De l'élicitation des besoins à la spécification formelle

Maritta Heisel [*] **— Jeanine Souquières** [**]

[*] *Fakultät für Informatik*
Universität Magdeburg
D-39016 Magdeburg
heisel@cs.uni-magdeburg.de

[**] *LORIA—Université Nancy 2*
B.P. 239
F-54506 Vandœuvre-lès-Nancy
Jeanine.Souquieres@loria.fr

RÉSUMÉ. *Cet article présente une méthode pour l'élicitation et l'expression des besoins. Elle est basée sur une analyse détaillée des besoins permettant une meilleure compréhension du problème à résoudre. L'approche – qui établit une distinction claire entre besoins et spécifications – conduit naturellement à l'expression d'une spécification formelle. Elle ne présente pas de nouveaux langages mais repose sur l'utilisation de techniques existantes. Les agendas sont utilisés pour décrire la méthode.*

ABSTRACT. *This paper proposes a method for the elicitation and the expression of requirements.It is based on a detailed analysis of requirements, leading to a better understanding of the problem to be solved. The approach – which clearly distinguishes between requirements and specifications – leads to the expression of a formal specification in a natural way. It does not introduce new languages but builds on known techniques. Agendas are used to describe the method.*

MOTS-CLÉS : *Agenda, élicitation des besoins, interactions, méthode, spécifications formelles, validation.*

KEY WORDS : *Agenda, requirements elicitation, interactions, formal specifications, method, validation.*

1. Introduction

L'élicitation des besoins pour un système informatique est une activité conceptuelle complexe, nécessitant des efforts importants de mise en œuvre. Basée sur la communication entre clients et analystes, elle conduit souvent à des incompréhensions. D'une part, les clients ne sont pas toujours conscients de leurs besoins. D'autre part, les analystes ne possèdent pas la même connaissance du domaine d'application que les clients. Certains besoins sont tellement évidents pour les clients qu'ils ne méritent pas, à leurs yeux, d'être mentionnés et demeurent ignorés des analystes. Les clients n'ont pas toujours une vision claire de ce que le système doit faire et changent d'avis en cours de développement. Ils savent enfin ce qu'ils veulent lorsqu'ils utilisent le système et qu'il ne répond pas à leur attente [ANT 98].

Proposer des guides méthodologiques dès les premières phases du cycle de vie est crucial pour augmenter la fiabilité du logiciel et en diminuer les coûts. Les erreurs d'analyse sont les plus nombreuses, les plus tenaces et les plus dangereuses pour la suite du développement. Elles sont aussi les plus coûteuses à corriger : une erreur d'analyse détectée en phase de conception coûte cinq fois plus cher que si elle était découverte pendant la phase d'analyse, ce rapport passant à deux cent fois si la détection a lieu en phase d'exploitation. Ces résultats, issus d'enquêtes récentes sur les réussites et les échecs des projets informatiques, ont été publiés par le Standish Group (disponibles à l'adresse //www.standishgroup.com/chaos.html).

L'utilisation de techniques formelles durant les premières phases du cycle de vie d'un logiciel est maintenant reconnue et acceptée. Cependant leur impact dans le monde industriel reste encore limité et la réalisation de logiciels sûrs est toujours un problème. Le passage à l'utilisation réelle de telles techniques par les développeurs nécessite la mise à disposition d'un ensemble de guides méthodologiques et d'outils. Durant ces dernières années, un effort de recherche important a été fourni pour la définition de langages formels. Ces travaux souffrent de certaines lacunes. D'une part, les méthodes formelles proposent un ensemble de notations mais peu de guides méthodologiques, laissant le spécifieur démuni quand à leur utilisation. D'autre part, les techniques formelles ne sont pas complètement intégrées dans le cycle de vie du développement du logiciel et plus particulièrement dans la phase d'analyse. La relation entre les différentes descriptions d'un système et le système réel n'est pas toujours évidente. Le point de départ du développement des spécifications formelles est difficile à identifier, le cahier des charges fourni par le client restant très difficile à exploiter. L'étape de spécification commence souvent avec une courte description du système à implanter et des détails sont ajoutés durant son élaboration. L'analyse des besoins du client et de son environnement n'est pas réellement prise en compte. Contrairement à cette pratique, nous sommes d'avis que l'élicitation des besoins et le passage à une spécification formelle devraient être abordées séparément.

La complémentarité entre description informelle et description formelle est importante. La description informelle ne doit pas être négligée ; elle constitue la base de départ de l'analyse et reste une référence pour le projet en cours de développement. Elle joue un rôle dans la validation par rapport aux besoins du client et favorise le

dialogue entre les différents intervenants du projet. La description formelle est utilisée pour la détection des incohérences et des ambiguïtés, et pour les raisonnements formels. Ces deux types de description ont une influence mutuelle et favorisent l'obtention d'un point de départ stable préalable aux phases de réalisation du système informatique.

Une tendance naturelle consiste à montrer le résultat final de la formalisation en oubliant toutes les étapes de raffinement et les remises en cause effectuées. Cependant, ces différentes étapes sont importantes dans les processus d'acquisition et de formalisation et la qualité du logiciel pourra être améliorée par une meilleure connaissance et maîtrise du processus de développement.

Notre approche explicite les différentes étapes de développement. Elle commence avec la phase d'élicitation des besoins, le résultat de cette phase servant de point de départ au développement de la spécification formelle.

Le paragraphe 2 présente une méthode pour l'élicitation des besoins. Elle prévoit une analyse détaillée des besoins amenant à une meilleure compréhension du problème. Le passage à la spécification formelle et sa vérification par rapport aux besoins sont présentés dans le paragraphe 3. L'approche est illustrée avec l'étude de cas d'un ascenseur. Le paragraphe 4 présente un état de l'art sur le sujet. Un résumé de notre contribution est proposé dans le paragraphe 5.

2. Elicitation des besoins

L'approche proposée pour guider l'élicitation des besoins s'inspire des travaux de Jackson et Zave [JAC 95, ZAV 97] et des premières phases des méthodes et des notations orientées objet comme OMT [RUM 91], FUSION [COL 94] ou UML [FOW 97]. Le processus d'élicitation commence par une réflexion sur le domaine d'application et une description informelle des besoins du client. Il consiste à reprendre les éléments du cahier des charges pour en faire une analyse exhaustive en vue de le préciser et de supprimer les ambiguïtés. Un passage de l'informel au formel est réalisé très tôt dans les étapes de développement, obligeant à analyser les besoins de manière détaillée. Ce processus est très délicat, car il fait intervenir différents cadres linguistiques. Des difficultés rencontrées lors de la formalisation d'un besoin peuvent être révélatrices d'un problème.

Les besoins font référence au système à réaliser dans son ensemble, alors que la spécification fait référence à la seule partie du système qui sera implantée par un logiciel, voir figure 1.

Agenda pour l'élicitation des besoins. L'élicitation des besoins s'effectue en six étapes, chacune d'elles apportant un guide méthodologique à l'analyse et à la détection des incohérences et des non-dits dans les cahiers des charges informels. Les agendas [HEI 98c] sont utilisés pour décrire notre méthode. Un agenda est constitué d'un ensemble d'étapes à réaliser pour satisfaire une tâche donnée. Le résultat de la résolution d'une tâche est un document exprimé dans un langage choisi. Un agenda contient une description informelle de chaque étape.

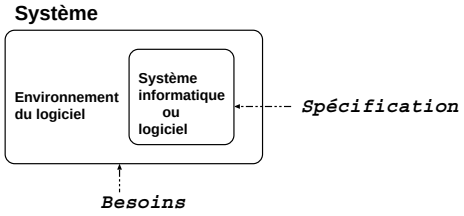

Figure 1. *Besoins/spécifications*

Les agendas sont présentés par des tables (exemple figure 2) ; ces tables expriment un sommaire de la méthode. Des explications complémentaires sont données pour chaque étape. Les agendas ne sont pas la seule manière de décrire et de guider le processus de développement [LÉV 97], mais ils assurent une certaine garantie de la qualité du produit obtenu grâce aux conditions de validation associées aux différentes étapes. Ces conditions de validation précisent des conditions sémantiques nécessaires que les résultats de l'étape doivent satisfaire. Certaines de ces conditions peuvent être automatisées, elles seront notées $\vdash$ dans la suite, d'autres doivent être vérifiées manuellement, notées $\circ$. De manière similaire au modèle de développement de logiciels en spirale [BOE 88], ces différentes étapes peuvent être répétées pour atteindre les objectifs, certaines étapes pouvant mettre en évidence des erreurs et des omissions dans les étapes précédentes.

Notre méthode s'applique aussi bien aux systèmes réactifs qu'aux systèmes transformationnels. Les systèmes réactifs réagissent à des événements en provenance de l'environnement. Les systèmes transformationnels, par contre, autorisent la manipulation de données de taille importante à l'aide d'opérations. Les événements jouent un rôle important dans notre approche car ils déclenchent des transitions d'état. Pour cela, nous associons à chaque opération d'un système transformationnel deux événements, l'un pour le déclenchement de l'opération et l'autre pour sa terminaison. Ainsi les systèmes partiellement réactifs et partiellement transformationnels peuvent être aussi traités.

Les résultats des cinq premières étapes de l'agenda A présenté figure 2 sont des documents informels. Ces étapes peuvent être effectuées en parallèle. La sixième étape consiste à formaliser les besoins et à détecter des interactions et des incohérences.

Etude de cas : l'ascenseur. Nous appliquerons l'approche proposée à l'élicitation des besoins d'un ascenseur desservant l'ensemble des étages d'un immeuble. Il s'agit d'une vue simplifiée de l'ascenseur, celui-ci ne prenant pas en compte la charge et ne proposant pas de mécanismes permettant d'interrompre la fermeture de la porte.

Dans la suite, nous détaillerons chacune des étapes avec son illustration sur l'étude de cas.

No	Étape	Moyen d'expression	Conditions de validation
1	Introduire le vocabulaire du domaine.	description informelle	
2	Identifier les opérations pertinentes avec leurs paramètres.	liste de noms	
3	Identifier tous les événements pertinents avec leurs paramètres.	liste de noms	
1 à 3		visualisations graphiques	∘ Chaque notion introduite dans l'étape 1 doit être en relation avec un événement ou une opération.
4	Classer les événements introduits.	table	⊢ Il n'y a pas d'événements commandés par le système informatique et non partagés par l'environnement.
5	Enoncer les faits, suppositions et besoins du système.	description informelle	∘ Toutes les notions utilisées dans les faits, les suppositions et les besoins doivent être mises en relation avec le domaine ou les événements introduits dans les étapes 1 à 3. ∘ Chaque notion du domaine et chaque événement doit être en relation avec au moins un fait, une supposition ou un besoin.
6	Formaliser faits, suppositions et besoins comme des contraintes sur les traces permises des événements du système.	formules sur les traces	∘ Chacun des besoins identifiés dans l'étape 5 doit être exprimé. ⊢ L'ensemble des contraintes doit être consistant. ∘ Pour chaque prédicat introduit dans les besoins, les événements qui le rendent vrai et ceux qui le rendent faux doivent être partagés par le système informatique.

Figure 2. *Agenda A pour l'élicitation des besoins*

2.1. *Introduire le vocabulaire du domaine*

Les différentes notions du domaine d'application sont exprimées textuellement.

Etude de cas. *L'ascenseur* est muni d'une *porte* et d'un ensemble de *boutons*. Les *utilisateurs* peuvent demander l'ascenseur depuis n'importe quel *étage* (chaque étage est muni d'un bouton d'appel) et sélectionner l'étage d'arrivée grâce aux boutons situés dans l'ascenseur.

2.2. *Identifier les opérations pertinentes avec leurs paramètres*

Cette étape concerne la partie non-réactive du système. Pour les systèmes purement réactifs, elle peut être vide. Les opérations du système sont habituellement indépendantes des composants physiques mais font référence aux informations disponibles dans le système informatique. A chaque opération Op, nous associons deux événements $OpInvocation$ et $OpTermination$, le premier étant contrôlé par l'environnement et partagé par le système informatique, le deuxième étant contrôlé par le système informatique et partagé par l'environnement (voir étape A4).

Etude de cas. L'ascenseur étant un système purement réactif, aucune opération n'est identifiée.

2.3. *Identifier tous les événements pertinents avec leurs paramètres*

Cette étape concerne la partie réactive du système. Dans le cas de systèmes transformationnels purs, cet ensemble peut être vide. Un événement est normalement en relation avec une partie du système physique, comme par exemple les boutons d'appel ou la porte d'un ascenseur.

La condition de validation associée aux trois premières étapes de l'agenda A vérifie que toutes les notions introduites dans le vocabulaire du domaine sont bien prises en compte, soit dans la partie réactive, soit dans la partie transformationnelle du système.

Des représentations graphiques peuvent être utilisées pour visualiser le résultat de ces trois premières étapes, celles-ci correspondant à l'expression de la partie statique du domaine d'application.

Etude de cas. Les événements suivants sont identifiés. Ils permettent de préciser le comportement attendu de l'ascenseur vis-à-vis de l'utilisateur : $open$ et $close$ font référence à la porte, $move$ et $stop$ font référence au mouvement de l'ascenseur, $enter$, $leave$ et $press(button)$ font référence aux utilisateurs de l'ascenseur.

La figure 3 présente une visualisation graphique du résultat de ces trois premières étapes. Les losanges représentent l'agrégation (un bouton est une partie de l'ascenseur) et les cercles la cardinalité (un ascenseur dispose de plusieurs boutons). Les demi-cercles représentent les relations. Cette visualisation ne préjuge pas d'une quelconque organisation future du système et les différentes entités ne se transformeront pas forcément en des objets informatiques.

Nous avons fait le choix de ne pas distinguer les boutons d'appels, qu'ils soient dans l'ascenseur ou dans les étages : dans les deux cas, ils servent à commander l'ascenseur. Ce choix, suffisant pour décrire les besoins de l'ascenseur simplifié, pourrait être remis en cause pour décrire des besoins plus complexes.

La condition de validation associée aux trois premières étapes peut facilement être vérifiée en regardant la figure 3 : chaque notion introduite dans l'étape A1 est en relation avec au minimum un événement.

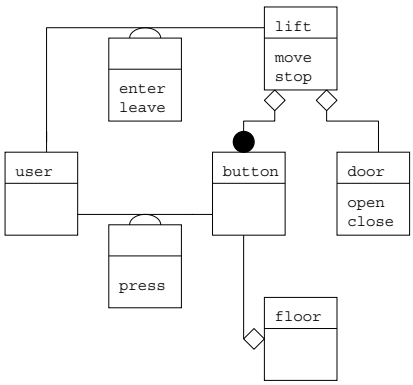

Figure 3. *Visualisation graphique résultante des trois premières étapes*

2.4. *Classer les événements introduits*

La classification adoptée est celle proposée par Jackson et Zave [JAC 95]. Elle consiste à identifier : (i) les événements contrôlés par l'environnement et non partagés par le système informatique, (ii) les événements contrôlés par l'environnement et observables par le système informatique, (iii) les événements contrôlés par le système informatique et observables par l'environnement, (iv) les événements contrôlés par le système informatique et non partagés par l'environnement. Cette classification est importante pour le développement car l'analyste est amené à décider quelle part de la tâche sera réalisée par le logiciel. Il est clair que les actions internes du système informatique ne font pas partie des besoins mais interviendront dans la spécification.

Etude de cas. Pour chaque événement, nous précisons qui le commande et qui l'observe. Cette classification est fortement liée aux caractéristiques du système physique et aux choix du système à implanter. Dans le cas de l'ascenseur, comme visualisé figure 3, l'environnement du système informatique est représenté par les utilisateurs. Les événements $enter$ et $leave$ sont commandés par l'environnement (par les utilisateurs). Si aucun contrôle sur la charge de l'ascenseur n'est effectué, il n'y a pas de raison pour que ces événements soient partagés par le système informatique. Par contre, si les besoins évoluent et précisent que l'ascenseur ne peut fermer ses portes lorsqu'il est en surcharge, alors ces deux événements devront être partagés par le système informatique. Cette évolution entraînera une évolution du système physique, avec par exemple, l'installation de capteurs.

Les événements commandés par l'environnement et non partagés par le système informatique sont $enter$ et $leave$. Un seul événement, $press(button)$, est commandé par l'environnement et observable par le système informatique. Les événements commandés par le système informatique et observables par l'environnement sont $open$, $close$, $move$ et $stop$. Comme requis par la condition de validation de cette étape, il n'y a pas d'événements commandés par le système informatique et non partagés par l'environnement.

2.5. *Enoncer les faits, suppositions et besoins du système*

L'élicitation des besoins vise à décrire l'ensemble des propriétés du système à réaliser. Ces propriétés peuvent être de différentes natures. Certains besoins ne peuvent être imposés et décrivent un comportement correspondant à un utilisateur parfait. Il est possible que le système ne soit pas à même de satisfaire correctement ces besoins dans tous les cas, mais seulement sous l'hypothèse d'un comportement attendu de l'utilisateur. Ces conditions sont exprimées à l'aide de *suppositions*. Les *faits* font référence à des propriétés du domaine d'application, comme par exemple des propriétés physiques : la porte de l'ascenseur ne peut s'ouvrir que si elle est dans l'état fermé. Les *besoins* contraignent la spécification et l'implantation du système. Leur satisfaction pourra être prouvée une fois la spécification et l'implantation terminées. Dans la phase de spécification, on pourra montrer que la modélisation du système satisfait les besoins en prenant en compte les faits sous l'hypothèse que les suppositions soient satisfaites.

Etude de cas. L'ascenseur considéré a le comportement attendu suivant.

Besoin 1. L'ascenseur est commandé à partir d'un bouton.

Besoin 2. Il est toujours possible d'appuyer sur un bouton.

Besoin 3. L'ascenseur ouvre sa porte après s'être arrêté.

Besoin 4. La porte reste ouverte un certain temps puis se referme automatiquement.

Besoin 5. S'il n'y a pas de demandes à servir, l'ascenseur reste au dernier étage servi, la porte fermée.

Besoin 6. Tant qu'il y a des demandes non servies, l'ascenseur continue à servir ces demandes.

Besoin 7. Si l'ascenseur est arrêté à un étage avec sa porte ouverte, une demande provenant de cet étage n'est pas prise en compte.

Besoin 8. Si l'ascenseur est arrêté à un étage avec sa porte fermée et qu'il reçoit une demande de cet étage, il ouvre sa porte.

Besoin 9. La porte est fermée pendant le mouvement de l'ascenseur.

Fait 1. La porte peut s'ouvrir seulement si elle est dans l'état fermé. Elle ne peut se fermer que si elle est dans l'état ouvert.

Supposition 1. Quand l'ascenseur arrive à l'étage demandé, l'utilisateur responsable de la demande entre/sort de l'ascenseur.

Les besoins décrits ici ne précisent pas le protocole de service de l'ascenseur. Les utilisateurs de l'ascenseur souhaitent que tout appel soit fatalement servi. Mais ce besoin est irréalisable car il dépend du comportement des utilisateurs. En effet, il est toujours possible de bloquer l'ascenseur à un étage en appuyant sur le bouton de cet étage dès que l'ascenseur a fermé ses portes.

Les conditions de validation de l'étape A5 sont faciles à vérifier : chaque notion et chaque événement introduit dans les étapes précédentes est utilisé dans au moins un fait, un besoin ou une supposition et vice versa (par exemple, le fait 1 relie la porte aux événements $open$ et $close$).

2.6. *Formaliser faits, suppositions et besoins du système*

Dans cette étape, les contraintes sont formalisées les unes après les autres. Chaque nouvelle contrainte est analysée en relation avec l'ensemble des contraintes déjà formalisées avant d'être intégrée à cet ensemble. Une procédure automatique de recherche des interactions correspondant aux comportements non désirables est décrite dans [HEI 98d]. Pour formaliser les besoins, nous utilisons des traces, c'est-à-dire des suites d'événements se produisant sur un état du système à un instant donné. Le système démarre à l'état S_1. Lorsqu'un événement e_1 se produit à l'instant t_1, le système passe à l'état S_2, et ainsi de suite :

$$S_1 \xrightarrow[t_1]{e_1} S_2 \xrightarrow[t_2]{e_2} \ldots \; S_n \xrightarrow[t_n]{e_n} S_{n+1} \; \ldots$$

Soit Tr l'ensemble des traces permises. Les contraintes sont des formules qui restreignent l'ensemble Tr. Pour une trace $tr \in Tr$, $tr(i)$ dénote le i-ème élément d'une trace, $tr(i).s$ l'état du i-ème élément et $tr(i).e$ l'événement qui se produit dans cet état. Chaque préfixe d'une trace valide est aussi une trace valide. Une spécification formelle des traces est donnée dans l'annexe A.

Pour exprimer les contraintes, des prédicats sur l'état global du système peuvent être introduits. Pour chaque prédicat introduit, on précisera les événements qui le rendent vrai et ceux qui le rendent faux.

Les contraintes décrites ne déterminent pas entièrement le comportement du système et n'ont pas pour objectif de restreindre sans nécessité la spécification. Toute spécification remplissant les contraintes exprimées est admissible. L'utilisation des contraintes pour décrire le comportement du système rend possible :

— l'expression de besoins négatifs, comme par exemple des propriétés de sécurité précisant les comportements non autorisés,

— la description de scénarios correspondant à des exemples de comportements attendus, ou à différents points de vue,

— la description de propriétés de vivacité.

Dans les documents fournis par les clients, les besoins sont souvent exprimés de manière inconsistante. Nous proposons de résoudre les conflits dès l'expression des besoins, le résultat de cette phase servant de contrat entre le client et le fournisseur de logiciels. De plus, une spécification correcte ne peut être obtenue à partir de besoins inconsistants. Une représentation formelle des besoins est plus adaptée pour détecter les inconsistances qu'une représentation informelle.

Notre approche présente, avec les étapes A1 à A5, des principes pour une analyse informelle et rigoureuse des besoins. L'étape A6 concerne la formalisation d'un ensemble de fragments autonomes décrivant les propriétés du système, cette formalisation facilitant la détection des incohérences et le passage à la spécification formelle.

Etude de cas. Pour des problèmes de place, nous présentons uniquement la formalisation du fait 1 et des besoins 1, 2, 7, 8 et 9. Une formalisation complète se trouve dans [HEI 98a].

Fait 1 : La porte peut s'ouvrir seulement si elle est dans l'état fermé. Elle ne peut se fermer que si elle est dans l'état ouvert.

$$\begin{array}{l}\forall\, tr : Tr \bullet \forall\, i : \mathrm{dom}\, tr \bullet \\ \qquad (tr(i).e = open \Rightarrow tr(i).s = door_closed) \wedge \\ \qquad (tr(i).e = close \Rightarrow tr(i).s = door_open)\end{array}$$

Nous avons introduit les prédicats $door_closed$ et $door_open$ sur l'état de l'ascenseur. Ils sont reliés aux événements $open$ et $close$ partagés par le système.

Besoin 1 : L'ascenseur est commandé à partir d'un bouton.

$$\begin{array}{l}\forall\, tr : Tr \bullet (\forall\, i : \mathrm{dom}\, tr;\ b : Button \bullet \\ \qquad call(tr(i).s, floor(b)) \Rightarrow \exists\, j : \mathrm{dom}\, tr \mid j < i \bullet tr(j).e = press(b))\end{array}$$

Chaque appel (exprimé par $call(tr(i).s, floor(b))$) est précédé d'un événement $press$ (exprimé par $tr(j).e = press(b)$). Le prédicat $call$ sert à mémoriser l'ensemble des appels à servir. Il est relié aux événements $press$ et $stop$. La fonction $floor$ donne l'étage associé à un bouton.

Besoin 2 : Il est toujours possible d'appuyer sur un bouton.

$$\forall\, tr : Tr;\ b : Button \bullet tr \frown \langle press(b) \rangle \in Tr$$

Cette formule exprime que toute trace peut être suivie de l'événement $press$.

Besoin 7 : Si l'ascenseur est arrêté à un étage avec sa porte ouverte, une demande provenant de cet étage n'est pas prise en compte.

Pour rendre possible l'expression de ce besoin, deux nouveaux prédicats sont nécessaires : $halted$ précise si l'ascenseur est arrêté et at si l'ascenseur se trouve à l'étage fl . Ces deux prédicats sont reliés aux événements $stop$ et $move$.

$$\begin{array}{l}\forall\, tr : Tr \bullet (\forall\, i : \mathrm{dom}\, tr;\ b : Button \mid i \neq \#tr \bullet \\ \quad halted(tr(i).s) \wedge at(tr(i).s, floor(b)) \wedge door_open(tr(i).s) \wedge \\ \quad tr(i).e = press(b) \\ \qquad \Rightarrow \forall\, fl : Floor \bullet call(tr(i+1).s, fl) \Leftrightarrow call(tr(i).s, fl))\end{array}$$

Considérons le i-ième élément de la trace tr, celui-ci n'étant pas son dernier élément. Dans cet état, l'ascenseur est à l'arrêt à l'étage fl (exprimé par $halted(tr(i).s)$ et $at(tr(i).s, floor(b))$) avec sa porte ouverte ($door_open(tr(i).s)$). Une demande provenant de cet étage fl (exprimée par $tr(i).e = press(b)$) ne modifie pas le prédicat $call$ dans l'état suivant du système, i.e. l'ensemble des appels à satisfaire demeure inchangé.

Besoin 8 : Si l'ascenseur est arrêté à un étage avec sa porte fermée et qu'il reçoit une demande de cet l'étage, il ouvre sa porte.

Un nombre indéterminé d'événements $press$ peut se produire. Il faut supprimer ces événements de la trace pour laquelle nos voulons exprimer des conditions de vivacité. C'est l'objectif de la fonction $remove$ décrite en annexe A, qui élimine certains événements d'une trace, ces événements étant donnés en paramètre.

$$\begin{array}{l}\textbf{let}\ evs == \{b : Button \bullet press(b)\} \bullet \\ (\forall\, tr : Tr \bullet \forall\, tr_1, tr_2 : TRACE \mid tr = tr_1 \frown tr_2 \wedge remove(tr_2, evs) \neq \langle\rangle \wedge \\ \quad halted(tr_2(1).s) \wedge door_closed(tr_2(1).s) \wedge tr_2(1).e = press(b) \wedge \\ \quad at(tr_2(1).s, floor(b)) \bullet remove(tr_2, evs)(1).e = open)\end{array}$$

Il existe un état dans lequel l'ascenseur est à l'arrêt avec sa porte fermée ($halted(tr_2(1).s) \wedge door_closed(tr_2(1).s)$). Considérons le comportement de l'ascenseur dans cet état. Si l'événement $press$ se produit ($tr_2(1).e = press(b)$), en provenance de l'étage d'arrêt ($at(tr_2(1).s, floor(b))$), le prochain événement différent de $press$ doit être $open$.

Besoin 9 : La porte est fermée pendant le mouvement de l'ascenseur.

Ce besoin décrit un invariant du système.

$$\forall\, tr : Tr \bullet (\forall\, i : \mathrm{dom}\, tr \bullet \neg\, halted(tr(i).s) \Rightarrow door_closed(tr(i).s))$$

Considérons le i-ième élément de la trace admissible tr. Si l'ascenseur n'est pas arrêté dans cet état, alors sa porte doit être fermée.

Ceci conclut la formalisation des besoins pour notre ascenseur. La première condition de validation de l'étape A6 est satisfaite. Pour vérifier la consistance des besoins exprimée par la deuxième condition de validation, une analyse complète de l'ensemble des contraintes est effectuée. Elle n'est pas présentée ici. La troisième condition de validation a été prise en compte au fur et à mesure de la formalisation.

Détection des interactions et validation des besoins par rapport au client.

L'introduction de nouveaux besoins comme par exemple la prise en compte de la charge de l'ascenseur (les portes restent ouvertes si la charge autorisée est dépassée) ou des appels prioritaires ont été étudiés ainsi que la détection d'interactions et leur résolution [HEI 98a].

Avec la démarche proposée, les activités de validation des besoins par rapport aux clients s'intègrent naturellement dans le processus. Les interactions avec le client sont importantes : à la fin de l'étape A5, lors de la détection d'une interaction pour la résoudre et à la fin du processus d'élicitation des besoins.

3. Passage à la spécification formelle

Jackson et Zave [JAC 95] considèrent les spécifications comme des besoins particuliers : un besoin est une spécification si tous les événements contraints par ce besoin sont contrôlés par le logiciel et toutes les informations dont il dépend sont partagées

par le logiciel et font référence au passé et non au futur. Les besoins ne parlent pas de l'état du logiciel.

Par nature, les besoins formalisés représentent des fragments. Ils ne constituent donc pas un modèle et ne déterminent pas entièrement le comportement du logiciel ; ils servent à en exprimer des propriétés.

C'est pourquoi, par opposition, nous considérons qu'une spécification n'est pas un ensemble de besoins. Une spécification est un modèle du logiciel à développer satisfaisant les besoins. Elle est développée pas à pas, en exprimant les différentes propriétés du logiciel à réaliser. Notre objectif est de faciliter le travail du spécifieur pour construire ce modèle, en partant des documents résultants de l'élicitation des besoins. L'approche proposée est incrémentale et permet de maîtriser la complexité : les besoins ne sont pas indépendants mais ils sont pris en compte les uns après les autres. La prise en compte d'un besoin peut faire évoluer en même temps différentes parties de la spécification. Celle-ci servant de base aux travaux d'implantation peut, par opposition aux besoins, faire des choix sur le logiciel et exprimer des détails qui ne sont pas directement observables dans l'environnement.

Les liens de traçabilité entre besoins et spécification sont établis via les agendas. Une confrontation entre les deux formalisations proposées permet de découvrir des incohérences et des incomplétudes.

La phase d'élicitation des besoins est indépendante du langage de spécification utilisé. Même si des constructions générales peuvent être identifiées [SOU 96], le développement de la spécification n'est pas complètement indépendant du langage de spécification et de son pouvoir d'expression. Dans la suite, nous utiliserons la notation Z [SPI 92]. Celle-ci a été choisie pour traiter l'exemple de l'ascenseur parce que l'état de l'ascenseur est assez complexe et qu'une description de cet état est indispensable pour spécifier le comportement du système. Le comportement de l'ascenseur sera contraint par les préconditions des opérations.

Le paragraphe 3.1 présente une description de la méthode à l'aide d'agendas. Cette méthode est ensuite appliquée à l'étude de cas de l'ascenseur, paragraphe 3.2. Le paragraphe 3.3 présente une notion formelle de correction d'une spécification par rapport à un ensemble de besoins formalisés à l'aide de traces.

3.1. *Agenda pour le développement de spécifications à partir des besoins*

Le point de départ de la spécification est constitué de l'ensemble des documents obtenu dans la phase d'élicitation des besoins présentée dans le paragraphe 2. L'approche est exprimée à l'aide de l'agenda B décrit figure 4.

Étape B 1 *Définir une première approximation de l'état du système et de son état initial*

Cette étape consiste à donner une première approximation de l'état du système de manière à ce que les prédicats introduits dans la phase d'élicitation des besoins

No	Etape	Conditions de validation
1	Définir une première approximation de l'état du système et de son état initial.	
2	Incrémenter la spécification en incorporant un à un chacun des besoins.	⊢ Les contraintes exprimant les faits ne doivent pas être violées. ⊢ Tous les événements introduits dans la phase d'élicitation des besoins doivent être pris en compte. ∘ Les préconditions des opérations doivent être définies de manière appropriée. ⊢ Les postconditions des opérations doivent être complètement définies. ⊢ Chaque prédicat utilisé dans l'élicitation des besoins doit être défini en terme de l'état du système.

Figure 4. *Agenda B pour le développement de la spécification*

No	Etape	Conditions de validation
1	Lister les événements présents dans la contrainte.	
2	Pour chaque événement de la liste, définir l'opération Z correspondante ou mettre à jour une opération existante.	⊢ Les contraintes exprimant les faits ne doivent pas être violées par la définition de l'opération.

Figure 5. *Agenda C pour la prise en compte d'un besoin*

puissent être définis. L'état initial correspondant est défini en parallèle.

Étape B 2 *Incrémenter la spécification en incorporant un à un chacun des besoins*

Cette étape est guidée par l'agenda C proposé figure 5. L'idée de base est de définir une opération Z pour chaque événement identifié dans l'étape A3 et partagé avec le logiciel ainsi que pour chaque opération identifiée dans l'étape A2.

L'étape C1 peut être effectuée par un simple examen syntaxique de la contrainte. L'étape C2 est plus complexe et peut nécessiter des révisions de l'état courant de la spécification [LÉV 96]. L'état du système et ses opérations doivent être revisités pour prendre en compte les évolutions possibles introduites par la nouvelle contrainte. La propagation des modifications est ici un élément important [SAD 97]. Rajouter une nouvelle contrainte peut entraîner les modifications suivantes :

– ajouter ou modifier des composants de l'état du système,

– ajouter ou modifier des types de données,
– ajouter ou modifier un invariant,
– propager ces modifications dans l'état courant de la spécification.

Toutes les définitions et modifications doivent respecter les propriétés du domaine. Cette condition de validation est propagée dans l'étape B2.

Concernant la troisième condition de validation de cette étape, les préconditions pour les opérations du système (voir l'étape A2) doivent être aussi faibles que possible pour garantir la robustesse du système. La situation est différente pour les opérations correspondant à la partie réactive, de fortes préconditions pouvant renforcer leur ordre d'application. Ceci est une manière d'exprimer la description du comportement en Z.

La quatrième condition de validation précise que chaque opération définit tous les attributs de l'état du système. Si le non-déterminisme est introduit délibéremment, il doit être justifié.

La cinquième condition de validation porte sur la définition finale de l'état du système. Celui-ci doit contenir suffisamment d'informations pour définir tous les prédicats utilisés pour exprimer l'ensemble des besoins.

Les agendas pour l'élicitation des besoins et le développement de la spécification permettent une approche intégrée avec introduction de textes formels dès que possible dans le processus de développement du logiciel. Ils établissent des liens de traçabilité entre les besoins et les éléments de la spécification. Ils laissent une place importante aux descriptions informelles qui sont mises à jour tout au long du processus. L'expression formelle des besoins, des suppositions et des faits sous la forme de contraintes apporte un guide au développement de la spécification.

3.2. *Etude de cas : l'ascenseur*

En prenant comme point de départ les documents résultants de la phase d'élicitation des besoins présentés dans le paragraphe 2, nous utiliserons les agendas B et C pour développer une spécification formelle de l'ascenseur.

3.2.1. *Étape B1 : Définir une première approximation de l'état du système et son état initial*

Dans la phase précédente, les prédicats $door_open$, $door_closed$, $call$, at et $halted$ ont été introduits. Notre objectif, en décrivant le schéma d'état $Lift$ est de définir ces prédicats, ce qui nous conduit aux définitions suivantes.

$$DOOR_STATE ::= open \mid closed$$

$$MOVEMENT ::= halted \mid moving$$

$$DIRECTION ::= up \mid down$$

$$[BUTTON]$$

$$FLOOR : \mathbb{P}\,\mathbb{N}$$

$$no_floor, bottom_floor, top_floor : FLOOR$$

$$no_floor \notin bottom_floor \,..\, top_floor$$

$$floor : BUTTON \longrightarrow FLOOR$$

$$\operatorname{ran} floor = bottom_floor \,..\, top_floor$$

Les constantes $bottom_floor$, top_floor et no_floor désignent respectivement l'étage du bas, le dernier étage et un étage indéfini. La fonction $floor$ établit le lien entre les boutons et les étages.

L'état du système et son état initial sont décrits par les schémas $Lift$ et $Init_lift$.

$Lift$

$$door : DOOR_STATE$$
$$mvt : MOVEMENT$$
$$direction : DIRECTION$$
$$call : \mathbb{P}\, FLOOR$$
$$at, passes_by : FLOOR$$

$$\exists_1 fl : \{at, passes_by\} \bullet fl = no_floor \quad (a)$$
$$mvt = halted \Leftrightarrow passes_by = no_floor \quad (b)$$
$$no_floor \notin call \quad (c)$$

$Init_lift$

$$Lift'$$

$$door' = closed$$
$$mvt' = halted$$
$$direction' = up$$
$$call' = \emptyset$$
$$at' = bottom_floor$$
$$passes_by' = no_floor$$

Les attributs de l'ascenseur désignent respectivement l'état de la porte, son mouvement, sa direction, la mémorisation de l'ensemble des appels à servir et le fait qu'il est à un étage donné ou en train de passer par un étage. L'invariant (a) précise que l'ascenseur ne peut pas, en même temps, être à l'arrêt et passer par un étage. L'invariant (b) établit une relation entre l'attribut de mouvement et le passage par un étage : si l'ascenseur est à l'arrêt, il ne passe pas par un étage et vice versa. L'invariant (c) précise qu'il ne peut pas y avoir d'appel provenant de no_floor, no_floor ne représentant pas un étage.

Cette première description de l'état de l'ascenseur permet de définir les prédicats introduits dans la formalisation des besoins (par exemple $door_open \Leftrightarrow door = open$ ou $at(fl) \Leftrightarrow at = fl$).

A l'état initial, l'ascenseur est arrêté à l'étage du bas, porte fermée et il n'y a pas d'appels à servir.

3.2.2. *Étape B2 : Incrémenter la spécification, en ajoutant un à un chacun des besoins*

Pour abréger l'écriture des schémas des opérations Z, nous utilisons la convention dans laquelle $\Delta Lift$ est remplacé par la liste des variables d'état modifiées par l'opération. Les variables non mentionnées restent inchangées.

Pour chaque événement partagé par le logiciel, nous introduisons une opération. Sa définition est guidée par l'analyse de chaque besoin en prenant en compte les propriétés du domaine exprimées par le fait 1.

Fait 1. La porte peut s'ouvrir seulement si elle est dans l'état fermé. Elle ne peut se fermer que si elle est dans l'état ouvert.

Ce fait utilise deux événements, $open$ et $close$ traduits par les deux opérations suivantes.

$$\begin{array}{|l} Open \\ \Delta(door) \\ \hline door = closed \\ door' = open \\ \hline \end{array} \qquad \begin{array}{|l} Close \\ \Delta(door) \\ \hline door = open \\ door' = closed \\ \hline \end{array}$$

Besoin 1. L'ascenseur est commandé à partir d'un bouton.

Ce besoin fait référence à l'événement $press$ qui modifie l'ensemble des appels à servir. La propriété décrite par le fait 1 est respectée par l'opération $Press$ correspondante.

$$\begin{array}{|l} Press \\ \Delta(call) \\ b? : BUTTON \\ \hline call' = call \cup \{floor(b?)\} \\ \hline \end{array}$$

Besoin 2. Il est toujours possible d'appuyer sur un bouton.

Ce besoin signifie que la précondition de $Press$ ne doit pas être être plus forte que $Lift$, c'est à dire que l'opération $Press$ doit toujours être possible. La définition de $Press$ donnée ci-dessus satisfait cette condition.

La prise en compte des besoins 3 à 6 nous amène à faire évoluer la spécification. Une première définition est donnée pour l'opération $Stop$ avec la prise en compte du besoin 3 ainsi que pour l'opération $Move$ pour le besoin 6. L'opération $Open$ est précisée avec le besoin 3 et l'opération $Close$ avec le besoin 4. Les modifications de la spécification par rapport aux versions précédentes sont notées en gras.

Lift

$door : DOOR_STATE$
$mvt : MOVEMENT$
$direction : DIRECTION$
$call : \mathbb{P}\ FLOOR$
$at, passes_by : FLOOR$
$\mathbf{timer} : \mathbb{N}$
$\mathbf{timer_runs} : \mathbf{YesNo}$
$\mathbf{next_stop} : \mathbf{FLOOR}$

$\exists_1\ fl : \{at, passes_by\} \bullet fl = no_floor$ (a)
$mvt = halted \Leftrightarrow passes_by = no_floor$ (b)
$no_floor \notin call$ (c)
$\mathbf{call} = \emptyset \Rightarrow \mathbf{mvt} = \mathbf{halted}$ (d)
$\mathbf{mvt} \neq \mathbf{halted} \Rightarrow \mathbf{next_stop} \in \mathbf{call}$ (e)
$\mathbf{timer_runs} = \mathbf{yes} \Leftrightarrow \mathbf{door} = \mathbf{open}$ (f)

Stop

$\Delta(mvt, at, passes_by, call, timer)$

$mvt \neq halted$
$next_stop = passes_by$

$mvt' = halted$
$at' = next_stop$
$passes_by' = no_floor$
$call' = call \setminus \{next_stop\}$
$timer' = opening_time$

Pour prendre en compte ces nouvelles définitions, de nouveaux attributs viennent compléter l'état du système : $timer$ pour gérer la durée d'ouverture de la porte exprimée par le besoin 4, $timer_runs$ pour l'état de ce timer (en fonctionnement ou non) et $next_stop$ pour désigner le prochain étage à servir (besoin 6). Ils nécessitent l'introduction d'un type énuméré $YesNo$ à deux valeurs et d'une constante $opening_time$ de type entier positif. L'introduction de ces nouveaux attributs définissant l'état du système doit être propagée dans l'ensemble des opérations existantes.

L'invariant de $Lift$ est complété : l'invariant (d) correspondant au besoin 5 précise que l'ascenseur reste à l'arrêt lorsqu'il n'y a pas d'appel à servir ; l'invariant (e) introduit pour satisfaire le besoin 6 précise que si l'ascenseur est en mouvement, il sert les demandes existantes. L'invariant (f), introduit pour satisfaire le besoin 4, précise que le timer est en fonctionnement lorsque la porte est ouverte.

Open

$\Delta(door, timer_runs)$

$\mathbf{mvt} = \mathbf{halted}$
$door = closed$
$\mathbf{timer} = \mathbf{opening_time}$
$door' = open$
$\mathbf{timer_runs}' = \mathbf{yes}$

Close

$\Delta(door, timer_runs)$

$door = open$
$\mathbf{timer_runs} = \mathbf{yes}$
$\mathbf{timer} = \mathbf{0}$
$door' = closed$
$\mathbf{timer_runs}' = \mathbf{no}$

Le timer, nécessaire pour déclencher la fermeture automatique de la porte après un certain temps, est initialisé à $opening_time$ et mis en marche lors de l'ouverture de la porte. Une opération $ChangeTimer$, non décrite ici, décrémente ce timer. La porte se ferme lorsque le timer prend la valeur zéro. Le timer doit aussi être initialisé dans

l'opération $Stop$ pour activer l'ouverture de la porte après l'arrêt de l'ascenseur.

L'opération $StartMoving$ met en marche l'ascenseur. Elle calcule son prochain arrêt. Ce calcul dépend du protocole de service de l'ascenseur qui est un paramètre de notre spécification. Il fait appel à la fonction $compute_next_stop$, paramétrée par l'étage de passage ou d'arrêt, la direction, le mouvement et l'ensemble des appels à servir.

$StartMoving$

$\Delta(mvt, direction, next_stop, passes_by)$

$mvt = halted$
$door = closed$
$call \neq \emptyset$
$timer = 0$

$mvt' = moving$
$next_stop' = compute_next_stop(at, direction, mvt, call)$
$next_stop' > at \Rightarrow direction' = up \wedge passes_by' = at + 1$
$next_stop' < at \Rightarrow direction' = down \wedge passes_by' = at - 1$

Afin de prendre en compte les appels pendant le mouvement, il faut de nouveau calculer le prochain arrêt. Il est aussi nécessaire de savoir où se trouve l'ascenseur. Cette information, modélisée dans l'état du système par l'attribut $passes_by$, doit être mise à jour. L'opération $GoOnMoving$ effectue l'ensemble de ces mises à jour.

$GoOnMoving$

$\Delta(passes_by, next_stop)$

$mvt = moving$
$direction = up \Rightarrow passes_by < next_stop$
$direction = down \Rightarrow passes_by > next_stop$
$next_stop' = compute_next_stop(passes_by, direction, mvt, call)$
$direction = up \Rightarrow passes_by' = passes_by + 1$
$direction = down \Rightarrow passes_by' = passes_by - 1$

$Move \mathrel{\widehat{=}} StartMoving \vee GoOnMoving$

Besoins 7 et 8. Si l'ascenseur est arrêté à un étage avec sa porte ouverte, une demande de cet étage n'est pas prise en compte. Si l'ascenseur est arrêté à un étage avec sa porte fermée et qu'il reçoit une demande de cet l'étage, il ouvre sa porte.

Ces besoins précisent l'opération $Press$. Il est nécessaire d'effectuer une analyse par cas. Si l'ascenseur se trouve à l'étage d'appel, l'événement $press$ ne modifie pas l'ensemble des appels. Il faut donc rebaptiser l'opération $Press$ en $PressNotAt$ et lui ajouter une nouvelle précondition, $floor(b?) \neq at$. Un nouvel invariant de $Lift$ doit être introduit, $at \notin call$.

Si l'ascenseur se trouve à l'étage d'appel, il faut distinguer si la porte est ouverte ou non en introduisant les opérations $PressAtOpen$ et $PressAtClosed$. La postcondition de $PressAtClosed$ limite l'ensemble des opérations applicables à l'opération $Open$.

$PressAtOpen$

$\Xi Lift$
$b? : BUTTON$

$floor(b?) = at$
$mvt = halted$
$door = open$

$PressAtClosed$

$\Delta(timer)$
$b? : BUTTON$

$floor(b?) = at$
$mvt = halted$
$door = closed$
$timer' = opening_time$

$$Press \mathrel{\widehat{=}} PressNotAt \lor PressAtOpen \lor PressAtClosed$$

Besoin 9. La porte est fermée pendant le mouvement de l'ascenseur.

Ce besoin introduit un nouvel invariant sur la porte, $mvt \neq halted \Rightarrow door = door_closed$.

Il est aisé de vérifier que le fait 1 est satisfait par l'ensemble des opérations introduites dans la spécification. Les autres conditions de validation doivent être vérifiées par inspection de la spécification.

3.3. *Preuve de la correction de la spécification relativement aux besoins*

Effectuer la preuve de la correction de la spécification obtenue par rapport aux besoins formalisés est une étape fastidieuse, permettant de déceler des erreurs voire des oublis dans l'expression des besoins. L'approche suivie permet de définir une notion formelle de la correction d'une spécification par rapport à l'ensemble des besoins, des faits et des suppositions formellement exprimés. Les contraintes utilisées pour exprimer les besoins font référence à l'ensemble Tr des traces admissibles du système.

Pour prouver que la spécification répond correctement aux besoins, il faut définir cet ensemble Tr. Pour cela, il faut définir l'ensemble des états, des événements et du temps, conformément à la définition des traces donnée en annexe A. Cette définition dépend du langage de spécification utilisé.

Construisons l'ensemble Tr pour des spécifications Z. Le type $STATE$ est défini par le schéma d'état, le schéma $Lift$ dans le cas de l'ascenseur. Le type $EVENT$ est défini par l'ensemble des opérations, soit dans le cas de l'ascenseur l'ensemble $\{Stop, Open, Close, ChangeTimer, Move, Press(b?)\}$. Pour le type $TIME$, nous prendrons $\mathbb{N}$, ensemble des entiers naturels.

Pour Z, l'ensemble Tr des traces possibles de la spécification est défini par l'ensemble des traces dans lequel chaque opération ne sera exécutée que si sa précondition est vraie.

La définition de l'ensemble Tr est inductive : la suite vide appartient à Tr. Une suite $\langle ti \rangle$ de longueur 1 appartient à Tr si $ti.s$ satisfait le prédicat défini par le schéma d'état initial et si $ti.s$ et les données $i?$ satisfont la précondition de $ti.e(i?)$, où e est une opération et $i?$ ses données.

Soit $tr \in Tr$ une trace valide avec $lti = last\ tr$ son dernier élément et $nti \in TraceItem$ un nouvel élément. On peut ajouter ce nouvel élément à la trace tr,
$tr \frown \langle nti \rangle \in Tr$ si

— $lti.s$ et $nti.s$ appartiennent à la relation définie par l'operation $lti.e(i?)$

— $nti.s$ et les données $i?$ satisfont la précondition de $nti.e(i?)$

— $nti.t \geq lti.t$

Une fois l'ensemble des traces Tr défini, il faut montrer que cet ensemble satisfait les contraintes exprimant les besoins et les faits sous l'hypothèse des suppositions.

Etude de cas. Une preuve complète a été réalisée pour l'exemple de l'ascenseur simplifié. A titre d'exemple, regardons le besoin 1. Il faut montrer qu'il n'y a pas d'appel à servir sans appuyer sur un bouton. Dans l'état initial du système, $call = \emptyset$. Seule l'opération $PressNotAt$ ajoute un élément à cet ensemble. Concernant le besoin 2, Il faut montrer que l'opération $Press$ est toujours possible, ce qui suffit pour conclure que chaque trace valide peut être augmentée par l'événement $press$. L'opération $Press$ est possible si sa précondition est satisfaite.

La précondition d'une opération définie par une disjonction est la disjonction des préconditions. Donc :

$$\text{pre } Press \mathrel{\widehat{=}} \text{pre } PressNotAt \vee \text{pre } PressAtOpen \vee \text{pre } PressAtClosed$$

Si ces schémas sont vus comme des prédicats, nous obtenons :

$$\text{pre } Press \Leftrightarrow \text{pre } PressNotAt \vee \text{pre } PressAtOpen \vee \text{pre } PressAtClosed$$

$$\text{pre } PressNotAt \Leftrightarrow Lift \wedge floor(b?) \neq at$$

$$\text{pre } PressAtOpen \Leftrightarrow Lift \wedge floor(b?) = at \wedge mvt = halted \wedge door = open$$

$$\Leftrightarrow Lift \wedge floor(b?) = at \wedge door = open$$

$$\text{pre } PressAtClosed \Leftrightarrow Lift \wedge floor(b?) = at \wedge mvt = halted \wedge door = closed$$

$$\Leftrightarrow Lift \wedge floor(b?) = at \wedge door = closed$$

La condition $mvt = halted$ dans les préconditions des opérations $PressAtOpen$ et $PressAtClosed$ est redondante avec $floor(b?) = at$. Elle découle des invariants (a) et (b) du schéma $Lift$. Nous avons montré que la disjonction des trois préconditions est $Lift$. Nous pouvons en conclure que $Press$ est toujours applicable.

4. Etat de l'art

Une distinction entre connaissance du domaine et besoins est souvent effectuée dans la littérature, avec vérification de la consistance entre ces deux vues. Par exemple, quand Parnas [PAR 95] décrit le contenu mathématique d'un cahier des charges pour un système de contrôle nucléaire, il introduit différentes relations mathématiques, l'une décrivant l'environnement du système automatique, l'autre décrivant ses besoins. Les deux sortes de relations doivent remplir des conditions de faisabilité. Pour Parnas, une étape critique dans la documentation des besoins concerne l'identification des informations de l'environnement devant être mesurées ou contrôlées, ainsi que la représentation de ces informations par des variables mathématiques. Il propose de caractériser les informations de l'environnement comme soit des informations contrôlées, soit des informations surveillées. Cette vision correspond à la classification des événements dans notre approche ainsi que dans celle de Jackson et Zave [JAC 95]. Lire une information surveillée correspond à la notion d'événements observables par le logiciel et contrôler une information correspond à celle d'événements contrôlés par le logiciel.

Alors que les suppositions ne sont pas utilisées par Parnas, ni par Jackson et Zave, elles sont utilisées dans KAOS [DAR 93, DAR 96]. KAOS supporte la construction de systèmes composites (voir aussi [FEA 91]). Un tel système est constitué d'un ensemble d'agents humains, logiciels et matériels, chacun d'eux ayant la responsabilité d'un ou plusieurs buts. L'approche proposée est orientée par les buts. Des buts généraux sont établis puis raffinés en une spécification KAOS. Le processus passe par l'expression des buts sous la forme d'une structure et/ou et se termine par l'affectation de responsabilités aux agents. Tout comme notre approche, KAOS propose des heuristiques pour l'élicitation des besoins et le développement de spécifications. KAOS se distingue de notre approche par la proposition de son propre langage et par des perspectives plus larges : le système et son environnement sont tous les deux modélisés en détail. Il en résulte une très riche terminologie. Par contraste, notre approche se focalise sur l'obtention de la spécification formelle des composants logiciels, avec modélisation des seuls aspects de l'environnement nécessaires à la spécification.

Easterbrook and Nuseibeh [EAS 96] ne font pas de distinction entre l'élicitation des besoins et le développement de la spécification. Ils élicitent les besoins lors de la détection d'inconsistances dans leurs spécifications. D'une part, leur approche permet de différer la résolution de conflits. D'autre part, cette sorte d'élicitation «paresseuse» des besoins diffère le moment ou un contrat entre clients et fournisseurs peut être établi. De plus, il est difficile de valider une spécification par rapport aux besoins quand ceux-ci ne sont pas rigoureusement décrits et validés relativement aux propriétés de complétude et de consistance.

Comparons notre approche avec celle de Jackson et Zave [JAC 95, ZAV 97]. Premièrement, nous établissons une différence entre besoins et spécifications. Notre méthode conduit à une spécification formelle exprimée dans un langage de spécification conventionnel, alors que leur approche s'arrête lorsque les besoins ont été transformés de telle manière qu'ils puissent être vus comme une spécification. Le langage d'ex-

pression des besoins (les formules du calcul du premier ordre) est le même durant tout le processus. Deuxièmement, l'apport le plus important de notre approche est le guide méthodologique proposé aux analystes et aux spécifieurs ainsi que la validation de produits développés. Ces points ne sont pas explicitement abordés par Jackson et Zave.

Par rapport aux approches et aux notations orientées objets mentionnées dans le paragraphe 2 [COL 94, RUM 91, FOW 97], nous avons seulement emprunté les premières étapes, où le vocabulaire pertinent est introduit de manière systématique. Ces méthodes n'ont pas pour objectif l'obtention d'une spécification formelle. Notre approche ne plaide pas spécialement en faveur d'un développement orienté objet.

5. Conclusion

Les besoins définissent un ensemble de conditions attendues d'un système ou d'un composant pour satisfaire un contrat, un standard ou un document imposé. Par exemple, le standard IEEE 1498 [IEE 94] définit un besoin comme une caractéristique possédée par un logiciel ou un composant logiciel pour être acceptable par le client. Les besoins doivent être complets et non ambigus.

Dans notre approche, la complétude est assurée par un processus de réflexion dès les premières phases de l'élicitation des besoins et par retour aux besoins si nécessaire après la phase de spécification. Dans le développement d'exemples, nous avons découvert des omissions dans les besoins lors du passage à la spécification formelle. Les besoins ont été clarifiés par la formalisation.

L'intégration de méthodes formelles dans le processus de développement de logiciels est un défi important du génie logiciel. Les stratégies d'intégration ont été classées par Fraser et al. [FRA 94] relativement à trois facteurs :

1. Est-ce que la stratégie permet de passer directement des besoins informels à la spécification formelle ou introduit-elle des modèles formels intermédiaires correspondant à un enrichissement des besoins ?

2. Si la stratégie introduit un modèle intermédiaire, est-ce que le processus suivi pour son élaboration est un processus qui raffine en parallèle les besoins et la spécification formelle, ou en séquence, i.e. la spécification formelle est dérivée des besoins, une fois ceux-ci complètement explicités ?

3. Dans quelle mesure la stratégie offre-t-elle des outils pour l'élicitation des besoins et la formalisation ?

Relativement à ces critères de classification, notre approche est une approche *transformationnelle* avec l'introduction de prédicats sur les traces d'événements du système comme représentation intermédiaire des besoins. Notre approche est en partie *séquentielle*, la phase de spécification ne commençant que lorsque les besoins sont estimés complets et corrects. Néanmoins, des allers et retours entre ces deux grandes phases sont possibles. Des outils d'aide à la validation de notre approche ne sont pas

encore disponibles mais sont aisément concevables : outils de vérification des conditions de validation, d'analyse des interactions et de développement des spécifications. Un prototype pour l'implantation de l'algorithme de recherche des interactions a été développé et la maquette Proplane [LÉV 97] est une base intéressante pour l'implantation de l'approche proposée.

Différentes études de cas ont été réalisées en suivant cette approche. Comme l'ascenseur, le four à micro-ondes est un système réactif. Avec un automate bancaire [HEI 98b], les aspects transformationnels ont aussi été abordés. L'étude des interactions de services dans les systèmes de téléphonie a été menée.

Les principales caractéristiques de notre approche peuvent être résumées comme suit. Besoins/élicitation des besoins d'une part, spécification/élaboration de spécifications d'autre part sont clairement identifiés. Un guide méthodologique détaillé pour ces deux activités est proposé, celles-ci se complétant et s'intégrant en douceur. Notre approche n'introduit pas de nouveaux langages ou formalismes. Le processus d'élicitation des besoins est indépendant du langage de spécification utilisé. Le processus de développement de spécifications peut être adapté pour utiliser d'autres langages que Z. Les systèmes transformationnels et réactifs peuvent être décrits ainsi que les besoins où le temps réel est nécessaire. Un lien de traçabilité entre le cahier des charges, les besoins et la spécification est maintenu via les agendas.

Nous proposons une manière standard pour exprimer les faits, suppositions et besoins. Les contraintes sur les traces possibles sont une manière flexible et puissante de décrire un système et ses interactions avec l'environnement. L'expression des besoins à l'aide de contraintes sur les traces permet de détecter systématiquement les conflits entre besoins et de définir une notion formelle de correction de la spécification relativement aux besoins.

6. Bibliographie

[ANT 98] ANTÓN A. I. et POTTS C., « The use of Goals to Surface Requirements for Evolving Systems ». In *Proc. of the 20 th International Conference on Software Engineering, ICSE'98*, Kyoto, Japan, 1998. IEEE Computer Society Press.

[BOE 88] BOEHM B. W., « A spiral model of software development and enhancement ». *IEEE Computer*, vol. 21, n° 5, p. 61–72, 1988.

[COL 94] COLEMAN D., ARNOLD P., BODOFF S., DOLLIN C., GILCHRIST H., HAYES F. et JEREMAES P., *Object-Oriented Development : The Fusion Method*. Prentice Hall, 1994.

[DAR 93] DARDENNE A., LAMSWEERDE A. et FICKAS S., « Goal-directed requirements acquisition ». *Science of Computer Programming*, vol. 20, p. 3–50, 1993.

[DAR 96] DARIMONT R. et LAMSWEERDE A., « Formal Refinement for Patterns for Goal-Driven Requirements Elaboration ». In *Proc FSE-4, ACM Symposium on the Foundation of Software Engineering*, p. 179–190. ACM Press, 1996.

[EAS 96] EASTERBROOK S. et NUSEIBEH B., « Using ViewPoints for inconsistency management ». *Software Engineering Journal*, p. 31–43, January 1996.

[FEA 91] FEATHER M., FICKAS S. et HELM R., « Composite System Design : the Good News and the Bad News ». In *Proc. 6th Knowledge-Based Software Engineering Conference*, p. 16–25. IEEE Computer Society Press, 1991.

[FOW 97] FOWLER M. et SCOTT K., *UML distilled. Applying the standard Object Modelling Language*. Addison-Wesley, 1997.

[FRA 94] FRASER M., KUMAR K. et VAISNAVI V., « Strategies for Incorporating Formal Specifications in Software Development ». *Communications of the ACM*, vol. 37, n° 10, p. 74–86, October 1994.

[HEI 98a] HEISEL M. et SOUQUIÈRES J., « Detecting Feature Interactions – A Heuristic Approach ». In SAAKE G. et TÜRKER C., Eds., *Proc. of the first FIREworks Workshop*, Preprint 10/98, p. 30–48, Fakultät für Informatik, 1998. Univ. Magdeburg.

[HEI 98b] HEISEL M. et SOUQUIÈRES J., « A Method to express requirements and transform them into a formal specification ». Rapport technique 98–R–011, LORIA Nancy, 1998.

[HEI 98c] HEISEL M., « Agendas – A Concept to Guide Software Development Activites ». In HORSPOOL R. N., Ed., *Proc. Systems Implementation 2000*, p. 19–32. Chapman & Hall London, 1998.

[HEI 98d] HEISEL M. et SOUQUIÈRES J., « A Heuristic Approach to Detect Feature Interactions in Requirements ». In KIMBLER K. et BOUMA W., Eds., *Proc. 5th Feature Interaction Workshop*, p. 165–171. IOS Press Amsterdam, 1998.

[IEE 94] IEEE, « Software development ». *IEEE publications office*, IEEE Standard 1498, Los Alamitos, CA, March 1994.

[JAC 95] JACKSON M. et ZAVE P., « Deriving Specifications from Requirements: an Example ». In *Proceedings 17th Int. Conf. on Software Engineering, Seattle, USA*, p. 15–24. ACM Press, 1995.

[LÉV 96] LÉVY N. et SOUQUIÈRES J., « A «Coming and Going» Approach to Specification Construction: a Scenario ». In SCHÄFER W., KRAMER J. et WOLF A., Eds., *Proc. 8th Int. Workshop on Software Specification and Design*, p. 115–118. IEEE Computer Society Press, 1996.

[LÉV 97] LÉVY N. et SOUQUIÈRES J., « Modelling Specification Construction by Successive Approximations ». In JOHNSON M., Ed., *6th International AMAST conference*, p. 351–364. Springer Verlag LNCS 1349, 1997.

[PAR 95] PARNAS D., Using Mathematical Models in the Inspection of Critical Systems. In HINCHEY M. et BOWEN J., Eds., *Applications of Formal Methods*, p. 17–31. Prentice Hall, 1995.

[RUM 91] RUMBAUGH J., BLAHA M., PREMERLANI W., EDDY F. et LORENSEN W., *Object-Oriented Modeling and Design*. Prentice-Hall International. Englewood Cliffs, New Jersey, 1991.

[SAD 97] SADAOUI S. et SOUQUIÈRES J., « Quelques approches de la réutilisation dans le modèle Proplane ». In *Conférence AFADL, Approches formelles dans l'assistance au développement de logiciels*, p. 85–96, Toulouse, 1997. Onera–Cert.

[SOU 96] SOUQUIÈRES J. et HEISEL M., « Expression of styles in Formal Specifications ». In *5th Software Quality Conference*, Dundee, Scotland, UK, July 1996. CRIN 96-R-093.

[SPI 92] SPIVEY J. M., *The Z Notation – A Reference Manual*. Prentice Hall, 2nd édition, 1992.

[ZAV 97] ZAVE P. et JACKSON M., « Four dark corners for requirements engineering ». *ACM Transactions on Software Engineering and Methodology*, vol. 6, n° 1, p. 1–30, January 1997.

A. Expression formelle des contraintes sur les traces

Pour décrire la manipulation des traces, nous utilisons la notation Z [SPI 92] comme métalangage. Dans le paragraphe 3, nous avons utilisé Z comme langage de spécification de l'application.

$$[STATE, EVENT, TIME]$$

$$\begin{array}{|l} \hline TraceItem \\ s : STATE \\ e : EVENT \\ t : TIME \\ \hline \end{array}$$

Chaque trace du système est une suite d'items, où un événement dans cette suite ne peut se produire à un temps t inférieur à celui d'un événement le précédant. Le symbole $\leq_t$ désigne une relation d'ordre partiel sur le temps :

$$\begin{array}{|l} TRACE : \mathbb{P}(\mathrm{seq}\ TraceItem) \\ \hline \forall\, tr : TRACE \bullet \forall\, i : \mathrm{dom}\, tr \bullet i = \#tr \vee (tr\ i).t \leq_t (tr(i+1)).t \end{array}$$

Soit Tr l'ensemble des traces possibles d'un système. Les contraintes s'expriment à l'aide de formules restreignant l'ensemble Tr. Le préfixe d'une trace est aussi une trace :

$$\begin{array}{|l} Tr : \mathbb{P}\ TRACE \\ \hline \forall\, tr : Tr \bullet (\forall\, tr' : TRACE \mid tr' \ \mathsf{prefix}\ tr \bullet tr' \in Tr) \end{array}$$

La fonction suivante supprime un ensemble d'événements donnés d'une trace donnée.

$$\begin{array}{|l} remove : TRACE \times \mathbb{P}\ EVENT \longrightarrow TRACE \\ \hline \forall\, tr : TRACE; \ evs : \mathbb{P}\ EVENT \bullet \\ \qquad remove(tr, evs) = tr \upharpoonright \{ti : TraceItem \mid ti.e \notin evs\} \end{array}$$

Maritta Heisel *a obtenu un doctorat d'informatique de l'Université de Karlsruhe et soutenu une habilitation à diriger des recherches à l'Université Technique de Berlin en Allemagne. Elle enseigne actuellement à l'Université de Magdeburg. Ses principaux thèmes de recherche portent sur la méthodologie pour l'application de techniques formelles en génie logiciel et le développement de systèmes critiques.*

Jeanine Souquières *est professeur à l'Université Nancy 2 depuis 1993 où elle assure les fonctions de directrice de l'UFR Mathématiques et Informatique. Elle dirige, au sein du LORIA à Nancy, l'équipe DEDALE. Ses principaux thèmes de recherche portent sur l'aide au développement de spécifications formelles avec le passage du semi-formel au formel, la prise en compte de la réutilisation et de la validation/vérification dans les différentes étapes du développement.*