans laquelle nous développons les spécifications Spec 1 et Spec 2 séparément l'une de l'autre. Une fois ces spécifications terminées, nous les assemblons dans une seule Spec. Ceci se réalise via l'outil Shared Event Composition [47] sous Rodin.
- *Approche decendante* permettant de partir d'une vue très abstraite du système et de la raffiner pas à pas en ajoutant des composants. Autrement dit, la Spec 2 du composant 2 raffine la Spec 1 du composant 1.

Au cours de l'étude du système de trains d'atterrissage, nous avons adopté la deuxième approche, descendante. Le collage dans cette approche est exprimé via la technique de raffinement. Les règles de raffinement aident à guider la validation de la Spec résultante du collage entre Spec 1 et Spec 2. Cette validation concerne essentiellement les comportements et les obligations. Si Spec 2 raffine Spec 1, nous avons :

1. Spec 2 doit respecter Spec 1 et ne doit pas la contredire en termes de comportement.
2. L'invariant de Spec 2 doit renforcer celui de Spec 1. Ceci nous guide à utiliser l'*invariant de collage*.
3. La notion d'*événements de collage* représente l'échange et la communication entre les deux composants. Ceci nous aide à décrire un comportement commun entre les deux spécifications.

La considération de ces trois règles pendant le raffinement nous apporte une aide à la validation des comportements et des obligations des spécifications utilisant le collage.

4.6 Vérification

Les outils de vérification de spécifications formelles, tels que les prouveurs et les générateurs d'obligations de preuves sous Rodin, permettent de :

- détecter des omissions dans le CdC concernant :
  - — l'état initial,
  - — certains besoins implicites et non décrits dans le CdC,
  - — l'absence de scénarios dans le CdC,
- détecter des contradictions entre la spécification mathématiquement correcte et les besoins. La spécification ne correspond pas aux besoins lorsque le CdC est trop pauvre.

Exemple.

Ajoutons la nouvelle exigence *FUN-err* suivante :

ID	*Description*
FUN-err	The gears can move only if doors are closed

47. http://wiki.event-b.org/index.php/Parallel_Composition_using_Event-B

Ce besoin est de type obligation. Il est modélisé en B événementiel par l'invariant :

$$FUN-err\_inv : gears\_moving = TRUE \Rightarrow doors\_pos = closed$$

dans lequel la variable booléenne gears_moving indique si les trains sont en déplacement. Avant l'introduction de cet invariant, la Spec en B événementiel était mathématiquement correcte. Après l'ajout de tel invariant, les prouveurs de Rodin ont donné lieu à un ensemble d'obligations de preuve non déchargées. Une de ces obligations de preuve a précisé que l'événement extend_gears ne respecte pas ce nouvel invariant FUN-err_inv :

- une contradiction est décelée entre la garde de cet événement et cet invariant :
  - garde : $doors\_pos = open$
  - invariant : $gears\_moving = TRUE \Rightarrow doors\_pos = closed$,
- cette garde a été modélisée à partir l'exigence FUN-3 (voir figure 4.6) et cet invariant a été modélisée à partir l'exigence FUN-err du CdC. Une contradiction entre ces exigences est détectée à l'aide des prouveurs de Rodin. Ce retour de la vérification mathématique à l'aide des preuves nous amène à poser des questions sur l'exigence "coupable" pour cet échec de preuve.

4.7 Conclusion

Dans ce chapitre, nous avons abordé notre contribution concernant la validation dans le processus de développement de logiciels et des systèmes complexes et hybrides. Nous avons traité la validation tout au long du développement. Pour y parvenir, nous préparons cette activité dès l'analyse des besoins et leur réécriture. Nous avons montré l'importance de la mémorisation de l'historique de la validation de la Spec et du collage pour gérer la complexité des systèmes et faciliter leur validation dans les niveaux raffinés.

Les limites de notre approche concernent les deux points suivants :

- la décision finale, pour dire qu'une Spec est valide, ne peut être prise qu'à la fin de tous les pas de raffinement et
- l'utilisation de la machine à états associée à la Spec pose le défit suivant : cette machine est construite à partir de la Spec dans l'objectif de construire de nouveaux scénarios de validation. Cette dépendance peut mener à des scénarios erronés si la Spec est erronée. Nous pensons que la demande de l'avis du client, les discussions et échanges avec lui sur ce point sont très importants pour qu'il puisse confirmer ces nouveaux scénarios.

Comme perspectives de ce travail, nous envisageons :

- utiliser le collage pour résoudre le problème de validation des Specs définies séparément. L'utilisation des invariants et des événements de collage apportera une aide importante pour la validation des modèles raffinés,
- exploiter l'apport des invariants et théorèmes pour aider à la validation. Comme c'est le cas des invariants/théorèmes exprimant l'absence de blocage (deadlock freness) dans le modèle formel, nous pouvons exprimer d'autres propriétés liées à la validation sous forme de théorèmes ou d'invariants. Une fois que ce dernier est prouvé correct mathématiquement, notre Spec demeure valide relativement à cette propriété ; ceci permet d'éviter de parcourir plusieurs chemins dans l'espace d'états lors de la validation et

- traiter le problème de mémorisation de l'historique de validation. La notion de hiérarchie est importante pour assurer cet objectif. L'historique sert à faciliter la validation des spécifications concrètes en réutilisant ce qui a été accompli pour les spécifications abstraites.

Chapitre 5

Évolution d'un système existant

Sommaire

L'évolutivité d'un système est un problème crucial lorsqu'il s'agit d'un système critique, hybride et complexe nécessitant de la précision dans son développement. Dans ce chapitre, nous abordons ce problème d'évolutivité par la prise en compte de nouveaux besoins relativement à un système existant. Ce dernier est décrit par son cahier des charges lié à sa spécification formelle. Les outils de vérification et de validation jouent un rôle important tout au long du développement. Ils assurent la correction de la spécification et l'amélioration de la qualité de son cahier des charges. Nous illustrons notre contribution par l'étude d'un distributeur automatique de billets (DAB).

5.1 Prise en compte d'un nouveau besoin

5.1.1 Analyse et compréhension

La prise en compte d'un nouveau besoin *Req* dans un système existant < CdC, Liens, Spec > permet d'obtenir un nouveau système < CdC', Liens', Spec' >. Le travail d'intégration de ce besoin commence par sa compréhension et son analyse. Ceci se réalise en amenant des éléments de réponse aux questions citées dans la table 5.1. La prise en compte de *Req* aide à déceler des problèmes tels que les incomplétudes et les incohérences dans le système existant.

Question	*Énoncé*
Q1.	Quel est son objectif ?
Q2.	Présente-t-il de nouveaux rôles associés aux acteurs du système ?
Q3.	A quel niveau du CdC va-t-il être intégré ?
Q4.	Comment va-t-il communiquer avec les autres besoins du CdC ?
Q5.	Quelles informations relatives à la validation apporte-t-il ?

TABLE 5.1 – Questions liées au besoin Req

5.1.2 Effets du nouveau besoin sur l'existant

5.1.2.1 CdC

Les évolutions de ce document concernent :

- l'introduction de nouvelles phrases issues du nouveau besoin,
- la composition/décomposition des phrases existantes,
- l'introduction des termes formels dans les énoncés informels des phrases,
- la mise à jour de la hiérarchie entre les phrases et
- l'expression de nouvelles informations telles que l'ordre entre phrases.

5.1.2.2 Spec

L'évolution de ce composant se répercute par :

- l'introduction de nouveaux éléments formels notamment des constantes, ensembles, variables, événements, axiomes, invariants et gardes,
- la suppression de certains éléments formels,
- l'introduction des précisions et de clarifications via des éléments formels (gardes et invariants) ou des commentaires et
- la description du collage entre les différents modèles de la Spec.

5.1.2.3 Liens

Représentés dans un *glossaire*, ces liens sont mis à jour automatiquement au fur et à mesure de l'évolution du CdC et de sa Spec correspondante.

5.2 Système existant

L'objectif de ce système est de permettre au client d'une banque de retirer de l'argent via sa carte bancaire associée à son compte. Nous partons d'un développement de ce système nommé *DAB 1* vérifié et validé. Il est décrit dans la figure 5.1 et caractérisé par les composants suivants :

CdC. Décrit sous ProR, ce document est constitué des phrases étiquetées et hiérarchisées. Les deux phrases *R5-1* et *R5-2* sont des enfants de *R5*. De même, la phrase *R5-1-1* est un enfant de *R5-1*. Ce document intègre des termes formels mis entre crochets *[ ]* et provenant de la *Spec*.

Spec. Elle est décrite en B événementiel par le contexte *DAB1_Ctx* et la machine *DAB1_Mch*.

Liens. Mémorisant les liens entre le CdC et sa Spec, ce composant est décrit par un glossaire.

CdC

ID	*Description*
	***** DAB 1 ***** Le client, identifié par sa carte bancaire et le code associé, retire de l'argent de son compte géré par la banque, à partir d'un distributeur. Le client peut retirer un certain montant chaque semaine. NB. Nous supposons que la carte est valide ainsi que son code. Le distributeur a assez d'argent dans sa caisse.
R1	Le client [insere_carte] dans le distributeur
R2	Le client [entre_pin] dans le distributeur
...	...
R5	La banque donne sa decision au client via le distribueur :
R5-1	Soit la banque accepte la demande du client :
R5-1-1	La banque [maj_compte] du client
R5-1-2	Le distributeur [sort_argent] pour le client
R5-1-3	Le distributeur [imprime_recu] pour le client
R5-2	Soit la banque refuse la demande du client : le distributeur [affiche_message_refus] au client
R6	Le distributeur [ejecte_carte]
R7	Le distributeur [termine_transaction]

Glossaire

Formal term	*Informal description*
insere_carte	insère sa carte bancaire
entre_pin	entre son code
maj_compte	met à jour le compte
sort_argent	sort l'argent
imprime_recu	imprime un reçu
affiche_message_refus	affiche un message de refus
ejecte_carte	éjecte la carte
termine_transaction	termine la transaction

Spec

CONTEXT DAB1_Ctx
SETS
CARTES, DECIDE, MSG
CONSTANTS
c1, c2, c3, pas_de_carte
OK, NotOK
solde_insuffisant, montant_depasse_plafond, code_invalide
pin
AXIOMS
axm1 : partition (CARTES, {c1}, {c2}, {c3}, {pas_de_carte})
axm2 : partition (DECIDE, {OK}, {NotOK})
axm3 : partition (MSG, {solde_insuffisant}, {montant_depasse_plafond}, {code_invalide})
axm4 : pin ∈ CARTES ↣ ℕ

MACHINE DAB1_Mch
SEES DAB1_Ctx
VARIABLES
carte, code, decision, message, nb_essai_pin
INVARIANTS
carte_Typ : carte ∈ CARTES
code_Typ : code ∈ ℕ
decision_Typ : decision ∈ DECIDE
message_Typ : message ∈ MSG
nbEssaiPin_Typ : nb_essai_pin ∈ ℕ
EVENTS
Initialisation
Event insere_carte //Client
Event entre_pin //Client
Event maj_compte //Banque
Event sort_argent //Distributeur
Event imprime_recu // Distributeur
Event affiche_message_refus //Distributeur
Event ejecte_carte //Distributeur
Event termine_transaction //Distributeur
...
END

FIGURE 5.1 – Système existant DAB 1

Le développement du système DAB 1 est hybride où les actions sont effectuées par :

- le client, comme *insere_carte*,
- la banque, via l'événement *maj_compte* et

- le distributeur, avec l'événement *sort_argent* par exemple.

5.3 Perte des cartes

5.3.1 Analyse du nouveau besoin

Soit le cahier des charges du nouveau système nommé *DAB 2* décrit par un nouveau besoin relativement à DAB 1 :

DAB 2. DAB 1 + *Le distributeur ne permet pas la perte de la carte bancaire du client.*

La table 5.2 apporte quelques éléments de réponse aux questions posées dans la section 5.1. Nous n'avons pas de réponse immédiate pour la question Q3. La réponse à cette question est initialement marquée par *"–"* et sera fournie au fur et à mesure de l'avancement de l'évolution du système.

Questions	***Réponses***
Q1.	non-perte de la carte bancaire
Q2.	Il présente un nouveau rôle pour le lecteur de cartes appartenant à l'interface du distributeur
Q3.	–
Q4.	- par la hiérarchie (soit enfant soit frère des autres besoins) et - par l'ordre
Q5.	- carte non-perdue - client récupère carte ou distributeur récupère carte

TABLE 5.2 – Éléments de réponses aux questions pour DAB 2

Après compréhension et analyse du nouveau besoin, son objectif se précise : il concerne le besoin R6, voir figure 5.1. Une solution consiste à récupérer la carte par le client, ou par le distributeur après un délai précisé.

5.3.2 Évolution du système

Cette évolution est décrite pas-à-pas. Nous présentons les différentes étapes.

5.3.2.1 Initialiser le développement

En partant du système existant DAB 1, la Spec est raffinée et caractérisée par son contexte nommé *DAB2_Ctx* et une machine *DAB2_Mch*, voir figure 5.2.

CONTEXT *DAB2_Ctx*
EXTENDS
DAB1_Ctx
...
END

MACHINE *DAB2_Mch*
REFINES
DAB1_Mch
SEES
DAB2_Ctx
...
END

FIGURE 5.2 – Initialisation de la Spec de DAB 2

5.3.2.2 Modifier R6

Ce nouveau besoin prévoit que la carte bancaire du client ne doit pas être perdue. Les raisons possibles de la perte sont :

1- la carte est perdue après être éjectée par le lecteur de cartes du distributeur ;

2- elle est oubliée durant le processus :

— après la sortie d'argent via le besoin R5-1-2 ou

— suite au refus de la requête de retrait via l'affichage du message associé. Le client peut partir dès qu'il voit qu'il ne peut pas obtenir l'argent demandé en oubliant sa carte. Ceci est exprimé dans R5-2.

Regardons la première raison et cherchons le besoin coupable dans le CdC. D'après le glossaire de la figure 5.1, nous trouvons les termes informels *"éjecte la carte"* associés au terme formel *ejecte_carte*. Ce dernier est exprimé dans le besoin R6. La carte n'est plus prise en charge par le distributeur. L'analyse de ce terme nous conduit à :

- *modifier* le texte de R6 afin de prendre en compte la non-perte de la carte et

- *remplacer* le terme formel *[ejecte_carte]* par les termes informels *"rend la carte"*.

Le terme formel *[ejecte_carte]* et son élément formel associé dans la Spec ne seront pas accessibles dans le système de DAB 2. Les nouveaux termes indiquent que la carte bancaire est rendue au client, c'est-à-dire qu'elle reste détenue par le distributeur jusqu'à ce qu'elle soit récupérée par le client, voir figure 5.3.

DAB 1	
ID	*Description*
R6	Le distributeur [ejecte_carte]

(a) Énoncé de R6

DAB 2	
ID	*Description*
R6	Le distributeur *rend la carte* au client qui la récupère pendant une durée d'attente

(b) Nouvel énoncé de R6

FIGURE 5.3 – Modification de l'énoncé de R6

NB. Nous traiterons la deuxième raison de perte de carte ultérieurement dans ce chapitre (section 5.3.2.5).

5.3.2.3 Réorganiser les besoins

5.3.2.3.1 Réécrire. Regardons la perte de la carte bancaire lorsque la banque accepte la demande du client, comme indiqué dans le besoin R5-1, voir figure 5.1. Afin d'éviter cette perte, la carte doit

sortir du lecteur de cartes et être récupérée avant de donner l'argent. Ceci est exprimé dans le nouvel énoncé du besoin R6 de la figure 5.3.

5.3.2.3.2 Hiérarchiser et renommer. Le client obtiendra l'argent demandé que s'il a récupéré sa carte. Afin de réaliser cet objectif, il est nécessaire de réorganiser les phrases : la phrase R6 doit être intégrée dans la décomposition de R5-1. Elle est renommée en R5-1-4. Nous nous focalisons ici sur la notion de *mise à jour de la hiérarchie* des besoins. Cette hiérarchie entre les quatre phrases, filles de R5-1, doit être précisée comme présenté dans la figure 5.4. Les évolutions de ces phrases sont présentées en gras.

NB. Afin de garder une trace du besoin R6, nous mettant son identifiant en commentaire - de la forme *"// R6"* - pour son nouvel identifiant R5-1-4.

ID	*Description*
R5-1	Soit la banque accepte la demande du client :
R5-1-1	La banque [maj_compte] du client
R5-1-2	Le distributeur [sort_argent] pour le client
R5-1-3	Le distributeur [imprime_reçu] pour le client
R5-1-4 // R6	**Le distributeur rend la carte au client qui la récupère pendant une durée d'attente**

FIGURE 5.4 – Intégration d'un nouvel enfant de R5-1

5.3.2.3.3 Ordonner. La mise à jour de la hiérarchie des phrases n'est pas suffisante pour résoudre le problème de la perte des cartes bancaires. Il faut indiquer l'ordre dans lequel les filles de R5-1 doivent être prises en compte lors de l'exécution des actions pour le retrait de l'argent du distributeur. Nous exprimons l'ordre informellement dans le CdC par l'introduction des numéros, allant de 1 à 4, indiquant l'enchaînement des opérations, voir figure 5.5.

ID	*Description*
R5-1	Soit la banque accepte la demande du client :
R5-1-1	**4-** La banque [maj_compte] du client
R5-1-2	**2-** Le distributeur [sort_argent] pour le client
R5-1-3	**3-** Le distributeur [imprime_reçu] pour le client
R5-1-4 // R6	**1-** Le distributeur rend la carte au client qui la récupère pendant une durée d'attente

FIGURE 5.5 – Réorganisation des besoins

5.3.2.3.4 Extraire des éléments de validation. La phrase informelle R5-1-4 précise deux actions nécessaires, à savoir *rend la carte* et *récupère la carte*, voir la table 5.3. Cette phrase fait apparaître le temps pour définir la durée d'attente durant laquelle le client récupère sa carte. Dans un déroulement normal, où le client respecte ce délai, la carte n'est pas perdue. Cependant, il se peut que le client ne récupère pas sa carte durant cette durée. Que ce passe-t-il dans ce cas ? En analysant le besoin R5-1-4,

nous constatons qu'il n'est pas suffisamment précis pour répondre à notre question. Pour cela, nous rajoutons une nouvelle action *"conserve la carte"* liée à la durée d'attente : si cette durée est dépassée, le distributeur conserve la carte du client. Cette dernière action permet de faire la différence entre l'action faite par le distributeur et celle faite par le client :

- le client récupère sa carte ou
- le distributeur la conserve dans sa boîte afin d'être remise à la banque concernée.

La nouvelle action est rajoutée comme élément de validation de type Fonctionality dans la table 5.3 mais aussi dans le besoin R5-1-4. L'énoncé de ce besoin devient le suivant :

ID	*Description*
R5-1-4 // R6	1- Le distributeur rend la carte au client qui la récupère pendant une durée d'attente. **A défaut, le distributeur conserve la carte.**

ID	*Term_Type*	*Terms*	*Event-B Model*
R5-1-4 // R6	Fact	durée d'attente	
	Functionality	- rend la carte - récupère la carte - conserve la carte	
	Behavior	- Le distributeur rend la carte au client –> le client la récupère ou - Le distributeur rend la carte au client –> le distributeur conserve la carte	

TABLE 5.3 – Prise en compte de la validation

5.3.2.4 Remplacer des éléments du système

Spec. Nous partons de la phrase R5-1-4 et spécifions le nouvel événement *rend_carte*. Il remplace l'événement *ejecte_carte*.

CdC. Avec ProR, le texte informel *"rend la carte"* est automatiquement remplacé par le terme formel *[rend_carte]* dans la phrase R5-1-4.

Liens. Le glossaire est mis à jour par un nouveau couple comme suit :

Formal term	*Informal description*
rend_carte	rend la carte

Outils

— *Validation.* Les éléments de validation de la table 5.3 sont mis à jour par l'introduction du terme formel *[rend_carte]* pour remplacer l'élément formel correspondant.

— *Vérification.* Nous ne disposons pas des informations suffisantes pour l'effectuer ; le développement de DAB 2 n'est pas vérifiable en l'état.

5.3.2.5 Décomposer la phrase R5-2

L'étape précédente nous a conduit à revoir la phrase R6 de la figure 5.1 pour satisfaire le premier cas où la banque accepte la demande du client. Cette phrase est modifiée, renommée en R5-1-4 et intégrée en tant qu'enfant de R5-1. Maintenant, regardons le deuxième cas où la banque refuse la demande du client. La carte doit être rendue au client ou conservée par le distributeur après la durée d'attente. Cette contrainte, exprimée par la phrase R5-1-4, doit aussi être prise en compte dans ce deuxième cas. Pour y parvenir, la phrase R5-2 est décomposée en deux besoins, voir figure 5.6. Le CdC mis à jour est présenté dans la partie droite de cette figure.

ID	*Description*
***** DAB 1 *****	
R5-2	Soit la banque refuse la demande du client : le distributeur [affiche_message_refus] au client
R6	Le distributeur [ejecte_carte]

ID	*Description*
***** DAB 2 = DAB 1 + Le distributeur ne permet pas la perte de la carte bancaire du client *****	
R5-1	Soit la banque accepte la demande du client :
R5-1-1	4- La banque [maj_compte] du client
R5-1-2	2- Le distributeur [sort_argent] pour le client
R5-1-3	3- Le distributeur [imprime_recu] pour le client
R5-1-4 // R6	1- Le distributeur [rend_carte] au client qui la récupère pendant une durée d'attente. A défaut, le distributeur conserve la carte.
R5-2	**Soit la banque refuse la demande du client :**
R5-2-1	**Le distributeur [affiche_message_refus] au client**
R5-2-2	**Le distributeur [rend_carte] au client qui la récupère pendant une durée d'attente. A défaut, le distributeur conserve la carte.**

FIGURE 5.6 – Décomposition de besoins

5.3.2.6 Introduire de nouvelles informations

5.3.2.6.1 Nouveaux événements

Spec. Il s'agit de spécifier l'événement [*recupere_carte*] effectué par le client.

Event *recupere_carte* // Client
when
...
then
act1 : carte := pas_de_carte
act2 : carte_recuperee := TRUE
end

CdC. Le texte informel *"la récupère"* est remplacé par le texte formel [*recupere_carte*] dans les besoins R5-1-4 et R5-2-2 de la figure 5.6. A noter ici que dans ce texte informel, le terme *"la"* se situant avant le verbe *"récupère"* désigne *"la carte"*

Liens. Avec l'arrivée du nouvel événement, le glossaire est mis à jour par le couple *(recupere_carte, la récupère)*.

Validation. Le client *récupère la carte*. Ce terme informel est lié à l'événement *recupere_carte*. Cette information de validation est prise en compte par la Spec et la table 5.3 est mise à jour par ce nouveau terme formel.

5.3.2.6.2 Temps

Spec. Elle est caractérisée par l'expression du concept *temps* via l'introduction :

- d'une constante *t_duree_carte* indiquant la durée maximale d'attente pendant laquelle le client doit récupérer sa carte,
- d'une variable *t_carte* pour mémoriser la durée pendant laquelle le client récupère sa carte et
- de deux événements *tick* et *avale_carte_temps_ecoule* décrivant respectivement un compteur qui incrémente le temps et l'action de conservation de la carte bancaire non-récupérée par le client.

CdC. Le texte informel *"une durée d'attente"* est remplacé par *[t_duree_carte]* dans les phrases *R5-1-4* et *R5-2-2*, voir la figure 5.8.

Liens. Le glossaire est mis à jour avec le couple *(t_duree_carte, une durée d'attente)*.

Validation. Le *temps* est un paramètre pour gérer la durée de récupération de la carte. Il est pris en compte par la Spec via la constante *t_duree_carte*, la variable *t_carte* et l'événement *tick*.

5.3.2.6.3 Invariant de collage

Spec. En plus des invariants de typage, un invariant de collage est introduit. Il exprime le fait que le distributeur ne contient pas de carte lorsque le client l'a récupérée :
glu_inv : carte_recuperee = TRUE $\Rightarrow$ *carte = pas_de_carte*

CdC. Le collage est exprimé à travers la relation de hiérarchie entre besoins.

Liens. Le glossaire est mis à jour automatiquement.

Validation. Nous extrayons des obligations exprimées sous forme de pré et post-conditions :
pré-condition : carte_recuperee = TRUE
post-condition : carte = pas_de_carte
Ces deux conditions sont prises en compte par la Spec via l'invariant *glu_inv*.

5.3.2.7 Exprimer l'ordre et la hiérarchie

Spec. La hiérarchie s'exprime par la notion de raffinement entre éléments formels, voir figure 5.7.

Exemple. L'événement *sort_argent′* est un enfant de l'événement abstrait *sort_argent* vu qu'il entre dans ces détails.

Quant à la notion d'ordre, elle est décrite via des invariants et des gardes.

Exemple. L'événement *s*ort_argent′ introduit la garde
grd_carte : carte_recuperee = TRUE
assurant que cet événement s'effectue après la récupération de la carte.

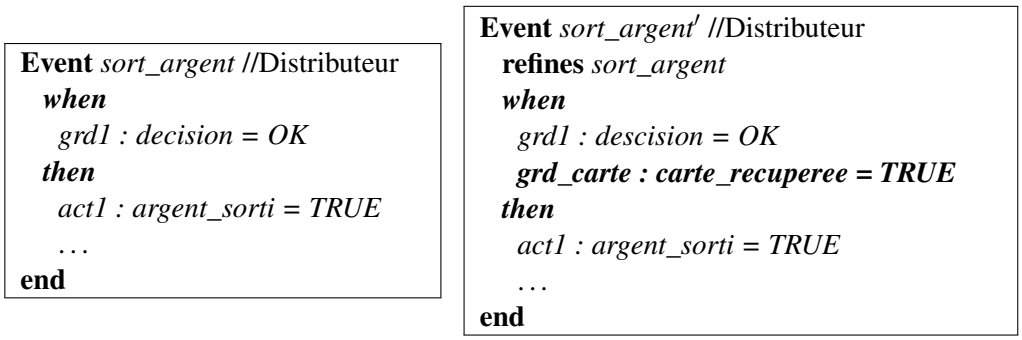


FIGURE 5.7 – Expression de l'ordre dans un événement raffiné

CdC. La hiérarchie s'exprime avec la notion d'*indentation* entre les besoins pour indiquer qu'un besoin enfant donne plus de détails qu'un besoin parent. L'ordre s'exprime via l'informel à travers la numérotation introduite au début des besoins comme pour les filles de R5-1.

Liens. La notion d'ordre ou de hiérarchie n'apparaît pas explicitement dans le glossaire.

Validation. Le distributeur remet la carte au client avant de lui remettre l'argent demandé. Ceci est exprimé par un *ordre d'enchaînement des événements* simulés en utilisant l'outil ProB :

rend_carte $\rightarrow$ *recupere_carte* $\rightarrow$ *sort_argent'* $\rightarrow$ *imprime_recu* $\rightarrow$ *maj_compte*

5.3.3 Bilan

5.3.3.1 Système résultant

Il est présenté dans la figure 5.8. La Spec de DAB 2 est correcte. Elle est vérifiée moyennant des OPs générées automatiquement par les outils disponibles dans la plateforme Rodin. Les éléments de validation accompagnant ce système ont également été pris en compte.

5.3.3.2 Points retenus

L'étude de l'intégration du besoin relatif à la perte des cartes bancaires nous a permis de mettre en compte les deux notions importantes :

- l'ordre et son expression dans les composants du système et

- la hiérarchie, exprimée par la relation parent-enfant ou frère-frère, et sa description aussi bien entre les besoins qu'entre les éléments formels de la Spec.

CdC

ID	*Description*
***** DAB 2 = DAB1 + Le distributeur ne permet pas la perte de la carte bancaire du client *****	
R1	Le client [insere_carte] dans le distributeur
R2	Le client [entre_pin] dans le distributeur
...	...
R5	La banque donne sa decision au client via le distribueur :
R5-1	Soit la banque accepte la demande du client :
R5-1-1	**4-** La banque [maj_compte] du client
R5-1-2	**2-** Le distributeur [sort_argent'] pour le client
R5-1-3	**3-** Le distributeur [imprime_recu] pour le client
R5-1-4 // R6	**1-** Le distributeur [rend_carte] au client qui [recupere_carte] pendant [t_duree_carte]. A défaut, le distributeur [avale_carte_temps_ecoule]
R5-2	Soit la banque refuse la demande du client :
R5-2-1	Le distributeur [affiche_message_refus] au client
R5-2-2	Le distributeur [rend_carte] au client qui [recupere_carte] pendant [t_duree_carte]. A défaut, le distributeur [avale_carte_temps_ecoule]
R7	Le distributeur [termine_transaction]

Glossaire

Formal term	*Informal description*
sort_argent'	sort l'argent
rend_carte	rend la carte
recupere_carte	recupère la carte
t_duree_carte	une durée d'attente
avale_carte_temps_ecoule	conserve la carte

Spec

CONTEXT DAB2_Ctx
EXTENDS DAB1_Ctx
CONSTANTS
t_duree_carte
AXIOMS
axm1 : t_duree_carte ∈ ℕ

MACHINE DAB2_Mch
REFINES DAB1_Mch
SEES DAB1_Ctx
VARIABLES
carte, code, decision, message, nb_essai_pin
t_carte, carte_recuperee
INVARIANTS
t_carte-Typ : t_carte ∈ ℕ
carte_recup-Typ : carte_recuperee ∈ BOOL
glu_inv : carte_recuperee = TRUE ⇒ carte = pas_de_carte
EVENTS
Initialisation

Event insere_carte //Client
Event entre_pin //Client
Event maj_compte //Banque
Event sort_argent' //Distributeur
Event imprime_recu //Distributeur
Event affiche_message_refus //Distributeur
Event rend_carte //Distributeur
Event recupere_carte //Client
Event avale_carte_temps_ecoule //Distributeur
Event tick //Distributeur
Event termine_transaction //Distributeur
...
END

FIGURE 5.8 – Système DAB 2

5.4 Clients malvoyants

Nous partons du système DAB 2 décrit par la figure 5.8.

5.4.1 Analyse du nouveau besoin

Soit le cahier des charges du nouveau système nommé *DAB 3* :

DAB 3. DAB 2 + *Le distributeur sert les clients malvoyants*

L'analyse du nouveau besoin

"Le distributeur sert les clients malvoyants"

nous guide par les questions présentées dans la table 5.1. Une des questions, Q1, s'intéresse au pourquoi de ce nouveau besoin : le DAB doit faciliter l'accès aux services offerts pour des personnes particulières telles que les malvoyants. Après plusieurs échanges, des évolutions sont apportées à ce nouveau besoin. Un client malvoyant est un client de la banque et a les mêmes droits d'accès aux services comme tout client.

5.4.2 Évolution du système

5.4.2.1 Décrire la notion de "client"

Nous raffinons le système existant DAB 2 pour développer le nouveau système DAB 3. Le nouveau besoin indique qu'il existe un nouveau type de client, le malvoyant. Ceci nous guide pour rajouter une abstraction de la notion du client : il peut être *malvoyant* ou pas. L'introduction de cette notion de client est accompagnée par la reprise des fonctionnalités existantes dans DAB 2 en les adaptant pour le malvoyant. Une mise à jour du système est observée de la manière suivante :

Spec. Une machine *DAB3_Mch* raffine la machine *DAB2_Mch* et utilise le contexte *DAB3_Ctx*. L'introduction de la notion de client affecte le reste des éléments de la Spec par :

— l'introduction d'une nouvelle variable *client*. Elle est de type *CLIENTS* déclaré dans le contexte *DAB3_Ctx*, voir figure 5.9 et

CONTEXT *DAB3_Ctx*
EXTENDS
DAB2_Ctx
SETS
CLIENTS
CONSTANTS
cl_autre, cl_malvoyant
AXIOMS
axm1 : partition (CLIENTS, {cl_autre}, {cl_malvoyant})
END

MACHINE *DAB3_Mch*
REFINES
DAB2_Mch
SEES
DAB3_Ctx
VARAIBLES
carte, code, decision
message, nb_essai_pin
t_carte, carte_recuperee
client
INVARIANTS
inv1 : client ∈ CLIENTS
...
END

FIGURE 5.9 – Démarrage de la spécification de DAB 3

— la mise à jour des événements hérités de *DAB2_Mch* par l'ajout d'une garde indiquant le type de client concerné. Par exemple, dans la figure 5.10, l'événement *affiche_message_refus* est raffiné par l'événement *affiche_message_refus′* dans lequel est introduite une garde *grd8*. Cette garde précise que le client doit être sans problème visuel.

Event *affiche_message_refus′* // Distributeur
  Refines
    affiche_message_refus
  any
    m
  where
    grd1 : m ∈ MSG // pour ce type, voir le contexte dans la figure 5.1
    ...
    grd8 : client = cl_autre // un client sans difficultés visuelles
  then
    act1 : message := m
end

FIGURE 5.10 – Événement évolué

CdC. De nouveaux besoins se rajoutent dans ce document. Ils permettent de distinguer le malvoyant des autres clients. Notons que le terme informel *"client"* existant dans DAB 2 doit être précisé : il signifie un *"client qui n'a pas de problème visuel"*. Par conséquent, les besoins de DAB 2 sont mis à jour dans DAB 3 par *[cl_autre]* qui remplace ce terme informel *"client"*. La figure 5.11 montre un aperçu sur les besoins introduits :

— *R-Client* décrit une abstraction du concept client en explicitant les deux types de client,

— *RMV0* exprime un besoin concernant un client malvoyant et

— les mises à jour des deux besoins *R1* et *R2*.

ID	*Description*
R-Client	**La banque sert des [cl_malvoyant] et des [cl_autre]**
RMV0	**Le [cl_malvoyant] se présente devant le distributeur**
...	...
R1	Le **[cl_autre]** [insere_carte] dans le distributeur
R2	Le **[cl_autre]** [entre_pin] dans le distributeur

FIGURE 5.11 – Mise à jour du CdC et introduction de nouveaux besoins

Liens. Le glossaire est mis à jour par l'introduction de trois couples de termes :

Formal term	*Informal description*
cl_autre	client
cl_malvoyant	client malvoyant
affiche_message_refus′	affiche un message de refus

5.4.2.2 Nouvelles fonctionnalités

Le nouveau besoin a permis de se poser la question *"comment un client malvoyant communiquera-t-il avec le distributeur ?"*. Le terme *"communiquer"* concerne à la fois l'entrée des informations par ce client et les messages de retour qu'il reçoit de la part du distributeur. Afin de répondre à cette question, nous prenons en compte l'aspect auditif de ce type particulier de client. L'écran du distributeur doit être désactivé et remplacé par des messages vocaux. Nous mettons en place de nouveaux éléments assurant ces fonctionnalités de la manière suivante :

Spec. — du côté du client, le distributeur doit savoir de quel type de client s'agit-il. Par exemple, le client malvoyant insère des écouteurs et le client n'ayant pas de problème de vue utilise l'écran. Par conséquent, la phrase *RMV0* du CdC de la figure 5.11 se précise : la séquence de termes *"se présente devant le distributeur"* est remplacée par le texte *"insère ses écouteurs dans le distributeur"*. Ce texte se formalise par un événement *insere_ecouteurs*, voir figure 5.12,

— du côté de la banque, pas de modification des données ou des fonctionnalités et

— du côté du distributeur, de nouvelles informations se rajoutent comme celles liées au guidage oral du client et à la saisie des données via un clavier braille. La figure 5.12 décrit une vue d'ensemble sur la signature de quatre événements réalisant ces services.

MACHINE *DAB3_Mch*
VARAIBLES
carte, code, decision, message, nb_essai_pin, t_carte, carte_recuperee // variables de DAB 2
client, ecran, clavier_braille, guidage_oral, message_vocal, ecouteurs // nouvelles informations
EVENTS
Event ***insere_ecouteurs*** // le client mal-voyant insère ses écouteurs
Event ***desactive_ecran*** // le distributeur désactive l'écran
Event ***active_guidage_oral*** // le distributeur active le guidage oral
Event ***active_clavier_Braille*** // le distributeur désactive le clavier normal et activer le clavier braille
Event ***envoie_message_vocal*** // le distributeur communique vocalement avec le client mal-voyant
...
END

FIGURE 5.12 – Étape intermédiaire du développement de DAB 3

CdC. Il évolue par l'introduction des termes formels dans son texte informel. Une version finale de ce document est fournie par la figure 5.13.

Liens. Le glossaire est mis à jour, voir figure 5.13.

5.4.3 Détections des lacunes

- Nous nous situons dans une catégorie des besoins oubliés : ceux qui sont évidents mais non-décrits dans le cahier des charges [Abrial, 2010]. Grâce au nouveau besoin, nous détectons des oublis dans les systèmes DAB 1 et DAB 2. Ces oublis concernent les échanges entre le client et l'interface du distributeur. Mis à part l'affichage du message de refus au client, aucun autre affichage

n'a été signalé dans la description de ces deux systèmes. Revenons à DAB 2 (figure 5.8) et examinons la phrase :

R2. Le client [entre_pin] dans le distributeur

Cette phrase ne prévient pas le client sur ce qu'il doit faire. Il s'agit d'un *besoin évident oublié*. Ce dernier s'intéresse à la description des échanges entre l'interface du distributeur et le client via des *messages affichés sur l'écran*. Nous avons détecté cet oubli lors de la prise en compte des clients malvoyants dans DAB 3 en se posant la question sur la manière de communication des deux acteurs.

- Nous détectons des *imprécisions* relativement aux entrées des données du client. La manière dont ces données doivent être introduites dans le distributeur n'est pas précisée dans les deux systèmes précédents. Pour un client sans problème visuel, ces données sont saisies directement via un écran tactile ou via un clavier séparé de l'écran du distributeur. Pour un client malvoyant, l'aspect visuel est absent. C'est pour cette raison que nous suggérons l'utilisation du clavier braille. Notons que dans des versions actuelles des distributeurs de billets, le clavier normal peut servir ces clients malvoyants en se basant sur des signes comme *O, X* et *V* gravés sur les touches de ce clavier.

- Nous détectons une *ambiguïté* dans les deux systèmes DAB 1 et DAB 2 concernant le terme *"client"*. Bien que ce terme a un sens très large, ces deux systèmes ne servent que des clients n'ayant pas de difficultés visuelles. L'introduction du besoin pour les clients malvoyants a mis en question cette notion de *"client"* et lui apporte une précision.

5.5 Conclusion

Dans ce chapitre, nous abordons le problème de la prise en compte d'un nouveau besoin *Req* dans un système existant. L'étape d'analyse de ce besoin est basée sur la compréhension et l'extraction de termes nécessaires au développement. Les outils disponibles dans la plateforme Rodin ont un rôle important tout au long du processus de développement. Nous avons utilisé l'outil ProR pour l'écriture des besoins, leur hiérarchisation et la mise à jour du glossaire. Nous utilisons les outils de vérification et de validation à savoir les générateurs d'obligations de preuves, les prouveurs, l'animateur et le model-checker ProB pour assurer la correction de la Spec. Le choix de l'étude de cas pour montrer et appliquer notre méthodologie est ciblé : bien qu'il s'agit d'un exemple "simple et classique", il nous a permis de montrer les découvertes liées au problème de prise en compte de nouveaux besoins dans des systèmes existants. Le système de distributeur DAB est hybride, critique et peu complexe. Nos découvertes, liées essentiellement aux oublis et aux imprécisions, sont applicables pour d'autres systèmes plus complexes. Les leçons dégagées de notre approche concernent :

- L'étape d'analyse et de compréhension d'un nouveau besoin. Elle est essentielle pour découvrir les informations nécessaires à la suite du développement.
- L'introduction d'un nouveau besoin dans un système existant. Il aide à déceler des lacunes dans ce système.
- La réutilisation de l'existant. Elle doit être exploitée avec précaution via des questions qui se posent au fur et à mesure du développement.
- Les questions. Quelque soit le choix pris avec l'arrivée d'un nouveau besoin, l'importance c'est de se poser des questions pour avancer dans le développement. Ces questions aident à comprendre

l'intérêt du nouveau besoin et favorisent son intégration dans l'existant. Elles concernent la mise à jour de la hiérarchie des besoins, la place et la manière d'intégrer ce besoin, son interaction avec les autres besoins du système, l'effet de son introduction sur l'existant et les découvertes qui accompagnent son intégration.

Pour la suite de notre travail, il est important de décrire rigoureusement les paramètres intervenant dans la prise en compte d'un nouveau besoin par rapport à un système existant. Ces paramètres serviront à semi-automatiser cette prise en compte.

CdC

ID	*Description*
***** DAB 3 = DAB 2 + Le distributeur sert les clients mal-voyants *****	
R-Client	La banque sert des [cl_malvoyant] et des [cl_autre]
RMV0	Le [cl_malvoyant] [insere_ecouteurs]
RMV0-1	Le distributeur [desactive_ecran] et [active_guidage_oral]
RMV0-2	Le distributeur [active_clavier_braille] et [desactive_clavier_normal]
RMV1	Le distributeur [envoie_m_vocal(entrez_carte)] au [cl_malvoyant] qui [insere_carte]
...	...
R1	Le [cl_autre] [insere_carte] dans le distributeur
R2	Le [cl_autre] [entre_pin] dans le distributeur

Glossaire

Formal term	*Informal description*
cl_autre	client (autre que malvoyant)
cl_malvoyant	client malvoyant
insere_ecouteurs	insère ses écouteurs dans le distributeur
desactive_ecran	désactive l'écran
active_guidage_oral	active le guidage oral
active_clavier_braille	active le clavier braille
desactive_clavier_normal	désactive le clavier normal
envoie_m_vocal	envoie un message vocal
entrez_carte	veuillez insérer votre carte

Spec

CONTEXT DAB3_Ctx
EXTENDS DAB2_Ctx
SETS
CLIENTS, MSG_VOCAL, ETAT_ECOUTEURS, ON_OFF
CONSTANTS
cl_autre, cl_malvoyant
bienvenue, entrez_carte, carte_en_sortie, solde_insuffisant, montant_depasse_plafond
branches, debranches
on, off
AXIOMS
axm1 : partition(CLIENTS, {cl_autre}, {cl_malvoyant})
axm2 : partition(MSG_VOCAL, {bienvenue}, {entrez_carte}, {carte_en_sortie}, {solde_insuffisant}, {montant_depasse_plafond})
axm3 : partition(ETAT_ECOUTEURS, {branches}, {debranches})
axm4 : partition (ON_OFF, {on}, {off})

MACHINE DAB3_Mch
REFINES DAB2_Mch
SEES DAB2_Ctx
VARIABLES
carte, code, decision, message, nb_essai_pin, t_carte, carte_recuperee
client, ecran, clavier_braille, guidage_oral, message_vocal, ecouteurs
INVARIANTS
t_carte-Typ : t_carte ∈ ℕ
ecran-Typ : ecran ∈ ON_OFF
clavier_Braille-Typ : clavier_braille ∈ ON_OFF
guidage_oral-Typ : guidage_oral ∈ ON_OFF
message_vocal-Typ : message_vocal ∈ MSG_VOCAL
ecouteurs-Typ : ecouteurs ∈ ETAT_ECOUTEURS
inv1 : guidage_oral = on ⇒ ecran = off ∧ ecouteurs = branches

EVENTS
Initialisation
Event insere_carte // Client sans difficultés visuelles
Event entre_pin // Client sans difficultés visuelles
...
Event insere_ecouteurs // Client malvoyant
Event desactive_ecran // Distributeur
Event active_guidage_oral // Distributeur
Event active_clavier_braille // Distributeur
Event desactive_clavier_normal // Distributeur
Event envoie_m_vocal // Distributeur
...
END

FIGURE 5.13 – Une partie du développement de DAB 3

Chapitre 6

Patron pour la description conditionnelle d'un besoin

Sommaire

Dans ce chapitre, nous décrivons *Dev-if*, un patron de développement pour le système < CdC, Liens, Spec >. Ce patron concerne les besoins décrits sous une forme conditionnelle et aide à la semi-automatisation du développement. Il est utilisé pour l'étude de cas d'un système de contrôle d'une machine d'hémodialyse.

Les résultats de ce chapitre ont fait l'objet de la publication [Sayar and Souquières, 2017].

6.1 Description

6.1.1 Forme générique conditionnelle

Rappelons que selon [Su et al., 2011], un cahier des charges est souvent décrit en deux parties distinctes :

- explicative regroupant des besoins sous forme de paragraphes, figures, tables et formules et

- référentielle contenant des exigences souvent décrites par un informaticien sous forme de phrases.

Pour certaines applications, les besoins de la partie référentielle sont décrits sous une forme particulière. Par exemple, pour la machine d'hémodialyse [Mashkoor, 2016], cette forme est essentiellement conditionnelle et s'intéresse aux cas d'échecs des composants physiques de la machine [48], voir figure 6.1. Un autre exemple est celui du train d'atterrissage d'un avion [Boniol and Wiels, 2014] dans lequel la partie référentielle décrit des contraintes sur :

- le fonctionnement normal du système dans un mode nominal via des phrases exprimées sous forme d'une *séquence d'actions* et
- le traitement des cas d'échecs dans un mode d'urgence via des besoins dont la majorité est décrite en suivant une forme *conditionnelle*.

En étudiant ces deux applications, nous définissons *Dev-if*, un patron de développement pour la forme conditionnelle des besoins. Ce patron assure les tâches suivantes :

- génération automatique d'une partie du développement,
- simplification de certaines tâches comme la vérification et la validation et
- indication des parties à compléter.

R-5	*During initiation, if the software detects that the pressure at the VP transducer exceeds the upper pressure limit, then the software shall stop the BP and execute an alarm signal.*
R-6	*During initiation, if the software detects that the pressure at the VP transducer falls below the lower pressure limit, then the software shall stop the BP and execute an alarm signal.*
R-8	*During initiation, if the software detects that the pressure at the AP transducer falls below the lower pressure limit, then the software shall stop the BP and execute an alarm signal.*
R-10	*While connecting the patient, if the software detects that the pressure at the VP transducer falls below the defined lower pressure limit for more than 3 seconds, then the software shall stop the BP and execute an alarm signal.*
R-21	*If the machine is in the initiation phase and if the temperature falls below the minimum temperature of 33C, then the software shall disconnect the dialyser from the DF and execute an alarm signal.*

FIGURE 6.1 – Quelques exigences dans le cahier des charges de l'hémodialyse

Nous décrivons la forme générique conditionnelle des besoins comme suit :

$$S_{p1}\ if\ condition(S_{p2})\ then\ action(S_{p3}) \tag{6.1}$$

dans laquelle :

- S_{p1} décrit l'ensemble des paramètres définissant l'environnement du besoin. Dans cet environnement, les clauses *if* et *then* sont applicables. Dans l'application de l'hémodialyse, S_{p1} est toujours composé d'un seul paramètre. Cependant, dans d'autres études de cas, nous avons identifié d'autres paramètres.

48. Cette étude de cas a été proposée dans la conférence ABZ 2016 : `https://www.scch.at/en/rse-news/abz-2016`

- S_{p2} désigne l'ensemble des paramètres utilisés dans la clause *if*.
- S_{p3} représente les paramètres décrits dans la clause *then*. Ceux-ci sont utilisés pour décrire les actions réalisées par le système dans le cas où la condition est vraie.

La première difficulté rencontrée à cette étape se résume par la question suivante : *comment identifier les valeurs des paramètres dans un besoin conforme à la forme conditionnelle* (6.1) *proposée ?* Nous donnons un premier élément de réponse à cette question en proposant de :

1. distinguer les trois clauses du besoin : son environnement, sa clause *if* et sa clause *then*,
2. localiser dans chaque clause l'élément qui change de valeur. Une compréhension du besoin est très importante dans cette étape.

Exemple.

Prenons les deux exigences *R-6* et *R-10* de la figure 6.1. Ces exigences se ressemblent et concernent la pression dans un composant appelé *VP transducer* où *VP* signifie la *pression veineuse*. En suivant la forme (6.1), nous extrayons les différentes valeurs des paramètres de ces besoins, voir table 6.1. Notons que parmi les valeurs des paramètres de S_{p2} de R-10 il y a un paramètre appelé *"a duration"*. Celui-ci n'apparaît pas explicitement dans l'énoncé du besoin. Il découle de notre compréhension de la suite des termes *"for more than 3 seconds"*.

S_{pi} / R	S_{p1}	S_{p2}	S_{p3}
R-6	During initiation	the pressure at the VP transducer	- the BP - an alarm signal
R-10	While connecting the patient	- the pressure at the VP transducer - a duration	- the BP - an alarm signal

TABLE 6.1 – Valeurs informelles des paramètres dans deux besoins

6.1.2 Définition du patron

Dev-if concerne les besoins suivant la forme conditionnelle (6.1). Il est utilisé pour développer un besoin *R-new* relativement à un besoin *R-dev* qui lui ressemble dans sa forme et qui est développé.

Ce patron fournit un modèle générique et des opérations internes favorisant le déroulement de son instanciation. Le modèle générique est présenté dans la figure 6.2. Il est constitué des trois composants : CdC_{if}, $Spec_{if}$ et $Glossaire_{if}$.

6.1.2.1 $Spec_{if}$

Elle est décrite par la machine *R-new'_Mch* raffinant la machine *R-dev_Mch* et utilisant le contexte *R-new'_Ctx*. Il est à noter que *R-dev_Mch* correspond à la machine associée au besoin *R-dev*.

- *Constantes et variables.* Chaque paramètre *pi* de l'ensemble $S_{pi} \mid i \in \{1,2,3\}$) est formalisé par trois éléments :

CdC_{if}	
ID	*Description*
R-new'	value([formal_p1]) if condition([formal_p2]) then action([formal_p3])
R-new'-init	[formal_p2] is [formal_p2_init]
R-new'-evolve	[formal_p2] can [change_formal_p2]
glue-R-new'-R-dev	...

$Glossaire_{if}$	
Formal term	*Informal description*
formal_p1	p1
formal_p2	p2
formal_p3	p3
formal_p2_init	initial value of p2
change_formal_p2	change p2

$Spec_{if}$

CONTEXT R-new'_Ctx
EXTENDS R-dev_Ctx
SETS
formal_p1_TYP, formal_p2_TYP, formal_p3_TYP
CONSTANTS
formal_p2_init,
formal_p1_val, formal_p2_val, formal_p3_val

AXIOMS
axm1 : partition(formal_p2_TYP, {formal_p2_init}, {formal_p2_val})
axm2 : partition (formal_p1_TYP, {formal_p1_val})
axm3 : partition (formal_p3_TYP, {formal_p3_val})

MACHINE R-new'_Mch
REFINES R-dev_Mch
SEES R-new'_Ctx
VARIABLES
formal_p1, formal_p2, formal_p3,
old_vars // old_vars coming from R-dev_Mch
INVARIANTS
R-new'-formal_p1-Typ : formal_p1 ∈ formal_p1_TYP
R-new'-formal_p2-Typ : formal_p2 ∈ formal_p2_TYP
R-new'-formal_p3-Typ : formal_p3 ∈ formal_p3_TYP
R-new'-normal : ¬(condition(formal_p2)) ∧ formal_p1 = formal_p1_val ⇒ ...
R-new'-anomaly : ... ⇒ condition(formal_p2) ∧ formal_p1 = formal_p1_val
glue-R-new'-R-dev : ... ⇒ condition(formal_p2) ∧ formal_p1 = formal_p1_val ∧ ...
EVENTS
Initialisation
begin
R-new'-init-formal_p1 : formal_p1 := formal_p1_val
R-new'-init-formal_p2 : formal_p2 := formal_p2_init
R-new'-init-formal_p3 : formal_p3 := formal_p3_val
end

Event change_formal_p2
any
value
where
grd1 : value ∈ formal_p2_TYP
then
R-new'-act : formal_p2 := value
end
Event treatment_R-new'
when
grd1 : formal_p1 = formal_p1_val
grd2 : condition(formal_p2)
then
R-new'-act1 : action(formal_p3)
end
Event glue_treatment_R-new'
when
grd1 : ... // condition on old_vars
grd2 : formal_p1 = formal_p1_val
grd3 : condition(formal_p2)
then
R-new'-act1 : action(formal_p3)
R-new'-act2 : ... // action in R-dev
end
END

FIGURE 6.2 – Composants de *Dev-if*

— une variable *formal_pi*,

— un type *formal_pi_TYP* pour la variable. Ce type est défini dans le contexte *R-new′_Ctx* et

— une constante *formal_pi_val* représentant une des valeurs prises par la variable correspondante.

Chaque paramètre *p2* de S_{p2} doit avoir une valeur initiale appelée *formal_p2_init*.

- *Invariants.* Mis à part les invariants de typage des variables, *Dev-if* formalise certaines propriétés du système via des prédicats. Nous disposons de trois invariants :

  — *R-new′-normal* décrivant une des propriétés évidentes et non exprimées explicitement dans le besoin : l'état de déroulement normal du futur système,

  — *R-new′-anomaly* exprimant l'état du système en cas d'échec. Ceci fait partie des propriétés exprimées explicitement par le besoin *R-new* et

  — *glue-R-new′-R-dev* décrivant le collage entre *R-new* et les autres besoins et/ou gérant la coopération entre les différentes parties du système. Cet invariant permet également d'assurer la préservation des propriétés abstraites de la spécification formelle par sa version concrète.

- *Événements.* Nous décrivons différents événements :

  — La valeur initiale des variables évolue pour garantir l'aspect dynamique de la spécification formelle. Ceci est réalisé par un événement nommé *change_formal_p2*.

  — Revenons sur l'objectif du patron *Dev-if* : la formalisation du besoin *R-new*. Ceci signifie que l'$action(S_{p3})$ sous la $condition(S_{p2})$ doit être formellement décrite. Cet objectif est accompli via un événement *treatment_R-new′* exprimant la condition à l'aide d'une garde nommée *grd2* et l'action avec sa partie *then*.

  — Nous décrivons le collage via ce que nous appelons *événements de collage* comme l'événement *glue_treatment_R-new′*.

- *Parties à compléter.* La $Spec_{if}$ intègre des "..." afin de signaler les parties où l'intervention manuelle du spécifieur est requise. *Dev-if* génère une partie automatique et des "..." dépendant d'un détail spécifique de l'application en cours et des choix de formalisation. Ces "..." permettent d'avertir le développeur pour qu'il n'oublie pas certaines parties essentielles telles que le collage et les propriétés décrites dans le besoin.

6.1.2.2 CdC_{if}

Il est décrit par :

- *R-new′*. C'est la forme réécrite de *R-new* dans laquelle les termes informels sont remplacés par leurs termes formels associés venant de la $Spec_{if}$.
- *R-new′-init.* Il s'agit d'un enfant de *R-new′* depuis lequel nous explicitons la relation de hiérarchie entre besoins. Il est destiné à donner une valeur initiale à tous les paramètres *p2* de S_{p2}. Avec ce besoin, nous soulignons un point important dans notre approche : expliciter un des besoins qui sont évidents et non mentionnés dans le cahier des charges du client [Abrial, 2010]. L'idée vient du fait que B événementiel exige que toute variable doit avoir une valeur initiale. *Dev-if* aide le développeur à générer des valeurs initiales de ces paramètres.
- *R-new′-evolve.* Nous devons trouver un chemin menant à un état où la $condition(S_{p2})$ est vérifiée. Ceci implique que les paramètres de S_{p2} doivent évoluer jusqu'à atteindre les valeurs souhaitées via ce besoin.

- *glue-R-new′-R-dev.* L'invariant *glue-R-new′-R-dev* et l'événement *glue_treatment_R-new′* sont les projections de ce besoin. Nous le décrivons avec des "..." vu que le collage dépend de la nature de l'application en cours d'étude et de la compréhension du développeur. Ce dernier doit les compléter manuellement après application du *Dev-if*.

6.1.2.3 $Glossaire_{if}$

Reliant le CdC_{if} avec la $Spec_{if}$, ce $Glossaire_{if}$ est constitué des couples *(Formal term, Informal description)*. Il est paramétré c'est-à-dire les paramètres $S_{pi} \mid i \in \{1,2,3\}$ de la forme conditionnelle (6.1) apparaissent dans sa description. Par exemple, le terme formel *formal_p1* est associé au terme informel *p1*, un élément de S_{p1}. Ce glossaire est automatiquement mis à jour quand *Dev-if* est appliqué. De même, une fois les "..." des deux autres composants sont complétés par le développeur, de nouveaux couples se rajoutent automatiquement au glossaire [Golra et al., 2018].

6.1.3 Utilisation

Le patron *Dev-if* prend comme opérandes un système $< CdC, Liens, Spec >$ et les besoins *R-dev* et *R-new*. Le *CdC* du système constitue toutes les exigences du système en cours de développement : celles qui sont formalisées en B événementiel telles que *R-dev* et celles qui ne sont pas développées telles que *R-new*. Les besoins développés sont réécrits en utilisant l'approche ProR [Jastram, 2010] ; ils contiennent des termes formels provenant de la *Spec*. Ce patron agit sur les opérandes citées ci-dessus pour fournir un nouvel état $< CdC', Liens', Spec' >$. Son application se résume par la formule mathématique suivante :

$$\textbf{Dev-if :} < CdC, Liens, Spec >, R-new, R-dev \rightarrow < CdC', Liens', Spec' > \qquad (6.2)$$

Interaction avec le développeur. L'application de *Dev-if* nécessite des entrées et fournit des sorties. Ces dernières concernent le système $< CdC', Liens', Spec' >$ et les *"..."* à compléter. Quant aux entrées, elles sont fixées en interagissant avec le développeur à travers des questions présentées dans la table 6.2. Au fur et à mesure de ces interactions, *Dev-if* effectue, en interne, des opérations telles que des affectations et des comparaisons entre anciens et nouveaux paramètres. Les comparaisons servent à éviter les redondances dans le résultat généré.

Question	***Entrées***
Q1.	Pour chaque paramètre *pi* dans *R-new*, préciser - un terme formel correspondant - un type à ce terme formel - une valeur du terme formel - une valeur initiale du terme formel
Q2.	Pour la $condition(S_{p2})$, saisir - sa représentation formelle - un nom à la condition
Q3.	Entrer la représentation formelle de l'$action(S_{p3})$

TABLE 6.2 – Entrées du patron *Dev-if* demandées au développeur

Un exemple détaillé des réponses fournies pour la question Q1 est décrit ultérieurement dans la section 6.2.

Collage. *Dev-if* utilise explicitement le collage issu du formel et plus précisément du raffinement à travers l'*invariant de collage* [49]. Dans la définition de ce patron, nous élargissons le champ de définition du collage pour couvrir d'autres notions :

- dans la *Spec*, nous ajoutons un concept d'*événement de collage* signifiant un traitement commun entre la machine abstraite et la machine raffinée.
- dans le *CdC*, ce concept peut avoir plusieurs significations qui dépendent de l'application en cours d'étude. Par exemples :
  - — pour le système d'hémodialyse, le collage concerne la formalisation de la réaction du système lorsque plus qu'une anomalie affecte les différents composants physiques en même temps. Il peut exprimer l'exclusion entre les cas et
  - — pour le train d'atterrissage d'un avion, ce concept exprime qu'on donne plus de détails sur les composantes du système.

6.1.4 Vérification et validation

Vérification. La correction du composant $Spec_{if}$ est prouvée : les obligations de preuve générées sont déchargées à l'aide des prouveurs automatiques et interactifs de la plateforme Rodin. Certaines d'entre elles ne peuvent pas être déchargées car elles concernent les "...". En adoptant ce patron, la tâche de vérification est automatiquement accomplie pour une partie du modèle formel en B événementiel de la même manière que celle proposée par les auteurs de [Hoang et al., 2013]. Trois OPs sont automatiquement déchargées pour la $Spec_{if}$ du *Dev-if*. Elles concernent essentiellement la préservation des invariants par les événements.
Validation. *Dev-if* est utilisé conjointement avec notre approche de validation pour la génération automatique des éléments de validation. Ces éléments sont génériques et ont été détaillés dans le chapitre 4.

Exemple. A partir de la figure 6.2, nous extrayons le scénario *générique* (6.3) suivant :

$$Initialisation \rightarrow (change\_formal\_p2)^* \rightarrow treatment\_R-new' \tag{6.3}$$

Il résume l'objectif du besoin *R-new* : en partant d'un état initial du système et en allant vers un état où la $condition(S_{p2})$ est évaluée à vrai, l'$action(S_{p3})$ s'effectue. Le symbole "*" signifie que l'événement en question (change_formal_p2) peut avoir zéro ou plusieurs occurrences.

6.2 Application de ce patron à l'hémodialyse

Cette étude de cas a pour objectif le développement d'un système de contrôle de la machine d'hémodialyse. La partie référentielle de son document d'exigences est décrite par trente-six besoins, quasiment tous décrits de la même forme conditionnelle. Nous allons étudier différents cas d'application de *Dev-if*. Chaque cas est identifié en localisant les différences entre les deux besoins en question : *R-new* qui va être développé et *R-dev* qui est développé. L'intérêt de l'étude de chaque cas est de montrer

49. Pour avoir plus de détails sur l'invariant de collage, voir la paragraphe *Machine* de la section 2.2.3.2, chapitre 2

l'apport de notre patron selon le degré de similarité entre les besoins. Afin de faciliter le développement des modèles formels en B événementiel de l'hémodialyse, nous avons choisi de formaliser un seul besoin à chaque niveau d'abstraction. Après avoir pris en compte un besoin, son modèle formel vérifié et validé est raffiné par un modèle associé à un autre besoin : c'est l'approche *descendante* du développement [50].

6.2.1 Initialiser le développement

Regardons l'application du patron en partant d'un développement initial. Commençons par le développement du besoin *R-5* de la figure 6.1. Dans l'énoncé de ce besoin, *BP* désigne la pompe à sang. L'état actuel du système de départ est décrit par :

- *CdC.* Il représente le document d'exigences informel contenant R-5.
- *Spec.* Aucun modèle formel n'est développé.
- *Liens.* Le glossaire est vide.

6.2.1.1 Construction du système

R-new correspond à R-5 et *R-dev* n'existe pas tant qu'il n'y a pas de spécification formelle développée à cette étape. Les valeurs informelles des paramètres de ce besoin sont décrites dans la table 6.3.

$\mathbf{S}_{pi}$	***Valeur***
S_{p1}	During initiation
S_{p2}	the pressure at the VP transducer
S_{p3}	- the BP - an alarm signal

TABLE 6.3 – Paramètres de R-5

En utilisant la formule mathématique (6.2), nous appliquons le patron *Dev-if* comme suit :

$$\textbf{\textit{Dev-if :}} < CdC, \varnothing, \varnothing > , R-5, \varnothing \rightarrow < CdC', Liens', Spec' > \tag{6.4}$$

Nous commençons par répondre aux questions présentées dans la table 6.2. A titre illustratif, les réponses à la question *Q1* sont fournies dans la table 6.4.

$\mathbf{S}_{pi}$	***formal_pi***	***formal_pi_TYP***	***formal_pi_value***	***formal_pi_init***
S_{p1}	phase	PHASES	initiat	initiat
S_{p2}	vp	$\mathbb{Z}$	// any value in $\mathbb{Z}$	0
S_{p3}	BP	BP_State	stopped	stopped
	alarm_vp	ALARMS	ALM_excess_vp	NULL

TABLE 6.4 – Réponses fournies au *Dev-if* relativement à R-5

50. Une version détaillée de l'application du patron *Dev-if* est fournie dans l'annexe B

Notre point de départ est un développement vide. Nous donnons une valeur initiale à chaque paramètre. Ceci est lié au choix du langage B événementiel pour lequel toutes les variables de la *Spec* doivent être initialisées. Une fois cette tâche effectuée, les paramètres de S_{p1} et de S_{p3} ne seront pas initialisés à nouveau dans les étapes suivantes de raffinement. Le système généré automatiquement est décrit dans la figure 6.3.

CdC

ID	*Description*
R-5'	[initiat] if [vp] exceeds [upper_press_limit] then stop [BP] and execute [alarm_vp]
R-5'-init	[vp] is 0 [BP] is [stopped]
R-5'-evolve	[vp] can [change_vp]

Glossaire

Formal term	*Informal description*
initiat	During initiation
vp	the pressure at the VP transducer
upper_press_limit	the upper pressure limit
BP	the BP
alarm_vp	an alarm signal
stopped	stop
change_vp	change the pressure at the VP transducer

Spec

CONTEXT R-5'_Ctx
SETS
PHASES, BP_State, ALARMS
CONSTANTS
initiat,
upper_press_limit,
stopped, NULL, ALM_excess_vp

AXIOMS
axm1 : partition (ℤ, {upper_press_limit})
axm2 : partition (PHASES, {initiat})
axm31 : partition (BP_State, {stopped})
axm32 : partition (ALARMS, {NULL}, {ALM_excess_vp})

MACHINE R-5'_Mch
SEES R-5'_Ctx
VARIABLES
phase, vp, BP, alarm_vp
INVARIANTS
R-5'-phase-Typ : phase ∈ PHASES
R-5'-vp-Typ : vp ∈ ℤ
R-5'-BP-Typ : BP ∈ BP_State
R-5'-alarm_vp-Typ : alarm_vp ∈ ALARMS
R-5'-normal : ¬(vp > upper_press_limit) ∧ phase = initiat ⇒ **...**
R-5'-anomaly : **...** ⇒ vp > upper_press_limit ∧ phase = initiat
EVENTS
Initialisation
begin
R-5'-init-phase : phase := initiat
R-5'-init-vp : vp := 0
R-5'-init-alarm_vp : alarm_vp := NULL
R-5'-init-BP : BP := stopped
end

Event change_vp
any
value
where
grd1 : value ∈ ℤ
then
R-5'-act : vp := value
end

Event treatment_R-5'
when
grd1 : phase = initiat
grd2 : vp > upper_press_limit
then
R-5'-act1 : BP := stopped ∧ alarm_vp :=ALM_excess_vp
end
END

FIGURE 6.3 – Développement de R-5

La séquence des mots *"execute an alarm signal"* est répétée dans toutes les exigences de cette étude de cas. Rappelons que l'objectif est de développer un système de contrôle qui réagit en cas d'échec d'un ou de plusieurs composants de la machine d'hémodialyse. Pour cela, il est important de distinguer les différentes erreurs afin d'aider l'utilisateur final à comprendre quel composant du système présente une anomalie. Pour y parvenir, nous avons effectué un choix d'affecter une alarme spécifique à chaque exigence et de lui donner une valeur expressive facilement interprétable. Par exemple, nous donnons le nom *"ALM_excess_vp"* à l'alarme décrite dans R-5 pour préciser que l'anomalie concerne un *excès de pression au niveau du transducteur VP*.

Dans cette étape, il n'y a pas de collage puisque nous partons d'une *Spec* vide. Le résultat donné automatiquement par ce patron est complété par le développeur en définissant les *"..."*. Ceux-ci concernent les deux invariants *R-5′-normal* et *R-5′-anomaly* de la figure 6.3. Par exemple, nous complétons ce dernier invariant comme suit :

R-5′-anomaly : **alarm_vp = ALM_excess_vp** $\Rightarrow$ vp > upper_press_limit $\wedge$ phase = initiat

Il exprime la propriété suivante : si l'alarme concernant la pression au composant transducteur VP est déclenchée, la valeur de la pression dans ce composant a dépassé la valeur maximale autorisée pendant la phase d'initiation.

6.2.1.2 Vérification et validation

- Pour la vérification, les OPs associées à la *Spec* sont toutes automatiquement déchargées.
- Concernant la validation, le scénario générique décrit par la formule (6.3) de la section **??** est instancié par les valeurs des paramètres de R-5 pour obtenir :

$$Initialisation \rightarrow (change\_vp)^* \rightarrow treatment\_R-5'$$

La *Spec* de R-5 prend en compte ce scénario en simulant la séquence d'événements correspondants à ce scénario sous ProB.

6.2.2 Clause if différente

Ce cas prend en compte les besoins *R-new* et *R-dev* ayant :

- des $S_{pi} \mid i \in \{1,2,3\}$ identiques,
- une même *action(*S_{p3}*)* et
- une *condition(*S_{p2}*)* différente.

Par exemple, nous traitons le besoin R-6 de la figure 6.1. En comparant ce besoin avec R-5 développé ci-dessus, nous trouvons que ces deux besoins ont les mêmes paramètres et diffèrent dans leur partie *condition(*S_{p2}*)*. L'énoncé de cette condition est *"the pressure at the VP transducer falls below the lower pressure limit"* dans R-6. L'application de *Dev-if* sur ce besoin, en utilisant le développement existant de R-5, génère automatiquement le système décrit par :

- *Spec.* Une nouvelle machine *R-6′_Mch* qui raffine *R-5′_Mch* est générée. Le patron introduit une nouvelle constante *lower_press_limit*. Il n'ajoute pas de nouvelle variable et prend en compte la condition(S_{p2}) dans les invariants et dans l'événement *treatment_R-6′*. La figure 6.4 détaille cet événement. Il n'y a toujours pas de collage dans cette étape de développement. La raison de son absence est que les deux exigences ont les mêmes paramètres et *vp* qui ne peut pas avoir deux valeurs en même temps.

Event treatment_R-6′
  when
    grd1 : phase = initiat
    grd2 : vp < lower_press_limit
  then
    R-6′-act1 : BP := stopped ∧ alarm_vp := ALM_deficit_vp
  end

FIGURE 6.4 – Un événement du développement de R-6

- *CdC.* Il est mis à jour par deux besoins : R-6′ représentant la forme réécrite de R-6 et son enfant nommé *R-6′-evolve* décrivant la diminution de *vp*. Ils sont décrits sous ProR comme présenté sur la figure 6.5.

ID	*Description*
R-6′	[initiat] if [vp] falls below [lower_press_limit] then stop [BP] and execute [alarm_vp]
R-6′-evolve	[vp] can [change_vp]

FIGURE 6.5 – Deux besoins du développement de R-6

- *Liens.* Il n'y a pas de nouveaux couples de termes ajoutés au glossaire.

6.2.3 Paramètres S_{p2} différents

Ce cas concerne les besoins ayant des paramètres S_{p2} différents, les mêmes $condition(S_{p2})$ et les mêmes $action(S_{p3})$. Développons le besoin R-8 en appliquant *Dev-if* et en utilisant le développement existant de R-6. La différence entre ces deux besoins est le paramètre S_{p2} :

- dans R-6, ce paramètre correspond à *"the pressure at the **VP** transducer"* et
- dans R-8, il correspond à *"the pressure at the **AP** transducer"*.

6.2.3.1 Evolution du système

S_{p1} et action(S_{p3}) sont les mêmes pour R-6 et R-8. Le paramètre de S_{p2} est différent. *Dev-if* génère automatiquement les composants suivants :

- *Spec.* Vu que nous avons un nouveau paramètre dans S_{p2}, deux nouvelles variables *ap* et *alarm_ap* et un événement *change_ap* sont introduits. Notre patron précise une nouvelle alarme liée à R-8. Cette alarme, appelée *ALM_deficit_ap*, est déclarée comme constante dans le contexte.
- *CdC.* Il est décrit dans la figure 6.6. Le besoin *glue-R-8′-R-6* est complété par le développeur : il s'intéresse au collage entre R-8 et R-6.

ID	*Description*
R-8′	[initiat] if [ap] falls below [lower_press_limit] then stop [BP] and execute [alarm_ap]
R-8′-init	[ap] is 0
R-8′-evolve	[ap] can [change_ap]
glue-R-8′-R-6	**[initiat] if [ap] and [vp] fall below [lower_press_limit] then stop [BP] and execute [alarm_ap] and [alarm_vp]**

FIGURE 6.6 – CdC de R-8

- *Liens.* Le glossaire est mis à jour automatiquement par trois couples de termes décrits dans la figure 6.7.

Formal term	*Informal description*
ap	the pressure at the AP transducer
alarm_ap	an alarm signal
change_ap	change the pressure at the AP transducer

FIGURE 6.7 – Glossaire de R-8

Le point important à souligner ici est la présence du collage. Comme mentionné dans la section 6.1, ce concept est exprimé par un invariant et un événement qui contiennent des parties à compléter par le développeur. Le collage exprime l'état du système lorsque la pression au niveau du transducteur AP et la pression au niveau du transducteur VP sont inférieures à la limite de pression la plus basse (*the lower pressure limit*). Les figures 6.8 et 6.9 montrent les contenus respectifs de l'invariant de collage *glue-R-8′-R-6* et de l'événement *glue_treatment_R-8′*. Les parties en gras représentent ce qui a été complété par le développeur, les autres parties sont automatiques.

glue-R-8′-R-6 : **alarm_ap = ALM_deficit_ap** $\wedge$ **alarm_vp = ALM_deficit_vp** $\Rightarrow$
ap < lower_press_limit $\wedge$ phase = initiat $\wedge$ **vp < lower_press_limit**

FIGURE 6.8 – Invariant de collage dans la Spec de R-8

Event glue_treatment_R-8′
  when
    grd1 : **vp < lower_press_limit** // condition in R-6
    grd2 : phase = initiat
    grd3 : ap < lower_press_limit
  then
    R-8′-act1 : BP := stopped $\wedge$ alarm_ap := ALM_deficit_ap
    R-8′-act2 : **alarm_vp := ALM_deficit_vp** // action in R-6
  end

FIGURE 6.9 – Événement de collage dans la Spec de R-8

6.2.3.2 Vérification et validation

- La *Spec* du développement de R-8 est automatiquement vérifiée en utilisant les outils disponibles sous la plateforme Rodin.
- Nous construisons un scénario fusionnant les deux besoins R-6 et R-8 :

$$Initialisation \rightarrow (change\_ap)^* \rightarrow (change\_vp)^* \rightarrow glue\_treatment\_R-8'$$

La *Spec* de R-8 prend en compte ce scénario en simulant, avec l'outil ProB, la séquence d'événements correspondants.

6.2.4 Paramètres S_{p1} différents

Ce cas traite les besoins R-new ayant des paramètres de S_{p1} différents de ceux de *R-dev* et S_{p2} qui est composé de plus d'un paramètre. Nous développons l'exigence R-10, décrite dans la figure 6.1, en utilisant le développement de R-8. Regardons de près les différences majeures entre ces deux besoins. Celles-ci sont extraites via une opération de comparaison interne au *Dev-if*. Dans R-10 :

- S_{p1} concerne *"While connecting the patient"*. Il s'agit d'une nouvelle phase de thérapie dans cette étude de cas (voir section 3.2 du [Mashkoor, 2016]).
- S_{p2} est composé de deux paramètres : *"the pressure at the VP transducer"* et *"a duration"*. Ce dernier est extrait d'après notre compréhension de la séquence des termes *"for more than 3 seconds"*.
- La *condition(*S_{p2}*)* est composée.

L'application du *Dev-if* sur R-10 génère automatiquement le système décrit par la figure 6.10 :

- *Spec.* Elle est présentée par la machine *R-10′_Mch*. En se basant sur les trois différences citées ci-dessus, notre patron se focalise sur :
  - S_{p1}. Une nouvelle constante *connect_patient* est introduite dans le contexte. Il n'y a pas de collage dans la *Spec* à cause de ce paramètre : R-10 est situé dans une phase temporellement différente de celle de R-8.
  - S_{p2}. Deux variables *vp* et *duration_vp* sont introduites. La première existe dans le développement de R-8 et la deuxième est nouvelle et est de type $\mathbb{N}$.
  - *condition(*S_{p2}*)*. Le patron agit sur les invariants et les gardes des événements. Par exemple, la garde *grd2* de l'événement *treatment_R-10′* est composée de deux parties.
- *CdC.* Les besoins associés à R-10 sont automatiquement générés. Ils contiennent des [termes_formels] intégrés dans leurs textes informels.
- *Liens.* Le glossaire est mis à jour par de nouveaux couples de termes.

CdC

ID	*Description*
R-10'	[connect_patient] if [vp] falls below [lower_press_limit] for [duration_vp] more than 3 seconds then stop [BP] and execute [alarm_vp]
R-10'-init	[duration_vp] is 0
R-10'-evolve	[duration_vp] can [change_duration_vp]

Glossaire

Formal term	*Informal description*
connect_patient	While connecting the patient
duration_vp	a duration
change_duration_vp	change a duration

Spec

CONTEXT R-10'_Ctx
EXTENDS R-8'_Ctx
CONSTANTS
connect_patient, ALM_deficit_vp_connect

AXIOMS
axm2 : partition (PHASES, {connect_patient})
axm3 : partition (ALARMS, {ALM_deficit_vp_connect})

MACHINE R-10'_Mch
REFINES R-8'_Mch
SEES R-10'_Ctx
VARIABLES
phase, vp, BP, alarm_vp
duration_vp
INVARIANTS
R-10'-duration_vp-Typ : duration_vp $\in \mathbb{N}$
R-10'-normal : $\neg$(vp $<$ lower_press_limit $\wedge$ duration_vp $>$ 3) $\wedge$ phase = connect_patient $\Rightarrow$ **...**
R-10'-anomaly : **...** $\Rightarrow$ vp $<$ lower_press_limit $\wedge$ duration_vp $>$ 3 $\wedge$ phase = connect_patient
EVENTS
Initialisation
begin
R-10'-init-phase : phase := connect_patient
R-10'-init-duration_vp : duration_vp := 0
end

Event change_duration_vp
any
value
where
grd1 : value $\in \mathbb{N}$
then
R-10'-act : duration_vp := value
end
Event treatment_R-10'
when
grd1 : phase = connect_patient
grd2 : vp $<$ lower_press_limit $\wedge$ duration_vp $>$ 3
then
R-10'-act1 : BP := stopped $\wedge$ alarm_vp :=ALM_deficit_vp_connect
end
END

FIGURE 6.10 – Système généré pour R-10

6.2.5 Cas général

Nous étudions le cas dans lequel les deux besoins ont des paramètres différents, des clauses *if* différentes et des clauses *then* différentes. Autrement dit, n'importe quel paramètre peut différer entre *R-new* et *R-dev*. Nous nous intéressons au développement de R-21, voir figure 6.1. Les paramètres de ce besoin sont :

- S_{p1}. Il est indiqué par la phrase *"If the machine is in the initiation phase"*. Selon notre compréhension, nous considérons qu'il a la même signification que la valeur de S_{p1} dans R-8 qui est *"During initiation"*.
- *condition*(S_{p2}). Elle est exprimée sous forme d'une séquence de termes *"the temperature falls below the minimum temperature of 33C"* dans laquelle S_{p2} concerne *"the temperature"*.

- *action(*S_{p3}*)*. Elle est composée par deux actions : *"disconnect the dialyser from the DF"* et *"execute an alarm signal"*. L'ensemble S_{p3} contient deux paramètres : *"the dialyser"* et *"an alarm signal"*.

Nous choisissons de développer R-21 en fonction du développement de R-8. Comme S_{p1} est commun à ces deux besoins, nous nous focalisons sur la *condition(*S_{p2}*)* et l'*action(*S_{p3}*)*. L'application de *Dev-if* génère :

- *Spec.* Elle est caractérisée par une machine *R-21′_Mch* qui utilise le contexte *R-21′_Ctx* et raffine *R-8′_Mch*. De nouvelles constantes *disconnected* et *ALM_low_temp* sont introduites dans ce contexte. Les trois parties du besoin R-21 sont prises en compte comme suit :
  - S_{p1}. Il existe déjà sous forme d'une constante nommée *initiat* dans le contexte *R-8′_Ctx*.
  - *condition(*S_{p2}*)*. Comme vu dans la section 6.2.2, *Dev-if* agit sur les invariants et les gardes des événements de la *Spec.*
  - *action(*S_{p3}*)*. Notre patron agit sur deux éléments : les actions des événements et les invariants. Par exemple, dans l'événement *treatment_R-21′* de la figure 6.11, l'action nommée *R-21′-act1* représente l'action composée *action(*S_{p3}*)* en utilisant un *"et logique* $\wedge$*"*.

Event treatment_R-21′
  when
    grd1 : phase = initiat
    grd2 : temperature < 33
  then
    R-21′-act1 : dialyser := disconnected $\wedge$ alarm_temp := ALM_low_temp
  end

FIGURE 6.11 – Un événement du développement de R-21

- *CdC.* Les besoins associés à R-21 sont automatiquement générés par *Dev-if*. Parmi ces besoins, il en existe un décrivant le collage avec R-8. Il a été initialement représenté par des *"…"*. La figure 6.12 montre une version complétée par le développeur.

ID	*Description*
glue-R-21′-R-8	**[initiat] if [temperature] falls below 33C and [ap] falls below [lower_press_limit] then disconnect [dialyser], execute [alarm_temp], stop [BP] and execute [alarm_ap]**

FIGURE 6.12 – Un besoin décrivant le collage dans le développement de R-21

- *Liens.* Le glossaire est automatiquement mis à jour.

6.3 Bilan

L'étude de plusieurs cas de systèmes critiques complexes et de taille importante révèle la nécessité d'automatiser ou semi-automatiser le développement. Une manière de réaliser cet objectif consiste à exploiter les formes communes des besoins du cahier des charges pour décrire des patrons comme *Dev-if*.

6.3.1 Apports de Dev-if

Ils concernent :

- *Aide à la construction.* Le patron interagit avec le développeur via des questions pour générer - par instanciation - un système < CdC, Liens, Spec > dont une partie considérable est obtenue automatiquement. En outre, il donne des indications sur les parties qui restent à définir dans le système afin d'éviter les oublis. Il aide également à renforcer les liens entre le *CdC* et la *Spec* en décrivant un glossaire instancié automatiquement.
- *Détection des oublis.* Les parties générées par le patron contenant des "..." permettent d'éviter les oublis en mentionnant explicitement, au développeur, des propriétés fondamentales telles que le collage. *Dev-if* permet la détection des lacunes dans le *CdC* telles que les initialisations et la description des propriétés évidentes.
- *Précision des ambiguïtés.* Au fur et à mesure de l'utilisation du patron, des ambiguïtés sont détectées. Elle concernent par exemple la notion d'alarme : une seule alarme générale pour tous les cas d'échec des composants ou plusieurs alarmes ?
- *Vérification.* Une partie des OPs est déchargée depuis le patron. Étant automatiquement déchargées pour *Dev-if*, ces OPs sont paramétrées et par conséquent l'instanciation de ce patron pour un besoin implique leur instanciation.
- *Validation.* Nous avons adopté la validation par animation et model-checking via ProB et en s'appuyant sur nos travaux [Sayar and Souquières, 2017] et [Sayar and Souquières, 2016]. Pour y parvenir, nous extrayons des éléments de validation génériques depuis la figure 6.2. Pendant l'application du *Dev-if*, ces éléments seront instanciés réduisant l'effort de validation.

6.3.2 Prise en compte des différents cas

Notre patron de développement considère les différents cas possibles. Il effectue des comparaisons entre les formes des besoins. Selon les différences entre deux besoins (dont un est développé), il se situe dans l'un des cas pour agir comme suit :

- Il instancie intégralement toutes les parties du modèle fourni en figure 6.2.
- Il ajoute de nouveaux invariants et un nouvel événement.
- Il introduit de nouvelles variables, de nouvelles constantes dans la partie *Spec* et de nouveaux couples de termes dans le glossaire.
- Il traite deux sous-cas :
  - — Quand les paramètres de S_{p1} sont différents, *Dev-if* crée de nouvelles constantes dans le contexte.
  - — Quand S_{p2} est composé de plus d'un paramètre, notre patron ajoute automatiquement de nouvelles variables, de nouveaux invariants et crée des gardes composées pour les événements concernés.
- Le cas général traite les points suivants :
  - — Une *condition(*S_{p2}*)* différente agit sur les invariants et les gardes des événements.
  - — Une *action(*S_{p3}*)* différente a un effet sur la partie des actions des événements concernés et sur les invariants.
  - — Le collage dépend des valeurs des paramètres du besoin :

- Si *R-new* et *R-dev* ont les mêmes paramètres dans S_{p2}, le collage n'existe pas.
- Si les deux besoins ont le même S_{p1} et des paramètres de S_{p2} différents, le collage peut avoir lieu.
- Si les deux besoins ont des S_{p1} différents, la présence du collage dépend de la propre compréhension du développeur et du sens de cet ensemble de paramètres dans S_{p1}. Par exemple, si cet ensemble concerne un aspect temporel tel qu'une phase située dans le temps, alors le collage peut exister si l'on est dans la même phase.

6.4 Conclusion

Dans ce chapitre, nous avons détaillé notre contribution concernant le patron de développement *Dev-if*. Celui-ci traite les besoins ayant une forme conditionnelle. Son application sur l'étude de cas de la machine d'hémodialyse a montré qu'il réduit l'effort de développement du système < CdC, Liens, Spec >, facilite les tâches de vérification et de validation et aide à éviter les oublis. En outre, il permet une participation active du développeur tout au long du développement via des interactions sous forme de questions et de réponses. Un autre atout de ce patron est qu'il n'est pas dépendant d'un langage de spécification. Il est utilisable dans divers langages.

Une limite de notre contribution est que son application est manuelle : le modèle de *Dev-if* est décrit sous une forme générique paramétrée en B événementiel, son application sur les besoins est fastidieuse vue qu'elle est faite à la main.

Comme première étape d'instanciation automatique de *Dev-if*, nous avons édité la partie $Spec_{if}$ de ce patron sous l'outil Pattern [51], plug-in de Rodin. Nous avons élaboré les deux autres constituants, CdC_{if} et $Glossaire_{if}$, en utilisant ProR.

51. http://wiki.event-b.org/index.php/Pattern

Chapitre 7

Patron pour la description séquentielle d'un besoin

Sommaire

Ce chapitre présente un de nos patrons de développement, le patron *Dev-seq*. Celui-ci n'est pas lié à un langage formel spécifique et s'intéresse aux besoins ayant une forme séquentielle.

7.1 Description

Nous nous intéressons aux besoins décrits par une forme séquentielle pour définir notre patron de développement nommé *Dev-seq*. Le terme *"séquentielle"* signifie qu'il s'agit d'une *séquence* ou *enchaînement* d'actions dans un *ordre* défini. Le patron proposé permet la gestion simultanée des besoins, de leurs modèles formels associés et des liens entre eux. Celui-ci s'applique dans les phases amont du processus de développement, l'analyse des besoins. *Dev-seq* favorise :

- l'automatisation de la prise en compte de la notion d'ordre entre opérations ou actions du système à développer,
- la préparation à l'activité de validation et/ou de test et
- l'aide à éviter les oublis dans le futur système.

Nous commençons par décrire une forme générique séquentielle des besoins. Par la suite, nous définissons notre patron en effectuant une analogie avec le triplet de Hoare [Hoare, 1969]. Contrairement aux patrons de conception et aux patrons de spécifications formelles, la description de *Dev-seq* n'est pas liée à un langage spécifique : nous le définissons dans deux langages, B événementiel et TLA$^+$. L'analogie avec le triplet de Hoare aide à déceler les éléments génériques constituants le patron indépendamment du langage utilisé, c'est-à-dire les éléments communs à toutes ses définitions. Ces éléments seront détaillés dans le paragraphe 7.1.3. Ce patron de développement agit sur un système < CdC, Liens, Spec >.

Dev-seq gère un modèle générique paramétré instancié selon le besoin en question. Il propose des éléments génériques de validation du futur système. Il effectue des opérations en interne telles que des comparaisons et des affectations et interagit avec le développeur via des questions pour générer un nouveau système.

7.1.1 Forme générique séquentielle

Prenons les besoins du cahier des charges du train d'atterrissage d'un avion [52] décrits dans la figure 7.1. Ce document comporte sept besoins décrits sous une forme séquentielle. Nous avons identifié plusieurs besoins décrits sous cette forme dans un autre document d'exigences : celui du système de contrôle de position de trains nommé "Hybrid ERTMS/ETCS Level 3" (European Rail Traffic Management System) publié dans la conférence ABZ 2018 [53].
Chaque besoin de cette figure 7.1 est décrit sous forme d'une séquence d'actions. Chaque action est exprimée à travers un ensemble de termes comportant un verbe. L'enchaînement entre les actions est exprimé de différentes manières :

- dans le besoin LS, cet enchaînement est décrit par le terme *"sequence"* au début, le signe de ponctuation *" :"* pour expliquer la séquence ainsi qu'une virgule *","* et un terme *"and"* connectant les actions,
- dans les besoins DCy et GCy, il s'exprime via des mots de liaison *"first"*, *"then"* et *"and finally"* et
- dans GEv, le séquencement est décrit par le terme *"sequence"* au départ suivi des actions *numérotées* de 1 à 8 pour indiquer leur ordre.

La répétition de cet enchaînement nous invite à définir une forme générique des besoins décrite par la forme (7.1)

$$\boxed{\begin{array}{l|l} \textit{S-new. (Requirement environment,) sequence name is :} & \begin{array}{l} 1.\ \textit{action}\ 1(Sp_1) \\ 2.\ \textit{action}\ 2(Sp_2) \\ \ldots \\ n.\ \textit{action}\ n(Sp_n) \end{array} \end{array}} \tag{7.1}$$

dans laquelle :

- *S-new* désigne le nom du besoin à développer,

52. Une partie de ce cahier des charges est décrite dans l'annexe A
53. https://www.southampton.ac.uk/abz2018/information/case-study.page

LS	*In nominal mode, the landing sequence is : open the doors of the landing gear boxes, extend the landing gears and close the doors.*
DCy	*When a door cylinder is locked in closed position and when it receives pressure from the extension hydraulic circuit,* - *first it is unlocked from the closed position* - *then it moves to the open position* - *and finally it is maintained in the open position as long as the pressure is maintained in the hydraulic extension circuit.*
GCy	*When a gear cylinder is locked in high position and when it receives pressure from the high hydraulic circuit,* - first it is unlocked from the high position - then it moves to the down position - and finally it is locked in the down position.
GEv	*When the gears are locked in retracted position, and the doors are locked in closed position, if the pilot sets the handle to "Down", then the software should have the following sequence of actions :* 1. *stimulate the general electro-valve isolating the command unit in order to send hydraulic pressure to the maneuvering electro-valves,* 2. *stimulate the door opening electro-valve,* 3. *once the three doors are in the open position, stimulate the gear outgoing electro-valve,* 4. *once the three gears are locked down, stop the stimulation of the gear outgoing electro-valve,* 5. *stop the stimulation of the door opening electro-valve,* 6. *stimulate the door closure electro-valve,* 7. *once the three doors are locked in the closed position, stop the stimulation of the door closure electro-valve,* 8. *and finally stop stimulating the general electro-valve.*

FIGURE 7.1 – Des exigences du cahier des charges du train d'atterrissage d'un avion

- *Requirement environment* décrit l'environnement du besoin *S-new*. En d'autres termes, cette partie décrit les conditions sous lesquelles ce besoin est défini. Nous la délimitons par des parenthèses pour indiquer que son existence est facultative.
- *Sequence name* affecte un nom à la séquence. Ce nom est soit prédéfini dans le besoin de départ (comme le cas du nom *"the landing sequence"* dans LS), soit donné par le développeur selon sa compréhension,
- Sp_i $(i \in \{1,..,n\})$ désigne l'ensemble des paramètres du besoin. Dans ce chapitre, nous étudions le

cas où cet ensemble ne contient qu'un seul élément p_i,

- *action i (Sp_i)* pour $(i \in \{1,..,n\})$ représente l'action effectuée sur Sp_i. Il s'agit de l'élément dynamique de cette forme. Une action peut être atomique ou composée. Néanmoins, vu que nous étudions le cas d'un Sp_i contenant un seul paramètre, cette action est souvent atomique et
- *les numéros de 1 à n* précisent explicitement l'ordre d'enchaînement des actions dans le besoin en question.

Exemple.

Regardons le besoin *LS* de la figure 7.1. Nous le réécrivons d'une manière équivalente afin qu'il soit conforme à la forme générique (7.1). Selon notre compréhension, les termes *"the doors of the landing gear boxes"* et *"the doors"* sont équivalents dans cette exigence. Par conséquent, nous gardons les termes *"the doors"* dans la nouvelle version de *LS*, voir la forme (7.2) :

LS. In nominal mode, the landing sequence is :	1. *open the doors* 2. *extend the landing gears* 3. *close the doors*	(7.2)

7.1.2 Analogie avec le triplet de Hoare

Ce triplet, décrit par la formule *{P}* I *{Q}*, exprime la propriété : *"pour tout état vérifiant P, si l'exécution de I se termine, alors Q doit être évalué à vrai"*. Dans ce triplet,

- *I* désigne :
  - — un programme, sous-programme ou une suite d'instructions dans le niveau programmation ou
  - — une spécification formelle décrite dans un langage formel comme B, B événementiel, LTL ou TLA^+,
- *P* est un prédicat décrivant une pré-condition sur les états en entrée de *I* et
- *Q* est un prédicat définissant une post-condition sur les états de sortie après la transformation effectuée par *I*.

Notre patron *Dev-seq* s'intéresse à deux aspects : l'aspect *sémantique* via l'expression de l'ordre entre les actions ou opérations du besoin et l'aspect de *validation* via la description d'un ou de plusieurs scénarios d'animation à partir du besoin. L'ordre entre les actions est préservé par notre patron via la prise en compte des notions de *pré-condition*, *post-condition* et *invariant*. Nous effectuons une analogie avec le triplet de Hoare, *{P}* I *{Q}*, par rapport auquel la forme générique séquentielle décrite dans (7.1) peut être comparée. Cette analogie aide à déceler l'ensemble des contraintes (pré-conditions, post-conditions et invariants) nécessaires pour la description du patron. Pour une *action i* précédée par une *action i-1* de la forme générique séquentielle, nous effectuons cette analogie :

- la partie *I* du triplet de Hoare est analogue à l'*action i*,
- la pré-condition *P* est équivalente à la disjonction, via un *ou logique*, de la partie *(Requirement environment)* avec :
  - — soit le résultat de l'*action i-1*,
  - — soit le vide si *action i* est en tête de la séquence et
- la partie *Q* du triplet est représentée par le *résultat d'action i*.

Nous présumons que le résultat de chaque action d'une séquence est à la fois :

- une post-condition pour l'action elle-même et
- une pré-condition pour l'action qui la suit si elle existe.

Dans la suite de ce chapitre, le terme *"pre"* désigne une contrainte de type pré-condition et le terme *"post"* indique une post-condition. Certaines contraintes sont à la fois des pré-conditions pour quelques actions et des post-conditions pour d'autres. Ceci assure l'enchaînement des actions permettant ainsi de préserver l'ordre. Afin de préciser la nature de ces contraintes, nous les notons avec un terme *"pre-post"*.

Exemple.

La figure 7.2 montre l'analogie entre le triplet de Hoare et la version réécrite (7.2) du besoins LS. A chaque action, nous procédons comme suit :

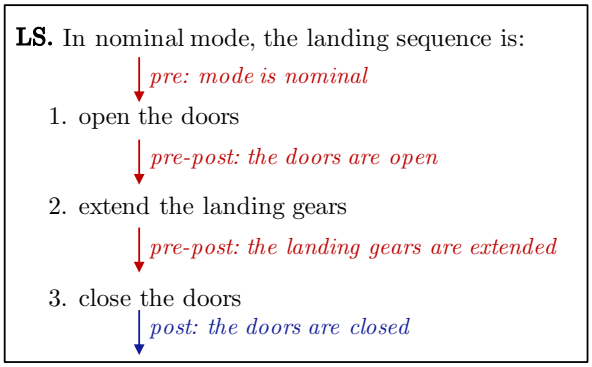


FIGURE 7.2 – Pré et post-conditions des actions de LS

1. L'action *"1. open the doors"* n'a explicitement pas de pré-condition vu qu'elle s'exécute en premier dans la chaîne. Cependant, nous remarquons la présence de la suite de termes *"In nominal mode"* au début du besoin. Celle-ci décrit une condition sur l'environnement du besoin *LS* : il faut être dans un *"nominal mode"* pour traiter ce besoin. En se référant à l'analogie avec le triplet de Hoare, nous considérons ceci comme une pré-condition de la première action *"open the doors"*. En outre, cette action engendre une post-condition que nous exprimons via les termes *"the doors are open"*.
2. L'action *"2. extend the landing gears"* a comme pré-condition le résultat de l'action 1 : *"the doors are open"*. Nous exprimons sa post-condition via l'ensemble des termes *"the landing gears are extended"*.
3. La dernière action de la chaîne *"3. close the doors"* a comme pré-condition *"the landing gears are extended"* et sa post-condition est *"the doors are closed"*.

7.1.3 Présentation du patron Dev-seq

Ce patron propose un modèle générique paramétré permettant l'automatisation du développement pour un besoin donné S-new. Il agit sur des opérandes représentées par S-new et un état d'un système existant < CdC, Liens, Spec >. *Dev-seq* gère ces opérandes pour fournir un nouvel état

$< CdC', Liens', Spec' >$. L'objectif est de garantir la prise en compte automatique de la notion d'ordre. L'application de notre patron se résume par la formule mathématique suivante

$$\textbf{\textit{Dev-seq :}} < CdC, Liens, Spec >, S-new \rightarrow < CdC', Liens', Spec' > \tag{7.3}$$

dans laquelle :

- CdC' désigne le *CdC* évolué par
  - — l'ajout de nouveaux besoins,
  - — le raffinement d'un ou de plusieurs besoins et/ou
  - — l'introduction d'un ordre entre les besoins existants via une numérotation.
- $Spec'$ représente la *Spec* mise à jour par des pré-conditions, post-conditions et des invariants.
- $Liens'$ sont les *Liens* mis à jour automatiquement.

Le modèle générique du patron est constitué de trois composants : CdC_{seq}, $Liens_{seq}$ et $Spec_{seq}$.

7.1.3.1 $Spec_{seq}$

Elle peut être décrite à l'aide de différents langages. Nous définissons ce composant en deux langages formels : B événementiel et TLA$^+$. Grâce à l'analogie faite entre la forme générique séquentielle des besoins et le triplet de Hoare, nous avons identifié les éléments formels communs entre les différentes définitions du composant $Spec_{seq}$ de notre patron. Ces éléments concernent :

- les pré-conditions, post-conditions et invariants décrivant l'ordre entre événements ou opérations,
- les variables décrivant l'état de l'environnement du besoin *S-new* et
- la prise en compte des variables représentant les paramètres des Sp_i (pour $(i \in \{1,..,n\})$) du besoin.

Sauf s'il s'agit d'une première application, *Dev-seq* ne crée pas forcément de nouvelle *Spec* : il est destiné à exprimer la notion d'*ordre*.

7.1.3.1.1 En B événementiel. La $Spec_{seq}$ de notre patron est décrite par un contexte et une machine, voir figure 7.3.

Le contexte, nommé *S-new_ctx*, est caractérisé par les constantes p_i*_val* désignant les valeurs des paramètres p_i des Sp_i $(i \in \{1,..,n\})$. Ces constantes ont chacune un type p_i*_TYPE*. Quant à la machine appelée *S-new_mch*, elle est décrite par :

- La clause *REFINES*. Elle est avec une astérisque ; cela signifie qu'elle est optionnelle. Elle dépend du cas de l'application du patron :
  - — si c'est une première application c'est-à-dire nous partons d'un développement vide, cette clause n'existe pas ou
  - — si le patron est appliqué sur un modèle existant, cette clause existe et fait appel à la machine abstraite nommée *Abstract_Model_Mch*.
- La clause *SEES* est également optionnelle. Son existence dépend de celle du contexte *S-new_ctx*.
- La liste des variables dépend de l'application. Nous introduisons deux sortes de variables :
  - — *formal_*p_i représentant une formalisation en B événementiel des paramètres informels de Sp_i $(i \in \{1,..,n\})$ et

CONTEXT *S-new_ctx*
EXTENDS *
Abstract_Model_Ctx
SETS *
p_i*_TYPE, ENV_TYPE*
CONSTANTS *
p_i*_val* // values of p_i $i \in 1..n$
vars_env_vals // values of environment variables
AXIOMS *
axm_i : partition(p_i*_TYPE,* $\{p_i\_val\}$*)* // giving types to the constants
axm_env : partition(ENV_TYPE, $\{vars\_env\_vals\}$*)*
END

MACHINE *S-new_mch*
REFINES *
Abstract_Model_mch
SEES *
S-new_ctx
VARIABLES
$formal\_p_i$ // $i \in \{1,..,n\}$
env_vars // variables describing the requirement environment
INVARIANTS
inv_Typ_pi : $formal\_p_i \in p_i\_TYPE$
inv_Typ_env_vars : env_vars $\in$ *ENV_TYPE*
inv_i : $\forall i \in 1..n \cdot post(formal\_action\_i(formal\_p_i)) = TRUE \Rightarrow pre(formal\_action\_i(formal\_p_i)) = TRUE$ // analogy with Hoare triplet
inv_order_i : $\forall i \in 1..n \cdot pre(formal\_action\_i{+}1(formal\_p_{i+1})) = post(formal\_action\_i(formal\_p_i))$
glu_inv_i : **...**
EVENTS
INITIALISATION
Event *evolve_env_vars* // evolving env_vars
Event *act_1_name* // action 1 of the sequence (i = 1)
when
grd1 : ... // Requirement environment : env_vars are checked
then
act1 : $formal\_action\_i(formal\_p_i)$
end
Event *act_i_name* // action i of the sequence ($i \in 2..n$)
refines * *abstract_act_i_name* // abstract event
when
grd1 : ... // abstract guards
grd2 : $post(formal\_action\_i\text{-}1(formal\_p_{i-1}))$
then
act1 : ... // abstract actions
act2 : $formal\_action\_i(formal\_p_i)$
end
END

FIGURE 7.3 – Description de $Spec_{seq}$ en B événementiel

— *env_vars* modélisant les variables d'environnement du besoin.

- La clause *INVARIANTS* est exprimée via quatre types d'invariants :
  — *inv_Typ_pi* pour le typage des paramètres du besoin *S-new*,
  — *inv_i* où i désigne le numéro de l'action en cours. Cet invariant exprime l'analogie avec le triplet de Hoare. Il assure que l'action numéro *i* ne s'exécute et ne donne une post-condition vraie que si sa pré-condition est vraie. Nous aurons autant de *inv_i* que d'actions dans le besoin,
  — *inv_order_i* pour exprimer l'ordre entre les actions et
  — *glu_inv_i* pour décrire le collage avec la Spec abstraite si elle existe.
- La clause *EVENTS* représente les événements :
  — assurant l'évolution de l'environnement du besoin via l'événement *evolve_env_vars*,
  — associés aux actions de la séquence *S-new*. Si ces événements existent, notre patron ajoute des contraintes sous forme de *gardes*, sinon il ajoute un nouvel événement. Dans ce dernier cas, ce patron associe à chaque action *action i (Sp$_i$)* un événement nommé *act_i_name*. Nous justifions ce choix par le fait que :
    - un événement est l'élément dynamique de la spécification formelle en B événementiel et
    - une action est l'élément dynamique dans la forme séquentielle des besoins.

    Notons que chaque événement a comme garde la post-condition (voir partie *then* de l'événement) de l'action précédente notée *post(formal_action_i-1(p$_{i-1}$))*. Ce type de gardes est essentiel vu qu'il exprime la notion d'ordre dans la spécification formelle. Un événement peut raffiner un autre événement abstrait appelé *abstract_act_i_name*. Ceci s'effectue lorsque cet événement donne plus de détails sur l'événement abstrait. La section 7.3 donne un exemple pour cet événement et
  — décrivant la première action de la séquence à travers un événement *act_1_name* associé à l'*action 1* de la forme (7.1). Cet événement a une garde nommée *"grd1"* formalisant les conditions sur l'environnement.

7.1.3.1.2 En TLA$^+$. L'état d'un système est décrit par les valeurs de ses variables. Pour notre patron, l'état de son composant $Spec_{seq}$ est décrit par les valeurs des paramètres des Sp_i ($i \in \{1,..,n\}$). La forme générique (7.1) décrit une séquence caractérisée par :

- une tête représentée par *action 1 (Sp$_1$)* et
- une fin décrite par *action n (Sp$_n$)*.

Chaque *action i (Sp$_i$)* agit sur les paramètres de Sp_i. Ces derniers prennent de nouvelles valeurs. En TLA$^+$, nous notons la nouvelle valeur des paramètres dans Sp_i par le suffixe $"'"$. Notre patron implémente son composant $Spec_{seq}$ en TLA$^+$ sous forme d'un *Module S-new_DevSeq*, voir figure 7.4. Ce module est caractérisé par :

- des variables *formal_pi* et *env_vars* représentant respectivement les paramètres de Sp_i et les variables d'environnement du besoin *S-new*,
- le théorème *PSinit* exprime le fait que pour assurer la première action (i = 1) nous avons des contraintes sur l'environnement à respecter. Ceci permet d'initialiser la première action,

- l'expression $\forall\, i \in (2..n+1) \cdot p'_{i-1} = post\,(act\_i\text{-}1\_name\ \ (p_{i-1}))$ indique que les nouvelles valeurs des paramètres p_{i-1} sont les post-conditions des actions $i-1$,
- le théorème *PSnext* exprime le fait que la pré-condition d'une action i prend la valeur des p'_{i-1} et
- le théorème *PS* est défini par la conjonction des deux théorèmes *PSinit* et *PSnext*. Celui-ci englobe la description formelle de la séquence du besoin *S-new* et permet d'assurer explicitement l'ordre entre les actions.

Module S-new_DevSeq

EXTENDS *Naturals*
VARIABLES $formal\_p_i$, *env_vars*

$PSinit \triangleq i = 1 \wedge env\_vars \wedge \Box\, formal\_p_i$
$\forall\, i \in (2..n+1) \cdot formal\_p'_{i-1} = post\,(act\_i\text{-}1\_name\ \ (formal\_p_{i-1}))$
$PSnext \triangleq \forall\, i \in (2..n+1) \cdot pre(act\_i\_name(formal\_p_i)) = formal\_p'_{i-1}$
$PS \triangleq PSinit \wedge \Box PSnext$

FIGURE 7.4 – Définition de $Spec_{seq}$ en TLA$^+$

7.1.3.2 CdC_{seq}

Il s'agit d'une version réécrite du besoin *S-new*. Nous choisissons d'utiliser l'approche ProR pour gérer, tracer et mémoriser ce document. Notre patron *Dev-seq* présente ce composant générique comme mentionné dans la figure 7.5.

ID	*Description*
S-new'	(condition ([env_vars])) the sequence name is :
S-new'-1	1. [act_1_name]
S-new'-2	2. [act_2_name]
...	...
S-new'-n	n. [act_n_name]

FIGURE 7.5 – Composant CdC_{seq} de *Dev-seq*

Dans cette représentation, *S-new'* indique le nouvel identifiant remplaçant *S-new*. Chacun des autres besoins de cette figure remplace une action. Ils sont hiérarchisés via un lien de parenté établi avec *S-new'*. Un tel lien est justifié par le fait que chaque action fait partie de la séquence principale exprimée dans le besoin parent. En outre, nous explicitons la notion d'ordre via la numérotation de *1* à *n* des besoins allant de *S-new'-1* jusqu'à *S-new'-n*. Les éléments mis entre crochets *[ ]* représentent des éléments formels issus de $Spec_{seq}$ élaborés en B événementiel ou en TLA$^+$. Par exemple, pour i = 1, *[act_1_name]* représente l'événement et/ou l'opération venant de la $Spec_{seq}$ et associé(e) à la description informelle d'*action 1*(Sp_1) de la forme générique (7.1) des besoins. Notons que l'ensemble

des termes *"(condition ([env_vars]))"* représentent la condition sur les variables de l'environnement du besoin si elle existe.

Nous considérons que la description des besoins dans ce composant est itérative afin de prendre en compte le raffinement. Cette notion signifie qu'un besoin donne plus de détails sur un autre besoin, tel est le cas de DCy et GCy qui raffinent respectivement les actions 1 et 2 de LS, voir figure 7.1. Ce besoin LS est considéré comme un besoin *abstrait*, tandis que DCy et GCy sont des besoins *raffinés*. Ils ont tous une forme séquentielle. Notre patron prend en compte cette notion de raffinement en intégrant automatiquement les besoins raffinés comme enfants des besoins abstraits concernés, voir section 7.3 pour plus de détails sur le raffinement via *Dev-seq*.

7.1.3.3 $Liens_{seq}$

Le contenu de ce composant ne dépend pas du langage choisi pour $Spec_{seq}$. Il est caractérisé par un glossaire décrit par des couples *(Formal term, Informal description)*. Ce glossaire est paramétré, c'est-à-dire que les paramètres de Sp_i présents dans les actions de la forme séquentielle (7.1) de *S-new* apparaissent dans sa description. Il est automatiquement mis à jour lors de l'application du *Dev-seq*. La figure 7.6 montre un aperçu de ce composant. Son premier couple décrit les variables d'environnement du besoin et leur formalisation. Son deuxième couple sera itérativement instancié selon la valeur de *i*. Un exemple de cette instanciation est fourni et expliqué ultérieurement par la figure 7.10.

Formal term	*Informal description*
env_vars	//environment variables
act_i_name	action i (Sp_i)

FIGURE 7.6 – Glossaire du patron *Dev-seq*

7.1.4 Vérification et validation

Vérification. Dans le cas de B événementiel, la correction du composant $Spec_{seq}$ a été prouvée en utilisant les outils de preuve disponibles sous la plateforme Rodin. Dans le cas de TLA$^+$, le module *S-new_DevSeq* nécessite une vérification.

Validation. Nous avons proposé une approche pour valider la spécification formelle élaborée en B événementiel dès l'analyse des besoins du client [Sayar and Souquières, 2017]. Outre la notion d'ordre, la forme séquentielle permet de déceler des informations utiles pour le développement de scénarios de validation et/ou de test du futur système ou logiciel. À partir du CdC_{seq} de la figure 7.5 et de notre compréhension de la forme générique (7.1) des besoins, nous extrayons des scénarios génériques pour la validation, comme indiqué dans la table 7.1. Ces éléments seront instanciés lors de l'application de *Dev-seq* sur un besoin de l'étude de cas en question.

Term_Type	*Terms*
Fact	- [formal_p_i] - [p_i_val] - [env_vars]
Functionality	- [act_i_name] //$i \in 1..n$
Obligation	- $\forall i \in 1..n$, postcondition([act_i-1_name]) = precondition([act_i_name]) - precondition([act_1_name]) = postcondition(evolve_vars_env)
Behavior	Initialisation $\rightarrow$ evolve_env_vars $\rightarrow$ [act_1_name] $\rightarrow$ [act_2_name] $\rightarrow \ldots \rightarrow$ [act_n_name]

TABLE 7.1 – Éléments génériques de validation associés à *Dev-seq*

7.2 Initialiser le développement

Prenons le besoin LS tel qu'il est décrit par sa forme présentée en (7.2). Le système de départ est décrit par les éléments suivants :

- *CdC.* Il représente le document informel rassemblant des exigences réécrites contenant LS.
- *Spec.* Ce composant est vide.
- *Liens.* Le glossaire est vide.

Ce besoin est constitué de trois actions. Commençons par extraire ses paramètres informels Sp_i ($i \in \{1,2,3\}$), voir table 7.2. Rappelons que le composant CdC_{seq} du patron est indépendant du langage utilisé pour $Spec_{seq}$.

Sp_i	*Valeur*
Sp_1	the doors
Sp_2	the landing gears
Sp_3	the doors

TABLE 7.2 – Paramètres informels de LS

Dev-seq fournit automatiquement le résultat décrit par le système < CdC, Liens, Spec >. Vu que nous partons d'un développement vide et qu'il s'agit de la première application de notre patron, celui-ci va fournir une panoplie d'informations concernant des constantes, variables, invariants et événements ou opérations.

7.2.1 CdC

Il est décrit par les besoins de la figure 7.7.

ID	*Description*
LS'	In [nominal] [op_mode], the landing sequence is :
LS'-1	1. [open_doors]
LS'-2	2. [extend_gears]
LS'-3	3. [close_doors]

FIGURE 7.7 – CdC de LS instancié à partir de *Dev-seq*

7.2.2 Spec

Elle est décrite en deux langages comme suit :

7.2.2.1 En B événementiel

Il est instancié automatiquement et décrit dans la figure 7.8.

CONTEXT *LS_ctx*
SETS
D_Positions, G_Positions, Mode
CONSTANTS
open, closed // values of p_1
g_extended, g_retracted // values of p_2
nominal, emergency // values of environment variables
AXIOMS
axm_1 : partition(D_Positions, {open}, {closed})
axm_2 : partition(G_Positions, {g_extended}, {g_retracted}))
axm_env : partition(Mode, {nominal}, {emergency})
END

MACHINE *LS_mch*
SEES
LS_ctx
VARIABLES
doors_pos // *p_1*
gears_pos // *p_2*
op_mode // env_vars
INVARIANTS
inv_Typ_doors_pos : doors_pos ∈ D_Positions
inv_Typ_gears_pos : gears_pos ∈ G_Positions
inv_Typ_op_mode : op_mode ∈ Mode
inv_1 : doors_pos = open ⇒ op_mode = nominal
inv_2 : gears_pos = g_extended ⇒ doors_pos = open
EVENTS
INITIALISATION
Event *evolve_op_mode* // evolving env_vars

Event *open_doors* // action 1 of the sequence
when
grd1 : op_mode = nominal
then
act1 : doors_pos := open
end
Event *extend_gears* // action 2 of the sequence
when
grd1 : doors_pos = open
grd2 : gears_pos = g_retracted
then
act2 : gears_pos := g_extended
end
Event *close_doors* // action 3 of the sequence
when
grd1 : gears_pos = g_extended
then
act2 : doors_pos := closed
end
END

FIGURE 7.8 – Description de Spec de LS en B événementiel

7.2.2.2 En TLA$^+$

Le module correspondant à la *Spec* est automatiquement instancié et présenté dans la figure 7.9.

Module LS_DevSeq

EXTENDS *Naturals*
VARIABLES *doors_pos, gears_pos, op_mode*

$PSinit \triangleq i = 1 \wedge op\_mode \wedge \Box doors\_pos$
$doors\_pos' = post\ (open\_doors\ )\ //\ =\ open$
$gears\_pos' = post\ (extend\_gears\ )\ //\ =\ g\_extended$
$doors\_pos' = post\ (close\_doors\ )\ //\ =\ closed$
$PSnext \triangleq \forall\ i\ \in (2..4) \cdot p'_{i-1} = pre(act\_i\_name(p_{i-1}))$
$PS \triangleq PSinit \wedge \Box PSnext$

FIGURE 7.9 – Spec de LS en TLA$^+$

7.2.3 Liens

Le glossaire est récursivement construit : le deuxième couple de la figure 7.6 est instancié autant de fois que du nombre d'actions dans le besoin LS, 3 fois entre autres. Le résultat est fourni dans la figure 7.10.

Formal term	*Informal description*
nominal	nominal
op_mode	mode
open_doors	open the doors
extend_gears	extend the landing gears
close_doors	close the doors

FIGURE 7.10 – Glossaire de LS issu de l'instanciation de *Dev-seq*

7.2.4 Vérification et validation

Ces deux activités ont été effectuées semi-automatiquement. Pour la validation, *Dev-seq* génère la table 7.3. Moyennant une activité d'animation avec l'outil ProB sous Rodin, tous les éléments de cette table ont été pris en compte par la *Spec* décrite en B événementiel.

7.3 Prise en compte d'un besoin enfant

Un apport de notre patron concerne sa prise en compte automatique du raffinement dans plusieurs étapes de développement :

- réécriture des besoins,

Term_Type	*Terms*
Fact	- [doors_pos], [gears_pos] - [open], [closed], [g_extended], [g_retracted] - [op_mode]
Functionality	- [open_doors], [extend_gears], [close_doors]
Obligation	- postcondition([open_doors]) = precondition([extend_gears]) - postcondition([extend_gears]) = precondition([close_doors]) - precondition([open_doors]) = postcondition(evolve_op_mode)
Behavior	Initialisation → evolve_op_mode → [open_doors] → [extend_gears] → [close_doors]

TABLE 7.3 – Éléments de validation de LS

- introduction d'un besoin raffiné dans un CdC abstrait et
- instanciation automatique du modèle générique < CdC_{seq}, $Liens_{seq}$, $Spec_{seq}$ >.

7.3.1 Réécriture du besoin enfant

Rappelons qu'une étape préliminaire à l'utilisation du patron *Dev-seq* consiste à réécrire les besoins selon la forme générique paramétrée proposée au début de ce chapitre. Si une exigence *S-new'* décrite sous une forme séquentielle est raffinée par une autre exigence de même forme, ce raffinement est pris en compte par notre patron lors de la phase de réécriture des besoins. Chacune des actions de *S-new'* est raffinée par une nouvelle séquence comme présenté dans la figure 7.11 :

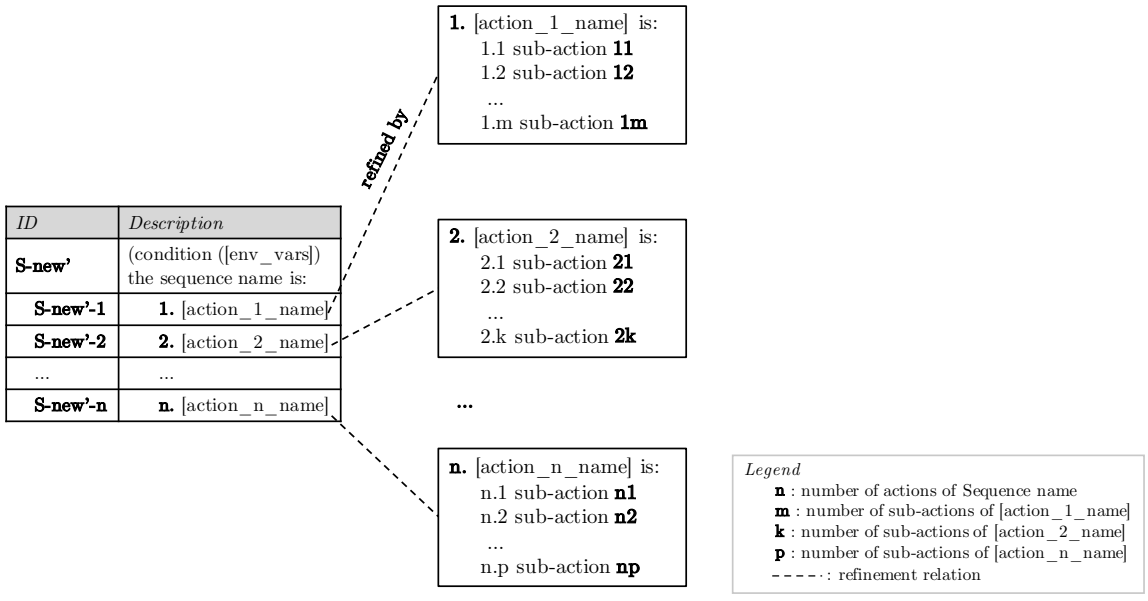


FIGURE 7.11 – Vue d'ensemble sur le raffinement des actions

- *action_1_name* est raffinée par les sous-actions allant de *sub-action 11* à *sub-action 1m*,
- Idem pour *action_2_name* ayant *k* sous-actions,

- ...

- *action_n_name* est raffinée par les sous-actions allant de *sub-action n1* à *sub-action np*.

Exemple.

En suivant la forme générique (7.1), les deux besoins DCy et GCy de la figure 7.1 sont réécrits. Le résultat est fourni par la figure 7.12.

DCy	*When a door cylinder is locked in closed position and when it receives pressure from the extension hydraulic circuit,* ***the opening doors sequence is :*** **1.** *unlock the door cylinder from the closed position* **2.** *move it to the open position* **3.** *maintain it in the open position as long as the pressure is maintained in the hydraulic extension circuit.*
GCy	*When a gear cylinder is locked in high position and when it receives pressure from the high hydraulic circuit,* ***the extension gears sequence is :*** **1.** *unlock the gear cylinder from the high position* **2.** *move it to the down position* **3.** *lock it in the down position.*

FIGURE 7.12 – Forme séquentielle réécrite de DCy et GCy

Les suites de termes respectives *"When a door cylinder is locked in closed position and when it receives pressure from the extension hydraulic circuit"* de DCy et *"When a gear cylinder is locked in high position and when it receives pressure from the high hydraulic circuit"* de GCy concernent la description d'une condition sur l'état de l'environnement englobant ces deux besoins. Il s'agit de la partie *Requirement environment* de la forme générique. Nous considérons que ces deux suites décrivent des pré-conditions respectives pour les deux besoins. Les actions *"LS'-1"* et *"LS'-2"* du besoin réécrit LS' sont raffinées respectivement par les nouvelles formes de DCy et GCy, voir figure 7.13.

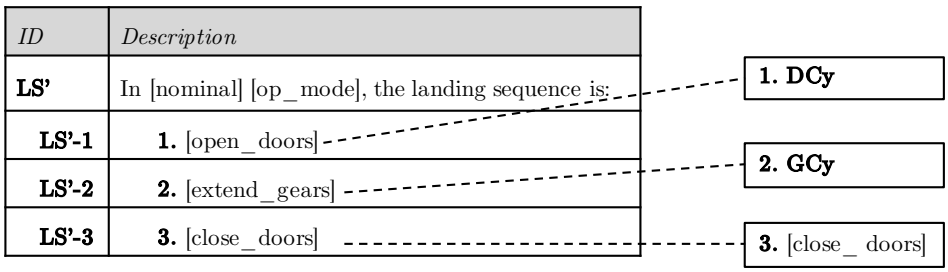


FIGURE 7.13 – Raffinement des actions de LS'

7.3.2 Préservation des pré et post-conditions abstraites

Nous partons de la figure 7.11 et nous nous intéressons à la préservation des pré et post-conditions abstraites entre les actions allant de 1 à n (partie gauche de cette figure) lors du raffinement. Nous nous basons sur l'analogie effectuée avec le triplet de Hoare (voir paragraphe 7.1.2) : pour i allant de 1 à n, le résultat d'une *[action_i-1_name]* est une pré-condition pour l'*action_i* qui la suit. Par exemple, le résultat de l'*[action_1_name]* est une pré-condition de l'*[action_2_name]* et ainsi de suite. En s'inspirant de la technique de raffinement en B événementiel, *Dev-seq* préserve les pré et post-conditions entre les sous-actions comme suit :

- le résultat de l'*[action_1_name]* est fourni par la *sub-action 1m* et
- ce résultat est considéré comme une pré-condition pour la *sub-action 21* représentant la tête de l'*[action_2_name]*.

Ceci se généralise pour toutes les autres actions.

Exemple.

Revenons à la forme réécrite LS' raffinée par DCy et GCy, voir figure 7.13. Nous avons fixé les pré-conditions et les post-conditions des actions de LS' dans la figure 7.2. Ces pré et post-conditions ont évolué en introduisant des termes formels dans leur description. Elles sont transmises aux sous-besoins qui raffinent LS', voir figure 7.14.

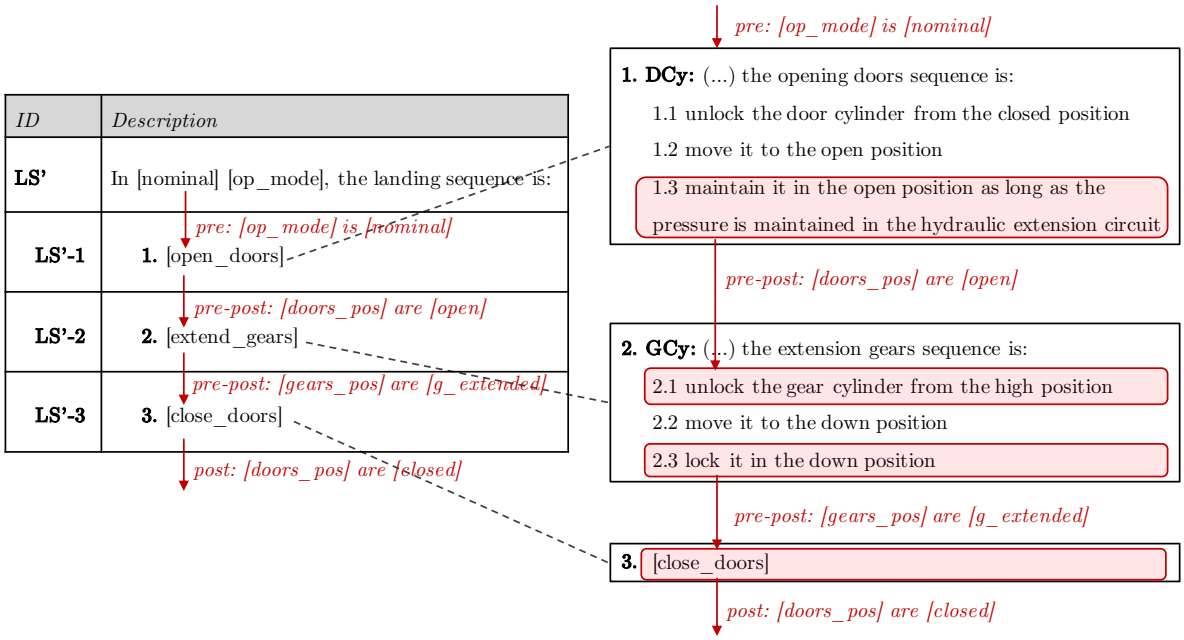


FIGURE 7.14 – Préservation des pré et post-conditions lors du raffinement des actions

7.3.3 Instanciation du patron

Notre patron permet le développement automatique des actions raffinées des besoins. Son instanciation génère les composants suivants :

- *Spec.* Le développement de ce composant est effectué en se basant sur le raffinement de la machine associée à S-new. Ceci est offert d'une façon native en B événementiel, voir figure 7.3.
- *CdC.* L'introduction d'un besoin raffiné se base sur la notion de *récursivité* : un besoin décrit sous une forme séquentielle et entrant dans les détails d'*[action_1_name]* est tout d'abord réécrit en instançiant le CdC_{seq} de la figure 7.5. Le résultat de cette instanciation est introduit comme *enfant* d'*[action_1_name]*. Cette récursivité permet d'introduire une *hiérarchie* entre les besoins. La figure 7.15 montre un cas de raffinement du besoin *S-new'-1*, enfant du besoin *S-new'*, voir partie (a) de la figure. Cet enfant est détaillé par de nouveaux besoins allant de *S-new'-1-1* à *S-new'-1-m*. Il devient leur parent comme présenté dans la partie (b) de la même figure.

ID	*Description*
S-new'	(condition ([env_vars])) The sequence name is :
S-new'-1	**1. [act_1_name]**
S-new'-2	2. [act_2_name]
...	...
S-new'-n	n. [act_n_name]

(a) Besoin S-new'-1 abstrait

ID	*Description*
S-new'	(condition ([env_vars])) The sequence name is :
S-new'-1	**1. [act_1_name] is :**
S-new'-1-1	**1.1 [act_11_name]**
...	**...**
S-new'-1-m	**1.m [act_1m_name]**
S-new'-2	2. [act_2_name]
...	...
S-new'-n	n. [act_n_name]

(b) Besoin S-new'-1 raffiné

FIGURE 7.15 – CdC_{seq} pour une action raffinée

- *Liens.* De nouveaux couples de termes sont automatiquement introduits dans le glossaire. Ceux-ci représentent les noms formels et leurs définitions informelles associées aux sous-actions du côté du *CdC*.

7.3.4 Application

Prenons le cas de développement de GCy. Nous avons mentionné précédemment que ce besoin entre dans les détails de la deuxième action de LS' : *"2. [extend_gears]"*. Il décrit également une séquence d'actions. Ce raffinement est pris en compte par notre patron. Celui-ci génère automatiquement les éléments suivants :

7.3.4.1 CdC

La prise en compte du raffinement entre séquences abstraite et raffinée est montrée via une *hiérarchie* entre besoins, voir figure 7.16 dans laquelle,

- LS' représente LS réécrit,
- GCy' définit GCy réécrit et
- GCy' est un enfant de LS'.

NB. Le besoin GCy remplace le besoin LS'-2 de la figure 7.7. Ce besoin, LS'-2, est renommé GCy'. Vu que nous voulons garder une trace de ce besoin, nous ajoutons son identifiant en commentaire *"//LS'-2"* dans la figure 7.16.

ID	*Description*
LS'	In [nominal] [op_mode], the landing sequence is :
LS'-1	1. [open_doors]
GCy' //LS'-2	**2. When [gears_cyln] is [locked_up] and when it recieves [pressure_high], [extend_gears] is :**
GCy'-1	**2.1 [unlock_up_gears_cylinders]**
GCy'-2	**2.2 [move_down_gears_cylinders]**
GCy'-3	**2.3 [lock_down_gears_cylinders]**
LS'-3	3. [close_doors]

FIGURE 7.16 – CdC de GCy enfant de LS

7.3.4.2 Spec

Dans ce composant, la prise en compte du raffinement est automatique via la relation *"refines"* en B événementiel. Le nouveau modèle introduit une nouvelle variable *gears_cyln* représentant les cylindres des trains d'atterrissage. Il introduit également de nouvelles constantes décrivant les états des cylindres : *locked_down*, *locked_up*, *moving_up* et *moving_down*.

Event lock_down_gears_cylinders
refines extend_gears
when
grd1 : doors_pos = open // garde héritée
grd2 : gears_pos = g_retracted // garde héritée
grd3 : gears_cyln = moving_down // nouvelle garde
then
act1 : gears_pos := g_extended // action héritée
act2 : gears_cyln := locked_down // nouvelle action
end

FIGURE 7.17 – Événement *lock_down_gears_cylinders*

Le raffinement de l'action 2 par GCy se répercute formellement par :

- introduction de deux événements *unlock_up_gears_cylinders* et *move_down_gears_cylinders* représentant les deux actions *"2.1 unlock the gear cylinder from the high position"* et *"2.2 move it to the down position"*. Le premier événement représente la *tête* de la séquence, il prend les mêmes gardes que l'événement abstrait *extend_gears* et
- raffinement de l'événement *extend_gears* par un nouvel événement *lock_down_gears_cylinders* comme présenté dans la figure 7.17. Celui-ci représente la *fin* de la séquence d'extension des trains : *"2.3 lock it in the down position"*.

7.3.4.3 Liens

Le glossaire est mis à jour automatiquement par les nouveaux éléments présentés dans la figure 7.18.

Formal term	*Informal description*
gears_cyln	a gear cylinder
locked_up	in high position
pressure_high	pressure from the high hydraulic circuit
unlock_up_gears_cylinders	unlock the gear cylinder from the high position
move_down_gears_cylinders	move it to the down position
lock_down_gears_cylinders	lock it in the down position

FIGURE 7.18 – Glossaire de GCy

7.4 Prise en compte d'un besoin frère

7.4.1 Expression de la notion d'ordre

Nous nous focalisons sur un cas où *Dev-seq* agit sur un modèle existant en rajoutant des détails dans le même niveau d'abstraction c'est-à-dire sans raffinement. L'introduction de la notion d'ordre entre deux besoins frères se répercute dans les différents composants comme suit :

- CdC_{seq}**.** Une numérotation est introduite montrant quel besoin vient avant l'autre. Par exemple, les besoins GCy et DCy sont de même niveau mais DCy se réalise avant GCy vu que l'ouverture des portes du train d'atterrissage s'effectue avant l'extension des trains. Nous attribuons le numéro *"1"* pour DCy et le numéro *"2"* pour GCy.
- $Spec_{seq}$**.** En B événementiel, la notion d'ordre est exprimée dans plusieurs endroits :
  - — *Au niveau des invariants.* Il s'agit d'exprimer cet ordre via l'invariant suivant nommé *inv_order_i*. Cet invariant est également décrit dans la figure 7.3. Celui-ci exprime que chaque pré-condition d'une action i+1 est égale à la post-condition de l'action i qui la précède.

    $$inv\_order\_i : \forall i \in 1..n \cdot pre(formal\_action\_i{+}1(formal\_p_{i+1})) = post(formal\_action\_i(formal\_p_i))$$

  - — *Au niveau des gardes entre événements.* Revenons sur la figure 7.3. La garde *grd2* de l'événement *act_i_name* exprime la relation d'ordre : pour effectuer cet événement il faut que la post-condition de l'événement précédent soit évaluée à vrai (TRUE).
- $Liens_{seq}$**.** La notion d'ordre n'apparaît pas explicitement dans le glossaire.

7.4.2 Application

Prenons l'exemple du développement de DCy. Nous partons du système < CdC, Liens, Spec > contenant le besoin GCy développé. Nous remarquons que les besoins DCy et GCy traitent tous les deux des cylindres associés aux portes et aux trains d'atterrissage. DCy est suivi temporellement par GCy. L'application du patron *Dev-seq* donne un nouveau système faisant évoluer le système existant *sans raffinement* et ajoutant de nouvelles informations apportées par DCy relativement à GCy. Le système résultant est décrit dans la figure 7.19 par :

CdC

ID	*Description*
LS'	In [nominal] [op_mode], the landing sequence is:
DCy' //LS'-1	**1. When [doors_cyln] is [locked_closed] and when it receives [pressure_extension], [open_doors] is:**
DCy'-1	**1.1 [unlock_closed_doors_cylinders]**
DCy'-2	**1.2 [move_open_doors_cylinders]**
DCy'-3	**1.3 [maintain_open_doors_cylinders]**
GCy' //LS'-2	2. When [gears_cyln] is [locked_up] and when it recieves [pressure_high], [extend_gears] is:
GCy'-1	2.1 [unlock_up_gears_cylinders]
...	...
LS'-3	3. [close_doors]

Glossaire

Formal term	*Informal description*
doors_cyln	a door cylinder
locked_closed	locked in closed position
pressure_extension	pressure from the extension hydraulic circuit
unlock_closed_doors_cylinders	unlock the door cylinder from the closed position
move_open_doors_cylinders	move it to the open position
maintain_open_doors_cylinders	maintain it in the open position as long as the pressure is maintained ...

Spec

CONTEXT GCy_ctx
SETS
GCylinders_Positions
DCylinders_Positions //issu de DCy
CONSTANTS
locked_up, locked_down, moving_up, moving_down, unlocked_up, unlocked_down
locked_closed, moving_open, maintained_open // issu de DCy
AXIOMS
axm1 : partition (GCylinders_Positions, {locked_up}, {locked_down}, {moving_up}, {moving_down}, {unlocked_up}, {unlocked_down})
axm2 : partition (DCylinders_Positions, {locked_closed}, {moving_open}, {maintained_open})

MACHINE GCy_mch
SEES GCy_ctx
VARIABLES
gears_pos, doors_pos, op_mode, gears_cyln
doors_cyln // issu de DCy
INVARIANTS
inv_Typ_doors_cyln: *doors_cyln* ∈ *DCylinders_Positions*
inv_order_1: gears_cyln = unlocked_up
⇒ doors_cyln = maintained_open // ordre
EVENTS
Initialisation
Event unlock_closed_doors_cylinders // issu de DCy

Event move_open_doors_cylinders // issu de Dcy
Event maintain_open_doors_cylinders // issu de DCy
refines open_doors
...
then
act1: doors_pos := open
act2: doors_cyln := maintained_open
end
Event unlock_up_gears_cylinders // issu de GCy
when
grd1: doors_pos =open
grd2: gears_pos = g_retracted
grd3: gears_cyln = moving_down
grd4: doors_cyln := maintained_open // expression de l'ordre
then
act1: gears_pos := g_extended
act2: gears_cyln := locked_down
Event move_down_gears_cylinders // issu de GCy
Event lock_down_gears_cylinders // issu de GCy
...
END

FIGURE 7.19 – Application du *Dev-seq* sur le besoin DCy

- *Spec.*

  — Trois nouvelles opérations associées aux trois actions du besoin DCy sont introduites. Celles-ci sont formalisées en B événementiel et/ou en TLA$^+$.

  — De nouvelles gardes sont introduites dans des opérations ou événements de la *Spec*. Ici, nous insistons sur les notions de pré-condition et de post-condition entre les deux besoins DCy et GCy. La notion d'ordre est assurée en ajoutant des pré-conditions ou des gardes entre les opérations ou événements associés à ces deux besoins. Prenons par exemple l'événement *unlock_up_gear_cylinders* dont le nom est mentionné dans le besoin *GCy'-1*. Nous partons d'un développement de GCy et nous assurons que cet événement est développé. Celui-ci représente le début de la séquence décrite dans GCy mais aussi vient après la dernière action de DCy *"1.3 maintain it in the open position as long as the pressure is maintained in the hydraulic extension circuit"*. Mis à part ses anciennes pré-conditions et post-conditions, notre patron constate que cet événement évolue sans raffinement en

lui introduisant de nouvelles gardes venant de DCy comme la garde *grd4* de l'événement *unlock_up_gears_cylinders* qui prend la même description de l'action nommée *act2* de l'événement *maintain_open_doors_cylinders*.

- *CdC.* Les noms formels donnés aux opérations ou aux événements sont introduits entre crochets *[ ]* dans le *CdC*. De même, dans ce document, le besoin *LS'-1* est renommé en *DCy'* qui le remplace. Nous gardons sa trace via un commentaire *"//LS'-1"*.
- *Liens.* Le glossaire est automatiquement mis à jour par de nouveaux couples.

7.5 Conclusion

Dans ce chapitre, nous avons présenté le patron de développement *Dev-seq* s'intéressant aux besoins décrits sous une forme séquentielle. Celui-ci concerne les besoins du *CdC* liés à leurs modèles formels *Spec* via des *Liens* représentés par un *Glossaire*. Notre patron est générique et n'est pas lié à un langage spécifique. Nous montrons deux implémentations possibles dans deux langages formels, B événementiel et TLA$^+$. Nous exploitons les techniques offertes par le génie logiciel telles que la récursivité et la programmation par contrats pour élaborer et utiliser notre patron. Les apports de *Dev-seq* concernent l'automatisation de la prise en compte de la notion d'ordre aussi bien dans le modèle formel que dans les besoins. Cette approche aide également à éviter les risques d'oublis et à faciliter l'activité de validation. Quel que soit le langage choisi pour implémenter notre patron, les mêmes concepts existent dans les différentes implémentations. Ces concepts concernent la description de l'environnement, la formalisation des paramètres du besoin ainsi que l'expression de l'enchaînement entre ses actions à travers des pré-conditions, des post-conditions et des invariants. *Dev-seq* permet de préserver la traçabilité des besoins et de la spécification formelle dans deux sens :

- dans le *CdC*, la trace des éléments formels apparaît dans les termes formels introduits entre crochets *[ ]* et
- dans la *Spec*, la trace des besoins est présentée par les étiquettes données aux éléments formels tels que les noms de la machine et des événements pour la définition en B événementiel et le nom du module et des actions pour la description en TLA$^+$.

Ne dépendant pas d'un langage spécifique, notre patron reste générique et n'est pas encore décrit en termes d'outils. Son application est dépendante des choix de développeur et son automatisation n'est pas effectuée. Nous avons proposé deux définitions possibles mais le champ peut être élargi par d'autres langages comme LTL. Nous envisageons également introduire dans notre patron d'autres concepts tels que la composition entre contraintes utilisée par le triplet de Hoare.

Chapitre 8

Conclusion et travaux futurs

Ce travail de thèse se situe dans le cadre du développement de systèmes et logiciels corrects à partir des besoins du client décrits dans son cahier des charges. Notre thèse traite deux mondes distincts : les besoins et leur spécification formelle correspondante. Notre objectif est de réduire l'écart entre ces deux mondes en établissant des liens et des interactions entre eux. Les documents issus de chaque monde forment un système présenté par (i) les besoins réécrits dans un *CdC*, (ii) leur spécification formelle correspondante nommée *Spec* et (iii) les liens entre eux représentés par un *glossaire*. Ces documents évoluent et interagissent en permanence ; grâce aux retours d'un monde, l'autre monde évolue et la qualité de ses documents s'améliore.

8.1 Nos contributions

Nous commençons par une étape de réécriture des besoins. Celle-ci, issue des travaux d'Abrial, vise la simplification de l'écriture des besoins et permet de les rendre accessibles et lisibles par toutes les parties prenantes. Par les termes *"partie prenante"*, nous désignons tout acteur - développeur, testeur ou client - utilisant le cahier des charges. Nous utilisons l'outil ProR pour cette étape de réécriture.

Nous avons élaboré notre approche autour de deux axes : le premier concerne la définition des interactions entre les constituants du système < CdC, Liens, Spec > et le deuxième s'intéresse à son évolution.

8.1.1 Définition des interactions entre CdC et Spec

Nous abordons cet axe à travers les deux activités de vérification et de validation, voir figure 8.1.

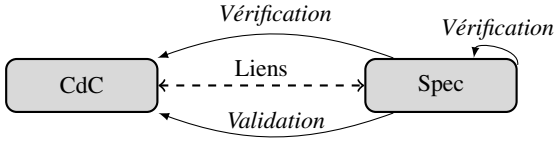


FIGURE 8.1 – Vérification et validation dans notre approche

8.1.1.1 Validation

Cette activité englobe plusieurs concepts : il ne s'agit pas de s'assurer que la Spec respecte un comportement décrit dans les besoins, mais plutôt de vérifier que tous les aspects décrits dans le CdC sont pris en compte par cette Spec. Comparant notre contribution par rapport aux approches existantes, nous présentons les apports suivants :

- Les approches traitant la validation des spécifications formelles s'occupent de l'aspect comportement. Elles utilisent des techniques telles que l'animation, le model-checking et la simulation approuvées par des outils comme ProB, AnimB et JeB pour étudier cet aspect. Outre le comportement, notre approche s'occupe également de nouveaux aspects classifiés en :
  - — données du futur système,
  - — opérations ou services offerts et
  - — obligations que ce système doit respecter lors de son fonctionnement.

  Sur le plan pratique, nous enrichissons la structure du CdC sous l'outil ProR par de nouveaux paramètres. Ceux-ci rassemblent les informations relatives aux différents aspects et présentent un accès rapide à la partie de la Spec concernée par la validation.
- L'activité de validation commence tôt dans le processus de développement, dès l'analyse des besoins. Elle commence par une étape d'extraction d'éléments nécessaires à cette activité avant de construire la Spec. Ces éléments sont mis à jour au fur et à mesure de l'évolution du système < CdC, Liens, Spec >.
- L'historique de la validation des Spec abstraites est mémorisé en vue de le réutiliser et l'exploiter lors de la validation des Spec raffinées. Cet historique aide à faciliter la tâche de validation et à prendre en compte l'aspect de collage entre les spécifications dans différents niveaux du développement.
- L'utilisation des outils pour la validation nous a servi à construire de nouvelles informations de validation. Par exemple, l'outil EventB Statemachines fournit une vue sur les différents états de la Spec et permet d'explorer différents scénarios de validation décrivant des comportements. Cette vue nous permet également de construire de nouveaux scénarios répartis entre autorisés et non-autorisés. Ces scénarios ne sont pas forcément présents dans les besoins. Ils enrichissent les éléments de validation extraits depuis le CdC.

8.1.1.2 Vérification

Bien que nous n'avons détaillé cette contribution qu'à travers un paragraphe dans ce mémoire, nous considérons que cette activité fait partie des interactions entre les deux mondes des besoins et des spécifications formelles. Contrairement aux autres approches qui considèrent que la vérification ne concerne que le monde du formel, les retours de cette activité nous signalent des failles aussi bien dans la Spec que dans les besoins. Nous faisons référence aux causes d'échec de preuves décrites par Abrial dans [Abrial, 2003] pour explorer cette piste. Cet échec peut être causé par (i) la Spec qui peut être prouvable, réfutable, pauvre et devant être enrichie ou (ii) par les besoins qui ne contiennent pas suffisamment d'informations ou qui contiennent des incohérences. Nous avons identifié un exemple dans lequel la vérification de la Spec du train d'atterrissage d'un avion signale que c'est le CdC qui est source du non-déchargement des obligations de preuve (OPs). Nous pensons que cette contribution, étant non encore détaillée, est un axe important à explorer. Cet axe sert à cerner les erreurs qu'encourent les spécifications formelles en désignant la source de l'erreur. Ceci aide à améliorer la qualité du système < CdC, Liens, Spec >.

8.1.2 L'évolution des besoins et de la Spec

Elle englobe :

- la prise en compte d'un nouveau besoin dans un système existant et
- l'utilisation des patrons de développement.

8.1.2.1 Prise en compte d'un nouveau besoin

Nous avons traité ce problème en partant du principe *"ne pas réinventer la roue"*. Admettant que nous disposons d'un système existant < CdC, Liens, Spec > perçu comme un seul bloc, l'objectif est de réutiliser ce système pour prendre en compte un nouveau besoin. Nous avons souligné les points suivants :

- la nécessité de la compréhension et de l'analyse du nouveau besoin,
- l'importance de la mise à jour de la hiérarchie entre les besoins,
- l'importance de la notion d'ordre entre les besoins et sa formalisation dans la Spec et
- les découvertes accompagnant l'intégration de ce nouveau besoin.

L'apport de notre contribution n'est pas d'introduire le besoin en soit, il s'agit de montrer l'ensemble des questions qui se posent au fur et à mesure de son intégration. Ces questions concernent la place de ce besoin parmi les besoins développés, sa formalisation et l'effet de son introduction sur le système existant. Quel que soit le choix effectué, des questions se posent et des découvertes émergent. Ces découvertes concernent le système existant et se focalisent autour des oublis, ambiguïtés et les besoins évidents et non décrits. Nous avons appliqué et montré notre contribution sur l'exemple d'un distributeur automatique de billets (DAB).

8.1.2.2 Utilisation des patrons de développement

Nous avons travaillé sur deux études de cas de taille importante et pour lesquelles un cahier des charges existe : une machine d'hémodialyse et un train d'atterrissage d'un avion. Dans chacun de ces documents, les besoins sont décrits selon deux parties :

- une explicative décrivant les détails du futur système. Elle est présentée par des spécialistes du domaine et
- une référentielle représentant "le quoi" du futur système et décrite par des informaticiens.

Cette contribution est liée à la forme des besoins de la partie référentielle :

- une forme conditionnelle *"if condition, then action"* dans la description de la machine d'hémodialyse et
- une autre forme séquentielle décrivant un enchaînement d'actions dans la présentation du train d'atterrissage.

Nous avons développé deux patrons de développement nommés *Dev-if* et *Dev-seq*. Le premier est associé à la forme conditionnelle des besoins et le deuxième à leur forme séquentielle. Comme les patrons de conception et ceux de spécification formelle, nos patrons sont utilisés par instanciation. Outre leur aide pour faciliter la tâche de développement, leur apport se résume par les trois points suivants :

- ils sont utilisés dans les phases amont du processus de développement dès l'analyse des besoins et s'intéressent à l'élaboration du système < CdC, Liens, Spec > ;
- ils sont indépendants du langage de spécification utilisé. Nous avons montré une définition de nos patrons en langage B événementiel. Nous avons focalisé spécifiquement sur ce langage mais nous montrons également la description et l'application du patron *Dev-seq* en TLA$^+$ et
- ils sont définis pour automatiser le développement et réduire les risques d'oubli autour du collage et des propriétés du futur système.

8.2 Difficultés et limites

Commençons par la première étape de réécriture des besoins du client. Celle-ci demande une compréhension de ces besoins afin de ne pas changer leur sens. Comme c'est le cas de plusieurs autres approches, une compréhension n'est toujours pas évidente ni directe vu que le cahier des charges souffre souvent des ambiguïtés et du manque de détails ; il faut du temps, des lectures à plusieurs reprises ainsi que l'utilisation de différents documents. Par exemple, pour le système de la machine d'hémodialyse qui décrit les cas de failles du système, nous avons eu besoin de comprendre le fonctionnement normal de cette machine avant de traiter les cas anormaux. Nous avons utilisé des images et des vidéos explicatives du processus de dialyse afin de comprendre le déroulement de ce système. La machine d'hémodialyse existait depuis plusieurs années, et son fonctionnement est connu. Ceci nous a permis d'accéder à différentes ressources de documentation. Cependant, cette méthode de documentation s'avère inutile en l'absence des ressources lorsqu'il s'agit de spécifier formellement un système inexistant.

Un autre point de notre approche concerne la gestion du système < CdC, Liens, Spec > : même en présence des outils, cette tâche est très difficile à accomplir. Nous avons (1) deux mondes différents, (2) des liens entre eux, (3) les retours des activités de vérification et de validation et (4) de différentes évolutions des documents. Les outils disponibles sous la plateforme Rodin pour le langage B événementiel donnent une vue d'ensemble sur les différents documents notamment les besoins et la Spec mais ne sont pas suffisants pour gérer simultanément ces documents.

Nous avons signalé que l'activité de vérification - à travers l'activité de preuve - concerne aussi bien la Spec que les besoins. Les retours de cette activité peuvent être incompréhensibles. La difficulté réside dans la localisation du constituant "coupable" causant le non-déchargement des OPs : le CdC ? la Spec ? les deux ? Aucun guide méthodologique n'est fourni pour aider à décider quel constituant revoir et corriger. D'après notre expérience, nous détectons qu'il se peut que le non-déchargement des OPs n'entre dans aucun des trois cas d'échec de preuve décrits par Abrial ; parfois ni la Spec ni le CdC sont coupables. Il s'agit des outils de preuve qui ne sont pas assez puissants pour réussir la preuve. Ceci complique à nouveau la tâche de localisation de la source d'échec de la preuve. Nous proposons l'idée d'analyse des OPs en question afin de résoudre ce problème.

8.3 Perspectives

Lors de l'application de notre contribution sur des études de cas de taille importante, nous avons obtenu des résultats encourageants. Nous avons utilisé des outils existants pour appliquer notre approche. Pour la suite, nous envisageons :

- Pour la validation dans les premières étapes de développement :
  - — automatiser l'extraction des éléments de validation. L'analyse syntaxique des besoins dans le cadre du traitement automatique des langues (TAL) ou l'utilisation des ontologies seront efficaces pour cet objectif. Par exemple, nous pourrons considérer que les verbes d'une phrase ou d'un besoin représentent des fonctionnalités du futur système. De même, les noms peuvent correspondre à des données devant exister dans ce système,
  - — exploiter l'apport des invariants et théorèmes pour aider à la validation. Comme c'est le cas pour l'expression de l'absence de blocage (deadlock-freeness) dans le modèle formel, nous pouvons exprimer d'autres propriétés liées à la validation sous forme de théorèmes ou d'invariants. Une fois que ces derniers sont prouvés corrects mathématiquement, notre Spec demeure valide relativement à cette propriété ; ceci permet d'éviter de parcourir plusieurs chemins dans l'espace d'états lors de la validation,
  - — traiter le problème de mémorisation de l'historique de validation : où mémoriser cet historique et comment le gérer ? La notion de hiérarchie est importante pour apporter un élément de réponse à cette question et
  - — utiliser le collage pour résoudre le problème de validation des Specs définies séparément. Ceci se réalise à travers l'utilisation des invariants et des événements de collage dans les Specs raffinées.

- Relativement à la vérification. L'exploration des retours de cette activité, étant fastidieuse, n'est toujours pas terminée. Nous envisageons de travailler sur cet axe en localisant la source du non-déchargement des OPs. Deux idées sont utiles pour réaliser cet objectif : l'utilisation d'un nouveau paramètre dans le CdC et l'automatisation des Liens entre le CdC et la Spec.

- Pour la prise en compte d'un nouveau besoin dans un système existant. La rédaction d'un guide méthodologique ou d'une heuristique permet de guider les étapes de développement lors de cette prise en compte. Un tel guide sera basé sur des questions pour aider le développeur et éviter les oublis.
- Relativement aux patrons de développement :
  - — améliorer la définition de ces patrons en introduisant l'expression des propriétés de sécurité comme l'absence de blocage dans la Spec. Ces propriétés sont communes à plusieurs systèmes et il s'avère important de discuter la possibilité de les introduire dans nos patrons. Pour y parvenir, il faut raisonner sur le sens de ces propriétés et leur utilité par rapport à nos patrons,
  - — définir une librairie de patrons de développement. De nouveaux patrons, tels que *Dev-set* pour les besoins décrits sous forme d'un ensemble, pourront être ajoutés à cette librairie et
  - — automatiser la définition de nos patrons en utilisant les outils de développement disponibles sous la plateforme Rodin. L'introduction de la composition entre les contraintes dans notre patron *Dev-seq* est également envisageable.

Bibliographie

[IEE, 1984] (1984). IEEE Guide for Software Requirements Specifications .

[IEE, 1990] (1990). IEEE Standard Glossary of Software Engineering Terminology.

[ISO, 2010] (2010). Systems and software engineering ? Vocabulary .

[ISO, 2011] (2011). Systems and software engineering - Life cycle processes - Requirements engineering.

[Abrial, 2003] Abrial, J.-R. (2003). B : passé, présent, futur. *Technique et Science Informatiques*, 22(1) :89–118.

[Abrial, 2005] Abrial, J.-R. (2005). *The B-book - assigning programs to meanings*. Cambridge University Press.

[Abrial, 2006a] Abrial, J.-R. (2006a). Formal Methods in Industry : Achievements, Problems, Future. In *28th International Conference on Software Engineering, Shanghai, China*, pages 761–768.

[Abrial, 2006b] Abrial, J.-R. (2006b). Train Systems. In *Rigorous Development of Complex Fault-Tolerant Systems RODIN project*, pages 1–36.

[Abrial, 2007] Abrial, J.-R. (2007). A System Development Process with Event-B and the Rodin Platform. In *Formal Methods and Software Engineering, 9th International Conference on Formal Engineering Methods, ICFEM, Boca Raton, FL, USA*, pages 1–3.

[Abrial, 2009] Abrial, J.-R. (2009). Faultless Systems : Yes We Can ! *IEEE Computer*, 42(9) :30–36.

[Abrial, 2010] Abrial, J.-R. (2010). *Modeling in Event-B : System and Software Engineering*. Cambridge University Press.

[Abrial et al., 2010] Abrial, J.-R., Butler, M., Hallerstede, S., Hoang, T. S., Mehta, F., and Voisin, L. (2010). Rodin : an open toolset for modelling and reasoning in Event-B. *International Journal on Software Tools for Technology Transfer*, 12(6) :447–466.

[Abrial et al., 2006] Abrial, J.-R., Butler, M. J., Hallerstede, S., and Voisin, L. (2006). An Open Extensible Tool Environment for Event-B. In *Formal Methods and Software Engineering, 8th International Conference on Formal Engineering Methods, ICFEM, Macao, China*, pages 588–605.

[Abrial and Hoang, 2008] Abrial, J.-R. and Hoang, T. S. (2008). Using Design Patterns in Formal Methods : An Event-B Approach. In *Theoretical Aspects of Computing - ICTAC, 5th International Colloquium, Istanbul, Turkey*, pages 1–2.

[Aceituna et al., 2014] Aceituna, D., Walia, G. S., Do, H., and Lee, S. (2014). Model-based requirements verification method : Conclusions from two controlled experiments. *Information & Software Technology*, 56(3) :321–334.

[Alexander et al., 1977] Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., Fiksdahl-King, I., and Angel, S. (1977). *A Pattern Language - Towns, Buildings, Construction*. Oxford University Press.

[Alkhammash et al., 2015] Alkhammash, E., Butler, M. J., Fathabadi, A. S., and Cîrstea, C. (2015). Building Traceable Event-B Models from Requirements. *Science of Computer Programming (Special Issue on Automated Verification of Critical Systems, AVoCS 2013)*, 111, Part 2 :318–338.

[Ameur and Méry, 2016] Ameur, Y. A. and Méry, D. (2016). Making explicit domain knowledge in formal system development. *Science of Computer Programming*, 121 :100–127.

[Banach, 2016] Banach, R. (2016). Hemodialysis Machine in Hybrid Event-B. In *Abstract State Machines, Alloy, B, TLA, VDM, and Z - 5th International Conference, ABZ, Linz, Austria*, pages 376–393.

[Beck, 1999] Beck, K. L. (1999). Embracing Change with Extreme Programming. *IEEE Computer*, 32(10) :70–77.

[Bell and Thayer, 1976] Bell, T. and Thayer, T. (1976). Software Requirements : Are They Really a Problem ? In *ICSE-2, 2nd International Conference on Software Engineering, San Francisco*, pages 61–68.

[Ben, 1996] Ben, D. V. (1996). Formalizing new navigation requirements for NASA's space shuttle. In *Formal Methods Europe FME '96, Lecture Notes in Computer Science, volume 1051, Springer-Verlag, Oxford, UK*, pages 160–178.

[Bendisposto et al., 2008] Bendisposto, J., Leuschel, M., Ligot, O., and Samia, M. (2008). La validation de modèles Event-B avec le plug-in ProB pour RODIN. *Technique et Science Informatiques*, 27(8) :1065–1084.

[Boehm, 1981] Boehm, B. W. (1981). An Experiment in Small-Scale Application Software Engineering. *IEEE Transactions on Software Engineering*, 7 :482–493.

[Boniol and Wiels, 2014] Boniol, F. and Wiels, V. (2014). Landing Gear System. In *ABZ Conference, Communications in Computer and Information Science, Springer*, volume 433, pages 1–18.

[Chemuturi, 2012] Chemuturi, M. (2012). *Requirements Engineering and Management for Software Development Projects*. Springer Publishing Company, Incorporated ©2012.

[Cimatti et al., 2008] Cimatti, A., Roveri, M., Susi, A., and Tonetta, S. (2008). From Informal Requirements to Property-Driven Formal Validation. In *Formal Methods for Industrial Critical Systems, 13th International Workshop, FMICS, L'Aquila, Italy*, pages 166–181.

[Colin et al., 2008] Colin, S., Lanoix, A., Kouchnarenko, O., and Souquières, J. (2008). Towards validating a platoon of cristal vehicles using csp||b. In *Algebraic Methodology and Software Technology, 12th International Conference, AMAST, Urbana, IL, USA*, pages 139–144.

[Cybulski et al., 1998] Cybulski, J. L., Neal, R. D. B., Kram, A., and Allen, J. C. (1998). Reuse of early life-cycle artifacts : workproducts, methods and tools. *Annals of Software Engineering*, 5(1) :227–251.

[Déharbe et al., 2012] Déharbe, D., Fontaine, P., Guyot, Y., and Voisin, L. (2012). SMT Solvers for Rodin. In *Abstract State Machines, Alloy, B, VDM, and Z - Third International Conference, ABZ Pisa, Italy*, pages 194–207.

[Delannoy, 2000] Delannoy, C. (2000). *Programmer en langage C++*. Eyrolles.

[Delor et al., 2003] Delor, E., Darimont, R., and Rifaut, A. (2003). Software quality starts with the modelling of goal-oriented requirements. In *16ème Journées Internationales "Génie Logiciel & Ingénierie de Systèmes et leurs Applications" ICSSEA, Paris, 6 pages*.

[Dijkstra, 1976] Dijkstra, E. W. (1976). *A Discipline of Programming*. Prentice-Hall.

[Dijkstra, 1970] Dijkstra, E. W. (avril 1970). Notes On Structured Programming. page 88.

[Driss, 2014] Driss, S. (2014). *From natural language specifications to formal specifications via an ontology as a pivot model*. Theses, Université Paris Sud - Paris XI.

[Eén and Sörensson, 2003] Eén, N. and Sörensson, N. (2003). An Extensible SAT-solver. In *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT, Santa Margherita Ligure, Italy,*, pages 502–518.

[Emerson and Clarke, 1980] Emerson, E. A. and Clarke, E. M. (1980). Characterizing Correctness Properties of Parallel Programs Using Fixpoints. In *Automata, Languages and Programming, 7th Colloquium, Noordweijkerhout, The Netherlands*, pages 169–181.

[Fathabadi et al., 2012] Fathabadi, A. S., Butler, M. J., and Rezazadeh, A. (2012). A Systematic Approach to Atomicity Decomposition in Event-B. In *Software Engineering and Formal Methods - 10th International Conference, Thessaloniki, Greece*, pages 78–93.

[Forsberg and Mooz, 1991] Forsberg, K. and Mooz, H. (1991). The Relationship of System Engineering to the Project Cycle. In *Proceedings of the First Annual Symposium of National Council on System Engineering*, pages 57–65.

[Gamma et al., 1996] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1996). *Design Patterns : Catalogue de modèles de conception réutilisables*. International Thomson Publishing France, Paris.

[Golra et al., 2018] Golra, F. R., Dagnat, F., Souquières, J., Sayar, I., and Guerin, S. (June 2018). Bridging the gap between informal requirements and formal specifications using model federation. In *SEFM 2018 International Conference on Software Engineering and Formal Methods, Toulouse, France*.

[Gotel and Finkelstein, 1994] Gotel, O. C. Z. and Finkelstein, A. (1994). An analysis of the requirements traceability problem. In *Proceedings of the First IEEE International Conference on Requirements Engineering, ICRE '94, Colorado Springs, Colorado, USA*, pages 94–101.

[Gunter et al., 2000] Gunter, C. A., Gunter, E. L., Jackson, M., and Zave, P. (2000). A Reference Model for Requirements and Specifications- Extended Abstract. In *4th International Conference on Requirements Engineering*, pages 17 : 37–43.

[Hallerstede et al., 2011] Hallerstede, S., Leuschel, M., and Plagge, D. (2011). Validation of Formal Models by Refinement Animation. *Science of Computer Programming*, 78(3) :272–292.

[Heisel and Souquières, 1999] Heisel, M. and Souquières, J. (1999). A Method for Requirements Elicitation and Formal Specification. In *18th International Conference on Conceptual Modeling, ER'99*, number 1728 in LNCS Springer-Verlag, pages 309–325.

[Hoang et al., 2013] Hoang, T. S., Fürst, A., and Abrial, J.-R. (2013). Event-B Patterns and their Tool Support. *Software and System Modeling*, 12(2) :229–244.

[Hoare, 1969] Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Communications of the ACM*, 12(10) :576–580 & 583.

[Idani, 2006] Idani, A. (2006). *B/UML : Mise en relation de spécifications B et de descriptions UML pour l'aide à la validation externe de développements formels en B. (B/UML : Bridging the gap between B specifications and UML graphical descriptions to ease external validation of formal B developments)*. PhD thesis, Joseph Fourier University, Grenoble, France.

[Jackson, 1997] Jackson, M. (1997). The meaning of requirements. *Annals of Software Engineering*, 3(1) :5–21.

[Jastram, 2010] Jastram, M. (2010). ProR, an Open Source Platform for Requirements Engineering based RIF. *Systems Engineering Infrastructure Conference, Munich, Germany*.

[Jastram et al., 2011] Jastram, M., Hallerstede, S., and Ladenberger, L. (2011). Mixing Formal and Informal Model Elements for Tracing Requirements. *Electronic Communications of the EASST*, 46.

[Johnson et al., 1994] Johnson, J., Mulder, H., and Group, S. (1994). The standish group report 1994.

[Jones et al., 1998] Jones, S., Till, D., and Wrightson, A. M. (1998). Formal methods and requirements engineering : Challenges and synergies. *Journal of Systems and Software*, 40(3) :263–273.

[Knight, 2002] Knight, J. C. (2002). Safety critical systems : challenges and directions. In *Proceedings of the 24th International Conference on Software Engineering, ICSE, Orlando, Florida, USA*, pages 547–550.

[Krings et al., 2015] Krings, S., Bendisposto, J., and Leuschel, M. (2015). From Failure to Proof : The ProB Disprover for B and Event-B. In *Software Engineering and Formal Methods - 13th International Conference, SEFM, York, UK*, pages 199–214.

[Ladenberger et al., 2017] Ladenberger, L., Hansen, D., Wiegard, H., Bendisposto, J., and Leuschel, M. (2017). Validation of the ABZ landing gear system using ProB. *International Journal on Software Tools for Technology Transfer (STTT)*, 19(2) :187–203.

[Lamport, 2002] Lamport, L. (2002). *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley.

[Leivant, 1994] Leivant, D. (1994). Higher order logic. In *Handbook of Logic in Artificial Intelligence and Logic Programming, Volume2, Deduction Methodologies*, pages 229–322.

[Leuschel and Butler, 2003] Leuschel, M. and Butler, M. (2003). ProB : A Model Checker for B. In *International Symposium of Formal Methods Europe, Pisa, Italy*, volume 2805 of *LNCS*, pages 855–874. Springer.

[Leuschel and Butler, 2008] Leuschel, M. and Butler, M. (2008). Prob : an automated analysis toolset for the B method. *STTT*, 10(2) :185–203.

[Lions et al., 1996] Lions, J.-L., Lübeck, L., Fauquembergue, J.-L., Kahn, G., Kubbat, W., Levedag, S., Mazzini, L., Merle, D., and O'Halloran, C. (19 juillet 1996). ARIANE 5 -Flight 501 Failure. Technical report.

[Mashkoor, 2016] Mashkoor, A. (2016). The Hemodialysis Machine Case Study. In *Abstract State Machines, Alloy, B, TLA, VDM, and Z - 5th International Conference, ABZ*, pages 329–343.

[Mashkoor and Jacquot, 2016] Mashkoor, A. and Jacquot, J.-P. (2016). Validation of Formal Specifications through Transformation and Animation. *Requirements Engineering, Springer Verlag, 16 pages*.

[Mashkoor et al., 2016] Mashkoor, A., Yang, F., and Jacquot, J.-P. (2016). Refinement-based Validation of Event-B Specifications. *Software and Systems Modeling, Springer Verlag, 33 pages*.

[Méry and Singh, 2011a] Méry, D. and Singh, N. K. (2011a). Automatic code generation from event-b models. In *Proceedings of the 2011 Symposium on Information and Communication Technology, SoICT, Hanoi, Viet Nam*, pages 179–188.

[Méry and Singh, 2011b] Méry, D. and Singh, N. K. (2011b). Formalization of heart models based on the conduction of electrical impulses and cellular automata. In *Foundations of Health Informatics Engineering and Systems - First International Symposium, FHIES, Johannesburg, South Africa*, pages 140–159.

[Méry and Singh, 2011c] Méry, D. and Singh, N. K. (2011c). Medical protocol diagnosis using formal methods. In *Foundations of Health Informatics Engineering and Systems - First International Symposium, FHIES, Johannesburg, South Africa*, pages 1–20.

[Moskewicz et al., 2001] Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L., and Malik, S. (2001). Chaff : Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference, Las Vegas, NV, USA*, pages 530–535.

[Neumann, 1995] Neumann, P. G. (1995). *Computer Related Risks*. ACM Press/Addison-Wesley Publishing Co.

[Owre et al., 1995] Owre, S., Rushby, J. M., Shankar, N., and von Henke, F. W. (1995). Formal Verification for Fault-Tolerant Architectures : Prolegomena to the Design of PVS. *IEEE Transactions on Software Engineering (TSE)*, 21(2) :107–125.

[Plagge and Leuschel, 2012] Plagge, D. and Leuschel, M. (2012). Validating B, Z and TLA + Using ProB and Kodkod. In *FM 2012 : Formal Methods - 18th International Symposium, Paris, France*, pages 372–386.

[Pnueli, 1977] Pnueli, A. (1977). The Temporal Logic of Programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA*, pages 46–57.

[Ponsard et al., 2015] Ponsard, C., Darimont, R., and Michot, A. (2015). Combining Models, Diagrams and Tables for Efficient Requirements Engineering : Lessons Learned from the Industry. In *Actes du XXXIIIème Congrès INFORSID, Biarritz, France*, pages 235–250.

[Sayar and Souquières, 2016] Sayar, I. and Souquières, J. (2016). La Validation dans le Processus de Développement. In *Actes du XXXIVème Congrès INFORSID, Grenoble, France*, pages 67–82.

[Sayar and Souquières, 2017] Sayar, I. and Souquières, J. (2017). Du cahier des charges à sa spécification. In *16 ème journées AFADL, Montpellier, France*.

[Sayar and Souquières, 2017] Sayar, I. and Souquières, J. (2017). La validation dans les premières étapes du processus de dévelopement. *Revue ISI-DAT, numéro spécial « Décisions, argumentation et traçabilité dans l'Ingénierie des Systèmes d'Information »*, 22(4) :11–41.

[Schwaber, 1995] Schwaber, K. (1995). SCRUM Development Process. In *10th Annual ACM Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA)*.

[Servat, 2007] Servat, T. (2007). BRAMA : A New Graphic Animation Tool for B Models. In *B 2007 : Formal Specification and Development in B, 7th International Conference of B Users, Besançon, France*, pages 274–276.

[Singh et al., 2015] Singh, N. K., Lawford, M., Maibaum, T. S. E., and Wassyng, A. (2015). Formalizing the cardiac pacemaker resynchronization therapy. In *Digital Human Modeling - Applications in Health, Safety, Ergonomics and Risk Management : Ergonomics and Health - 6th International Conference, DHM, Held as Part of HCI International, Los Angeles, CA, USA, Part II*, pages 374–386.

[Snook and Butler, 2006] Snook, C. F. and Butler, M. J. (2006). UML-B : Formal Modeling and Design Aided by UML. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 15(1) :92–122.

[Su and Abrial, 2017] Su, W. and Abrial, J.-R. (2017). Aircraft landing gear system : approaches with Event-B to the modeling of an industrial system. *International Journal on Software Tools for Technology Transfer (STTT)*, 19(2) :141–166.

[Su et al., 2011] Su, W., Abrial, J.-R., Huang, R., and Zhu, H. (2011). From Requirements to Development : Methodology and Example. In *13th International Conference on Formal Engineering Methods, Durham, UK*, pages 437–455.

[Su et al., 2015] Su, W., Abrial, J.-R., Pu, G., and Fang, B. (2015). Formal development of a real-time operating system memory manager. In *20th International Conference on Engineering of Complex Computer Systems, ICECCS 2015, Gold Coast, Australia*, pages 130–139.

[Su et al., 2014] Su, W., Abrial, J.-R., and Zhu, H. (2014). Formalizing Hybrid Systems with Event-B and the Rodin Platform. *Science of Computer Programming*, 94 :164–202.

[van Lamsweerde, 2000] van Lamsweerde, A. (2000). Requirements Engineering in the Year 00 : a Research Perspective. In Ghezzi, C., Jazayeri, M., and Wolf, A. L., editors, *Proceedings of the 22nd International Conference on on Software Engineering, Limerick Ireland*, pages 5–19. ACM.

[van Lamsweerde, 2008] van Lamsweerde, A. (2008). Requirements Engineering : from Craft to Discipline. In Harrold, M. J. and Murphy, G. C., editors, *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Atlanta, Georgia, USA*, pages 238–249.

[van Lamsweerde, 2009] van Lamsweerde, A. (2009). *Requirements Engineering - From System Goals to UML Models to Software Specifications*. Wiley.

[Voisin and Abrial, 2014] Voisin, L. and Abrial, J.-R. (2014). The Rodin Platform Has Turned Ten. In *Abstract State Machines, Alloy, B, TLA, VDM, and Z - 4th International Conference, ABZ, Toulouse, France*, pages 1–8.

[Wang, 1998] Wang, L. (1998). A Case-study of Requirements Reuse through Product Families. *Annals of Software Engineering*, 5(1) :253–277.

[Wirth, 1971] Wirth, N. (1971). Program Development by Stepwise Refinement. *Communications of ACM*, 14(4) :221–227.

[Wright, 2009] Wright, S. (2009). Automatic Generation of C from Event B. In *Workshop on Integration of Model-based Formal Methods and Tools, Düsseldorf, Germany*, pages 130–139.

[Yang, 2013] Yang, F. (2013). *A Simulation Framework for the Validation of Event-B Specifications. (Un environnement de simulation pour la validation de spécifications B événementiel)*. PhD thesis, University of Lorraine, Nancy, France.

[Yang and Jacquot, 2011a] Yang, F. and Jacquot, J.-P. (2011a). An Event-B Plug-in for Creating Deadlock-Freeness Theorems. In *4th Brazilian Symposium on Formal Methods, São Paulo, Brazil*.

[Yang and Jacquot, 2011b] Yang, F. and Jacquot, J.-P. (2011b). Scaling Up with Event-B : A Case Study. In *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA*, pages 438–452.

[Yang et al., 2014] Yang, F., Jacquot, J.-P., and Souquières, J. (2014). Proving the Fidelity of Simulations of Event-B Models. In *15th International IEEE Symposium on High-Assurance Systems Engineering, Miami Beach, FL, USA*, pages 89–96.

[Zave, 1997] Zave, P. (1997). Classification of research efforts in requirements engineering. *ACM Comput. Surv.*, 29(4) :315–321.

Annexe A

Train d'atterrissage d'un avion

Annexe B

Machine d'hémodialyse

Résumé

Le développement de spécifications formelles correctes pour des systèmes et logiciels commence par l'analyse et la compréhension des besoins du client. Entre ces besoins décrits en langage naturel et leur spécification définie dans un langage formel précis, un écart existe et rend la tâche de développement de plus en plus difficile à accomplir. Nous sommes face à deux mondes distincts.
Ce travail de thèse a pour objectif d'expliciter et d'établir des interactions entre ces deux mondes et de les faire évoluer en même temps. Par interaction, nous désignons les liens, les échanges et les activités se déroulant entre les différents documents. Parmi ces activités, nous présentons la validation comme un processus rigoureux qui démarre dès l'analyse des besoins et continue tout au long de l'élaboration de leur spécification formelle. Au fur et à mesure du développement, des choix sont effectués et les retours des outils de vérification et de validation permettent de détecter des lacunes aussi bien dans les besoins que dans la spécification. L'évolution des deux mondes est décrite via l'introduction d'un nouveau besoin dans un système existant et à travers l'application de patrons de développement. Ces patrons gèrent à la fois les besoins et la spécification formelle associée ; ils sont élaborés à partir de la description de la forme des besoins. Ils facilitent la tâche de développement et aident à éviter les risques d'oublis. Quel que soit le choix, des questions se posent tout au long du développement et permettent de déceler des lacunes, oublis ou ambiguïtés dans l'existant.

Mots-clés: exigences, spécification formelle, liens, outils, vérification, validation

Abstract

The development of correct formal specifications for systems and software begins with the analysis and understanding of client requirements. Between these requirements described in natural language and their specification defined in a specific formal language, a gap exists and makes the task of development more and more difficult to accomplish. We are facing two different worlds. This thesis aims to clarify and establish interactions between these two worlds and to evolve them together. By interaction, we mean all the links, exchanges and activities taking place between the different documents. Among these activities, we present the validation as a rigorous process that starts from the requirements analysis and continues throughout the development of their formal specification. As development progresses, choices are made and feedbacks from verification and validation tools can detect shortcomings in requirements as well as in the specification. The evolution of the two worlds is described via the introduction of a new requirement into an existing system and through the application of development patterns. These patterns manage both the requirements and their associated formal specifications ; they are elaborated from the description of the form of the requirements in the client document. They facilitate the task of development and help to avoid the risk of oversights. Whatever the choice, the proposed approach is guided by questions accompanying the evolution of the whole system and makes it possible to detect imperfections, omissions or ambiguities in the existing.

Keywords: Requirements, Formal Specification, Links, Tools, Verification, Validation