ÉVALUATION DE MÉTHODES FORMELLES DE SPÉCIFICATION

par

Hassan Diab

mémoire présenté au Département de mathématiques
et d'informatique en vue de l'obtention du grade de maître ès sciences (M.Sc.)

FACULTÉ DES SCIENCES
UNIVERSITÉ DE SHERBROOKE

Sherbrooke, Québec, Canada, mai 1999

Sommaire

Avec l'utilisation toujours plus massive de l'informatique, dans tous les domaines de la vie quotidienne, les exigences accrues en qualité des applications rendent incontournable l'amélioration du processus de développement du logiciel. Les travaux s'orientent, de plus en plus, vers l'utilisation des méthodes formelles de spécification qui permettent de répondre à ces exigences. Cependant, les applications pour lesquelles les méthodes formelles ont été utilisées dans l'industrie sont encore limitées à des domaines bien particuliers.

Un des écueils pour l'acceptation des méthodes formelles dans l'industrie est l'absence d'un langage formel de spécification permettant d'exprimer et de vérifier de manière optimale tous les aspects d'un système informatique. En effet, différents langages sont proposés pour exprimer les spécifications relatives aux structures de données manipulées, au contrôle des différentes tâches à effectuer, aux contraintes temps réels, etc. Pour cela, l'emploi d'une méthode formelle inappropriée au domaine de l'application peut ajouter des difficultés notationnelles qui ne font souvent qu'obscurcir le problème. Il s'agit donc de choisir la méthode appropriée au domaine de l'application à développer, et aussi de choisir la manière de l'appliquer dans un cadre et une méthode de travail donnés.

Une approche pour l'évaluation des méthodes formelles est proposée dans ce mémoire afin d'aider le développeur à choisir la méthode appropriée au domaine de l'application. Cette approche n'examine pas la qualité théorique de méthodes, mais plutôt leurs qualités pratiques concernant les notations et les outils, en les appliquant à un système de la

facturation de commandes. Les deux étapes principales de cette approche sont :

1. la définition d'une liste de critères pour l'évaluation de méthodes formelles ;
2. l'évaluation de l'applicabilité des méthodes formelles au problème choisi, en se basant sur ces critères.

Les critères ont été systématiquement dérivés de nos modélisations du système de la facturation de commande et du contexte d'utilisation du document de spécification dans les phases du cycle de développement. En se basant sur cette liste de critères, une évaluation de quatre méthodes formelles, soit la méthode relationnelle inductive, la méthode boîte noire par entités, la méthode B et la méthode assertions de traces, a été faite.

Remerciements

Qu'il me soit permis de remercier tous ceux qui ont apporté une contribution à la réalisation de ce mémoire.

Je voudrais remercier mon directeur de recherche, le professeur Marc Frappier, pour m'avoir aidé à corriger et à parfaire le texte et le support multiple qu'il m'a apporté tout au long de cette recherche. Je lui suis également reconnaissant pour m'avoir encouragé dans mon travail et pour la patience qu'il a portée à mon égard et pour ses conseils judicieux tout au long de mes études de deuxième cycle à l'Université de Sherbrooke. Il a ma profonde reconnaissance.

Merci au professeur Richard St-Denis et au professeur Djemel Ziou de l'Université de Sherbrooke pour avoir accepté d'être membres du jury.

Enfin, j'adresse tous mes remerciements à tout le personnel du Département de mathématiques et d'informatique de l'Université de Sherbrooke.

Table des matières

Liste des tableaux

Liste des figures

Introduction

L'industrie développe maintenant des systèmes larges et complexes. Malgré le fait qu'un grand nombre de logiciels aient été développés, les processus utilisés et la qualité des résultats obtenus sont encore pauvres. Le coût et le temps nécessaires pour développer un logiciel sont imprévisibles et souvent très élevés. Le développement de systèmes larges et complexes est souvent achevé en retard par rapport au plan initial. Pour résoudre ces problèmes, il est indispensable d'apporter des améliorations au processus de développement. Plus particulièrement, ce mémoire s'intéresse à la phase de spécification.

Dans la suite, nous présentons brièvement quelques modèles de développement du logiciel. Ensuite, nous abordons la phase de spécification. Enfin, nous définissons le problème abordé dans ce mémoire et l'approche proposée pour son étude.

Modèles de développement du logiciel

Le modèle du cycle de développement est un des concepts les plus importants dans l'ingénierie moderne du logiciel. Sa prémisse est que le développement et l'implantation d'un système sont réalisés en plusieurs phases distinctes, chacune ayant une tâche bien définie.

Plusieurs modèles ont été proposés : le modèle en cascade (*Waterfall*) de Royce [41] reconnaît l'importance de l'interaction entre les phases et il fait en sorte que cette interaction soit limitée aux phases adjacentes (voir figure 1). Dans le modèle en spirale de

Boehm [9], on préconise une approche plus itérative comprenant du prototypage (voir figure 2).

Dans tous les cas, le produit final de chaque phase est un document qui sert à évaluer le résultat de cette même phase et qui forme la spécification utilisée par les phases subséquentes. Par exemple, toutes les exigences décrites dans la spécification devraient être satisfaites par l'implantation, et l'implantation ne devrait fournir que les fonctions décrites dans la spécification.

Une des motivations de ces modèles est de minimiser le nombre des fautes résiduelles dans les premières phases, car il est moins coûteux de corriger une faute le plus tôt possible. Par exemple, il est souvent facile de corriger une faute de programmation durant les tests unitaires et il est également facile d'insérer une exigence durant la revue de la spécification. Par contre, il est très coûteux d'insérer ou de corriger une telle exigence si elle est détectée après la phase de codage ou durant la phase de test d'intégration. Les données présentées par Fairly [22] montrent qu'il est cinq fois plus coûteux de corriger une faute de spécification durant la phase de conception que durant la phase initiale, dix fois plus coûteux durant la phase de codage, vingt à cinquante fois plus coûteux dans la phase de test et cent fois plus coûteux si le système est en opération. Ces données sont corroborées par plusieurs autres études [8, 12] .

La fiabilité et le moindre coût sont les motivations principales d'un processus de développement bien structuré. L'utilisation de méthodes systématiques, pour éviter les oublis et détecter et corriger des fautes, est la meilleure façon d'augmenter la probabilité que le système désiré n'ait aucune faute critique.

La spécification du logiciel

La spécification est un élément critique dans le processus de développement du logiciel. Selon Clark et Wing [14], l'écriture de la spécification permet une compréhension

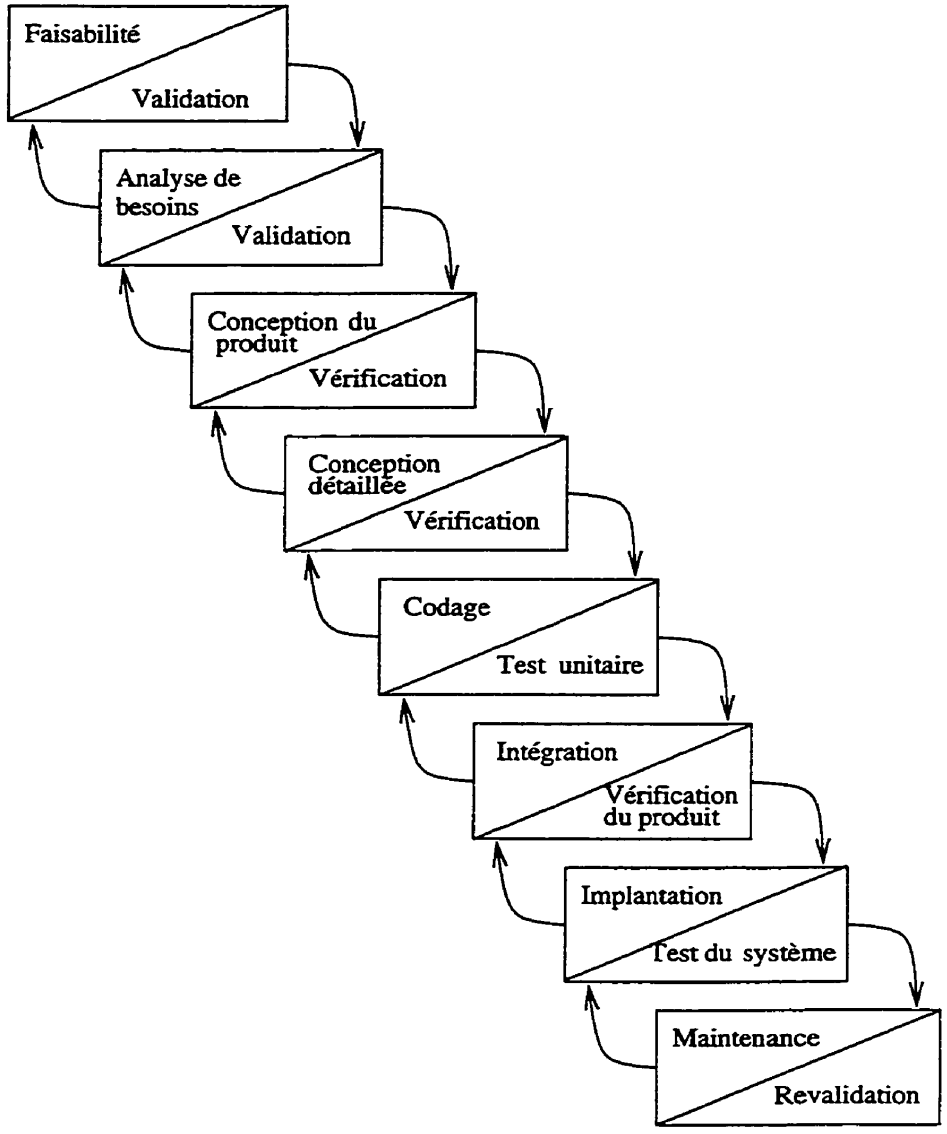


FIG. 1 – *Modèle en cascade*

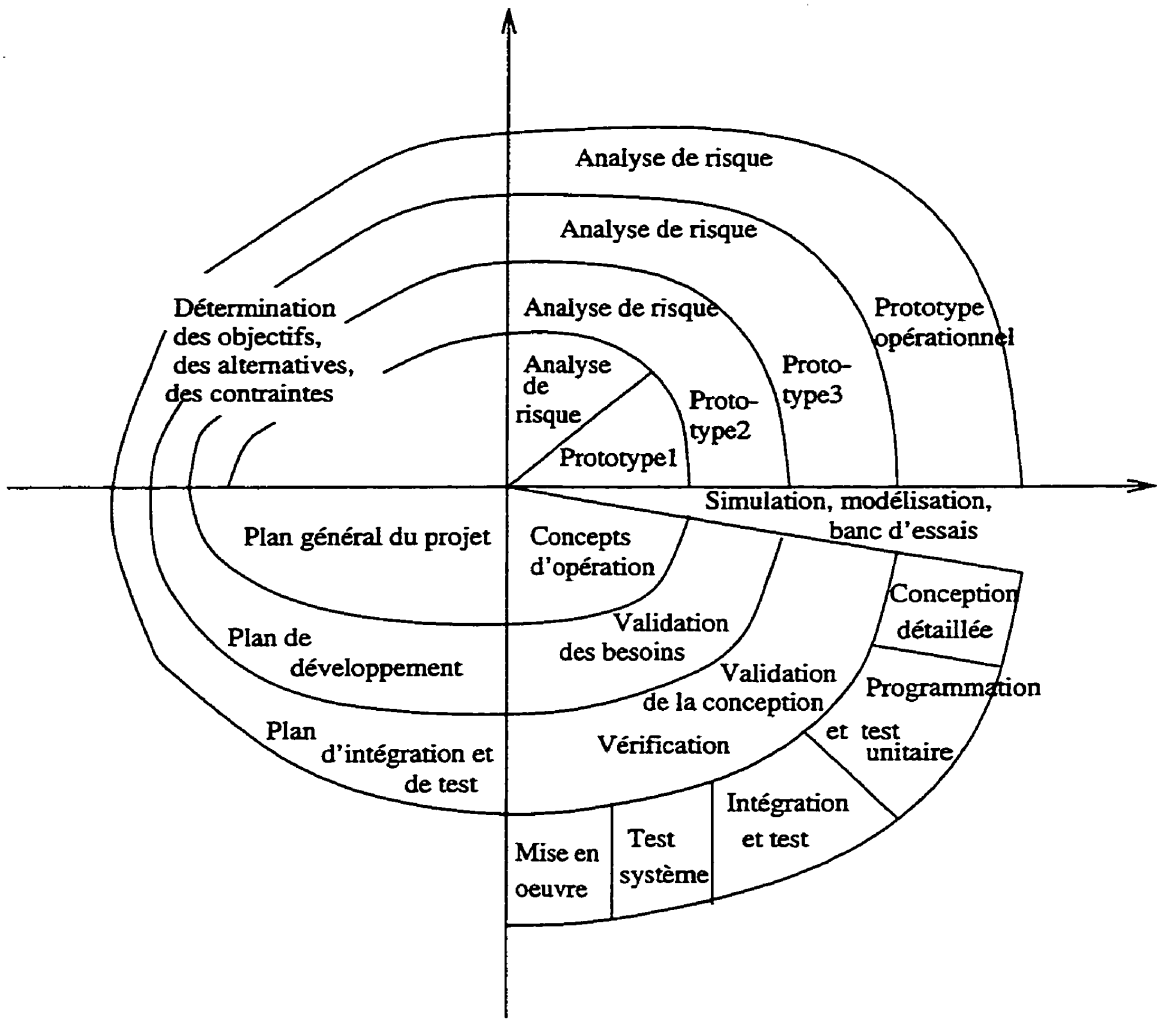


FIG. 2 – *Modèle en spirale*

approfondie du logiciel à développer. C'est en fait un moyen pour assurer une meilleure communication entre toutes les personnes concernées par le développement.

La spécification est la documentation des exigences du système. Cette vision du système se situe à un niveau très abstrait. Cependant, elle doit être complète et précise, de sorte que tout système qui satisfait ces exigences documentées comble correctement les besoins des utilisateurs. La spécification est comme un véhicule qui transmet les besoins des utilisateurs aux développeurs du système. Pour cela, la spécification doit être utilisée et comprise par toutes les personnes concernées par le développement du système.

Des études ont démontré que les fautes de spécification sont les plus coûteuses à corriger [8, 22]. Ces fautes, lorsqu'elles sont détectées durant le développment du logiciel, occasionnent du travail supplémentaire, ce qui augmente la durée du développement et la rend imprévisible. Si ces fautes n'ont pas été détectées, alors le système tombe en panne durant l'exécution. De ces constatations, on peut déduire que moins il y aura de fautes résiduelles dans la spécification, plus le processus de développement et la qualité du logiciel seront améliorés.

La spécification informelle

Actuellement, la plupart des concepteurs écrivent les spécifications en langage naturel. Le langage naturel est compréhensible par les utilisateurs, les analystes et les développeurs. Chacun d'eux est habitué à lire et à écrire des documents en langage naturel, puisqu'il n'exige pas une spécialisation particulière. L'organisation de ces documents est bien familière : table des matières, chapitres, sections, etc. La spécification informelle s'accorde bien avec les méthodes actuelles utilisées pour développer un logiciel. Cependant, il est possible qu'une spécification en langage naturel ait plusieurs interprétations. Cela signifie que les spécifications informelles sont prédisposées à être incomplètes et incohérentes, à cause de leur ambiguïté et de l'incapacité d'en faire une vérification méthodique et rigoureuse.

La spécification formelle

Les méthodes de spécification formelles utilisent des techniques mathématiques pour décrire un problème. L'utilisation de notations formelles résoud le problème de la variété d'interprétations par la rigueur du formalisme, l'abstraction, la syntaxe et la sémantique mathématiques bien définies. Elles engendrent des spécifications précises que l'on peut vérifier de manière systématique à l'aide d'outils. En identifiant tôt les problèmes du système, il est encore possible de les corriger avec un coût minimal. Ceci peut réduire le coût et la durée du développement. Ces améliorations pourraient aboutir à un processus prévisible qui produit un logiciel ayant un nombre réduit de fautes. Des études, [14, 15, 37], ont montré que l'addition de méthodes formelles au cycle de développement améliore le processus de développement et la qualité du logiciel, en imposant un contrôle dès les premières phases du développement, notamment la spécification.

Les méthodes formelles ne sont pas utiles si elles ne sont pas compréhensibles et faciles à utiliser. Différentes notations formelles ont différents niveaux de compréhension, mais elles sont relativement faciles à apprendre [20]. Depuis qu'on a reconnu le potentiel des méthodes formelles pour produire de meilleures spécifications, plusieurs notations appuyées par des outils ont été introduites. Les notations prennent différentes formes, comme tabulaire, graphique, mathématique, etc. Différents outils ont été développés, comme des éditeurs, des animateurs et des vérificateurs, pour manipuler ces notations.

La problématique

Actuellement, les méthodes formelles ne sont pas encore largement utilisées dans l'industrie. Une raison est la difficulté d'incorporer la technologie formelle dans l'industrie. Par exemple, l'utilisation d'une méthode formelle allonge typiquement la phase de spécification et réduit la phase de test. En conséquence, le code est produit un peu plus tard dans le cycle de développement, ce qui donne l'impression que le projet n'avance

pas aussi vite qu'avec un processus traditionnel. Aussi, elles exigent une formation particulière du personnel ou bien elles sont incompatibles avec les méthodes existantes. Par ailleurs, la notation formelle n'empêche pas la production d'une spécification de mauvaise qualité. Une cause potentielle de ce problème est que les notations choisies ne sont pas adaptées aux propriétés que l'on veut spécifier et vérifier. De plus, il n'existe pas de méthode de spécification formelle universelle. Dans certains cas, il peut être utile d'avoir plusieurs spécifications utilisant des notations distinctes, pour spécifier différents aspects d'un même système. Pour cela, il est indispensable de choisir une ou plusieurs méthodes formelles appropriées et compatibles l'une par rapport à l'autre pour spécifier un système.

Une étude comparative des méthodes formelles peut résoudre ce type de problème. Les études comparatives sont importantes, d'une part, parce qu'elles aident les concepteurs à choisir la méthode appropriée pour décrire leurs projets. D'autre part, elles aident les chercheurs à proposer des nouvelles méthodes, en montrant les limitations et les avantages des méthodes existantes.

L'approche

L'objectif de ce travail est de comparer et d'évaluer les méthodes formelles, en se basant sur des expériences. Les expériences consistent à appliquer plusieurs méthodes formelles au même problème, et à observer les avantages et les limitations de chaque méthode. Pour s'assurer que les observations soient pertinentes, six étapes ont été définies et appliquées :

1. sélection d'un cas d'étude ;
2. analyse du cas d'étude et génération d'hypothèses ;
3. rédaction des spécifications formelles avec quatre notations différentes ;
4. définition de critères d'évaluation ;
5. évaluation des spécifications en fonction des critères définis dans l'étape précédente ;

6. déduction des conclusions.

Le cas d'étude concerne la gestion de commandes. L'énoncé, posé par Henri Habrias, consiste à facturer des commandes, autrement dit à faire passer des commandes de l'état "en_ attente" à "facturée". Notons que le problème de la facturation a déjà été modélisé par plusieurs méthodes formelles et semi-formelles dans le cadre de *"International Workshop on Comparing System Specification Techniques"* à l'Institut de Recherche en Informatique de Nantes (IRIN) [26].

Notons que notre travail est limité à un cadre d'étude. La démarche est donc pédagogique et non celle d'un document descriptif du système.

Structure du mémoire

Dans le chapitre 1, nous discutons les rôles des méthodes formelles dans le cycle de développement. Une présentation de chaque méthode évaluée, suivie par un exemple simple pour faciliter sa compréhension et son application, est donnée dans le chapitre 2. Dans le chapitre 3, nous présentons nos spécifications du problème de la facturation sous quatre formalismes, en expliquant la technique de spécification selon chacun des formalismes utilisé. La dérivation de critères est présentée dans le chapitre 4. Les résultats et les questions soulevées durant notre étude sont décrits au chapitre 5. Enfin, nous terminons par une conclusion où l'on fait une synthèse de la contribution de chaque méthode à la résolution du cas d'étude.

Chapitre 1

Les méthodes formelles

Les méthodes formelles ont considérablement évolué depuis les études de McCarthy, Floyd, Hoare et plusieurs autres pionniers [33]. Une tendance de cette évolution fut l'étude de méthodes pour des systèmes de plus en plus complexes, par exemple les systèmes distribués. Une autre tendance, peut-être la plus déterminante de l'utilisation des méthodes formelles, a été le deplacement de l'intérêt des chercheurs des dernières phases aux premières phases du cycle de développement, soit de la vérification de programme vers la spécification de système.

Le contenu de ce chapitre décrit succinctement le rôle de la spécification formelle dans le cycle de développement d'un système informatique. Nous exposons quelques définitions, ses avantages et son utilisation dans l'industrie.

1.1 Définitions

Dans le recueil de normes français en génie logiciel ISO/AFNOR 1997 [31], la spécification formelle est définie comme suit :

1. ≪ *Spécification pouvant être utilisée afin de démontrer mathématiquement la validité de la mise en oeuvre d'un système ou encore de dériver mathématiquement la*

mise en oeuvre du système ≫ ;

2. ≪ *Spécification écrite en notation formelle, souvent utilisée pour une démonstration d'exactitude* ≫.

Selon Gaudel, Marre, Bernot et Schlienger [34], une spécification est dite formelle si :

1. ≪ *elle est écrite en suivant une syntaxe bien définie, comme celle d'un langage de programmation* ≫ ;
2. ≪ *la syntaxe est accompagnée d'une sémantique rigoureuse qui définit des modèles mathématiques représentant les réalisations acceptables de chaque spécification syntaxiquement correcte* ≫.

Selon Choppy [13], la spécification formelle est définie comme suit :
≪ *on parle de spécification informelle lorsque le langage utilisé n'a pas une syntaxe précise, de spécification semi-formelle lorsque la syntaxe du langage utilisé est définie de façon précise, et de spécification formelle lorsque la syntaxe et la sémantique du langage utilisé sont définies* ≫.

1.2 Les avantages

L'avantage principal des méthodes formelles est l'utilisation de concepts de la logique et de la technique mathématique. Ces concepts fournissent des outils effectifs qui organisent les pensées des concepteurs et qui facilitent la communication entre toutes les personnes concernées par le développement. De plus, ils nous permettent de décrire de manière précise, non ambiguë, les demandes énoncées par l'utilisateur du système logiciel à réaliser. Les notions d'ensemble, de relation, de fonction et leurs différentes propriétés et opérations, avec les quantifications universelles et existentielles, nous permettent d'établir une spécification d'une manière simple et claire et de démontrer mathématiquement les propriétés de la spécification.

Les principaux avantages techniques d'une spécification formelle [11, 10, 14, 36, 43],

par rapport à une spécification informelle, sont la précision et la clarté. Des imprécisions et des ambiguïtés peuvent facilement se glisser dans les spécifications informelles. Ceci peut ouvrir la voie à plusieurs interprétations. Par contre, les termes de spécifications formelles n'ont qu'une seule interprétation.

Un autre avantage des spécifications formelles est que les questions sont posées et répondues avec précision et d'une manière scientifique. De plus, les méthodes formelles fournissent des spécifications qui peuvent être rigoureusement vérifiées, analysées et testées dès les premières étapes du cycle de développement, ce qui n'est pas le cas dans les méthodes informelles. Cela signifie qu'il possible de détecter et de corriger des fautes dès les premières étapes, ce qui réduit le coût et la durée du développement et améliore la qualité du logiciel.

Les méthodes formelles nous permettent de spécifier ce qui est nécessaire à un niveau d'abstraction particulier. Certains comportements et propriétés peuvent être volontairement exclus s'il est préférable que leurs élaborations soient remises aux prochaines phases du cycle de développement.

Selon Hoffman et Stooper [29], les spécifications rigoureuses jouent un triple rôle dans le développement du logiciel. D'abord, les spécifications documentent avec précision les décisions de conception, indépendamment de l'implantation, et servent de base pour la revue de cette conception. Durant l'implantation, les mêmes spécifications supportent le développement en parallèle. D'une part, elles renseignent les utilisateurs de ce qu'ils peuvent s'attendre et, d'autre part, elles renseignent les programmeurs de ce qu'ils doivent faire, et servent de base pour la phase de test. Finalement, durant la maintenance, ces mêmes spécifications supportent l'analyse de changements et aident à la formation de nouveaux personnels.

1.3 Usage industriel

Dans cette section nous présentons brièvement quelques utilisations documentées des méthodes formelles dans l'industrie.

En 1993, Craigen, Gerhart et Ralston [15] ont rédigé un rapport sur l'utilisation de méthodes formelles dans l'industrie, notamment dans le domaine de l'énergie nucléaire. Leurs objectifs n'étaient pas la dérivation d'une liste de critères mais plutôt la documentation de l'utilisation actuelle de méthodes formelles dans l'industrie. Ils ont étudié les effets de méthodes formelles sur le cycle de développement d'un projet, en particulier la satisfaction du client, le coût et la qualité du produit. Plusieurs entrevues ont été faites avec des praticiens en utilisant des questionnaires pour examiner l'utilisation des méthodes formelles dans un contexte industriel pour les applications commerciales. Les domaines de ces applications vont de l'énergie nucléaire au transport, en passant par l'aéronautique et l'espace, la physique et le domaine médical.

L'article de Hall [27] constitue une bonne introduction sur l'utilisation des méthodes formelles dans l'industrie. C'est un des meilleurs articles non techniques disponibles sur les méthodes formelles. Hall cite nombre d'applications qui ont été développées avec succès, en utilisant la notation Z. Parmi ces applications, on note la modélisation des oscilloscopes de Tektronix [17].

Une autre application qui a connu un succès important consiste à développer une nouvelle version du moniteur de télétraitement CICS [30, 39] en utilisant la notation Z. Cette nouvelle version contient 500 000 lignes de code non-modifiées et 268 000 lignes de code modifiées dans lesquelles 37 000 lignes furent produites à partir des spécifications formelles écrites en Z et le reste de lignes furent écrites à l'aide de méthodes informelles. Environ 2 000 pages de spécifications ont été produites. La plupart des fautes de la partie spécifiée formellement ont été détectées durant les premières phases du processus de développement, notamment la spécification, ce qui n'était pas le cas dans la partie

spécifiée informellement. Il a été estimé que l'utilisation de la notation Z a réduit le coût de 9 %.

La station nucléaire *Darlington*, gérée par Hydro-Ontario, utilise un système logiciel pour arrêter les réacteurs en cas d'urgence [15]. Avant d'émettre une autorisation pour la mise en opération de la centrale, l'AECB, *Atomic Energy Control Board*, a exigé qu'on lui démontre la fiabilité du logiciel. Avec l'assistance de David Parnas [37], la méthode formelle SCR, *Software Cost Reduction,* dérivée de la méthodologie A7 [21], a été utilisée pour vérifier que l'implantation du logiciel satisfaisait parfaitement les exigences [4]. La spécification et le code ont été formalisés séparément en SCR. Le projet a été complété avec succès et la station a eu son autorisation.

Un autre exemple qui englobe l'utilisation de méthodes formelles est le développement du TRANSPUTER [5] par le manufacturier de semi-conducteur Inmos. Il a été estimé que l'utilisation de méthodes formelles a permis de réduire le temps de développement de moitié par rapport aux méthodes traditionnelles.

Les méthodes formelles ont été utilisées avec succès dans le domaine du transport, où la sûreté des automatismes est primordiale. Les logiciels tiennent une part importante dans le fonctionnement des systèmes de contrôle/commande du mouvement. L'utilisation de méthodes formelles, pour développer ce type de logiciels, est une solution au double impératif: la qualité et la sûreté de fonctionnement. Une des applications, parmi les applications les plus connues dans le domaine du transport, est celle du Métro de Paris, en utilisant la méthode B de Abrial [2, 7].

Dans les facteurs de succès figurent, bien sûr, un support technique suffisant, mais également une réflexion sur l'intégration de la méthode choisie au sein du processus de développement requis par l'entreprise et la prise en compte des contraintes qui en découlent. Les applications pour lesquelles les techniques formelles ont été utilisées sont variées, avec un accent sur les domaines où la sûreté de fonctionnement est primordiale. Les résultats obtenus sont très positifs et les applications aux systèmes développés dans

l'industrie sont généralement facteurs de progrès, en ce sens que la qualité des logiciels est très nettement améliorée. Mais une mauvaise utilisation des méthodes formelles, en choisissant des objectifs inconvenables, en sélectionnant des langages et des techniques inappropriés, sont de bonnes raisons pour annihiler les avantages des méthodes formelles.

1.4 Conclusion

Il est clair que la spécification joue une variété de rôles dans le cycle de développement. Pour les gestionnaires de projet, elle peut être la base pour définir le plan de développement et ses critères de progression. Pour les ingénieurs de système, elle précise le rôle du logiciel dans l'ensemble du système et ses relations avec les composants matériels. Pour les développeurs, elle spécifie les contraintes sur les comportements et la performance. Pour les programmeurs, elle définit les choix acceptables de comportements du système. Et pour les testeurs, elle est la base pour la planification de tests et pour la vérification du système. Pour cela, une méthode qui fournit des spécifications précises, complètes et rigoureuses peut largement améliorer le processus de développement et la qualité du logiciel.

Concernant l'utilisation des méthodes formelles dans l'industrie, on peut dire que les méthodes et les spécifications formelles sont l'objet de nombreux développements dans le monde. Si des chiffres précis sur le coût et le gain occasionnés par l'utilisation des spécifications formelles ne sont pas nombreux, certaines entreprises ont réellement tenté l'expérience avec satisfaction. Cependant, l'absence d'un guide méthodologique durant l'utilisation d'une méthode particulière peut certainement contribuer à l'échec du projet.

Chapitre 2

Les notations

Dans ce chapitre, nous commençons par introduire la notation mathématique nécessaire à notre exposé. Ensuite, nous présentons les notations des quatre méthodes formelles suivantes : relationnelle inductive, boîte noire par entités, langage B et assertions de traces. Notons que nous ferons référence à ces notations dans les chapitres suivants. Pour chaque méthode, nous commençons par introduire ses concepts de base, sa syntaxe et sa sémantique. Au cours de chaque présentation, l'application de méthode sera illustrée avec un exemple simple, le compteur.

2.1 La notation mathématique

Nous utilisons les structures mathématiques suivantes pour la spécification. Nous dénotons l'ensemble des sous-ensembles de X par $\mathbb{P}(X)$. Nous utilisons $X \times Y$ pour dénoter le produit cartésien de deux ensembles. Nous utilisons le tuple $\langle x, y \rangle$ pour dénoter un élément du produit cartésien. Nous dénotons les relations par R. L'expression $x \lhd R \rhd y$ dénote que le tuple $\langle x, y \rangle$ est un élément de R (i.e., $\langle x, y \rangle \in R$). Une fonction (partielle ou totale) d'un ensemble dans un autre ensemble est dénotée par $f : X \longrightarrow Y$. La composition de deux fonctions, dénotée par $f \circ g$, est définie par $f \circ g\ (x) = f\ (g\ (x))$.

Nous dénotons la séquence vide par ε. Nous utilisons les variables x, y, z pour dénoter les séquences et nous dénotons par u, v, w des éléments de ces séquences. Les propriétés des données sont exprimées par des formules du calcul des prédicats du premier ordre, en utilisant les opérateurs ET ($\wedge$) OU ($\vee$) NON ($\neg$) implication ($\Rightarrow$) et équivalence ($\Leftrightarrow$) et les quantificateurs universel ($\forall$) et existentiel ($\exists$). Des constantes qui sont dénotées par a, b, c sont également considérées.

2.2 La méthode relationnelle inductive

La méthode relationnelle inductive est une méthode de spécification formelle. Elle se situe à un niveau très abstrait (boîte noire). Elle se concentre seulement sur la description du comportement observable du système [23].

Les particularités de cette méthode sont : l'induction, le raffinement, la lisibilité et la possibilité de faire une vérification automatique. La méthode inductive s'articule autour de quatre concepts : la spécification relationnelle, la logique du premier ordre, la définition inductive et la sémantique donnée par le plus petit point fixe d'une fonction. Pour spécifier un problème en utilisant cette méthode, on doit appliquer les deux formalismes présentés dans les deux sections suivantes.

2.2.1 Spécification relationnelle

Ce formalisme consiste à identifier toutes les séquences d'entrées possibles du système et leurs sorties correspondantes. Formellement, soit I l'ensemble de toutes les entrées possibles du système, soit I^+ l'ensemble de toutes les séquences finies et non vides construites avec des éléments de I, et soit O l'ensemble de toutes les sorties possibles du système. Alors une spécification représentée par la relation R est définie comme un sous-ensemble de $I^+ \times O$, c'est-à-dire $R \subseteq I^+ \times O$. Pour illustrer l'application de cette méthode, nous utilisons un exemple très simple, un compteur.

Informellement, il existe trois actions dans ce système :

1. **inc**, qui incrémente le compteur par la valeur 1 et n'a aucune sortie visible ;
2. **dec**, qui décrémente le compteur par la valeur 1 et n'a aucune sortie visible ;
3. **val**, qui retourne la valeur courante du compteur.

Alors, l'ensemble de toutes les entrées du système est donné par :

$$I = \{ \textbf{inc, dec, val} \}$$

L'ensemble de toutes les sorties possibles du système est donné par :

$O = \mathbb{Z} \cup \{ \tau \}$, où τ dénote la sortie invisible du système.

Considérons quelques éléments de la relation R du compteur :

$$\textbf{val} \lhd \mathrm{R} \rhd 0$$

$$\textbf{inc} \lhd \mathrm{R} \rhd \tau$$

$$\textbf{inc.val} \lhd \mathrm{R} \rhd 1$$

$$\textbf{inc.val.dec} \lhd \mathrm{R} \rhd \tau$$

$$\textbf{inc.val.dec.val} \lhd \mathrm{R} \rhd 0$$

L'interprétation de ces séquences est comme suit : le dernier symbole de la séquence d'entrée représente le dernier symbole reçu par le système. Les symboles précédents représentent l'historique de tout ce qui a été soumis au système depuis son démarrage. Le point "." dénote la concaténation de séquences. On ne fait pas de distinction entre une séquence d'un élément et cet élément, par souci de simplicité.

D'une manière générale, la relation R consomme une séquence d'entrées et produit la sortie correspondante. Comme il existe un nombre infini de séquences, alors il est impossible d'établir une spécification complète du système en utilisant cette simple énumération de couples. La définition inductive utilisée dans cette méthode peut contribuer à la résolution de ce problème [23]. Celle-ci fait l'objet de la prochaine section.

2.2.2 La définition des axiomes

Lorsque les espaces d'entrée et de sortie ont été définis, la définition des axiomes doit être entamée. En utilisant la logique du premier ordre, une définition inductive est donnée par un ensemble d'axiomes de la forme $\wedge_{i=1}^{n} A_i \Rightarrow B$, où n est un nombre naturel et A_i et B sont des méta-variables. Par souci de lisibilité, on représente ces axiomes de la manière suivante :

$$\frac{A_1 \wedge ... \wedge A_n}{B}$$

On peut diviser ces axiomes en trois classes : axiomes de base, axiomes de réduction et axiomes de permutation.

Les axiomes de base

Ils définissent toutes les sorties possibles pour les séquences de base. Dans l'exemple du compteur, trois axiomes de base sont proposés :

$$\text{B-1} : \frac{}{x.\mathbf{inc} \lhd R \rhd \tau},$$

$$\text{B-2} : \frac{}{x.\mathbf{dec} \lhd R \rhd \tau},$$

$$\text{B-3} : \frac{}{\mathbf{val} \lhd R \rhd 0}.$$

Dans les deux axiomes B-1 et B-2, le symbole x est une variable qui peut être n'importe quel élément de I^*. Les axiomes B-1 et B-2 indiquent que la soumission de l'entrée **inc** ou **dec** ne produit aucune sortie visible, peu importe l'historique. L'axiome B-3 indique que la valeur initiale du compteur est zéro si on soumet l'entrée **val** juste après le démarrage du système.

Pour des raisons de lisibilité, cette méthode utilise des tableaux pour présenter les axiomes. Le tableau 1 illustre les trois axiomes de base définis ci-dessus. Ce tableau se traduit ligne par ligne. Une ligne se traduit de la manière suivante : soit c une condition apparaissant sous la colonne "Prémisse", soit e une séquence d'entrées apparaissant sous la colonne "Entrée" et soit s une sortie apparaissant sous la colonne "Sortie". On dit que le système retourne la sortie s si on soumet l'entrée e au système à condition que c soit satisfaite. La traduction de cette ligne, sous forme d'un axiome, est la suivante :

$$\frac{c}{e \lhd R \rhd s}$$

La colonne "No" indique le numéro de l'axiome.

No	Prémisse	Entrée	Sortie
B-1		$x.\mathbf{inc}$	τ
B-2		$x.\mathbf{dec}$	τ
B-3		$\mathbf{val}$	0

TAB. 1 – *Axiomes de base du compteur*

Les axiomes de réduction

Ils servent à éliminer des symboles d'une séquence complexe donnée, afin de la réduire à une séquence de base. Pour illustrer ce concept, Nous proposons les trois axiomes suivants pour le compteur :

$$\text{R-1} : \frac{x.\mathbf{val} \lhd R \rhd y}{x.\mathbf{inc}.\mathbf{val} \lhd R \rhd y + 1},$$

$$\text{R-2} : \frac{x.\mathbf{val} \lhd R \rhd y}{x.\mathbf{dec}.\mathbf{val} \lhd R \rhd y - 1},$$

$$\text{R-3} : \frac{x.x' \lhd R \rhd y \wedge x' \neq \varepsilon}{x.\mathbf{val}.x' \lhd R \rhd y}.$$

L'axiome R-1 indique que l'entrée **inc** incrémente le compteur par la valeur 1. L'axiome R-2 indique que l'entrée **dec** décrémente le compteur par la valeur 1. L'axiome R-3 indique que l'entrée **val** ne change pas la valeur du compteur.

Comme les axiomes de base, les axiomes de réduction peuvent être écrits sous forme de tableaux. Le tableau 2 donne les axiomes de réduction. La traduction de ce tableau se fait ligne par ligne. Une ligne se traduit de la manière suivante : soit c une condition apparaissant sous la colonne "Condition", e_1, e_2 et e_3 des séquences apparaissant respectivement sous la colonne "Préfixe", "Supprimer" et "Suffixe", et s est un terme apparaissant sous la colonne sortie. On distingue deux cas :

1. si la colonne "Sortie" est vide, alors on obtient l'axiome suivant :

$$\frac{e_1.e_3 \lhd R \rhd y \wedge c}{e_1.e_2.e_3 \lhd R \rhd y},$$

   en supposant que la variable y n'apparaisse pas dans c, e_1, e_2 et e_3.

2. si un terme est spécifié dans la colonne "Sortie" on obtient l'axiome suivant :

$$\frac{e_1.e_3 \lhd R \rhd y \wedge c}{e_1.e_2.e_3 \lhd R \rhd y'},$$

   en supposant que la variable y est utilisée dans le terme spécifié dans la colonne "Sortie" y'.

No	Condition	Préfixe	Supprimer	Suffixe	Sortie
R-1		x	**inc**	**val**	y+1
R-2		x	**dec**	**val**	y-1
R-3	$x' \neq \varepsilon$	x	**val**	x'	

TAB. 2 – *Axiomes de réduction du compteur*

Ces axiomes peuvent être utilisés pour démontrer qu'un couple appartient à la relation R. Voici un exemple de preuve permettant de calculer, à l'aide des axiomes B-1, R-1, R-2 et R-3, la sortie correspondante à une séquence donnée.

inc.inc.val.dec.val ⊲R⊳ 1

Voici la démonstration de preuve du couple ci dessus :

inc.inc.val.dec.val $\lhd R \rhd$ 1

$\Leftarrow$ R-2

inc.inc.val.val $\lhd R \rhd$ 2

$\Leftarrow$ R-3

inc.inc.val $\lhd R \rhd$ 2

$\Leftarrow$ R-1

inc.val $\lhd R \rhd$ 1

$\Leftarrow$ R-1

val $\lhd R \rhd$ 0

$\Leftarrow$ B-1

VRAI

Les axiomes de permutation

Ils transforment une séquence donnée, qui n'est pas calculable par les axiomes de base ni par les axiomes de réduction, à une séquence calculable, en permutant des éléments. Dans l'exemple du compteur, tous les axiomes de permutation peuvent être déduits des autres axiomes. Pour cela, il n'est pas nécessaire de définir ce type d'axiomes. Par exemple l'axiome P-1 peut être déduit des axiomes R-1 et R-2. Dans ce cas, P-1 peut être omis de la spécification.

$$\text{P-1} : \frac{x.\mathbf{inc}.\mathbf{dec}.x' \lhd R \rhd y \wedge x' \neq \varepsilon}{x.\mathbf{dec}.\mathbf{inc}.x' \lhd R \rhd y}$$

2.3 La méthode boîte noire par entités

La méthode boîte noire par entités décrit formellement le comportement du système avec une relation d'entrée-sortie $R \subseteq I^+ \times O$, comme dans la méthode relationnelle

inductive [24]. Elle s'inspire d'un concept issu de la méthode *JSD* (*Jackson System Development*) [32], soit le diagramme de structure d'entité. De plus, elle est fondée sur l'algèbre de processus, les expressions régulières et la logique des prédicats du premier ordre. Ceux-ci nous permettent de décrire la relation R entre les entrées et les sorties du système.

2.3.1 Les opérateurs de base des séquences

Plusieurs opérateurs classiques des séquences sont employés dans cette méthode. Nous utilisons $x \vdash u$ pour dénoter l'ajout de l'élément u à la droite de la séquence x. Nous utilisons $u \dashv x$ pour dénoter l'ajout de l'élément u à la gauche de la séquence x . Nous utilisons $x \downarrow$ A pour dénoter la projection de la séquence x sur l'alphabet A. Nous utilisons $A^* \triangleq A^+ \cup \{ \varepsilon \}$. Nous utilisons *prefix(x)* pour dénoter l'ensemble de tous les préfixes de la séquence x. Nous utilisons *last(x)* pour dénoter le dernier élément de la séquence non vide x. Nous utilisons *head(x)* pour dénoter le premier élément de la séquence non vide x. Nous utilisons *tail(x)* pour dénoter la séquence non vide x amputée de son premier élément. On utilise *front(x)* pour dénoter la séquence non vide x amputée de son dernier élément. On utilise $\preceq$ pour dénoter l'inclusion entre les séquences. Enfin, nous utilisons $\#x$ pour dénoter la longueur de x. D'autres notations peuvent être définies et utilisées au cours de la spécification d'un problème particulier.

2.3.2 Présentation de la méthode

Une spécification sous cette méthode se développe en quatre étapes: 1) la définition des espaces d'entrée et de sortie, 2) la définition des entités du système en termes de séquences d'entrée, 3) la définition des contraintes et enfin, 4) la définition de la relation entre les entrées et les sorties. Voici une description de ces étapes :

La définition des espaces d'entrée et de sortie

Cette phase consiste à déterminer toutes les entrées du système et leurs attributs, et à définir les ensembles de sorties correspondant à chaque entrée, comme dans la méthode inductive. La distinction entre les deux méthodes, concernant la définition des espaces d'entrée et de sortie, est la numérotation des entrées et la notion d'attribut qui sont absents dans la méthode inductive.

Par exemple, l'ensemble d'entrées $I_1 \triangleq \{ \textbf{inc} \} \times C\_ID$, avec l'attribut C_ID (identificateur de compteur), a le singleton $O_1 \triangleq \{ \tau \}$ comme ensemble de sorties. Le symbole τ dénote que le système ne retourne aucune sortie visible. Les ensembles d'entrée et de sortie du compteur sont les suivants :

$$I_1 \triangleq \{ \textbf{inc} \} \times C\_ID \qquad O_1 \triangleq \{ \tau \}$$
$$I_2 \triangleq \{ \textbf{dec} \} \times C\_ID \qquad O_2 \triangleq \{ \tau \}$$
$$I_3 \triangleq \{ \textbf{val} \} \times C\_ID \qquad O_3 \triangleq \mathbb{Z}$$

Alors que les espaces d'entrée et de sortie du système sont donnés par :

$$I \triangleq \bigcup_{i=1}^{3} I_i \qquad \text{et} \qquad O \triangleq \bigcup_{i=1}^{3} O_i$$

Et l'alphabet de l'entité *Compteur* est donné par : $\Sigma_{Compteur} \triangleq I$.

On constate que l'espace d'entrée du compteur est différent de celui utilisé dans la méthode relationnelle inductive. Nous avons fait ce choix afin de mieux illustrer la méthode boîte noire par entité. Ce système comporte plusieurs compteurs, au lieu d'un seul. Un compteur est identifié par un élément $c\_id \in C\_ID$. L'entrée $\langle \textbf{inc}, c\_id \rangle$ sert à incrémenter le compteur *c_id*. De la même manière, on peut décrémenter le compteur *c_id* avec $\langle \textbf{dec}, c\_id \rangle$ et interroger sa valeur avec $\langle \textbf{val}, c\_id \rangle$. L'espace de sortie de l'entrée **val** est l'ensemble des nombres entiers $\mathbb{Z}$. Nous supposons que C_ID est l'ensemble des nombres naturels $\mathbb{N}$.

La définition des entités du système

L'objectif de cette phase est de définir toutes les entités du système et de décrire leurs comportements individuels en termes de séquences d'entrées. Pour ce faire, on utilise le diagramme de structure d'entité inspiré de la méthode *JSD* pour exprimer l'ordre des séquences valides d'entrée.

Dans cette méthode, chaque entité peut être interprétée comme un ensemble d'objets. Chaque objet a plusieurs instances possibles. Une instance représente un comportement possible d'un objet particulier. Par exemple, un objet du type compteur, *c_id*, peut être incrémenté et décrémenté, ou décrémenté et incrémenté. Ces deux instances sont deux comportements possibles du même objet *c_id* de l'entité compteur.

Dans le cas présent, le système compteur n'a qu'une seule entité, nommée *Compteur*. La figure 3 montre le comportement possible de cette entité. Dans cette figure, les deux symboles "*" et "|" dénotent respectivement l'itération et la sélection.

Pour définir une entité, on peut utiliser les opérateurs des expressions régulières ou d'une algèbre de processus comme CSP [28] ou CCS [35]

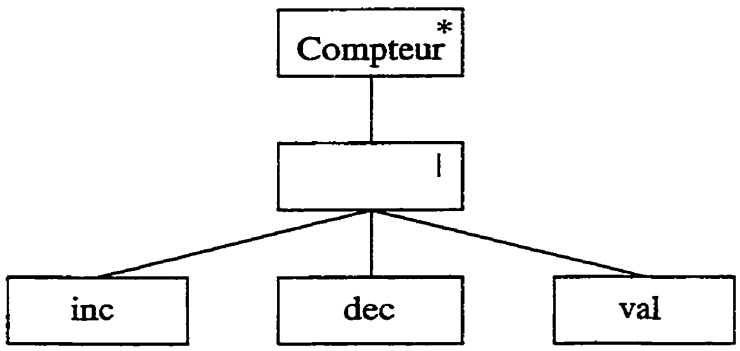


FIG. 3 – *Diagramme de structure de l'entité "Compteur"*

L'ensemble de clés identifiant les entités du système est donné par *C_ID*. La fonction totale qui fournit l'ensemble des séquences d'entrées pour une clé $c\_id \in C\_ID$ est dénotée par $f_c \in C\_ID \longrightarrow \mathbb{P}(\Sigma_c^*)$, où Σ_c^* est l'ensemble de toutes les séquences d'entrées (incluant

la séquence vide). Dans l'exemple du compteur, la fonction f_c est définie comme suit :

$$f_c(c\_id) \triangleq Compteur(c\_id).$$

La description du comportement du compteur, en termes des séquences d'entrées, est donnée par :

$$Compteur(c\_id) \triangleq (\ \langle \mathbf{inc}, c\_id \rangle \mid \langle \mathbf{dec}, c\_id \rangle \mid \langle \mathbf{val}, c\_id \rangle\ )^*$$

Dans ce paragraphe, on présente la sémantique formelle du diagramme défini dans la phase précédente. Elle consiste à décrire la façon par laquelle les instances seront combinées pour définir les séquences d'entrée bien formées. Comme $f_e(\ k\ )$ est l'ensemble des séquences d'entrées de l'entité e ayant une clé k, cela signifie que $f_e(k)$ est l'ensemble de toutes les instances de l'objet de clé k. Notons que le diagramme de structure de l'entité e représente l'arbre syntaxique de $f_e(k)$.

Progressivement, on définit l'ensemble d'instances de l'entité e, *i.e.* de tous les objets de e, et l'ensemble d'instances du système.

Une séquences d'entrée bien formée est générée de la manière suivante :

1. choisir un ensemble d'objets pour chaque entité ;
2. choisir une instance pour chacun de ces objets ;
3. calculer l'entrelacement des instances d'une entité ;
4. calculer le produit parallèle avec synchronisation sur les entrées communes entre les entités.

Comme il n'y a qu'une seule entité dans notre exemple, l'étape 4 est triviale.

Deux prédicats ont été définis pour extraire les instances d'une séquence d'entrées :

1. *instance_de (x, y, f_e)* : détermine si l'instance x de l'entité définie par f_e est une sous-séquence de la séquence d'entrées y.
2. *instance_de_clé (x, y, f_e, k)* : détermine si l'instance x de l'entité définie par f_e, ayant k comme clé, est une sous-séquence de la séquence d'entrées y.

Par exemple ces deux prédicats sont satisfaits si :

$$x = \langle \textbf{inc}, 1\rangle.\langle \textbf{dec}, 1\rangle,$$
$$y = \langle \textbf{inc}, 1\rangle.\langle \textbf{inc}, 2\rangle.\langle \textbf{dec}, 2\rangle.\langle \textbf{dec}, 1\rangle.\langle \textbf{val}, 2\rangle,$$

avec $k = 1$ et $f_e = \textit{Compteur}$.

La définition des contraintes

Dans cette phase, on définit des contraintes sur les séquences d'entrées bien formées, afin de définir l'ensemble des séquences valides. Ces contraintes nous permettent d'exprimer des conditions sur les séquences d'entrées qui ne peuvent pas être exprimées avec des expressions régulières. L'ensemble de toutes les séquences valides est donné par :

$$\textbf{Valide}_{\text{E}} = \{\ x \mid x \in S_E \wedge \textit{Contraintes (x)}\ \}$$

où x est une séquence d'entrée. On dénote par S_E l'ensemble des séquences bien formées du système E. Le prédicat *Contraintes* (x) est défini comme suit :

$$\textit{Contraintes (x)} \Leftrightarrow \bigwedge_{i=1}^{n} P_i,$$

où P_i est un prédicat qui représente une contrainte particulière.

Par exemple, supposons qu'on doit définir une contrainte, $P(y)$, pour garantir que la valeur du compteur soit plus grande ou égale à zéro. La définition mathématique de cette contrainte est donnée par :

$$P(y) \Leftrightarrow$$
$$\forall\, x, z : \textit{instance\_de}(x, y, f_{\textit{Compteur}}) \wedge z \in \textit{prefix(x)}$$
$$\Rightarrow$$
$$\#\ (z \downdownarrows \{\ \textbf{inc}\ \})\ -\ \#\ (z \downdownarrows \{\textbf{dec}\ \}) \geq 0$$

On utilise l'opérateur "$\downdownarrows$" pour dénoter la projection sur l'étiquette des éléments d'entrée de la séquence x.

La définition de la relation d'entrée-sortie

Cette dernière phase consiste à fixer toutes les sorties possibles pour chaque entrée, autrement dit définir le comportement d'entrée-sortie du système. Ce comportement est défini par des axiomes de la forme suivante :

$$x \vdash u \in \mathbf{Valide}_{\mathrm{E}} \quad \wedge \quad \mathit{Conjonction}$$
$$\Rightarrow$$
$$x \vdash u \lhd \mathrm{R} \rhd o$$

où u dénote le dernier élément de la séquence d'entrée et *Conjonction* est la conjonction des prédicats spécifiés pour calculer la sortie de l'entrée u, en tenant en compte de l'historique d'entrées représenté par x. Pour l'exemple du compteur, trois axiomes ont été définis :

$$1\text{-}\ x \vdash \langle \mathbf{inc}, c\_id \rangle \in \mathbf{Valide}_{\mathbf{Compteur}}$$
$$\Rightarrow$$
$$x \vdash \langle \mathbf{inc}, c\_id \rangle \lhd \mathrm{R} \rhd_{\mathcal{T}}$$
$$2\text{-}\ x \vdash \langle \mathbf{dec}, c\_id \rangle \in \mathbf{Valide}_{\mathbf{Compteur}}$$
$$\Rightarrow$$
$$x \vdash \langle \mathbf{dec}, c\_id \rangle \lhd \mathrm{R} \rhd_{\mathcal{T}}$$
$$3\text{-}\ x \vdash \langle \mathbf{val}, c\_id \rangle \in \mathbf{Valide}_{\mathbf{Compteur}}$$
$$\Rightarrow$$
$$x \vdash \langle \mathbf{val}, c\_id \rangle \lhd \mathrm{R} \rhd \mathit{Calcul}(x, c\_id)$$

Comme cette méthode ne spécifie que les séquences valides d'entrées et que, dans le cas du compteur, une séquence x étendue par n'importe quelle entrée est toujours valide, les axiomes ci-dessus peuvent être écrits de la manière suivante :

$$1\text{-}\quad x \vdash \langle \mathbf{inc}, c\_id \rangle \lhd \mathrm{R} \rhd_{\mathcal{T}}$$
$$2\text{-}\quad x \vdash \langle \mathbf{dec}, c\_id \rangle \lhd \mathrm{R} \rhd_{\mathcal{T}}$$
$$3\text{-}\quad x \vdash \langle \mathbf{val}, c\_id \rangle \lhd \mathrm{R} \rhd \mathit{Calcul}(x, c\_id)$$

où $\mathit{Calcul}(x, c\_id)$ est une fonction qui retourne une valeur entière, en calculant le nombre d'entrées **inc** de la séquence x moins le nombre d'entrées **dec**.

La fonction $Calcul(x, c\_id)$ est définie comme suit :

instance_de_clé (c, x, Compteur, c_id)

$\Rightarrow$

$Calcul(x, c\_id) = \#\ (\ c \downdownarrows \{\ \mathbf{inc}\ \})\ -\ \#\ (\ c \downdownarrows \{\ \mathbf{dec}\ \})$

2.4 La méthode B

La méthode B est une méthode pour la spécification et la conception de logiciels, fondée sur la logique du premier ordre, la théorie des ensembles et la théorie du raffinement. Elle fait partie de la famille des méthodes dites orientées modèle qui représentent les logiciels par des données, caractérisées par leurs propriétés invariantes, et des services qui manipulent ces données [2, 3]. Les caractéristiques de la méthode B sont les suivantes :

- elle dispose d'un seul cadre formel pour décrire abstraitement et concrètement les logiciels, autrement dit, pour les spécifier et les concevoir ;
- elle autorise le développement progressif des modèles pour des transformations successives de leur spécification, appelées raffinements ;
- elle intègre les concepts d'encapsulation des données et de masquage de l'information, et a été conçue pour la construction structurée et modulaire de logiciels ;
- elle fixe les conditions de vérification qui garantissent la cohérence de la spécification ainsi que la cohérence et la conformité à cette spécification, en ce qui concerne ces raffinements.

2.4.1 Les notations

La méthode B est composée de trois notations : la notation mathématique, la notation des substitutions généralisées et la notation des machines abstraites.

La notation mathématique

Dans le langage de la méthode B, la notation mathématique permet de modéliser les données et les propriétés des données des logiciels. Cette notation est essentiellement la notation de la théorie des ensembles. Cependant, à la différence de la théorie classique des ensembles, la théorie des ensembles de B est typée : les ensembles sont constitués d'éléments ayant tous la même structure fondamentale.

La notation des substitutions généralisées

La notation des substitutions généralisées permet de modéliser les services des logiciels, autrement dit, les actions que les logiciels peuvent effectuer pour remplir leurs fonctions. Une substitution généralisée est un transformateur de prédicats [18] qui, appliqué aux prédicats qui caractérisent les données avant l'activation d'un service, produit les prédicats qui caractérisent les données après l'accomplissement de ce service.

La substitution élémentaire est la substitution lexicale de la logique classique : si x est une variable et e est une expression ensembliste, la substitution élémentaire $x := e$, appliquée à un prédicat P (ce que l'on note $[x := e\,]P$) transforme P en un prédicat P', obtenu en remplaçant dans P toutes les occurences libres de x par e.

Les substitutions généralisées sont des extensions de la substitution élémentaire. Les principales substitutions sont les suivantes :

- La substitution avec précondition, notée **pre** P **then** S **end**, qui définit les conditions d'utilisation d'une autre substitution. Si P est vrai, S s'exécute. Sinon la substitution peut ne pas terminer.
- La substitution avec garde, notée **select** P **then** S **end**, qui définit le contexte d'application d'une autre substitution. Si P est vrai, S s'exécute. Sinon la substitution peut produire n'importe quel résultat désiré.

- La substitution de choix non déterministe borné, notée choice $S1$ or $S2$ end, qui définit deux substitutions possibles.
- La substitution de choix non déterministe non borné, notée any X where P then S end, qui définit un nombre indéterminé de substitutions possibles, une pour chaque valeur des variables X qui satisfait le prédicat P.
- La composition parallèle de substitutions, notée $S1 \parallel S2$, qui effectue simultanément deux substitutions.
- La substitution identité, notée skip , qui laisse inchangé les prédicats.
- La substitution conditionnelle déterministe, notée if C then $S1$ else $S2$.
- La composition séquentielle de substitution , notée $S1 ; S2$.
- La substitution d'itération, notée while C do S invariant I variant V end.

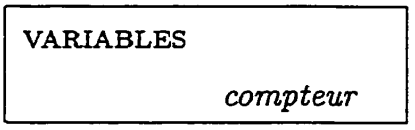
Les quatre premières substitutions de l'énumération précédente sont utilisées pour la spécification, c'est-à-dire pour définir ce que les services doivent réaliser, alors que les trois dernières sont utilisées pour la programmation, c'est-à-dire pour définir des services exécutables.

La notation des machines abstraites

La notation des machines abstraites définit les composants et les liens de composition des composants qui permettent de construire des modèles de logiciels structurés et modulaires. Il y a trois types de composants dans le langage de la méthode B : les machines abstraites, les raffinements et les implantations. Les sous-sections suivantes présentent plus en détail leurs caractéristiques.

Machines abstraites – Une machine abstraite définit les ensembles, les constantes, les variables et les opérations qui modélisent abstraitement les données et les services d'un logiciel. Ce faisant, elle définit l'interface opérationnelle du logiciel, parce qu'elle fixe le mode et les conditions d'utilisation de ses constituants. Prenons l'exemple du compteur.

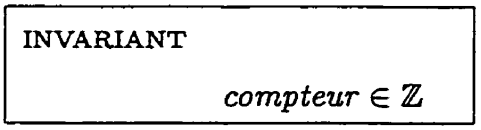
Le nom de la machine est **Compteur.** On ne lui passe aucun paramètre.



Le symbole *compteur* est une variable. On spécifie dans la rubrique INVARIANT quel est son type.

Les ensembles et les constantes d'une machine abstraite représentent les données immuables du logiciel. Les ensembles peuvent être ou bien des ensembles abstraits ou bien des ensembles énumérés. Un ensemble abstrait est un ensemble dont on ne veut connaître, à ce stade de la modélisation, ni le nombre, ni la nature exacte de ses éléments. Un ensemble énuméré est un ensemble défini par la liste complète de ses éléments. Les constantes sont caractérisées par une conjonction de prédicats de la notation mathématique.

Les variables d'une machine abstraite représentent les données modifiables du logiciel. Comme les constantes, les variables sont définies par une conjonction de prédicats, appelée INVARIANT, qui fixe les propriétés que leurs valeurs effectives doivent toujours satisfaire.



On apprend ici que *compteur* appartient à l'ensemble d'entiers $\mathbb{Z}$. Les ensembles, les constantes et les variables définissent la partie statique du logiciel. Les opérations définissent la partie dynamique, c'est-à-dire les actions atomiques que le logiciel peut effectuer pour remplir sa fonction. Dans les machines abstraites, les opérations modélisent les effets observables des actions sur les variables et non la manière dont ces effets

sont obtenus. Une opération particulière, l'initialisation, fixe les valeurs initiales des variables. Les opérations sont définies par des substitutions de la notation des substitutions généralisées.

```
INITIALISATION
                    compteur:=0
```

La rubrique INITIALISATION permet d'indiquer que la valeur initiale de la variable *compteur* est égale à 0. La rubrique OPERATIONS contient la définition des opérations offertes par la machine abstraite. La forme générale d'une opération est la suivante :

Nom des paramètres de sortie ← *Nom de l'opération (Nom des paramètres d'entrée)*

Dans l'exemple du compteur, on a les deux opérations suivantes qui n'ont aucun paramètre d'entrée ni de sortie :

```
OPERATIONS
    inc ≙
        begin
            compteur:=compteur +1
        end ;
    dec ≙
        begin
            compteur:=compteur -1
        end ;
```

L'opération **val**, qui affiche la valeur de compteur, a le paramètre de sortie *valeur*.

```
valeur ← val ≙
            begin
                valeur:=compteur
            end ;
```

Des conditions de vérification sont associées aux machines abstraites. Ce sont des prédicats dont la démonstration garantit les propriétés suivantes : premièrement, que l'initialisation établit l'invariant, c'est-à-dire que les valeurs initiales des variables satisfont aux propriétés fixées par l'invariant ; deuxièment, que chaque opération préserve l'invariant, c'est-à-dire que dans l'hypothèse où les valeurs des variables satisfont l'invariant avant l'activation de l'opération, alors, après l'accomplissement de l'opération, les nouvelles valeurs des variables satisfont également les propriétés fixées par l'invariant.

Raffinements – Généralement, une machine abstraite n'est pas un modèle exécutable du logiciel qu'elle définit, parce que ses données et ses opérations sont trop abstraites pour pouvoir être automatiquement traduites dans un langage de programmation. Pour obtenir un modèle exécutable, il est nécessaire d'enrichir et transformer la représentation abstraite des données et des opérations en une représentation concrète, c'est-à-dire exécutable, fonctionnellement équivalente. Le changement de représentation est réalisé par un ou plusieurs raffinements successifs. Leur nombre n'est pas imposé, mais est en fonction de la distance qui sépare la représentation abstraite de la représentation concrète. Ainsi, raffiner un composant (une machine abstraite ou un autre raffinement d'une machine abstraite) consiste à enrichir ou à représenter l'information de ses variables abstraites par des variables concrètes dont la structure se rapproche de la structure de données programmable, et à représenter ses opérations abstraites par des opérations concrètes définies par des substitutions qui introduisent des structures de contrôle de programmation. Concrètement, un composant raffinement, comme une machine abstraite, est composé d'ensembles, de constantes, de variables et d'opérations. Toutefois, les opérations de la machine abstraite doivent avoir exactement la signature des opérations et de la machine abstraite raffinée. De plus, son invariant doit formaliser la relation de représentation qui lie les variables du raffinement aux variables du composant raffiné.

Des conditions de vérification sont associées à tout raffinement. Leur démonstration garantit que, compte tenu de la relation de raffinement entre les variables du raffinement

et les variables du composant raffiné, le modèle du raffinement est fonctionnellement équivalent au modèle du composant raffiné. Ceci signifie qu'un appel à une opération d'un raffinement a exactement les mêmes effets visibles qu'un appel à l'opération, de même signature, du composant raffiné de la machine abstraite initiale.

Implantations – Une implantation est le dernier raffinement d'une machine abstraite. C'est un modèle exécutable, le sous-ensemble du langage de la méthode B qui offre les structures de données fondamentales de programmation (entiers compris dans un intervalle MININT..MAXINT, booléens et tableaux) et les substitutions fondamentales de programmation (affectation, séquencement, if then else, case of, boucle while, appel à une opération, etc.)

Une implantation fixe les valeurs effectives des ensembles abstraits et des constantes introduits au cours du développement. Une implantation peut être construite avec des machines abstraites qui offrent les données et les opérations nécessaires à la réalisation des données et des opérations implantées.

Les conditions de vérification associées à une implantation sont celles d'un raffinement, auxquelles s'ajoutent les conditions qui montrent que les valeurs effectives des ensembles abstraits et des constantes satisfont aux propriétés qui les caractérisent.

Liens de composition des composants – La méthode B autorise différents liens de composition entre composants d'une même application. Les principaux liens de composition sont le lien *sees* qui permet à une machine abstraite, un raffinement ou une implantation de voir une autre machine abstraite, pour accéder, en lecture seulement, à ses ensembles, ses constantes et ses variables, et le lien *imports,* qui permet à une implantation de s'appuyer sur une machine abstraite M pour réaliser ses propres données et opérations avec les données et les opérations de M.

Notons que notre spécification du cas d'étude se limite à la notation de machine abstraite et n'inclut pas le raffinement ni l'implantation.

2.5 La méthode des assertions de traces

La méthode des assertions de traces, proposée par Parnas [38] et améliorée par Bartussek [6] et puis par Wang [45], permet de spécifier les comportements observables d'un système sans référer aux structures internes de données. Une trace d'un module est définie comme une séquence d'appels des opérations appelées par les programmes d'accès.

2.5.1 Présentation

La méthode des assertions de traces est basée sur les concepts suivants : l'encapsulation de données, les séquences, les équations explicites et les machines à états. Pour modéliser un système avec cette méthode, on doit le décomposer en modules. Ces modules sont vus comme des boîtes noires ; on spécifie seulement leurs comportements observables.

Un module est considéré comme un ensemble de programmes qui fournissent l'accès à une structure de données. Le comportement d'un module est caractérisé par les événements reçus de l'environnement (appels de programmes d'accès). Trois types de programmes sont distingués : les programmes d'affectation qui changent les valeurs des données, les programmes d'interrogation qui consultent les données sans changer leurs valeurs et les programmes d'affectation et d'interrogation qui consultent les données et changent leurs valeurs.

Un module est spécifié comme une machine à états où les entrées sont représentées par les appels de programmes d'accès et les sorties sont représentées par leurs valeurs de sortie. L'état d'un module correspond à un intervalle de temps entre deux appels de programmes d'affectations ou d'affectations et d'interrogations. Chaque état est représenté par une classe d'équivalence de traces. Deux traces s, s' sont considérées comme équivalentes SSi :

$$\forall\ s'' : s.s'' \text{ est une trace légale} \Leftrightarrow s'.s'' \text{ est une trace légale}$$

Une trace légale est une séquence finie d'appels, syntaxiquement correcte, des programmes

séparés par l'opérateur de concaténation ".", et elle fournit l'historique complet du comportement observable du module, en incluant tous les appels et toutes les sorties produites. Une transition est un changement d'état déclenché par un appel, autrement dit un changement de la classe d'équivalence. La trace vide, dénotée par "_", représente l'état initial du module.

La méthode des assertions de traces utilise des tableaux pour présenter la syntaxe des programmes d'accès, la fonction de transition d'état et les valeurs de sorties. Tout ceci fait l'objet de la prochaine section.

2.5.2 La structure de la spécification

La méthode des assertions de traces propose un document de spécification bien structuré. Il est divisé en cinq sections :

1. la syntaxe,
2. les traces canoniques,
3. les traces équivalentes,
4. les valeurs de sortie,
5. le dictionnaire.

La syntaxe

Le tableau 3 illustre le format général du tableau des programmes d'accès d'un module. La traduction de ce tableau se fait ligne par ligne. Chaque ligne décrit la syntaxe d'un programme d'accès, en spécifiant le nom du programme, les types de ses arguments, et le type de sa valeur de sortie. On utilise le symbole "< >" pour indiquer le type d'argument. On utilise le symbole ": O" pour indiquer que l'argument est un paramètre d'entrée-sortie. Le type de la valeur de sortie d'un programme apparaît sous la colonne "Valeur". Notons que les lignes du tableau sont triées par ordre alphabétique de nom de

programme.

Par exemple, la première ligne du tableau 3 est traduite de la manière suivante : $\mathbf{P_1}$ est le nom de programme. Son premier argument qui est de type "Booléen" et il est considéré comme un paramètre d'entrée-sortie. Les deux arguments Arg#i et Arg#n sont respectivement de type "Entier" et "Char". La valeur de sortie du programme $\mathbf{P_1}$, apparaissant sous la colonne "Valeur", est de type "Booléen". La signature de ce programme est définie comme suit :

$$\mathbf{P_1}(\text{Arg\#1, Arg\#}i\text{, Arg\#}n\text{, Booléen})$$

À titre d'exemple, la traduction de cette signature en langage de programmation C++ est comme suit :

$$\text{bool } \mathbf{P_1}(\text{bool\& Arg\#1,..., int Arg\#}i\text{, ..., char Arg\#}n)$$

Nom de Programme	Arg#1	...	Arg#i	...	Arg#n	Valeur
$\mathbf{P_1}$	< Booléen > : O	...	< Entier >	...	< Char >	Booléen
...	...	...	...	...	...	...
$\mathbf{P_m}$	< Entier >					

TAB. 3 – *La syntaxe des programmes d'accès*

Nom de Programme	Valeur
dec	
inc	
val	Entier

TAB. 4 – *La syntaxe des programmes d'accès du Compteur*

Le système *Compteur* a les trois programmes d'accès suivants : **dec, inc** et **val**. Comme ces programmes n'ont aucun paramètre d'entrée, toutes les colonnes d'arguments peuvent être omises. Le tableau 4 montre la syntaxe de chacun de ces programmes. On remarque que les types des valeurs de sortie des programmes **dec** et **inc** ne sont pas

déclarés. Ceci signifie que ces deux programmes n'ont aucune sortie visible. Par contre, le programme d'accès **val** a une sortie visible puisque son type a été déclaré sous la colonne "Valeur".

Les traces canoniques

Comme les programmes d'interrogation ne changent pas les données de module, alors ils n'ont aucun effet sur l'état du module et, par conséquent, leur présence n'est pas nécessaire dans les traces canoniques. Certains programmes d'affectation et certains programmes d'affectation et d'interrogation fournissent des données au module ; ils sont appelés *Information Contributors* (ICT) ou constructeurs de données. D'autres éliminent une partie de données existantes ; ils sont appelés *Information Consumers* (ICS) ou consommateurs de données. Finalement, d'autres changent les valeurs de données existantes ; ils sont appelés *Information Modifiers* (IMD) ou modificateurs de données [25].

Pour définir les traces canoniques, une heuristique est de choisir seulement les programmes de type ICT pour définir les traces canoniques. Toute trace qui contient des programmes de type ICS ou IMD peut être remplacée par une trace canonique plus courte qui ne contient que de programmes de type ICT. Par exemple, **inc** et **dec** sont de type ICT dans le module *Compteur*. La définition mathématique d'une trace canonique du *Compteur* est :

$$\text{Canonique}(T) \Longleftrightarrow (T \in \mathbf{inc}^* \cup \mathbf{dec}^*)$$

Les traces équivalentes

Cette section définit la fonction de transition d'état, r. Cette section contient toutes les assertions qui définissent les transitions d'état, sous la forme du tableau 5. Une ligne de ce tableau est traduite de la manière suivante : soit C_1 une condition, T_1 une trace quelconque. On dit que la trace T étendue par le programme d'accès E est équivalente à

T_{r1} si $T = T_1$ et $T.E$ satisfait la condition C_1. La traduction formelle de cette ligne est : $C_1(T.E) \wedge T = T_1 \Rightarrow T.E =_r T_{r1}$

$T.E =_r$

Condition	Patron de trace	Equivalente
C_1	T_1	T_{r1}
...	...	...
C_n	T_n	T_{rn}

TAB. 5 – *Le format général du tableau des traces équivalentes*

On peut interpréter une ligne du tableau comme une transition d'état de la manière suivante. Dans l'état T_1, si le programme E est appelé, le module peut aller à l'état T_{r1}, si la condition C_1 est satisfaite.

Une valeur "vrai" apparaissant sous la colonne de "Patron de trace" indique qu'il n'est pas nécessaire d'introduire un patron de trace contenant des variables, autrement dit, il n'y a pas de contrainte sur la trace T. Si toutes les entrées dans la colonne de "Patron de trace" sont vrai, alors la colonne peut être entièrement omise. De plus, si toutes les entrées apparaissant dans la colonne "Condition" sont **vrai**, i.e., il n'existe pas de contraintes sur la trace canonique T étendue par E, alors la colonne peut aussi être omise.

Par exemple, le tableau 8 peut être remplacée par :

$T.\mathbf{val} =_r T$, où T est une trace quelconque.

On a énuméré les tableaux des traces équivalentes en ordre alphabétique selon les noms de programmes E. Dans l'exemple du compteur, les tableaux 6, 7 et 8 illustrent respectivement les traces équivalentes pour les programmes **dec**, **inc** et **val**.

Les valeurs de sortie

Le tableau 9 illustre le format général d'un tableau de valeurs de sortie pour chaque programme d'accès ayant une valeur de sortie. La traduction de ce tableau se fait de la

T.**dec** $=_r$

Condition	Patron de trace	Équivalente
vrai	-	**dec**
$T \in$ **inc***	S. **inc**.S'	$S.S'$
$T \in$ **dec***	**vrai**	T.**dec**

TAB. 6 – *Les traces équivalentes du programme d'accès* **dec**

T.**inc** $=_r$

Condition	Patron de trace	Équivalente
vrai	-	**inc**
$T \in$ **dec***	S. **dec**.S'	$S.S'$
$T \in$ **inc***	**vrai**	T.**inc**

TAB. 7 – *Les traces équivalentes du programme d'accès* **inc**

manière suivante. Soit C_1 une condition, T_1 une trace quelconque étendue par le programme d'accès E. On dit que le programme d'accès E retourne la valeur V_1 si $T = T_1$ et $T.E$ satisfait la condition C_1. Autrement dit, si le module est dans l'état $T = T_1$ et que le programme E est appelé, il produit en sortie la valeur V_1, si la condition C_1 est satisfaite.

Le tableau 10 indique les valeurs de sortie possibles du programme **val**. La fonction auxiliaire f (T) calcule la longueur de la trace. La définition mathématique de cette fonction sera donnée dans la section dictionnaire.

Le dictionnaire

La section du dictionnaire présente les définitions des termes, des fonctions auxiliaires, des types et de toute autre structure utilisée dans le corps de la spécification.

T.**val** $=_r$

Condition	Patron de trace	Équivalente
vrai	**vrai**	T

TAB. 8 – *Les traces équivalentes du programme d'accès* **val**

$O(T, E) =$

Condition	Patron de trace	Valeur de sortie
C_1	T_1	V_1
...	...	...
C_n	T_n	V_n

TAB. 9 – *Format général du tableau de valeurs de sortie*

O(T, **val**) =

Condition	Patron de trace	Valeur de sortie
vrai	_	0
$T \in$ **inc***	**vrai**	$f(T)$
$T \in$ **dec***	**vrai**	$-f(T)$

TAB. 10 – *Les valeurs de sortie du programme d'accès* **val**

Ces définitions sont souvent utilisées dans les situations suivantes.

1. Certains prédicats ou fonctions utilisés dans le corps de la spécification sont trop complexes pour être complètement développés dans un seul endroit. Les fonctions auxiliaires sont introduites pour décomposer les fonctions complexes en fonctions plus simples.
2. D'autres prédicats utilisent des expressions qui apparaissent dans plusieurs endroits de la spécification. Alors, il est très utile de faire la factorisation et les définir tous dans un même endroit.
3. Certaines fonctions doivent être définies d'une manière récursive, d'où la nécessité d'être explicitement nommées.

Les fonctions auxiliaires sont définies comme des fonctions mathématiques. Dans le cas où une fonction auxiliaire est très complexe, elle peut être définie en termes d'autres fonctions auxiliaires.

Dans l'exemple du compteur, la seule fonction définie est $f(T)$. Cette fonction retourne la longueur de la trace.

$$f(T) = \#T$$

2.5.3 Le cas d'erreur

Une erreur a lieu lorsque l'élément d'extension d'une trace canonique définit un comportement indésirable, ce qu'on appelle la trace illégale. Rappelons qu'une trace est dite légale si elle représente correctement l'état du module. Notons que les valeurs de sortie ne sont pas définies pour les traces illégales. Dans le cas où la trace est illégale, une indication d'erreur sera donnée.

Chapitre 3

L'exemple de la facturation

Dans ce chapitre, nous traitons le problème de la facturation de commandes sous différentes approches. Quatre approches, présentées dans le chapitre précédent, sont utilisées pour modéliser ce problème : relationnelle inductive, boîte noire par entités, B et assertions de traces. Pour chaque modélisation, nous présentons la technique de spécification et ensuite nous donnons des remarques tirées de nos modélisations.

3.1 Présentation du cas d'étude

Le cas d'étude choisi consiste à facturer des commandes. Ce cas d'étude a déjà été modélisé avec plusieurs méthodes formelles et semi-formelles dans le cadre du *"International Workshop on : Comparing Systems Specification Techniques"* à l'Institut de Recherche en Informatique de Nantes (IRIN) [26].

3.1.1 Les exigences

Dans la suite de ce document, nous nous basons sur les exigences suivantes, tirées de [26], pour la réalisation de nos spécifications.

1. *≪ Facturer consiste à changer l'état d'une commande (le faire passer de "en attente" à "facturée")*
2. *Sur une commande, il y a une et une seule référence à un produit commandé en une certaine quantité commandée.*
3. *La quantité peut être différente d'une commande à l'autre.*
4. *La même référence peut être commandée sur plusieurs commandes.*
5. *L'état d'une commande devient "facturée" si la quantité commandée est inférieure ou égale à la quantité en stock pour la référence du produit commandé.*
6. *Considérez les deux cas suivants :*
   - *(a)* 1^{er} *cas :*
     - *i. Toutes les références commandées sont des références en stock. Le stock et l'ensemble des commandes varient, du fait de l'entrée de nouvelles commandes ou de l'annulation de commandes (pour l'ensemble des commandes) et du fait de l'entrée de quantité de produits en stock.*
     - *ii. On n'a pas à prendre en compte ces entrées. Cela signifie qu'on ne reçoit pas deux flots d'entrée (commandes, entrées en stock). Le stock et l'ensemble des commandes nous sont toujours donnés dans un état à jour.*
   - *(b)* 2^{e} *cas : nous avons à prendre en compte les entrées :*
     - *i. des nouvelles commandes,*
     - *ii. des annulations de commandes,*
     - *iii. des quantités en stock ≫.*

3.1.2 Les hypothèses

En analysant l'énoncé présenté ci-dessus, nous posons des hypothèses pour rendre notre spécification plus complète. Généralement, ces hypothèses dépendent de deux facteurs : la technique utilisée et la connaissance de l'analyste. Comme notre objectif est l'évaluation des méthodes en question, nous prenons les hypothèses les plus simples pour spécifier formellement notre cas d'étude. D'autres hypothèses seront posées en fonction des questions soulevées selon la technique utilisée. Les hypothèses posées sont les suivantes :

1. chaque commande donne lieu à une seule facture ;
2. une commande est un ensemble de lignes de commande ;
3. chaque ligne de commande correspond à un produit commandé pour une quantité donnée ;
4. un produit ne peut apparaître plus d'une fois dans une commande ;
5. une commande vide (sans référence) ne peut être facturée ;
6. une commande ne peut être modifiée une fois facturée ;
7. la saisie d'une commande se fait ligne par ligne sans vérification de la quantité en stock ;
8. la vérification d'une commande se fait ligne par ligne durant la facturation ;
9. une commande est dite valide si toutes ses quantités commandées sont disponibles en stock ;
10. une ligne appartient à une et une seule commande.

3.2 La méthode inductive

Dans cette section, nous appliquons la méthode relationnelle inductive pour spécifier notre cas d'étude.

3.2.1 1^{er} Cas

Une spécification relationnelle inductive se présente comme une relation binaire entre les entrées et les sorties du système [23]. Intuitivement, la première question qui se pose est la suivante : quelles sont les entrées possibles du système et leurs sorties correspondantes?

La stratégie utilisée pour répondre à cette question est de sélectionner, à partir de cas réels du système, toutes les entrées possibles. Par exemple, pour qu'une commande soit facturée, on doit créer la commande, créer les lignes de commande (une ou plusieurs), la facturer, à condition que toutes les quantités commandées soient disponibles en stock.

En procédant de cette manière, nous proposons les entrées ci-dessous que nous présentons informellement afin d'aider le lecteur à comprendre la fonctionnalité de chacune d'elles.

1. ⟨**Créer_cde,** *cde*⟩ sert à créer une commande *cde*. Si la commande n'a pas été créée précédemment, le système retourne OK, sinon il retourne le message ERREUR.
2. ⟨**Ajouter_ligne,** *cde, l, p, q*⟩ sert à ajouter une ligne de commande *l*. Si le produit *p* n'a pas été ajouté précédemment dans la commande *cde*, le système retourne OK, sinon il retourne le message ERREUR. Le symbole *q* représente la quantité commandée du produit *p*.
3. ⟨**Facturer_cde,** *cde, Q*⟩ sert à facturer une commande *cde* à condition que toutes les quantités de produits commandés soient disponibles en stock et que la commande a été créée précédemment. Si la commande est valide, le système retourne OK, sinon, il retourne le message ERREUR. Notons que le symbole Q représente la quantité disponible en stock pour chaque produit *p*.
4. ⟨**État_cde,** *cde*⟩ sert à afficher l'état courant d'une commande *cde*. Il existe trois cas possibles :
   (a) Si la commande n'a pas été créée précédemment, le système retourne le message INEXISTANTE.

(b) Si la commande a été créée et non facturée, le système retourne le message EN_ATTENTE.

(c) Si la commande a été créée et facturée, le système retourne le message FACTURÉE.

L'espace d'entrée et de sortie

L'espace d'entrée du système est le suivant :

$$I \triangleq \{ \textbf{Créer\_cde} \} \times Id\_COMMANDE \cup$$
$$\{ \textbf{Ajouter\_ligne} \} \times Id\_COMMANDE \times Id\_LIGNE \times Id\_PRODUIT \times \mathbb{N} \cup$$
$$\{ \textbf{Facturer\_cde} \} \times Id\_COMMANDE \times QUANTITÉ\_STOCK \cup$$
$$\{ \textbf{État\_cde} \} \times Id\_COMMANDE$$

où *Id_COMMANDE, Id_LIGNEE* et *Id_PRODUIT* sont respectivement des ensembles élémentaires des numéros de commandes, des numéros de lignes et des numéros de produits du système. La quantité disponible en stock est représentée par la fonction *QUANTITÉ_STOCK*. Sa définition mathématique est :

$$QUANTITÉ\_STOCK = Id\_PRODUIT \longrightarrow \mathbb{N}$$

L'espace de sortie du système est le suivant :

$$O \triangleq \{ \text{INEXISTANTE, EN\_ATTENTE, FACTURÉE, OK, ERREUR} \}$$

La génération des axiomes

Lorsqu'on a défini les espaces d'entrée et de sortie, il nous reste à générer les trois classes d'axiomes : les axiomes de base, les axiomes de réduction et les axiomes de permutation. Ces axiomes doivent calculer les sorties correspondantes aux séquences d'entrées admissibles. Comme on ne peut pas démontrer la complétude de la spécification et pour

éviter les oublis, nous suggérons d'appliquer les étapes suivantes afin de générer ces axiomes :

1. déterminer tous les cas de base du système pour des séquences d'entrées :
   (a) valides,
   (b) invalides,
   (c) retournant des messages d'erreur ;
2. trouver les entrées qui n'ont aucun effet sur la valeur de sortie ;
3. déterminer les entrées qui sont dépendantes entre elles, autrement dit, celles dont la soumission au système changent la valeur de sortie selon leurs positions dans la séquence ;
4. s'assurer que toutes les sorties qui sont définies dans l'espace de sortie peuvent être calculées par les axiomes générés.

La génération des axiomes est itérative. Pour simplifier, nous commençons par la génération des axiomes de base, puis des axiomes de réduction et enfin des axiomes de permutation.

Les axiomes de base – La génération commence par l'inventaire des entrées issues de la lecture du texte. Ensuite, nous raisonnons sur ces entrées en analysant les questions suivantes : quelles sont leurs sorties correspondantes ; quels sont les liens entre ces entrées et quelles sont les combinaisons possibles entre ces entrées ?

Nous établissons alors une première version initiale et intuitive des axiomes. Pour notre cas d'étude, nous divisons les axiomes de base en trois classes : les axiomes qui calculent une commande facturée (cas valides), ceux qui calculent une commande refusée (cas invalides) et ceux qui produisent un message d'erreur (cas d'erreurs). Notons que, vu la complexité de spécifier le système de la facturation avec la présente méthode, notre modélisation est limitée aux cas valides du système. La spécification complète est donnée à l'annexe A.

En effet, nous avons défini les axiomes de base suivants :

- pour créer une commande :

  B1 - $\dfrac{}{\langle \textbf{Créer\_cde}, cde\rangle \lhd \mathrm{R} \rhd \mathrm{ok}}$

- et pour afficher l'état initial d'une commande :

  B2 - $\dfrac{}{\langle \textbf{État\_cde}, cde\rangle \lhd \mathrm{R} \rhd \text{inexistante}}$

Les axiomes de réduction – Une démarche systématique est une meilleure façon pour définir ce type d'axiomes afin d'éviter les oublis, en fixant une entrée et en la combinant avec les autres y compris l'entrée elle-même. La génération de ces axiomes met en évidence les relations entre entrées.

Prenons les deux séquences suivantes :

$\langle \textbf{Créer\_cde}, cde\rangle.\langle \textbf{Créer\_cde}, cde'\rangle.\langle \textbf{Ajouter\_ligne}, cde, l, p, q\rangle \lhd \mathrm{R} \rhd \textsc{ok}$

$\langle \textbf{Créer\_cde}, cde'\rangle.\langle \textbf{Ajouter\_ligne}, cde, l, p, q\rangle \lhd \mathrm{R} \rhd \textsc{erreur}$

Nous remarquons que l'entrée courante $\langle \textbf{Ajouter\_ligne}, cde, l, p, q\rangle$ a deux sorties différentes dépendamment de l'entrée $\langle \textbf{Créer\_cde}, cde\rangle$. Cet exemple montre que le problème de la facturation est de type dynamique, c'est-à-dire que la valeur de sortie ne dépend pas seulement de l'entrée courante, mais aussi de l'historique d'entrées. Par contre, prenons les deux séquences suivantes :

$\langle \textbf{Créer\_cde}, cde\rangle.\langle \textbf{État\_cde}, cde\rangle.\langle \textbf{Ajouter\_ligne}, cde, l, p, q\rangle \lhd \mathrm{R} \rhd \textsc{ok}$

$\langle \textbf{Créer\_cde}, cde\rangle.\langle \textbf{Ajouter\_ligne}, cde, l, p, q\rangle \lhd \mathrm{R} \rhd \textsc{ok}$

Il est clair que l'entrée $\langle \textbf{État\_cde}, cde\rangle$ n'a aucun effet sur la valeur de sortie de l'entrée courante $\langle \textbf{Ajouter\_ligne}, cde, l, p, q\rangle$. Nous remarquons qu'il y a des entrées qui ont un effet sur la valeur de sortie de l'entrée courante et d'autres qui n'en ont pas. Pour fixer les idées, prenons l'exigence 2 présentée dans la section 3.1.1 qui est la suivante : ≪ *Sur une commande, il y a une et une seule référence à un produit commandé en une certaine quantité commandée* ≫.

Afin d'exprimer cette contrainte, nous avons défini les trois axiomes suivants :

- Ajouter ligne de commande (une commande) :

$$\text{AR1} - \frac{x.\langle \textbf{Créer\_cde}, cde\rangle \lhd \mathrm{R} \rhd \mathrm{ok}}{x.\langle \textbf{Créer\_cde}, cde\rangle.\langle \textbf{Ajouter\_ligne}, cde, l, p, q\rangle \lhd \mathrm{R} \rhd \mathrm{ok}}$$

- Ajouter ligne de commande (plusieurs commandes) :

$$\text{AR2} - \frac{\begin{array}{c} cde \neq cde' \wedge \\ x.\langle \textbf{Créer\_cde}, cde'\rangle \lhd \mathrm{R} \rhd \mathrm{ok} \wedge \\ x.\langle \textbf{Ajouter\_ligne}, cde, l, p, q\rangle \lhd \mathrm{R} \rhd \mathrm{ok} \end{array}}{x.\langle \textbf{Ajouter\_ligne}, cde, l, p, q\rangle.\langle \textbf{Créer\_cde}, cde'\rangle \lhd \mathrm{R} \rhd \mathrm{ok}}$$

- Ajouter plusieurs lignes :

$$\text{AR3} - \frac{\begin{array}{c} p \neq p' \wedge l \neq l' \wedge \\ x.\langle \textbf{Ajouter\_ligne}, cde, l, p, q\rangle \lhd \mathrm{R} \rhd \mathrm{ok} \wedge \\ x.\langle \textbf{Ajouter\_ligne}, cde, l', p', q'\rangle \lhd \mathrm{R} \rhd \mathrm{ok} \end{array}}{\begin{array}{c} x.\langle \textbf{Ajouter\_ligne}, cde, l, p, q\rangle. \\ \langle \textbf{Ajouter\_ligne}, cde, l', p', q'\rangle \lhd \mathrm{R} \rhd \mathrm{ok} \end{array}}$$

où x est une séquence quelconque d'entrées.

Nous remarquons que pour traiter une contrainte nous avons du définir trois axiomes de réduction calculant par induction l'ajout d'une ligne de commande, ce qui n'est pas très pratique et qui rend la spécification longue et complexe.

Les axiomes de permutation – La dernière étape, en appliquant la méthode inductive, est de générer des axiomes qui servent à transformer, par permutation de symboles, des séquences d'entrée en d'autres séquences qui ont les mêmes comportements. L'analyste examine toutes les combinaisons possibles des entrées. Par exemple, les deux entrées ⟨**Créer_cde** , *cde*⟩ et ⟨**Créer_cde,** *cde'*⟩ sont indépendantes l'une par rapport à l'autre. Alors, la permutation de ces deux entrées n'a aucun effet sur le comportement de leur séquence, à condition que les deux entrées ne soient pas les dernières entrées de la séquence. L'axiome de permutation devient :

$$\text{AP1}: \frac{cde \neq cde' \wedge x' \neq \varepsilon \wedge}{x.\langle \textbf{Créer\_cde}, cde\rangle.\langle \textbf{Créer\_cde}, cde'\rangle.x' \lhd \mathrm{R} \rhd y}$$
$$x.\langle \textbf{Créer\_cde}, cde'\rangle.\langle \textbf{Créer\_cde}, cde\rangle.x' \lhd \mathrm{R} \rhd y$$

3.2.2 2^e Cas

Dans le deuxième cas, nous devons tenir compte de deux nouvelles entrées. Nous proposons de réutiliser la spécification du premier cas et de la compléter. Les deux nouvelles entrées à spécifier sont :

1. $\langle$**Annuler_ligne,** *cde, l*$\rangle$ sert à annuler une ligne de commande *l*. Si la ligne *l* a déjà été créée et non annulée, le système retourne OK , sinon, il retourne le message ERREUR ;
2. $\langle$**Annuler_cde,** *cde*$\rangle$ sert à annuler une commande *cde*. Si la commande n'a pas été créée précédemment, le système retourne le message ERREUR, sinon il retourne le message OK ;
3. $\langle$**Ajouter_produit,** *p*$\rangle$ sert à ajouter un produit *p* ;
4. $\langle$**Ajouter_stock,** *p, q*$\rangle$ sert à ajouter la quantité *q* à la quantité disponible en stock du produit *p*. Le système retourne toujours OK (en supposant que tous les produits ont déjà été ajoutés au catalogue) ;
5. $\langle$**Afficher_stock,** *p*$\rangle$ sert à afficher la quantité disponible en stock du produit *p*.

L'espace d'entrée du système devient :

$$\begin{aligned} I \triangleq\ & \{\ \textbf{Créer\_cde}\ \} \times Id\_COMMANDE \cup \\ & \{\ \textbf{Ajouter\_ligne}\ \} \times Id\_COMMANDE \times Id\_LIGNE \times \\ & \qquad\qquad Id\_PRODUIT \times \mathbb{N} \cup \\ & \{\ \textbf{Annuler\_ligne}\ \} \times Id\_COMMANDE \times Id\_LIGNE \cup \\ & \{\ \textbf{Annuler\_cde}\ \} \times Id\_COMMANDE \cup \\ & \{\ \textbf{Facturer\_cde}\ \} \times Id\_COMMANDE \cup \end{aligned}$$

$$\{\ \textbf{État\_cde}\ \} \times \mathit{Id\_COMMANDE} \cup$$
$$\{\ \textbf{Ajouter\_produit}\ \} \times \mathit{Id\_PRODUIT} \cup$$
$$\{\ \textbf{Ajouter\_stock}\ \} \times \mathit{Id\_PRODUIT} \times \mathbb{N} \cup$$
$$\{\ \textbf{Afficher\_stock}\ \} \times \mathit{Id\_PRODUIT}$$

L'espace de sortie du cas présent est défini comme suit :

$$O \triangleq \{\ \text{INEXISTANTE, EN\_ATTENTE, FACTURÉE, OK, ERREUR}\ \} \cup \mathbb{N}$$

Pour spécifier le comportement de système en entier, nous devons nous assurer que tous les axiomes du premier cas sont vérifiés. Dans le cas où un axiome n'est pas vérifié, nous devons le corriger ou l'éliminer si nécessaire. En ajoutant de nouvelles entrées à l'ancien espace, le nombre de combinaisons possibles entre les entrées sera plus grand que celui du premier cas. Par conséquent, la spécification devient plus longue et complexe. En effet, pour facturer une commande il faut que :

- la commande existe (il est possible qu'elle soit créée et annulée plusieurs fois) et non facturée ;
- la commande soit non vide, autrement dit, qu'elle contienne au moins une ligne de commande (une ligne peut être ajoutée et annulée plusieurs fois) ;
- la quantité commandée de chaque produit apparaissant dans la commande soit disponible en stock.

Pour traiter le cas de la facturation de commande, nous avons défini les axiomes ci-dessous. Notons que la spécification avec la méthode relationnelle inductive se trouve dans l'annexe A de ce document.

- pour facturer une commande :

$$\text{AR1} - \frac{\begin{array}{c} \mathit{cde} \neq \mathit{cde}' \wedge \\ x.\langle \textbf{Créer\_cde}, \mathit{cde}' \rangle \lhd \mathrm{R} \rhd \mathrm{ok} \wedge \\ x.\langle \textbf{Facturer\_cde}, \mathit{cde} \rangle \lhd \mathrm{R} \rhd \mathrm{ok} \end{array}}{x.\langle \textbf{Facturer\_cde}, \mathit{cde} \rangle.\langle \textbf{Créer\_cde}, \mathit{cde}' \rangle \lhd \mathrm{R} \rhd \mathrm{ok}}$$

- pour facturer une commande avec une ligne de commande :

$$\text{AR2} - \frac{\begin{array}{c} q \geq q' \wedge \\ x.\langle \textbf{Créer\_cde}, cde\rangle.\langle \textbf{Ajouter\_ligne}, cde, l, p, q'\rangle. \\ \langle \textbf{Ajouter\_stock}, p, q\rangle \lhd \mathrm{R} \rhd \mathrm{ok} \end{array}}{\begin{array}{c} x.\langle \textbf{Créer\_cde}, cde\rangle.\langle \textbf{Ajouter\_ligne}, cde, l, p, q'\rangle. \\ \langle \textbf{Ajouter\_stock}, p, q\rangle.\langle \textbf{Facturer\_cde}, cde\rangle \lhd \mathrm{R} \rhd \mathrm{ok} \end{array}}$$

- pour facturer une commande avec plusieurs lignes :

$$\text{AR3} - \frac{\begin{array}{c} q \geq q' \wedge \\ x.\langle \textbf{Facturer\_cde}, cde\rangle \lhd \mathrm{R} \rhd \mathrm{ok} \wedge \\ x.\langle \textbf{Ajouter\_ligne}, cde, l, p, q'\rangle.\langle \textbf{Ajouter\_stock}, p, q\rangle \lhd \mathrm{R} \rhd \mathrm{ok} \end{array}}{\begin{array}{c} x.\langle \textbf{Ajouter\_ligne}, cde, l, p, q'\rangle. \\ \langle \textbf{Ajouter\_stock}, p, q\rangle.\langle \textbf{Facturer\_cde}, cde\rangle \lhd \mathrm{R} \rhd \mathrm{ok} \end{array}}$$

De la même manière nous avons défini le reste des axiomes.

3.2.3 Remarques

La méthode inductive se concentre seulement sur ce que le système doit faire et non sur comment le faire. Le coeur de cette méthode est basé sur la notion d'induction. Cette notion rend la spécification plus ou moins concise dépendamment du type de problème à résoudre.

L'article [23] présente une spécification boîte noire d'un système téléphonique, en utilisant la méthode relationnelle inductive. D'après [23], il est clair que la méthode relationnelle inductive nous permet de donner une représentation compréhensible du système téléphonique et de raisonner facilement sur ce système. Par contre, dans l'exemple de la facturation, la notion inductive engendre à une explosion combinatoire entre les entrées, ce qui rend la spécification longue. Les contraintes sur les attributs des entrées du système de facturation sont plus nombreuses que celles du système téléphonique. Ceci entraîne une augmentation du nombre d'axiomes à générer.

3.3 La méthode boîte noire par entités

Nous traitons dans cette section le problème de la facturation de commandes sous la méthode boîte noire par entités [24]. Au cours de la modélisation, nous présentons la technique de spécification et enfin nous donnons quelques remarques.

3.3.1 Les étapes de la spécification

La modélisation obéit à deux approches différentes. L'une, plutôt intuitive, relève de la spécification et repose sur la vision que se fait l'analyste de son système d'information. L'autre répond à une démarche systématique permettant de faciliter la validation du modèle obtenu.

La manière la plus aisée de commencer la spécification consiste à travailler sur les quatre plans suivants :

1. **la définition des espaces d'entrée et de sortie :** cette étape est similaire à l'étape de la définition des espaces entrée et de sortie de la méthode relationnelle inductive (voir section 3.2.1) ;
2. **la définition des entités et la description de leurs comportements individuels :**
   - repérer les entités,
   - décrire le comportement individuel de chaque entité à l'aide des diagrammes de structure,
   - formaliser les diagrammes de structure des entités ;
3. **la définition des contraintes du système ;**
4. **la définition de la relation d'entrée-sortie.**

3.3.2 1^{er} cas

La définition des espaces d'entrée et de sortie

1. **Établir la liste d'entrées** – Cette étape détermine la liste des entrées du système. Dans le cas présent, les entrées attendues par le système sont les suivantes :
   - **Créer_cde** sert à créer une commande ;
   - **Ajouter_ligne** sert à ajouter une ligne de commande ;
   - **État_cde** sert à afficher l'état courant d'une commande ;
   - **Facturer_cde** sert à facturer une commande.
2. **Définir les attributs de chaque entrée** – Dans cette étape, nous attribuons à chacune des entrées définies dans l'étape précédente un ou plusieurs paramètres d'entrée que nous appelons attributs. Par exemple, supposons que nous voulons créer une commande. Le seul paramètre qui doit être fourni par l'utilisateur, afin d'achever l'exécution de cette entrée, est le numéro de commande. L'ensemble de tous les numéros de commandes est dénoté par *Id_COMMANDE*. La définition formelle de cette entrée est :

   $$\{ \textbf{Créer\_cde} \} \times \mathit{Id\_COMMANDE}$$

   De la même manière nous avons défini les attributs du reste des entrées du système. Le tableau 11 illustre la liste des entrées.

$I_1 \triangleq \{ \textbf{Créer\_cde} \} \times \mathit{Id\_COMMANDE}$
$I_2 \triangleq \{ \textbf{Ajouter\_ligne} \} \times \mathit{Id\_COMMANDE} \times \mathit{Id\_LIGNE} \times \mathit{Id\_PRODUIT} \times \mathit{QUANTITÉ\_CDÉE}$
$I_3 \triangleq \{ \textbf{Facturer\_cde} \} \times \mathit{Id\_COMMANDE}$
$I_4 \triangleq \{ \textbf{État\_cde} \} \times \mathit{Id\_COMMANDE}$

TAB. 11 – *La liste des entrées*

3. **Pour chaque entrée, recueillir toutes les sorties possibles** – On définit pour une entrée son ensemble de sorties possibles.

Par exemple, pour l'entrée **Créer_cde** il existe deux cas possibles :

- si le numéro de commande a déjà été créé, le système produit un message d'erreur. Comme dans cette méthode ne spécifie pas les cas d'erreurs, nous n'avons pas besoin de définir la sortie ERREUR ;
- si le numéro n'a pas été créé, le système produit la sortie OK.

Comme cette méthode ne spécifie que les cas valides, alors l'entrée **Créer_cde** a le singleton {OK} comme ensemble de sorties.

De la même manière nous avons recueilli le reste des sorties. Le tableau 12 illustre les espaces d'entrée et de sortie du système.

$I_1 \triangleq$ { **Créer_cde** } × *Id_COMMANDE*	$O_1 \triangleq$ { OK }
$I_2 \triangleq$ { **Ajouter_ligne**} × *Id_COMMANDE* × *Id_LIGNE* × *Id_PRODUIT* × *QUANTITÉ_CDÉE*	$O_2 \triangleq$ { OK }
$I_3 \triangleq$ { **Facturer_cde** } × *Id_COMMANDE*	$O_3 \triangleq$ { OK }
$I_4 \triangleq$ { **État_cde** } × *Id_COMMANDE*	$O_4 \triangleq$ { INEXISTANTE, EN_ATTENTE, FACTURÉE }

TAB. 12 – *Les espaces d'entrée et de sortie*

Les deux ensembles I et O sont définis par l'union des espaces comme suit :

$$I \triangleq \bigcup_{i=1}^{4} I_i \text{ et } O \triangleq \bigcup_{i=1}^{4} O_i$$

La définition des entités et la description de leurs comportements individuels

1. **Repérer les entités** – Les entités sont les objets de gestion essentiels du système d'information. Dans notre exemple que gère-t-on ? Dans le cas présent, nous gérons uniquement la facturation de commandes. Deux entités ont été identifiées : *COMMANDE* et *REQUÊTE*. L'entité *COMMANDE* est un ensemble dont chaque élément est une commande particulière. L'entité *REQUÊTE* n'a aucun identifiant ni propriétés. Elle sert à consulter les données du système. C'est sans doute cette première étape qui est la plus délicate. Ici, ce qui doit guider, ce sont les objectifs de gestion.

2. **Décrire le comportement individuel de chaque entité à l'aide des diagrammes de structure** – Après la définition des entités du système, nous devons décrire le comportement individuel de chacune de ces deux entités. Pour chaque entité, nous décrivons tous ses comportements possibles, en utilisant le diagramme de structure d'entité de la méthode JSD [32]. Toutes les instances possibles d'une entité seront représentées par un diagramme.

   Dans le cas présent, le comportement possible de l'entité *COMMANDE* peut être : créer une commande particulière (une seule fois), ajouter des lignes à cette commande (une ou plusieurs fois) et enfin facturer cette commande (une seule fois). En termes de séquences d'entrées, il pourrait être :

   ⟨Créer_cde , ...⟩.⟨Ajouter_ligne , ...⟩.⟨Facturer_cde , ...⟩

   ⟨Créer_cde , ...⟩.⟨Ajouter_ligne , ...⟩.⟨Ajouter_ligne, ...⟩.⟨Facturer_cde , ...⟩

   ⟨Créer_cde , ...⟩.⟨Ajouter_ligne , ...⟩.⟨Ajouter_ligne , ...⟩.⟨Ajouter_ligne, ...⟩.⟨Facturer_cde , ...⟩

   ...

   Rappelons que l'objectif principal de cette étape est d'introduire des contraintes sur l'ordonnancement des entrées. Pour cela, nous devons nous baser sur les contraintes décrites dans l'énoncé du problème pour spécifier le comportement individuel de chaque entité. Prenons par exemple la contrainte suivante : *"Une commande vide (sans référence) ne peut pas être facturée"*. Elle est exprimée par l'utilisation du symbole "+". Ce symbole signifie l'itération de l'entrée **Ajouter_ligne** au moins une fois. La figure 4 illustre les comportements possibles d'une commande particulière.

3. **Formaliser les diagrammes de structure des entités** – Cette étape consiste à transformer les diagrammes construits à l'étape précédente en termes d'expressions de processus.

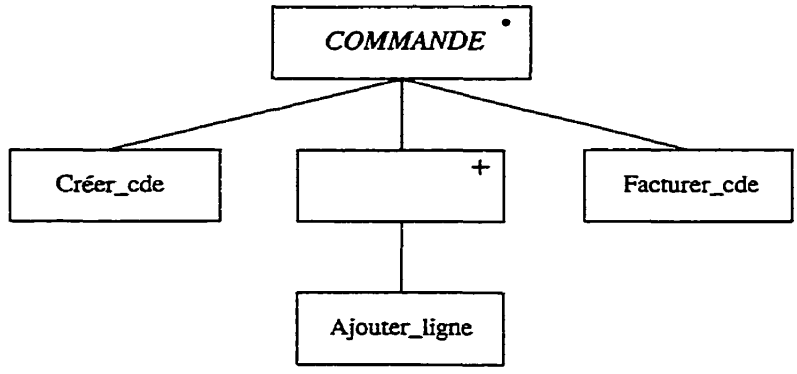


FIG. 4 – *Le diagramme de structure de l'entité COMMANDE*

Les définitions formelles du diagramme de l'entité *COMMANDE* et de l'entité *REQUÊTE* sont :

$$\textit{COMMANDE}\ (\textit{cde\_id}) \triangleq \langle \textbf{Créer\_cde}, \textit{cde\_id}\rangle.((\textbf{Ajouter\_ligne}, \textit{cde\_id}, \textit{id\_ligne}, \_, \_))^{+}.\langle \textbf{Facturer\_cde}, \textit{cde\_id}\rangle$$

$$\textit{REQUÊTE} \triangleq \langle \textbf{État\_cde}, \_\rangle^{*}$$

Notons que le diagramme de la figure 4 représente l'arbre syntaxique de la définition de l'entité *COMMANDE*.

La définition des contraintes

Dans l'étape précédente, nous avons étudié les entrées pouvant être soumises au système, en imposant des contraintes sur l'ordonnancement des entrées. Mais, certaines contraintes n'ont pas encore été traitées. Cette étape débute donc par le recensement des contraintes et des conditions d'activation du système.

Rappelons que deux prédicats ont été définis afin de faciliter la formalisation des contraintes :

$$\textit{instance\_de}\ (x, y, f) \text{ et } \textit{instance\_de\_clé}(x, y, f, k)$$

où x, y sont des séquences, f est une fonction et k est la clé (identifiant) d'une entité.

En prenant les hypothèses dérivées de la section 3.1.2, on a deux contraintes à formaliser :

1. *un produit ne peut apparaître plus d'une seule fois dans une commande ;*
2. *une commande est dite valide si toutes ses quantités commandées sont disponibles en stock.*

Les définitions formelles de ces contraintes sont respectivement les suivantes :

1. $Produit\_unique(x) \Leftrightarrow$

   $(\forall\ cde,\ p\_id\colon instance\_de\ (cde,\ x,\ prefix \circ COMMANDE\ )$

   $\Rightarrow$

   $\#\ (cde \downarrow \{\ \langle \textbf{Ajouter\_ligne}, \_, \_, p\_id, \_\rangle\ \}) \leq 1\ )$

2. $Quantité\_suffisante(x) \Leftrightarrow$

   $(\forall\ cde\_id,\ p\_id,\ q\colon instance\_de\_clé(cde,\ x,\ COMMANDE,\ cde\_id) \wedge$

   $\langle \textbf{Ajouter\_ligne}, cde\_id, \_, p\_id, q\rangle \preceq cde$

   $\Rightarrow$

   $Stock(x,\ p\_id) \geq q)$

où $Stock(x,\ p\_id) =$

let

$L = \{\ i \mid \exists\ cde\_id\colon x[i] = \langle \textbf{Ajouter\_ligne}, cde\_id, \_, p\_id, \_\rangle \wedge$
$instance\_de\_clé(\_,\ x,\ COMMANDE,\ cde\_id)\ \}$

in

return $Quantité\_stock(p\_id)$ - $\Sigma_{i \in L}\, QUANTITÉ\_CDÉE(x[i]))$

où $Quantité\_stock = Id\_PRODUIT \longrightarrow \mathbb{N}$ et $QUANTITÉ\_CDÉE$ est un attribut de l'entrée **Ajouter_ligne**.

La variable *cde* est une sous-séquence de la séquence x. Les identifiants d'une commande et d'un produit quelconques sont respectivement dénotés par *cde_id* et *p_id*. La fonction $Stock(x,\ p\_id)$ retourne la quantité disponible en stock du produit *p_id* pour une séquence d'entrée x. Notons que, dans le cas présent, nous ne gérons que les quantités

qui doivent être supprimées du stock par une facturation. Pour modéliser le stock, nous utilisons la fonction *Quantité_stock* qui donne, par hypothèse, la quantité en stock du produit *p_id* lors du démarrage du système.

La définition de la relation d'entrée-sortie

Cette dernière étape consiste à définir une relation entre les séquences d'entrées et les sorties, autrement dit, le comportement d'entrée-sortie du système. Cette relation est représentée par des axiomes qui seront appliqués aux séquences valides définies dans l'étape précédente.

Dans le cas présent, l'entrée **Créer_cde** peut seulement produire en sortie la valeur OK. Lorsque cette entrée sera soumise, le système produira la sortie OK qui indique la création d'une nouvelle commande.

Un seul axiome suffit pour spécifier la sortie d'une séquence dont le dernier symbole est associé à un singleton comme ensemble de sorties.

$x \vdash u \in \mathbf{Valide_E} \wedge u \in I_i \wedge O_i = \{ o \}$

$\Rightarrow$

$x \vdash u \lhd \mathrm{R} \rhd o$

Par exemple, avec cet axiome, on peut déduire la formule suivante :

$x \vdash u \in \mathbf{Valide_E}$

$\Rightarrow$

$x \vdash \langle \mathbf{Créer\_cde}, \_ \rangle \lhd \mathit{FACTURATION} \rhd \mathrm{OK}$

Néanmoins, si une entrée a plus d'une sortie possible, nous devons séparément examiner la relation existant entre cette entrée et chacune de ses sorties.

Considérons l'entrée **État_cde**. Si cette entrée est soumise au système, différentes sorties pourront être produites, dépendamment de l'historique d'entrées. Les sorties possibles de l'entrée **État_cde** sont les suivantes :

1. INEXISTANTE, dans le cas où la commande n'a été pas créée ;

2. EN_ATTENTE, dans le cas où la commande a déjà été créée ;
3. FACTURÉE, dans le cas où la commande a déjà été créée et facturée.

Les axiomes qui correspondent aux trois cas ci-dessus sont respectivement les suivants :

1. $\neg$ ($\exists$ *cde*: *instance_de_clé*(*cde*, *x*, *prefix* $\circ$ *COMMANDE*, *cde_id*))

   $\Rightarrow$

   $x \vdash \langle$**État_cde**, *cde_id*$\rangle \triangleleft$ *FACTURATION* $\triangleright$ INEXISTANTE
2. *instance_de_clé*(*cde*, *x*, *prefix* $\circ$ *COMMANDE*, *cde_id*) $\wedge$

   label(*last*(*cde_id*)) $\neq$ **Facturer_cde**

   $\Rightarrow$

   $x \vdash \langle$**État_cde**, *cde_id*$\rangle \triangleleft$ *FACTURATION* $\triangleright$ EN_ATTENTE
3. *instance_de_clé*(*cde*, *x*, *COMMANDE*, *cde_id*)

   $\Rightarrow$

   $x \vdash \langle$**État_cde**, *cde_id*$\rangle \triangleleft$ *FACTURATION* $\triangleright$ FACTURÉE

3.3.3 2^e Cas

Pour modéliser le 2^e cas, nous devons refaire les étapes utilisées pour le 1er cas et adapter la spécification.

1. **Ajout des entrées et des sorties** – Afin de pouvoir gérer le stock et les commandes, il est nécessaire de spécifier de nouvelles entrées et sorties. D'après l'énoncé du problème, cinq entrées doivent être ajoutées :
   (a) **Annuler_cde** sert à annuler une commande ;
   (b) **Annuler_ligne** sert à annuler une ligne de commande ;
   (c) **Ajouter_produit** sert à ajouter un nouveau produit ;
   (d) **Ajouter_stock** sert à ajouter une nouvelle quantité d'un produit à sa quantité en stock ;

(e) **Afficher_quantité_produit** sert à afficher la quantité disponible en stock d'un produit particulier.

De la même manière, nous définissons les attributs de ces nouvelles entrées ainsi que leurs sorties correspondantes. Le tableau 13 illustre le nouvel espace d'entrée-sortie du système.

$I_1 \triangleq$ { **Créer_cde** } × *Id_COMMANDE*	$O_1 \triangleq$ { OK }
$I_2 \triangleq$ { **Ajouter_ligne**} × *Id_COMMANDE* × *Id_LIGNE* × *Id_PRODUIT* × *QUANTITÉ_CDÉE*	$O_2 \triangleq$ { OK }
$I_3 \triangleq$ { **Facturer_cde** } × *Id_COMMANDE*	$O_3 \triangleq$ { OK }
$I_4 \triangleq$ { **État_cde** } × *Id_COMMANDE*	$O_4 \triangleq$ { INEXISTANTE, EN_ATTENTE, FACTURÉE }
$I_5 \triangleq$ { **Annuler_cde** } × *Id_COMMANDE*	$O_5 \triangleq$ { OK }
$I_6 \triangleq$ { **Annuler_ligne** } × *Id_COMMANDE* × *Id_LIGNE*	$O_6 \triangleq$ { OK }
$I_7 \triangleq$ { **Ajouter_produit** } × *Id_PRODUIT*	$O_7 \triangleq$ { OK }
$I_8 \triangleq$ { **Ajouter_stock** } × *Id_PRODUIT* × *QUANTITÉ*	$O_8 \triangleq$ { OK }
$I_9 \triangleq$ { **Afficher_quantité_produit** } × *Id_PRODUIT*	$O_9 \triangleq \mathbb{N}$

TAB. 13 – *Le nouvel espace d'entrée-sortie*

Les ensembles I et O sont définis par l'union des espaces d'entrée et de sortie comme suit :

$$I \triangleq \bigcup_{i=1}^{9} I_i \text{ et } O \triangleq \bigcup_{i=1}^{9} O_i$$

2. **Ajout des entités** – Deux nouvelles entités doivent être ajoutées : *PRODUIT* et *LIGNE*. Les figures 5 et 6 montrent respectivement le comportement de chaque entité.

   L'ajout de nouvelles entrées (par exemple, **Annuler_cde**) change le comportement de l'entité *COMMANDE* décrit dans le cas précédent. Ce changement entraîne une restructuration de son diagramme. La figure 7 illustre le nouveau comportement de l'entité *COMMANDE*. On remarque que l'entrée **Ajouter_ligne** est commune aux deux entités *COMMANDE* et *PRODUIT*. Le symbole $|||_{l\_id \in L}$ représente l'entrelacement entre les instances de l'entité *LIGNE*, où L est l'ensemble de toutes les clés de lignes d'une commande particulière. Les transformations des diagrammes

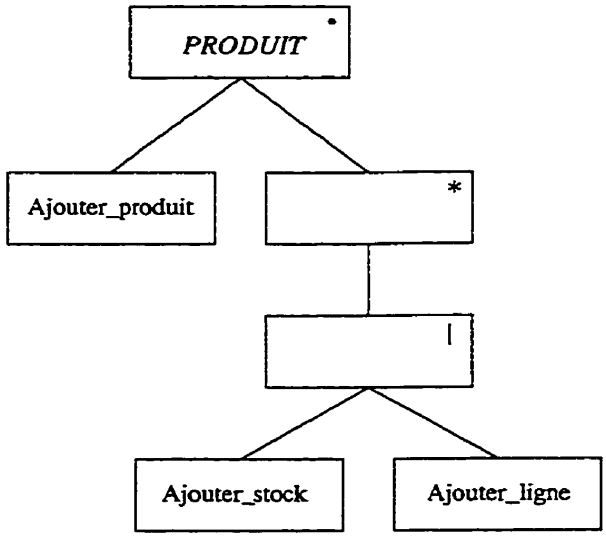


FIG. 5 – *Le diagramme de structure de l'entité PRODUIT*

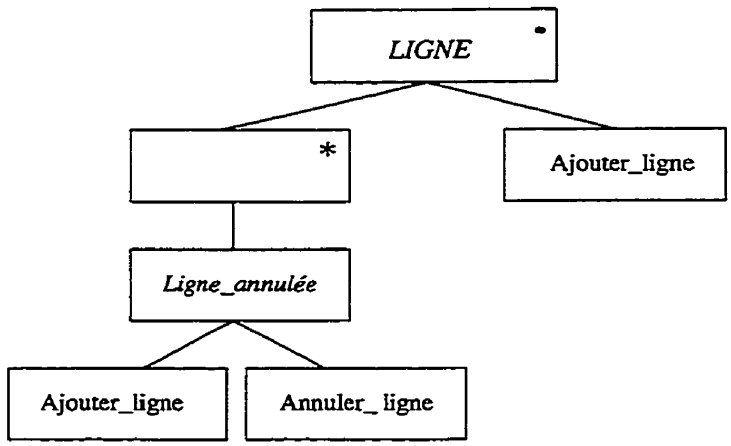


FIG. 6 – *Le diagramme de structure de l'entité LIGNE*

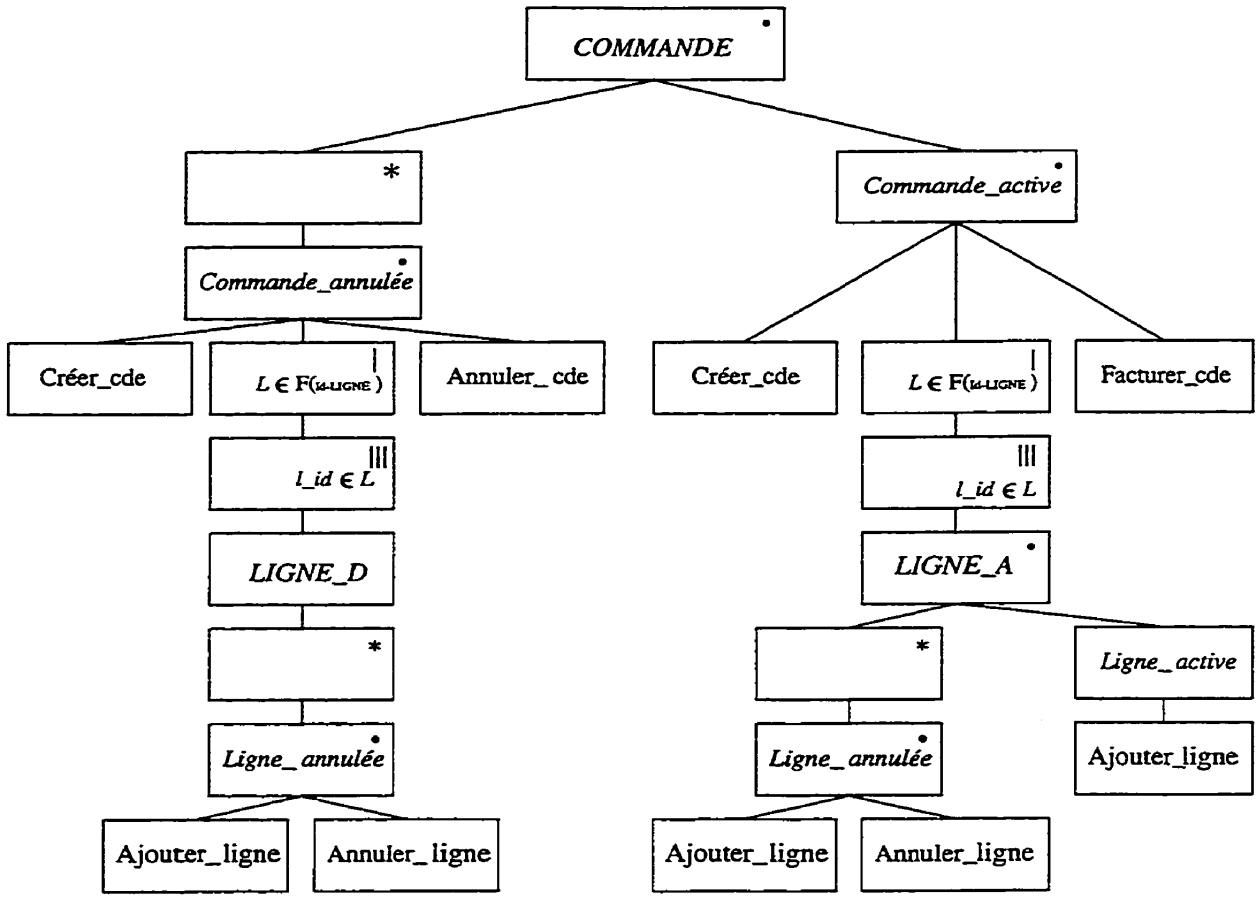


FIG. 7 – *Le diagramme de structure de l'entité COMMANDE*

des entités *COMMANDE, PRODUIT* et *LIGNE* en termes de séquences d'entrée sont les suivantes :

- $COMMANDE(cde\_id) = (Commande\_annulée(cde\_id))^* . Commande\_active(cde\_id)$
- $Commande\_active(cde\_id) = \langle$**Créer_cde**, $cde\_id\rangle.(|L \in F(\text{ID\_LIGNE})(|||_{l\_id \in L}\ (LIGNE\_A(cde\_id, l\_id)))).$ $\langle$**Facturer_cde**, $cde\_id\rangle$
- $Commande\_annulée(cde\_id) = \langle$**Créer_cde**, $cde\_id\rangle.(|L \in F(\text{ID\_LIGNE})(|||_{l\_id \in L}\ (LIGNE\_D(cde\_id, l\_id)))).$ $\langle$**Annuler_cde**, $cde\_id\rangle$
- $PRODUIT(p\_id) = \langle$**Ajouter_produit**, $p\_id\rangle.(\langle$ **Ajouter-stock**, p_id, _ $\rangle \mid \langle$**Ajouter_ligne**, _, _, p_id, _$\rangle)^*$
- $LIGNE(cde\_id, l\_id) = (Ligne\_annulée\ (cde\_id, l\_id))^*.\langle$**Ajouter_ligne**, cde_id, l_id, _, _$\rangle$
- $LIGNE\_A(cde\_id, l\_id) = (Ligne\_annulée\ (cde\_id, l\_id))^*.(Ligne\_active(cde\_id, l\_id))$
- $LIGNE\_D(cde\_id, l\_id) = (Ligne\_annulée\ (cde\_id, l\_id))^*$
- $Ligne\_active(cde\_id, l\_id) = \langle$**Ajouter_ligne**, cde_id, l_id, _, _$\rangle$
- $Ligne\_annulée(cde\_id, l\_id) = \langle$**Ajouter_ligne**, cde_id, l_id, _ _$\rangle.\langle$**Annuler_ligne**, $cde\_id, l\_id\rangle$
- $REQUÊTE = (\langle$**État_cde**, _$\rangle \mid \langle$**Afficher_quantité_produit**, _$\rangle)^*$

3. **Ajout et modification de contraintes** – Dans cette étape, nous devons vérifier si les contraintes définies dans le premier cas sont toujours valables et s'il y a de nouvelles contraintes à formaliser. L'ajout de l'entrée **Annuler_ligne** nous amène à modifier la contrainte $Produit\_unique(x)$. Une ligne peut être créée et annulée plusieurs fois avant de facturer une commande. Ceci signifie qu'un produit peut être commandé et annulé plusieurs fois. En conséquence, nous ne devons pas tenir compte des produits qui sont commandés par l'entrée **Ajouter_ligne** et annulés par l'entrée **Annuler_ligne**. Alors, la définition formelle de $Produit\_unique(x)$ devient :

$Produit\_unique\ (x) \Leftrightarrow$

$(\forall\ cde, p\_id\text{: } instance\_de\ (\ cde, x, prefix \circ COMMANDE\ )$

$\Rightarrow$

$(\#\ (cde \downarrow \{\ \langle$**Ajouter_ligne**, _, l_id, _, _$\rangle\ \}) - \#\ (cde \downarrow \{\ \langle$**Annuler_ligne**, _, $l\_id\rangle$ $\})) \leq 1)$

De la même manière nous avons modifié la contrainte $Quantité\_suffisante(x)$, puisqu'il suffit que les quantités en stock soient suffisantes pour les lignes qui sont commandées et non annulées. La définition formelle de cette contrainte devient :

Quantité_suffisante(x) $\Leftrightarrow$

($\forall$ *cde_id*, *p_id*, *q*, : *instance_de_clé* (*l*, *x*, *LIGNE*, (*cde_id*, *l_id*)) $\wedge$

last(l) = $\langle$**Ajouter_ligne**, *l_id*, _, *p_id*, *q*$\rangle$ $\Rightarrow$ *Stock* (*x*, *p_id*) $\geq$ *q*)

Nous remarquons que dans le cas présent il est possible d'ajouter de nouvelles quantités au stock par l'entrée **Ajouter_stock**. Dans ce cas, la fonction *Stock* (*x*, *p_id*) devient :

Stock(x, *p_id*) =

if $\exists\ p, a, L$:

instance_de_clé(*p*, *x*, *PRODUIT*, *p_id*)

$a = p \downarrow\downarrow$ { **Ajouter_stock** }

L = { $c \mid \exists\ l$: *instance_de_clé*(l, *x*, *prefix* $\circ$ *LIGNE*, *l_id*) $\wedge$

last(l) = $\langle$**Ajouter_ligne**, *cde_id*, *l_id*, *p_id*, _$\rangle$ $\wedge$

instance_de_clé(_, *x*, *COMMANDE*, *cde_id*)}

then

return ($\Sigma_{i=1}^{\#a}$ *QUANTITÉ*($a[i]$) - $\Sigma_{c \in L}$ *QUANTITÉ_CDÉE*(c),

où *QUANTITÉ* et *QUANTITÉ_CDÉE* sont respectivement les attributs des entrées **Ajouter_stock** et **Ajouter_ligne**.

4. **Ajout des axiomes** – L'ajout de nouvelles entrées au système nécessite évidemment la définition de nouveaux axiomes afin de calculer les sorties correspondantes. Comme dans le cas précédent, cette définition débute par l'inventaire de toutes les sorties possibles pour chaque entrée. Par exemple, l'entrée $\langle$**Ajouter_ligne**,*cde_id*, _, *p_id*, _$\rangle$ a comme seule sortie possible OK, puisqu'il est certain que le produit *p_id* ne peut apparaître au plus qu'une seule fois dans la commande *c_id* , en respectant la contrainte *Produit_unique*(x). La définition mathématique de cet axiome est :

$x \vdash \langle$**Ajouter_ligne**, *cde_id*, _, *p_id*, _$\rangle \triangleleft$ *FACTURATION* $\triangleright$ OK

De la même manière nous définissons le reste des axiomes.

Notons que la spécification entière de cet exemple se trouve à l'annexe B.

3.3.4 Remarques

La méthode boîte noire par entités couvre seulement la phase de spécification d'un logiciel. Elle est principalement fondée sur la notion de diagramme de structure d'entité de JSD, l'algèbre de processus et la logique des prédicats du premier ordre.

Le cas de la facturation est de type mémorisation de données et contrôle global centralisé. À notre avis, la méthode boîte noire par entités est plus adaptée à notre cas d'étude que la méthode relationnelle inductive. Le principe de spécification est le suivant : on extrait les instances d'une séquence pour définir le comportement du système et spécifier seulement les cas valides. Ceci implique moins d'effort de la part de l'analyste, car il n'a pas à spécifier les séquences invalides.

La validation d'une spécification basée sur les entités n'est pas triviale. Il serait utile d'avoir des outils pour exécuter une spécification. Dans notre cas d'étude, nous avons procédé par inspection pour valider notre spécification.

3.4 La méthode B

Le développement d'un composant en B [3] est réalisé de manière descendante, par une alternance de phases abstraites et de phases concrètes. On définit d'abord le modèle abstrait du composant par une machine abstraite, puis on enrichit et concrétise ce modèle en raffinant progressivement cette machine abstraite et, enfin, on définit le modèle exécutable par l'implantation. Dans la suite, notre spécification se limite seulement aux machines abstraites.

3.4.1 1^{er} Cas

Représentation formelle des données

Dans cette section, nous ne décrivons pas de constantes, car la spécification informelle du problème de facturation ne mentionne pas de données constantes. La représentation formelle de la facturation des commandes s'applique :

1. aux ensembles de base.
2. aux variables d'état (invariant).

Les ensembles de base

Il existe deux types d'ensembles de base : différés et énumérés.

Les ensembles différés :

1. Pour représenter la notion de "commande", nous définissons l'ensemble différé : *COMMANDE*. Cet ensemble est celui de toutes les commandes connues et inconnues.
2. Pour représenter la notion de "item", nous définissons l'ensemble différé : *LIGNE*. Cet ensemble est celui de toutes les lignes connues et non encore connues.
3. Pour représenter la notion de "produit", nous définissons l'ensemble différé : *PRODUIT*. Cet ensemble est celui de tous les produits connus et non encore connus.

Les ensembles énumérés :

1. Nous avons répertorié deux types de réponses : OK et ERREUR. Le système génère la réponse OK quand l'exécution d'une opération se termine avec succès et il génère la réponse ERREUR dans le cas inverse. Nous pouvons définir l'ensemble énuméré *RÉPONSE* qui contient ces deux types de réponses.

La définition mathématique de cet ensemble est :

$$RÉPONSE = \{ \text{OK, ERREUR} \}$$

2. Nous définissons un deuxième ensemble énuméré. Les éléments de cet ensemble représentent tous les états possibles d'une commande. Il existe trois états possibles : INEXISTANTE, EN ATTENTE et FACTURÉE. La définition mathématique de cet ensemble est :

$$ÉTAT = \{ \text{INEXISTANTE, EN\_ATTENTE, FACTURÉE} \}$$

Les variables d'état

L'ensemble de commandes connues – Nous devons définir une variable représentant toutes les commandes connues. Cette variable doit être incluse dans l'ensemble *COMMANDE* (ensemble de toutes les commandes connues et inconnues). Nous appelons cette variable *Commande*. La variable *Commande* est un sous-ensemble de *COMMANDE*. Sa définition mathématique est :

$$Commande \subseteq COMMANDE$$

L'ensemble de lignes connues – Nous devons définir une variable représentant toutes les lignes connues. Cette variable doit être incluse dans l'ensemble de toutes les lignes (connues et inconnues) *LIGNE*. Cette variable est appelée *Ligne*. La variable *Ligne* est un sous-ensemble de *LIGNE*. Sa définition mathématique est :

$$Ligne \subseteq LIGNE$$

L'ensemble de produits connus – Nous devons définir une variable représentant tous les produits connus. Cette variable doit être incluse dans l'ensemble de tous les produits (connus et inconnus) *PRODUIT*. Cette variable est appelée *Produit*. La variable *Produit* est un sous-ensemble de *PRODUIT*. Sa définition mathématique est :

$$Produit \subseteq PRODUIT$$

La répartition des lignes dans les commandes – Nous définissons la variable *Commande_de* qui associe un numéro de commande pour un item donné. Cette variable est définie comme une fonction totale entre les deux ensembles de base *LIGNE* et *COMMANDE*. La définition mathématique de cette fonction est :

$$Commande\_de \in Ligne \longrightarrow Commande$$

La figure 2 illustre graphiquement la fonction *Commande_de* qui existe entre les deux ensembles *Ligne* et *Commande*.

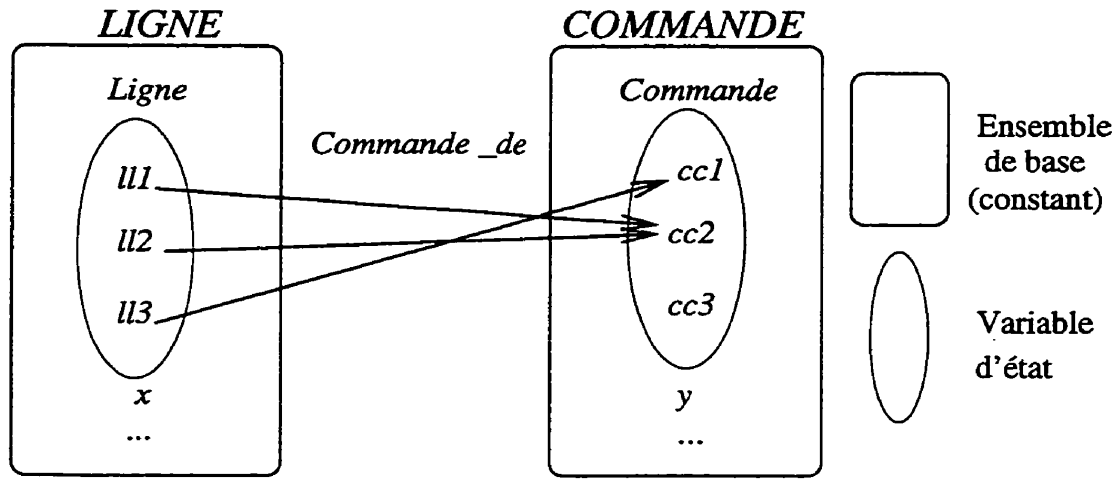


FIG. 8 – *Répartition des lignes aux commandes*

De la même manière, nous pouvons définir les variables suivantes : *Produit_réf, Quantité_commandée, État* et *Quantité_en_stock*. Les définitions mathématiques de ces fonctions sont respectivement les suivantes :

- $Produit\_réf \in Ligne \longrightarrow Produit$;
- $Quantité\_commandée \in Ligne \longrightarrow \mathbb{N}$;
- $État \in Commande \longrightarrow ÉTAT$;
- $Quantité\_en\_stock \in Produit \longrightarrow \mathbb{N}$.

Les lignes des commandes – Soit la contrainte suivante du cas d'étude.

≪ *Sur une commande, on a une et une seule référence à un produit commandé en une*

certain-e quantité commandée ≫ . Nous pouvons formaliser cette contrainte par le produit direct des deux fonctions *Commande_de* et *Produit_réf* comme suit :

$$Commande\_de \otimes Produit\_réf = \{$$
$$ll, (pp, cc) \mid ll, (pp, cc) \in Ligne \times (Produit \times Commande) \wedge$$
$$(ll, cc) \in Commande\_de \wedge (ll, pp) \in Produit\_réf \}$$

Le produit direct de deux relations *Commande_de* et *Produit_réf* doit être une fonction injective totale de la variable *Ligne* vers le produit cartésien des variables *Commande*, et *Produit*. La définition mathématique de cette propriété est la suivante :

$$Commande\_de \otimes Produit\_réf \in (Ligne \rightarrowtail (Produit \times Commande))$$

La figure 9 donne une représentation graphique de ce produit.

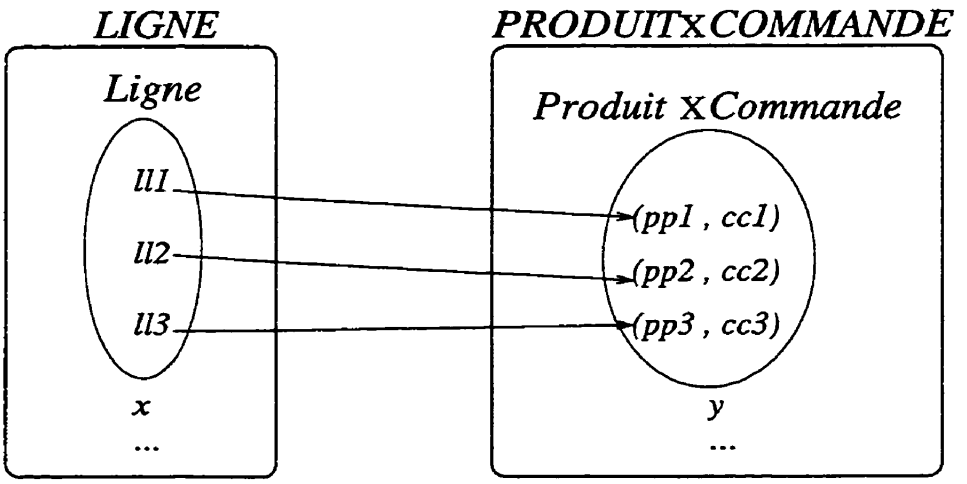


FIG. 9 – *Le produit direct de deux relations Commande_de* et *Produit_réf*

La spécification formelle

Dans cette section, nous définissons deux machines abstraites pour les variables définies dans les étapes précédentes.

La machine "Produit1" – Nous commençons par la définition d'une machine comprenant les deux variables suivantes : *Produit* et *Quantité_en_stock*.

Rappelons que dans le cas présent nous ne gérons pas le stock. C'est pour cela que nous ne tenons compte que de ces deux variables. La machine comprend une seule opération qui sert à mettre à jour le stock.

1. **Les variables et leur initialisation** – Au départ, nous considérons qu'il n'y a aucun produit en stock. Pour cela, nous initialisons les deux variables *Produit* et *Quantité_en_ stock* à l'ensemble vide. Le tableau 14 montre les variables et leur initialisation.

MACHINE
Produit1
SETS
PRODUIT
VARIABLES
Produit, Quantité_en_stock
INVARIANT
$Produit \subseteq PRODUIT \wedge$
$Quantité\_en\_stock \in \text{Produit} \longrightarrow \mathbb{N}$
INITIALISATION
$Produit := \emptyset \parallel$
$Quantité\_en\_stock := \emptyset$

TAB. 14 – *La machine* **Produit1**

2. **Les opérations de changement d'état** – La définition d'une opération se limite à dire que ce programme modifie les variables de la machine, mais de telle manière que les propriétés invariantes soient préservées.

   L'opération **Supprimer_stock** (voir le tableau 15) sert à mettre à jour la quantité en stock lors de la facturation d'une commande. Elle comporte un seul paramètre d'entrée, *stock*. Ce paramètre est défini comme une fonction partielle entre l'ensemble de tous les produits qui sont en stock et l'ensemble des entiers naturels $\mathbb{N}$ (ou NAT en B), tel que l'indique la première partie de la conjonction de la précondition PRE. Cette fonction représente les quantités qui doivent être soustraites

du stock. À cette fin, il faut que la quantité en stock soit plus grande ou égale à la quantité commandée pour chaque produit. Cette contrainte est exprimée dans la suite de la précondition de l'opération.

L'opérateur $\lhd\!\!-$ est le symbole de surcharge des relations. Il permet de modifier l'image d'un élément dans une relation. La figure 10 représente l'effet de l'opération **Supprimer_stock**. Sur cette figure, la flèche en pointillés indique l'image du produit *pp1* avant l'exécution de la substitution.

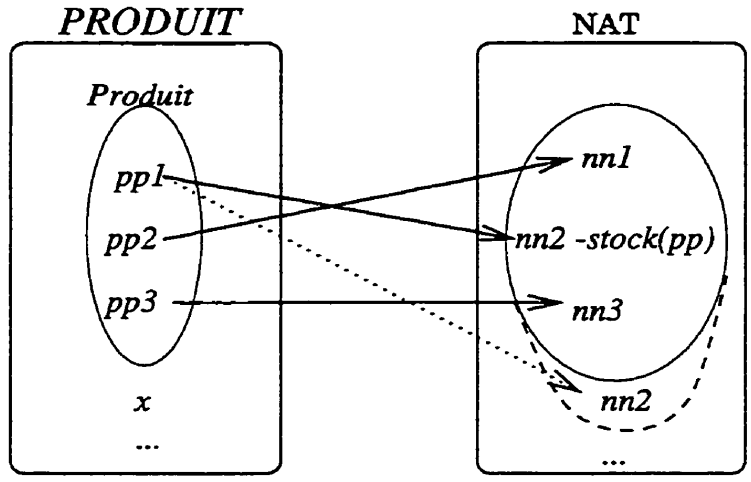


FIG. 10 – *La surcharge des relations*

La machine "Facturation1" – Cette machine est définie à partir des données suivantes : *item, commande, Commande_de, Produit_réf, Quantité_commandée, État.* Cette machine sert surtout à la facturation de commandes.

1. **Les variables et leur initialisation** – Au départ, nous considérons que toutes les variables sont initialisées à l'ensemble vide. Le tableau 16 illustre toutes ces variables et leur initialisation.
2. **Les opérations de changement d'état** – L'opération **Facturer_cde** comporte un paramètre de sortie indiquant si la commande a été facturée et un paramètre

OPERATIONS
Supprimer_stock (*stock*) =
PRE
stock ∈ *Produit* ⇸ NAT ∧
∀ *pp*. (*pp* ∈ **dom**(*stock*) ⇒
Quantité_en_stock(*pp*) ≥ *stock*(*pp*))
THEN
Quantité_en_stock:= *Quantité_en_stock* ⩤
(λ *pp*. (*pp* ∈ **dom**(*stock*) |
Quantité_en_stock(*pp*) - *stock*(*pp*)))
END ;

TAB. 15 – *L'opération* **Supprimer_stock**

d'entrée, le numéro de commande. Cette opération permet de facturer une commande si :

(a) elle est en attente ;

(b) elle contient au moins un item ;

(c) pour chaque item de la commande, la quantité disponible en stock est plus grande ou égale à la quantité commandée.

Le tableau 17 illustre la spécification formelle de l'opération **Facturer_cde**. Dans cette opération, nous utilisons la surcharge fonctionnelle *État*(*cc*) := FACTURÉE afin de changer la valeur de la variable *État*. Dans la figure 11, nous présentons une commande qui, associée à l'état EN_ATTENTE, devient associée à l'état FACTURÉE (la flèche en pointillés indique l'image de *commande* avant la surcharge fonctionnelle). De plus, cette opération fait l'appel de l'opération **Supprimer_stock** afin de mettre à jour le stock après chaque facturation d'une commande.

Promotion d'opérations

L'opération **Facturer_cde** consiste à mettre à jour la quantité en stock de chaque produit apparaissant dans la commande à facturer et à changer l'état d'une commande.

MACHINE
Facturation1

SETS

$COMMANDE$;
$LIGNE$;
$\acute{E}TAT = \{$ INEXISTANTE, EN_ATTENTE, FACTURÉE $\}$;
$R\acute{E}PONSE = \{$ OK, ERREUR $\}$;

INCLUDES

Produit1

VARIABLES

$Ligne$, $Produit\_réf$, $Quantité\_commandée$,
$Commande$, $Commande\_de$, $État$

INVARIANT

$Ligne \subseteq LIGNE \wedge$
$Commande \subseteq COMMANDE \wedge$
$Commande\_de \in Ligne \longrightarrow Commande \wedge$
$Produit\_réf \in Ligne \longrightarrow Produit \wedge$
$Quantité\_commandée \in Ligne \longrightarrow$ NAT $\wedge$
$État \in Commande \longrightarrow \acute{E}TAT \wedge$
$(Commande\_de \otimes Produit\_réf) \in (Ligne \rightarrowtail (Commande \times Produit))$

INITIALISATION

$Ligne := \emptyset \parallel$
$Commande := \emptyset \parallel$
$Commande\_de := \emptyset \parallel$
$Produit\_réf := \emptyset \parallel$
$Quantité\_commandée := \emptyset \parallel$
$État := \emptyset$

TAB. 16 – *La machine* **Facturation1**

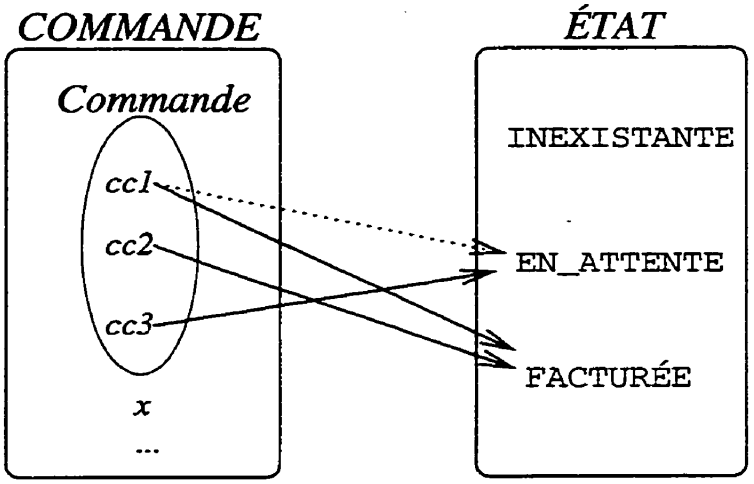


FIG. 11 – *La surcharge fonctionnelle*

OPERATIONS
réponse ⟵ **Facturer_cde**(*cc*) =
PRE
  $cc \in COMMANDE$
THEN
  IF
    $État(cc) = \text{EN\_ATTENTE} \wedge$
    $\mathbf{card}(Commande\_de^{-1}[\{\ cc\ \}]) > 0 \wedge$
    $\forall\ ll.\ (\ ll \in Ligne \wedge Commande\_de(ll)) = cc \Rightarrow$
      $Quantité\_en\_stock(produit\_réf(ll)) \geq Quantité\_commandée(ll)$
  THEN
    Supprimer_stock$((produit\_réf \otimes Quantité\_commandée)$
      $[Commande\_de^{-1}[\{cc\}]]) \parallel$
    $État(cc) := \text{FACTURÉE} \parallel$
    réponse := OK
  ELSE
    réponse := ERREUR
  END
END ;

TAB. 17 – *L'opération* **Facturer_cde**

Pour réaliser les traitements sur la quantité en stock de chaque produit (suppression de quantité commandée), il est nécessaire que la machine **Facturation1** puisse utiliser l'opération **Supprimer_stock** de la machine **Produit1**. Pour utiliser les opérations ou les variables de la machine **Produit1**, la machine **Facturation1** doit inclure la machine **Produit1**. Ceci peut être fait par l'ajout de la clause INCLUDES (voir le tableau 16).

3.4.2 2^e Cas

Une extension de la spécification du 1^{er} Cas

Dans la spécification précédente, nous avons défini les opérations qui servent seulement à facturer des commandes. Dans le 2^e cas, nous ajoutons les opérations qui gèrent le stock et les commandes. Dans cette extension de la spécification, nous définissons les deux machines suivantes : **Produit2** et **Facturation2**.

Nous proposons d'étendre la spécification précédente, en nous intéressant uniquement aux opérations suivantes :

- **Ajouter_stock** sert à ajouter une certaine quantité au stock ;
- **Créer_cde** sert à créer une nouvelle commande ;
- **Ajouter_ligne** sert à ajouter une ligne de commande ;
- **Annuler_cde** sert à annuler une commande ;
- **Annuler_ligne** sert à annuler une ligne de commande.

La machine "Produit2" – Dans cette extension de la spécification, nous définissons une nouvelle machine que nous appelons **Produit2**. Cette nouvelle machine est destinée à gérer le stock. Nous y déclarons une nouvelle opération : **Ajouter_stock.** Cette opération comporte un paramètre d'entrée : *stock.* Ce paramètre est une fonction partielle de *Produit* vers l'ensemble des nombres naturels NAT. Le tableau 18 illustre la spécification de cette opération.

La machine "Facturation2" – Cette nouvelle machine est destinée à gérer les

OPERATIONS
Ajouter_stock(*stock*) =
PRE
stock ∈ *Produit* ⇸ NAT ∧
(∀ *pp*. (*pp* ∈ **dom**(*stock*) ⇒
Quantité_en_stock + *stock*(*pp*) ≤ MAXINT))
THEN
Quantité_en_stock := *Quantité_en_stock* ⩤
(λ *pp*.(*pp* ∈ **dom**(*stock*) |
Quantité_en_stock(*pp*) + *stock*(*pp*)))
END
END

TAB. 18 – *L'opération* **Ajouter_stock**

commandes. Nous y déclarons quatre nouvelles opérations : **Créer_cde, Ajouter_ligne, Annuler_cde** et **Annuler_ligne**. Le tableau 19 illustre la première partie de la spécification de **Facturation2**. Dans ce tableau, nous spécifions la définition *lignes*(*cc*) dans la clause DEFINITIONS. Cette définition nous permet de décrire une formule par un nom et d'utiliser ce nom dans l'invariant ou dans les opérations de la machine. Dans le tableau 21, l'opération **Annuler_cde** permet d'illustrer l'utilisation de la clause DEFINITIONS.

Le tableau 20 illustre la spécification des deux opérations suivantes : **Créer_cde** et **Ajouter_ligne**. L'opération **Créer_cde** est destinée à créer une nouvelle commande. Elle est munie d'une précondition : *Commande* ≠ *COMMANDE*. La commande *cc* ne doit pas appartenir à l'ensemble de commandes créées *Commande*. On utilise la substitution indéterministe à choix non borné, ANY, pour choisir une valeur de *cc* satisfaisant cette contrainte ; on ajoute ensuite la commande *cc* au sous-ensemble *Commande* avec l'énoncé *Commande* := *Commande* ∪ {*cc*}, et on affecte à l'état de la commande la valeur EN_ATTENTE en utilisant la surcharge fonctionnelle *État(cc)* := EN_ATTENTE.

L'opération **Ajouter_ligne** ajoute un nouvel item à une commande donnée. Cette

MACHINE
Facturation2
SETS
COMMANDE;
LIGNE;
ÉTAT = { INEXISTANTE, EN_ATTENTE, FACTURÉE };
DEFINITIONS
$lignes(cc) = Commande\_de^{-1}\ [\{cc\}]$
INCLUDES
Produit2
VARIABLES
*les mêmes variables que la machine **Facturation1***
INVARIANT
*les mêmes invariants que la machine **Facturation1***
INITIALISATION
*la même initialisation que la machine **Facturation1***

TAB. 19 – *La machine* **Facturation2**

opération comporte un paramètre de sortie qui indique si l'ajout a été fait et trois paramètres d'entrée. La machine **Facturation2** a deux façons de répondre après l'exécution de cette opération :

1. accepter l'ajout du nouvel item, dans ce cas la valeur de *réponse* est OK ;
2. refuser l'ajout du nouvel item, dans ce cas la valeur de *réponse* est ERREUR.

Le tableau 21 illustre la spécification des deux opérations **Annuler_cde** et **Annuler_ligne**.

L'opération **Annuler_cde** sert à annuler une commande. Elle comporte aussi un paramètre de sortie, *réponse*, qui indique le succès de l'annulation de la commande *cc*. Elle est munie d'une précondition $cc \in COMMANDE$, indiquant que *cc* doit appartenir à l'ensemble de toutes les commandes connues et inconnues *COMMANDE*.

L'opération **Annuler_ligne** est très similaire à l'opération précédente. Elle comporte un paramètre de sortie qui indique le succès ou l'échec. Cette opération élimine l'item

ll et toutes ses données relatives dans les fonctions *Commande_de*, *Produit_réf* et *Quantité_Commandée*.

Notons que la spécification entière de cet exemple se trouve à l'annexe C.

3.4.3 Les obligations de preuve

Une obligation de preuve est un théorème à démontrer. En fait, la démonstration des théorèmes n'est pas obligatoire, mais elle est souhaitable pour s'assurer que ce qui est écrit est parfaitement correcte. Il existe deux modes pour démontrer les obligations de preuve.

1. La preuve en mode automatique s'acquitte de la plupart des obligations de preuve sans l'intervention de l'utilisateur. Pour offrir un compromis entre sa rapidité et son taux de preuve, le prouveur automatique peut être paramétré par un niveau de puissance. Typiquement, le niveau de 0 s'acquitte rapidement de 90 % des obligations de preuve démontrables automatiquement par l'outil ; les autres niveaux s'acquittent plus lentement d'une partie des 10 % restants.
2. La preuve en mode interactif est utilisée lorsque le mode automatique a échoué. L'utilisateur guide le prouveur interactif dans sa démonstration d'une obligation de preuve à l'aide de commandes (ajout d'hypothèses, preuve par cas, etc) et en ajoutant de nouvelles règles.

Une des obligations consiste à prouver qu'après l'exécution de la substitution, l'invariant est toujours satisfait. Cette obligation de preuve s'exprime par la formule suivante :

$$\text{PRE} \wedge \text{INVARIANT} \Rightarrow [\text{Substitution}]\ \text{INVARIANT}$$

La formule "[Substitution] INVARIANT" dénote la précondition la plus faible telle que, après l'exécution de substitution, la formule INVARIANT est vraie.

L'obligation de preuve indique que si on démarre la substitution dans un état où PRE et INVARIANT sont satisfaits, alors la substitution termine dans un état ou l'invariant est

```
OPERATIONS
Créer_cde=
PRE
      Commande ≠ COMMANDE
THEN
      ANY cc WHERE
         cc ∈ COMMANDE - Commande
      THEN
         Commande := Commande ∪ {cc } ||
         État(cc) := EN_ATTENTE
      END
END ;
réponse ⟵ Ajouter_ligne (cc, pp, qq ) =
PRE
      qq ∈ NAT ∧
      cc ∈ COMMANDE ∧
      pp ∈ PRODUIT
THEN
      IF
         (cc, pp ) ∉ ran(Commande_de ⊗ Produit_réf) ∧
         Ligne ≠ LIGNE ∧
         État(cc ) = EN_ATTENTE
      THEN
         ANY ll WHERE
            ll ∈ LIGNE - Ligne
         THEN
            Produit_réf (ll ) := pp ||
            Commande_de (ll ) := cc ||
            Ligne := Ligne ∪ {ll } ||
            Quatité_commandée (ll ) := qq ||
            réponse := OK
         END
      ELSE
         réponse := ERREUR
      END
END ;
```

TAB. 20 – *Les opérations* **Créer_cde** et **Ajouter_ligne**

```
OPERATIONS
réponse ⟵ Annuler_cde(cc) =
PRE
      cc ∈ COMMANDE
THEN
      IF
         État(cc) = EN_ATTENTE
      THEN
         Commande := Commande - {cc } ||
         État := {cc} ⩤ État ||
         Ligne := Ligne - lignes(cc) ||
         Commande_de :=lignes(cc) ⩤ Commande_de ||
         Produit_réf := lignes(cc) ⩤ Produit_réf ||
         Quatité_commandée := lignes(cc) ⩤ Quatité_commandée ||
         réponse := OK
      ELSE
         réponse := ERREUR
      END
réponse ⟵ Annuler_ligne(ll) =
PRE
      ll ∈ LIGNE
THEN
      IF
         État (Commande_de(cc)) = EN_ATTENTE
      THEN
         Ligne := Ligne - {ll } ||
         Commande_de := {ll } ⩤ Commande_de ||
         Produit_réf := {ll } ⩤Produit_réf ||
         Quantité_commandée := {ll } ⩤ Quantité_commandée ||
         réponse := OK
      ELSE
         réponse := ERREUR
      END
END ;
END ;
```

TAB. 21 – *Les opérations* **Annuler_cde** et **Annuler_ligne**

satisfait. En appliquant cette formule à l'opération **Créer_cde**, nous obtenons l'obligation du tableau 22. Cette obligation peut-être décrite de la manière suivante : la propriété invariante doit être vraie peu importe la nouvelle commande *cc* ajoutée à *Commande*.

$$
\begin{array}{l}
Commande \subseteq COMMANDE \wedge \\
Commande \neq COMMANDE \wedge \\
cc \in COMMANDE \wedge \\
cc \notin Commande \\
\Rightarrow \\
Commande \cup \{\ cc\ \} \subseteq COMMANDE
\end{array}
$$

TAB. 22 – *Obligation de preuve*

Nous utilisons Atelier-B pour générer toutes les obligations de preuve et pour faire leur démonstration. Le tableau 23 donne les statistiques des obligations de preuve par machine. La colonne "Preuve automatique" montre le nombre d'obligations de preuve démontrées automatiquement. La colonne "Preuve interactive" montre le nombre d'obligations de preuve que le prouveur automatique n'a pu démontrer et qui ont été démontrées de manière interactive. La colonne "Obligation non prouvée" montre le nombre d'obligations de preuve que nous n'avons pas pu démontrer. Enfin, la colonne "% Prouvée" montre le pourcentage des obligations démontrées par la machine. Au cours de la démonstration des théorèmes, une faute a été détectée dans la spécification.

Nom de machine	Obligations de preuve	Preuve automatique	Preuve interactive	Obligation non prouvée	% Prouvée
Produit1	3	3	0	0	100
Facturation1	9	9	0	0	100
Produit2	7	3	2	2	71
Facturation2	48	40	6	2	95
Total	67	55	8	4	94

TAB. 23 – *Statistique sur les obligations de preuve*

3.4.4 Remarques

Nous avons donné des éléments de la spécification. Une spécification complète contient des théorèmes et des preuves. Différents types de théorèmes sont à démontrer : l'existence d'un état initial, la validité de la précondition et les propriétés générales du système (par exemple les invariants). La validation de la spécification se fait au travers des preuves de propriétés générales du système.

La méthode B couvre la spécification et la conception des logiciels avec une notation homogène, et permet la traduction automatique en code exécutable, si on a suffisamment raffiné la spécification. La méthode B est fondée sur le principe d'obligations de preuve. La méthode B permet de construire des logiciels intégralement prouvés, c'est-à-dire conforme à leur spécification.

La conception consiste à raffiner une spécification jusqu'à l'obtention d'un programme B exécutable. L'outil Atelier-B [19] peut traduire le code B exécutable dans un langage de programmation classique comme C++.

Contrairement aux méthodes relationnelle inductive et boîte noire par entités, la méthode B oblige à réfléchir sur une construction modulaire du système ainsi que sur les variables d'état, les types et les opérations.

L'apprentissage du prouveur de théorème d'Atelier-B [19] nécessite un effort significatif. L'utilisateur doit être familier avec la logique des prédicats et la théorie des ensembles.

3.5 La méthode des assertions de traces

La méthode des assertions de traces [45] repose sur la notion de type abstrait et la modularité. Elle nous permet de spécifier formellement l'interface du module représentant le comportement observable du module. Dans la suite, nous traitons le problème de la facturation avec cette méthode en appliquant les étapes suivantes.

3.5.1 Les étapes de la spécification

1. **La syntaxe**
   - décomposer le système en modules et énumérer les programmes d'accès de chaque module ;
   - pour chaque programme, définir les types des arguments et des valeurs de sortie ;
   - définir la syntaxe pour chaque programme d'accès.
2. **Les traces canoniques**
   - identifier les programmes d'accès qui sont de type constructeur de données (ICT) ;
   - pour chaque module, définir les traces canoniques par les programmes identifiés.
3. **Les équivalences de traces**
   - énumérer tous les patrons possibles de traces canoniques ;
   - formaliser toutes les conditions et les contraintes du système ;
   - définir les équivalences de traces pour toutes les traces canoniques étendues par chacun des programmes d'accès.
4. **Les valeurs de sortie**
   - lister les programmes d'accès ayant des sorties visibles ;
   - déterminer et formaliser toutes les conditions et les contraintes du système relatives à ces programmes ;
   - définir les sorties possibles pour toutes les traces étendues par chacun de ces programmes.
5. **Le dictionnaire**
   - définir toutes les fonctions auxiliaires utilisées dans les étapes précédentes.

3.5.2 1^{er} Cas

La syntaxe

Décomposer le système en modules et lister leurs programmes d'accès - Dans cette étape il s'agit de décomposer le système en modules et déterminer les programmes d'accès de chaque module.

En se basant sur l'énoncé de problème, nous avons défini les deux modules suivants : *COMMANDE* et *PRODUIT*. Ces deux modules sont décrits séparément. Cependant, il y a une connexion entre ces modules.

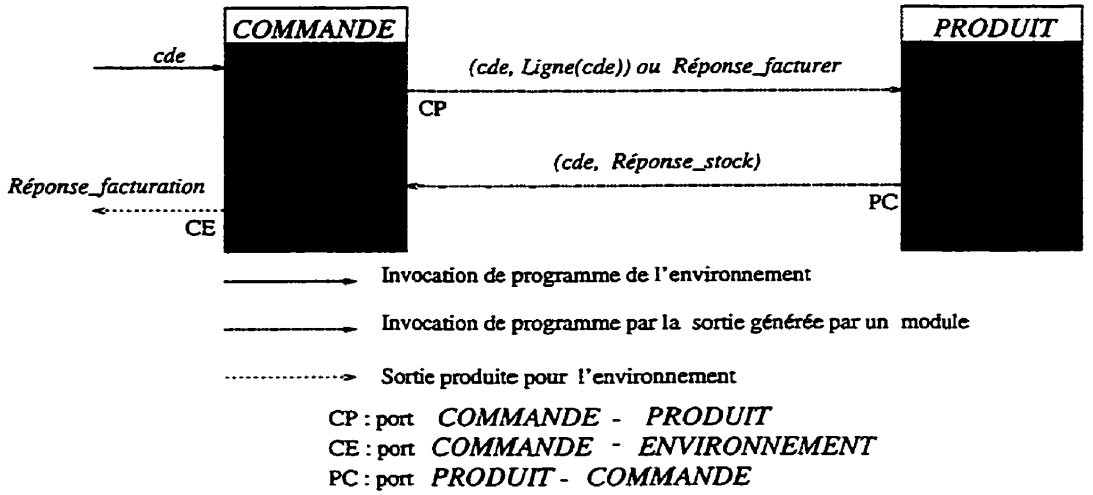


FIG. 12 – *Scénario de la facturation de commande*

La figure 12 illustre le scénario de connexion de la facturation de commande. Cette facturation commence par la saisie du numéro de commande à facturer qui invoque le programme **Facturer_cde**. Pour facturer une commande, deux conditions doivent être satisfaites. D'une part, il faut que la commande ait déjà été créée. D'autre part, il faut que les quantités figurant dans la commande soient disponibles en stock. Pour cela, nous devons consulter le module *PRODUIT* et mettre à jour le stock s'il y a suffisamment de stock pour la commande. Cette consultation et mise à jour de stock seront faites

par l'invocation du programme **Supprimer_stock**. Cette invocation sera faite par les deux paramètres de sortie, *cde* et *Ligne*, générés par le module *COMMANDE* après l'exécution du programme **Facturer_cde** dans le cas où la commande est en attente (si la commande n'a pas été créée, le module *COMMANDE* génère un paramètre de sortie de type booléen). Le paramètre *cde* représente le numéro de commande à facturer. Le paramètre *Ligne* représente l'ensemble des lignes des commandes du système. Il est de type statique, c'est-à-dire qu'il n'est pas modifiable par les appels de programme des modules *PRODUIT* et *COMMANDE* . Sa définition formelle est la suivante :

$$Ligne \in COMMANDE \longrightarrow (PRODUIT \nrightarrow \mathbb{N})$$

Le programme **Supprimer_stock** met à jour le stock s'il y a suffisamment de stock pour tous les produits figurant dans la commande. Dans ce cas, le module *PRODUIT* retourne deux paramètres de sortie, *cde* et *Réponse_stock*, qui indique si l'exécution du programme **Supprimer_stock** se termine avec succès ou non. Ces deux paramètres invoquent le programme **Facturation_cde** qui sert à modifier l'état de EN_ATTENTE à FACTURÉE. Ce changement sera fait seulement dans le cas où le paramètre *Réponse_stock* prend la valeur VRAIE. Dans ce cas, le module *COMMANDE* produit le paramètre de sortie *Réponse_facturation* qui indique si l'opération de la facturation de commande s'achève avec succès ou non.

Pour chaque programme, définir les types des arguments et des valeurs de sortie – La facturation d'une commande consiste à modifier son état de EN_ATTENTE à FACTURÉE. Dans le cas présent, nous supposons qu'une commande donne lieu à une seule facture et qu'une facture correspond à une seule commande.

Selon ce scénario, le module *COMMANDE* comporte les trois programmes d'accès suivants :

- **Facturer_cde** – Ce programme comporte un paramètre d'entrée, *cde*, qui est le numéro de commande à facturer. De plus, il comporte deux paramètres de sortie, *cde* et *Quantité_Cdée*, dans le cas où la commande est en attente. Si la commande

n'a pas été créée, le programme **Facturer_cde** comporte un paramètre de type *Réponse* qui indique l'échec de la facturation.

- **Facturation_cde** – Ce programme sert à modifier l'état de la commande de EN_ATTENTE à FACTURÉE. Il comporte deux paramètres d'entrée, *cde* et *Réponse_stock*, qui indique s'il y a suffisamment de stock pour tous les produits figurant dans la commande *cde*. De plus, il comporte un paramètre de sortie, *Réponse_facturation*, qui est de type *Réponse*. Il prend la valeur OK dans le cas où la facturation d'une commande est effectuée avec succès et ÉCHEC dans la cas inverse. Rappelons que l'invocation de ce programme sera automatiquement effectuée après chaque exécution du programme **Supprimer_stock**.
- **État_cde** – Ce programme sert à afficher l'état courant d'une commande donnée. Il comporte le numéro de commande comme paramètre d'entrée. Il existe trois sorties possibles après l'exécution de ce programme :
  1. INEXISTANTE – Cette sortie sera produite dans le cas où la commande n'a pas été créée.
  2. EN_ATTENTE – Cette sortie sera produite dans le cas où la commande a déjà été créée.
  3. FACTURÉE – Cette sortie sera produite dans le cas où la commande a déjà été créée et facturée

Le module *PRODUIT* a un seul programme : **Supprimer_stock**. Ce programme met à jour la quantité en stock dans le cas où les quantités commandées sont disponibles en stock. Il comporte deux paramètres d'entrée, *cde* et *Quantité_Cdée*. De plus, il comporte deux paramètres de sortie, *cde* et *Réponse_stock*, qui est de type booléen indiquant au succès ou à l'échec de l'exécution de programme.

Définir la syntaxe des programmes d'accès pour chaque module – Le tableau 24 montre la syntaxe des programmes du module *COMMANDE*. Par exemple, le programme **Facturation_cde** comporte deux paramètres d'entrée et un paramètre de sortie. Notons

que toutes les fonctions auxiliaires seront définies dans la section dictionnaire (*Quantité_Cdée*, *E*, etc).

Nom de programme	1^{er} arg	2^{e} arg	Valeur
État_cde	< *COMMANDE* >		*E*
Facturation_cde	< *COMMANDE* >	< *Booléen* >	*Réponse*
Facturer_cde	< *COMMANDE* >		(*COMMANDE*, *Quantité_Cdée*) ou *Réponse*

TAB. 24 – *La syntaxe des programmes d'accès du module COMMANDE*

De la même manière, nous avons défini la syntaxe des programmes d'accès du module *PRODUIT*. Dans le cas présent, ce module a un seul programme d'accès : **Supprimer_stock**. Le tableau 25 illustre la syntaxe de ce programme ainsi que les types de ses arguments et de ses valeurs de sortie.

Nom de programme	1^{er} arg	2^{e} arg	Valeur
Supprimer_stock	< *COMMANDE* >	< *Quantité_Cdée* >	(*COMMANDE*, *Booléen*)

TAB. 25 – *La syntaxe des programmes d'accès du module PRODUIT*

Les traces canoniques

Dans cette section, nous définissons les traces canoniques du système. Il s'agit de faire les deux étapes suivantes.

Identifier les programmes d'accès qui sont de type constructeur de données (ICT) – Selon Wang [45], la trace canonique représentant tous les états possibles d'un module doit seulement comporter les programmes d'accès de type constructeur de données, c'est-à-dire ceux qui fournissent les données au module. Dans le cas présent, les modules *COMMANDE* et *PRODUIT* ont respectivement les programmes **Facturation_cde**

et **Supprimer_stock** qui sont de type constructeur de données.

Pour chaque module, définir les traces canoniques par les programmes identifiés – Concernant le module *COMMANDE*, la trace canonique consiste en la disjonction entre la trace vide, dénotée par _ et représentant l'état initial du module, et la trace canonique composée du programme **Facturation_cde**. Sa définition mathématique est :

$$T_{C_c} = \_ \vee T_{C_F}$$

où T_{C_F} est définie comme suit :

$T_{C_F} = [\textbf{Facturation\_cde}(cde_i, \text{vrai})]_{i=1}^{n}$,
où $cde_i \in COMMANDE$

De la même manière, nous définissons la trace canonique du module *PRODUIT* comme suit :

$$T_{C_p} = \_ \vee T_{C_S}$$

où T_{C_S} est définie comme suit :

$T_{C_S} = [\textbf{Supprimer\_stock}(cde_i, \textit{Quantité\_Cdée})]_{i=1}^{n}$,
où $cde_i \in COMMANDE$

Les équivalences de traces

Dans cette étape, nous définissons un certain nombre de règles de transition d'un état à un autre du module.

Énumérer tous les patrons de traces canoniques – La définition des équivalences de traces à une trace canonique étendue par un programme d'accès débute par l'énumération de toutes les traces canoniques dont les séquences d'appels de programmes d'accès, composant ces traces, ont des effets sur les valeurs de sortie de la trace T étendue par chacun des programmes du module. Par exemple, pour le programme d'accès **Facturation_cde** nous avons défini les patrons de traces canoniques suivants :

1. $T = X.\textbf{Facturation\_cde}(cde, \text{vrai}).Y$ représente une trace canonique qui contient

le programme **Facturation_cde** dont l'exécution s'est terminée avec succès ;

2. $\neg\exists X, Y$: $T = X.$**Facturation_cde**(*cde*, vrai).Y indique l'inexistante d'une trace canonique qui contient le programme **Facturation_cde**.

Formaliser toutes les conditions et les contraintes du système – Dans cette étape, nous définissons les conditions qui doivent être satisfaites par les patrons de traces canoniques définies dans l'étape précédente. Dans le tableau 26, la colonne "Condition" illustre ces conditions.

Définir les équivalences de traces pour toutes les traces canoniques étendues par chacun des programmes ICT – Le tableau 26 illustre toutes les traces équivalentes à la trace canonique T étendue par le programme d'accès **Facturation_cde**. Par exemple, dans la troisième ligne de ce tableau illustre les conditions et le patron de trace T qui définissent la trace équivalente à la trace canonique quelconque T étendue par le programme **Facturation_cde**. Dans cette ligne, le patron de trace indique que la commande *cde* n'a pas été facturée. La condition "*Quantité_suffisante* = faux" indique qu'il n'y a pas suffisamment de stock. Dans ce cas, l'invocation du programme **Facturation_cde** n'a aucun effet sur l'état du module. En conséquent, la trace T étendue par le programme **Facturation_cde** est équivalente à la trace T.

$T.$**Facturation_cde**(*cde*, *Quantité_suffisante*) $=_r$

Condition	Patron de trace	Trace équivalente
vrai	$T = X.$**Facturation_cde**(*cde*, vrai).Y	$T \wedge$ % COMMANDE_FACTURÉE %
Quantité_suffisante = vrai	$\neg\exists X, Y$: $T = X.$**Facturation_cde**(*cde*, vrai).Y	$T.$**Facturation_cde**(*cde*, vrai)
Quantité_suffisante = faux	$\neg\exists X, Y$: $T = X.$**Facturation_cde**(*cde*, vrai).Y	$T \wedge$ % STOCK_INSUFFISANT %

TAB. 26 – *Trace équivalente à* T étendue par **Facturation_cde**

Les valeurs de sortie

Dans cette section, on doit attribuer à chaque module les valeurs de sorties retournées par chacun de ses programmes d'accès.

Lister les programmes d'accès ayant des sorties visibles – Cette étape consiste à lister tous les programmes d'accès ayant des valeurs de sortie figurant dans le tableau de syntaxe. D'après les tableaux de syntaxe, nous remarquons que tous les programmes d'accès ont des sorties visibles.

Déterminer et formaliser toutes les conditions et les contraintes du système relatives à ces programmes – Dans cette étape, nous définissons les conditions qui doivent être satisfaites par les patrons de traces canoniques T. Dans le tableau 27, la colonne "Condition" illustre ces conditions.

Définir les sorties possibles pour toutes les traces étendues par chacun de ces programmes – Le tableau 27 illustre toutes les valeurs de sorties possibles retournées par le programme **Facturation_cde**. Prenons la deuxième ligne de ce tableau. Le paramètre de sortie *Réponse_facturation* prend la valeur OK (le patron de trace et la condition indiquent respectivement que la commande *cde* n'a pas été facturée et qu'il y a suffisamment de stock).

$O(T$,**Facturation_cde**(*cde*, *Quantité_suffisante*)) =

Condition	Patron de trace	Sortie	Port
vrai	$T = X$.**Facturation_cde**(*cde*, vrai).Y	ÉCHEC	CE
Quantité_suffisante= vrai	$\neg\exists X, Y$: T= X.**Facturation_cde**(*cde*, vrai).Y	OK	CE
Quantité_suffisante= faux	$\neg\exists X, Y$: T= X.**Facturation_cde**(*cde*, vrai).Y	ÉCHEC	CE

TAB. 27 – *Valeurs de sortie de* T étendue par **Facturation_cde**

Le dictionnaire

Rappelons que cette dernière étape consiste à définir toutes les fonctions auxiliaires utilisées dans les étapes précédentes. Dans le cas présent, les définitions mathématiques de ces fonctions sont les suivantes :

1. *COMMANDE* est l'ensemble de toutes les commandes du système ;
2. *PRODUIT* est l'ensemble de tous les produits du système ;
3. $E = \{$ INEXISTANTE, EN_ATTENTE, FACTURÉE $\}$;
4. $Quantité\_Cdée = PRODUIT \nrightarrow \mathbb{N}$;
5. $Ligne \in COMMANDE \longrightarrow Quantité\_Cdée$;
6. $Quantité\_suffisante \in T_{C_p} \times Quantité\_Cdée \longrightarrow Booléen$

   $Quantité\_suffisante(T, Qc)$

   $\Leftrightarrow$

   $\forall\ cde, p : (cde, p) \in \mathbf{dom}(Qc)$

   $\Rightarrow$

   $Quantité\_en\_stock(T, p) \geq Qc(p)$;
7. $Quantité\_en\_stock \in T_{C_p} \times PRODUIT \longrightarrow \mathbb{N}$;

   $Quantité\_en\_stock(T.\mathbf{Supprimer\_stock}(cde, Qc), p) =$

   If $p \in \mathbf{dom}(Qc)$

   then

   return $Quantité\_en\_stock(T, p)$ - $Qc(p)$

   else

   return $Quantité\_en\_stock(T, p)$;

   $Quantité\_en\_stock(\_, p) = Stock\_initial(p)$;
8. $Stock\_initial \in PRODUIT \longrightarrow \mathbb{N}$ est la quantité initiale en stock du système ;
9. $Réponse = \{$ OK, ÉCHEC$\}$.

3.5.3 2^e Cas

On doit modifier la spécification précédente pour gérer les commandes et le stock. Pour cela, nous devons ajouter de nouveaux programmes, permettant de réaliser ces deux tâches, aux programmes décrits dans le cas précédent. Évidement, l'ajout de nouveaux programmes nous amène à définir de nouvelles traces équivalentes, à décrire de nouvelles valeurs de sortie et à définir de nouvelles fonctions auxiliaires.

La syntaxe

Nous ajoutons les quatre programmes suivants pour le module *COMMANDE*:

1. **Créer_cde**(*cde*) sert à créer une commande;
2. **Annuler_cde**(*cde*) sert à annuler une commande;
3. **Ajouter_ligne**(*cde, p, q*) sert à ajouter une ligne de commande où q représente la quantité commandée du produit p;
4. **Annuler_ligne**(*cde, p, q*) sert à annuler une ligne de commande.

Nom de programme	1^{er} arg	2^e arg	3^e arg	Valeur
Ajouter_ligne	$< COMMANDE >$	$< PRODUIT >$	$< \mathbb{N} >$	*Réponse*
Annuler_cde	$< COMMANDE >$			*Réponse*
Annuler_ligne	$< COMMANDE >$	$< PRODUIT >$		*Réponse*
Créer_cde	$< COMMANDE >$			*Réponse*
État_cde	$< COMMANDE >$			*E*
Facturation_cde	$< COMMANDE >$	$< Booléen >$		*Réponse*
Facturer_cde	$< COMMANDE >$			(*COMMANDE*, *Quantité_Cdée*) ou *Réponse*

TAB. 28 – *La syntaxe des programmes d'accès du module COMMANDE*

Nous ajoutons les deux programmes suivants pour le module *PRODUIT*:

1. **Ajouter_produit**(p) sert à ajouter un produit où p dénote le numéro de produit à ajouter;
2. **Ajouter_stock**(p, q) sert à ajouter une nouvelle quantité q à la quantité disponible en stock du produit p.

Les tableaux 28 et 29 illustrent respectivement la syntaxe de programmes d'accès des modules *COMMANDE* et *PRODUIT*.

Nom de programme	1^{er} arg	2^{e} arg	Valeur
Ajouter_produit	$< PRODUIT >$		*Réponse*
Ajouter_stock	$< PRODUIT >$	$< \mathbb{N} >$	*Réponse*
Supprimer_stock	$< COMMANDE >$	$<$ *Quantité_Cdée* $>$	(*Char, Booléen*)

TAB. 29 – *La syntaxe des programmes d'accès du module PRODUIT*

Les traces canoniques

Dans cette section, nous définissons les traces canoniques des deux modules. Rappelons que chaque trace canonique représente un état d'un module. D'une manière générale, une trace canonique ne contient que les programmes d'accès qui sont de type constructeur de données (ICT). Concernant le module *COMMANDE*, il y a trois programmes qui sont de type constructeur de données: **Créer_cde**, **Ajouter_ligne** et **Facturation_cde**. Pour des raisons de simplicité, nous avons défini la trace canonique du module *COMMANDE* comme suit:

$$T_{C_c} = T_C.T_I.T_F,$$

1. $T_C = \_ \vee T_{C'}$, où $T_{C'} = [\textbf{Créer\_cde}(cde_i)]_{i=1}^{n}$
2. $T_I = \_ \vee T_{I'}$, où $T_{I'} = [\textbf{Ajouter\_ligne}(cde_i, p_j, q_j)]_{i=1,\ j=1}^{n,m}$
3. $T_{C_F} = \_ \vee T_{C_{F'}}$, où $T_{C_{F'}} = [\textbf{Facturation\_cde}(cde_i, \text{vrai})]_{i=1}^{n}$

De la même manière, nous définissons la trace canonique du module *PRODUIT* avec les programmes ICT **Ajouter_produit** et **Ajouter_stock**. Sa définition mathématique est la suivante :

$$T_{C_p} = T_{C_{Ap}}.T_{C_{As}}$$

1. $T_{C_{Ap}} = \_ \vee T_{C_{Ap'}}$, où $T_{C_{Ap'}} = [\mathbf{Ajouter\_produit}(p_i)]_{i=1}^{n}$
2. $T_{C_{As}} = \_ \vee T_{C_{As'}}$, où $T_{C_{As'}} = [\mathbf{Ajouter\_stock}(p_i, q_j)]_{i=1,\ j=1}^{n,m}$

Les équivalences de traces

Comme dans le premier cas, nous définissons, dans cette section, les traces qui sont équivalentes à une trace canonique quelconque, T, étendue par chacun des programmes définis dans cette extension. Le tableau 30 illustre les traces qui sont équivalentes à la trace canonique T étendue par le programme **Créer_cde**(*cde*).

$T.\mathbf{Créer\_cde}(cde) =_r$

Condition	Patron de trace	Trace équivalente
$cde \in COMMANDE\_(T)$	vrai	$T \wedge$ % COMMANDE_CRÉÉE %
$cde \notin COMMANDE(T)$	vrai	$T.\mathbf{Créer\_cde}(cde)$

TAB. 30 – *Trace équivalente à T étendue par* **Créer_cde**

Les valeurs de sortie

Comme dans le premier cas, nous définissons, dans cette section, les valeurs de sortie des programmes d'accès. Par exemple, le tableau 31 illustre les valeurs de sortie de la trace T étendue par le programme **Créer_cde**(*cde*).

$O(T,$ **Créer_cde**$(cde)) =$

Condition	Patron de trace	Valeur	Port
$cde \in COMMANDE(T)$	vrai	ÉCHEC	CE
$cde \notin COMMANDE(T)$	vrai	OK	CE

TAB. 31 – *Valeurs de sortie de* T étendue par **Créer_cde**

Le dictionnaire

L'ajout des nouveaux programmes d'accès nous amène à définir des nouvelles fonctions auxiliaires et à modifier les définitions de certaines fonctions utilisées dans le premier cas.

Notons que la spécification entière de cet exemple se trouve à l'annexe D.

3.5.4 Remarques

La méthode des assertions de traces donne la structure du système en termes de modules et de liens entre ces modules. Un avantage de ce type de spécification est que la modification d'un module entraîne peu de modifications sur les autres modules. Une spécification par assertion est un automate où les états sont représentés par des traces. Comme pour tout automate, la vérification de la complétude de la spécification est simple ; on doit s'assurer que pour chaque appel de programme, la fonction de transition et la fonction de sortie traitent tous les états possibles. Ceci nous permet de vérifier, d'une façon systématique, la complétude de la spécification.

L'utilisation de tableaux à deux dimensions, pour présenter des formules complexes, rend la compréhension de ces formules plus facile. De plus, elles servent à diminuer la répétition des sous-expressions et à localiser la partie importante dans l'expression.

Nous avons choisi de modéliser le système de facturation en le décomposant en modules, ce qui est classique dans cette approche. La modélisation des liens entre modules pose un problème : comment peut-on identifier et décrire les liens entre les différentes sorties des modules. Nous aurions pu modéliser le système comme un seul module. Cette spécification serait formée essentiellement des mêmes énoncés, car les traces canoniques du

système seraient constituées de la même concaténation des traces canoniques de chaque module.

Au niveau de la notation, la méthode par assertion de trace utilise plusieurs abréviations qui ne sont pas conformes à la syntaxe classique de la logique du premier ordre. Par exemple, on écrit $T_C = \_ \vee [\textbf{Créer\_cde}(cde_i)]_{i=1}^{n}$ pour décrire un ensemble de traces. Il y a plusieurs ambiguités dans cette définition. Premièrement, cette définition est une égalité. On s'attend à des termes comme opérandes du symbole "=". Or, on a à droite une disjonction entre deux termes, ce qui est syntaxiquement incorrect. De plus, le terme $[\textbf{Créer\_cde}(cde_i)]_{i=1}^{n}$, $cde_i \in COMMANDE$, $i = 1..n$ est aussi ambigu. Veut-on signifier que tous les numéros de commandes sont distincts ou peut-il y avoir des répétitions?

3.6 Conclusion

Dans ce chapitre, nous avons modélisé le problème de la facturation sous quatre méthodes formelles. Chaque modélisation apporte sa contribution à l'éclaircissement de l'énoncé en soulevant des problèmes spécifiques.

Finalement, pour achever notre étude de cas il nous reste à dériver des critères afin d'évaluer les quatre méthodes en question. Cette dérivation doit être basée sur les activités de cycle de développement du logiciel. Tout ceci fait l'objet de chapitre suivant.

Chapitre 4

Critères d'évaluation

Dans ce chapitre, chaque phase du cycle de vie est examinée. Les activités de chaque phase qui impliquent le document de spécification sont énumérées afin d'identifier les besoins qu'ils imposent sur la notation du document de spécification. Ces besoins sont traduits en une liste concise de critères qui peuvent être employés pour évaluer des méthodes formelles. Ces critères sont dérivés directement du processus de développement du logiciel plutôt que d'une expérience avec un projet particulier.

Dans la suite, les contextes d'utilisation des spécifications formelles dans chaque phase du cycle de développement du logiciel sont définis. Ensuite, nous présentons quelques travaux connexes dans le domaine de l'évaluation de méthodes formelles. Enfin, nous définissons notre liste de critères afin d'évaluer les méthodes formelles en question.

4.1 Contexte d'utilisation de méthodes formelles

Généralement, le cycle de développement d'un logiciel se compose de plusieurs phases. Celles-ci représentent une liste de tâches à accomplir pour créer un logiciel. L'ordre de l'accomplissement de ces phases est différent d'un modèle à un autre. Par exemple, le modèle *Waterfall* prescrit la réalisation d'une phase avant le commencement d'une autre.

Dans le modèle *Spiral,* l'ordre de développement de différentes parties du système est dicté selon la priorité associée à chacune d'elles.

La figure 13 illustre le contexte d'utilisation des méthodes formelles dans le cycle de développement d'un logiciel. Dans cette figure, les seuls utilisateurs de ces méthodes sont les analystes et les personnes concernées par les phases de conception, d'implantation et de maintenance. Le client et les personnes concernées par la vérification sont des utilisateurs occasionnels. Ils peuvent consulter le document de spécification, mais jamais le produire ou le modifier.

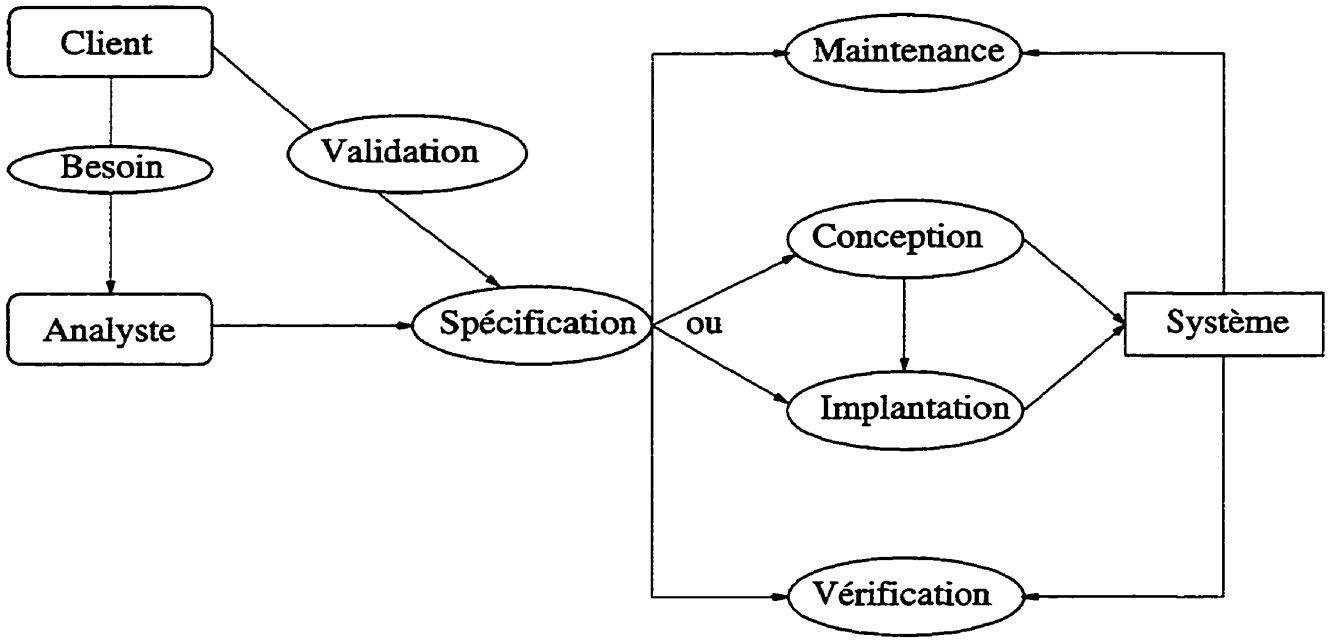


FIG. 13 – *Contexte d'utilisation de spécifications formelles*

Il est clair que le document de spécification est utilisé dans chaque phase du cycle de développement. Pour chaque phase, il y a un contexte d'utilisation du document de spécification. Pour cela, toute dérivation d'une liste de critères doit se baser sur les contextes d'utilisation de la spécification durant la réalisation de chaque phase.

Notons que les phases présentées ne sont pas spécifiques à un modèle particulier.

4.1.1 Spécification

Généralement, deux acteurs interviennent dans cette phase. D'abord, il y a le client, qui exprime ses besoins sous la forme d'un texte en langage naturel. Ensuite, il y a l'analyste, qui a la responsabilité primordiale de traduire le discours nécessairement informel de son client en un document de spécification formel qui doit être non ambiguë, afin de permettre aux développeurs de réaliser un système satisfaisant les exigences du client. C'est l'analyste qui a la tâche difficile de passer de l'informel au formel et d'articuler le "quoi faire" tel qu'exprimé par le client, avec le "comment faire" que comprend le programmeur. Pour cela, il est important que la méthode utilisée permette à l'analyste d'exprimer facilement tous les aspects du système en satisfaisant les besoins du client tout en ayant une spécification compréhensible pour le concepteur et le programmeur.

4.1.2 Conception

Le document de conception doit satisfaire les besoins enregistrés dans le document de la phase précédente. Il doit fournir les détails nécessaires pour créer une implantation concrète. La conception peut être décrite par une notation de spécification, par un langage d'implantation ou par d'autres notations particulières. Si une notation de spécification est utilisée, le document de conception sera un document additionnel au document de spécification. Dans tous les cas, le concepteur a la responsabilité de raffiner la spécification du système afin de déterminer l'architecture du système, de caractériser les interfaces, de structurer les données et de détailler les procédures. Pour cela, plus le document de spécification est clair, lisible, compréhensible et bien structuré, plus la réalisation de cette phase sera facile à achever.

4.1.3 Implantation

L'objectif de la phase d'implantation est de produire des programmes exécutables satisfaisant les documents de spécification et de conception. cette phase. L'implantation est typiquement accomplie par un groupe de programmeurs distincts qui n'a pas participé à la phase de spécification ou de conception. Pour cela, on sera très attentif à la continuité entre le symbolisme utilisé dans la phase de spécification et raffiné dans la phase de conception et le(s) langage(s) de programmation du système. Le but est que les différentes parties du système, accomplies par différentes personnes, soient compatibles entre elles. Mises ensembles, elles doivent satisfaire les besoins du client. La vérification de chaque programme est sous la responsabilité des programmeurs.

4.1.4 Vérification et validation

L'objectif de cette phase est de s'assurer que l'implantation satisfait les besoins documentés dans la phase de spécification. D'une part, le testeur devra consulter la spécification de façon à produire des scénarios d'essais permettant la vérification du système. D'autre part, le client devra consulter le document de la spécification. Dans sa démarche, il pourra se faire assister d'experts pour améliorer la précision de ses exigences et pour valider que la spécification correspond à ses exigences.

L'usage d'outils permet de réduire le temps de vérification du système, de validation de la spécification et de minimiser le nombre de fautes résiduelles dans la spécification. Pour cela, il est important qu'une méthode formelle dispose d'outils permettant au testeur de vérifier automatiquement le système et d'un simulateur permettant au client de valider la spécification.

4.1.5 Maintenance

Dans cette phase, on applique au système des modifications afin de corriger des fautes, de modifier sa fonctionnalité ou d'améliorer sa conception. Pour réaliser un changement, on répète le cycle de développement. De nouvelles personnes s'ajoutent à l'équipe de développement, d'autres quittent. Des modifications introduisent des nouvelles fautes. Il faut donc que la méthode formelle utilisée facilite l'évolution de la spécification pour que la maintenance soit facile à réaliser.

4.2 Travaux connexes

Ardis *et al.* ont présenté une évaluation de six méthodes de spécification : Mode Chart, VFSM, ESTEREL, LOTOS, Z et SDL en plus le langage de programmation C [20]. Pour évaluer ces méthodes, les critères utilisés ont été dérivés de leur expérience, en spécifiant un système de commutation téléphonique avec chacune de ces notations. Les critères ont été divisés en deux catégories : fondamentale et importante. Une table présente les critères associés à chaque phase du cycle de développement. Chaque critère est décrit par un paragraphe. La spécification et l'outil de chaque notation ont été examinés et des indices (+, 0 et -) ont été alloués pour chaque critère.

Pfleeger et Halton ont étudié l'influence de méthode formelles sur un système de contrôle du trafic aérien, *Air traffic Control System,* réalisé par Praxis [40]. Le développement de ce système a fourni un contexte de comparaison entre les méthodes formelles et informelles, car plusieurs parties du système ont été spécifiées en utilisant des méthodes formelles et informelles.

Faulk a présenté une liste de propriétés pour évaluer la qualité de spécification [44]. Ces propriétés ont été classifiées en deux catégories : propriétés sémantiques et propriétés de progiciels. Si la spécification satisfait les propriétés sémantiques alors elle est complète, précise, non ambiguë, cohérente et vérifiable. Si la spécification satisfait les propriétés de

progiciels alors elle est lisible, claire et bien structurée pour la revue et l'inspection.

Rushby a fourni une liste considérable de critères pour sélectionner une méthode formelle dans un rapport pour la NASA [42]. La liste est divisée en deux sortes de critères : les critères de notations et les critères d'utilisation.

Davis a présenté une comparaison de dix méthodes formelles et d'une méthode informelle (langage naturel, machine à états finie, DT, PDL, SA/RT, Statecharts, REVS, RLP, SDL, PAISLey et réseaux de Petri), en spécifiant le comportement externe d'un système téléphonique [16]. Huit critères ont été définis pour l'évaluation de ces méthodes. Les méthodes ont été analysées et comparées, en utilisant chacun des huit critères. Les résultats d'évaluation ont été présentés dans un table sur une échelle de 0 (pauvre) à 10 (excellent).

Notons que tous les critères de ces travaux ne sont pas systématiquement dérivés d'une base d'évaluation bien définie, mais plutôt des expériences de chercheurs sur l'utilisation de méthodes formelles.

4.3 Critères d'évaluation

Les phases du cycle de développement d'un logiciel décrivent la pratique courante du développement du logiciel. Pour cela, elles servent de base pour toute sorte d'évaluation. Dans notre étude, les critères générés, en évaluant les méthodes formelles choisies, s'appliquent seulement à la phase de spécification. Généralement, cette phase consiste en deux activités : la documentation et la validation. La documentation est accomplie par l'analyste, tandis que la validation est accomplie par l'analyste et le client. Pour cela, nous divisons les propriétés voulues d'un document de spécification en deux catégories principales de critères : la documentation et la validation de spécification.

4.3.1 La documentation de spécification

Effort de spécification

Le temps exigé pour écrire une spécification formelle est une considération importante. Pour cela, la méthode utilisée doit soutenir un prototypage rapide de spécification et faciliter l'élaboration de son détail. L'objectif principal, dans l'industrie, est de diminuer le temps développement du système. L'utilisation de méthodes formelles peut allonger le temps de développement des premières phases, comme la documentation de spécification et la validation. En contrepartie, elle peut raccourcir le temps de développement des dernières phases, comme le codage, le test et la maintenance.

Couverture du système

Le langage naturel est infiniment large et flexible. Il peut être utilisé pour exprimer la plupart des exigences fonctionnelle ou non fonctionnelle. La difficulté de décrire toutes les parties du système est une faiblesse commune à plusieurs notations formelles. Pour qu'une méthode soit généralement applicable, il faut que cette méthode nous permette d'exprimer facilement tous les aspects du système.

Support d'évolution

La spécification ne se dérive pas en une seule itération. Elle sera développée et changée autant de fois qu'il est nécessaire, au fur et à mesure que les spécifieurs améliorent leur compréhension du système. La méthode de spécification doit soutenir une évolution logique de la spécification et faciliter son changement (l'addition de nouvelles données, de nouveaux services, etc).

Support d'utilisabilité

La capacité à localiser des informations pertinentes est une partie vitale de l'utilité de la spécification. La spécification est un moyen de communication entre toutes les personnes concernées par le développement. Comme un document de spécification est typiquement volumineux, il faut que la notation de spécification fournisse un mécanisme structuré pour faciliter la navigation. Dans un document de spécification informelle, la table de matières et les index aident à localiser les informations demandées. Les méthodes formelles doivent fournir un moyen similaire pour enrichir l'utilisabilité de leurs documents.

4.3.2 La validation de spécification

Durant les premières phases du cycle de développement, l'attention est mise sur la validation. D'une part, le client doit valider que la description du système satisfait tous ses besoins. D'autre part, le développeur doit aussi valider la complétude et la cohérence à travers le système. Généralement, ces validations peuvent être faites de deux manières : manuelle, où la clarté et la lisibilité de la spécification sont exigées, ou automatique, où un prouveur de théorèmes permet la vérification de la cohérence (e.g. préservation de l'invariant en langage B) et un simulateur du modèle permet de montrer le comportement du système.

4.3.3 La liste de critères

Dans cette section, nous présentons cette liste de critères afin d'évaluer les quatre méthodes formelles choisies. Nous divisons cette liste en deux parties de critères : la documentation et la validation de spécification.

La documentation de spécification

1. **couverture** – la notation permet d'exprimer tous les aspects du système ;
2. **élicitation** – durant la modélisation, la méthode nous permet de poser des questions critiques afin de collecter les informations pertinentes ;
3. **évolution** – la méthode fournit un support pour faciliter la modification et l'évolution de la spécification ;
4. **navigation** – la notation facilite la navigation et la recherche ;
5. **facilité de rédaction** – le temps exigé pour écrire la spécification, la taille et la complexité de la notation sont raisonnablement appropriés ;
6. **facilité d'apprentissage** – la notation est raisonnablement facile à apprendre.

La validation de spécification

1. **facilité de validation** – le temps exigé pour valider la spécification est raisonnablement approprié ;
2. **facilité de compréhension/clarté** – la spécification est lisible, claire, bien structurée et facile à comprendre et à examiner par le client ou l'expert du domaine pour assurer la complétude, la cohérence et l'exactitude ;
3. **automatisation** – la méthode fournit un outil permettant de simuler la spécification afin de visualiser le comportement du système.

Dans une évaluation, on peut utiliser tous les critères ou seulement un sous-ensemble qui est considéré le plus approprié à un projet particulier. Dans le meilleur des cas, on associe un poids à chaque critère selon l'importance relative de la satisfaction du critère afin de rencontrer les objectifs d'un projet particulier. Les résultats de cette évaluation peuvent être employés pour évaluer l'applicabilité d'une méthode formelle à un projet ou pour identifier les aspects qui nécessitent une amélioration. Dans ce document, les résultats d'évaluation seront présentés et des poids (faible, moyen et fort) seront alloués à chaque

méthode afin de comparer les méthodes en question. Tout ceci fait l'objet du prochain chapitre.

Chapitre 5

Les résultats de l'évaluation

En se basant sur les critères définies dans le chapitre précédent, une évaluation de quatre méthodes formelles est présentée: relationnelle inductive, boîte noire par entités, le langage B et assertions de traces.

Dans la suite, nous définissons brièvement l'idée d'une évaluation idéale. Ensuite, nous présentons les résultats d'évaluation des méthodes en question et les implications tirées de cette étude.

5.1 L'évaluation idéale

La manière idéale pour évaluer l'utilité d'une méthode formelle dans l'industrie est d'appliquer cette méthode à une variété et un grand de nombre de projets. Les projets choisis doivent être nombreux et inclure plusieurs domaines d'applications, par exemple: systèmes critiques, systèmes d'information, systèmes distribués, etc. Les objectifs des projets doivent également être variés. Les équipes de développement doivent être formées des praticiens de l'industrie, y compris les gestionnaires, les analystes, les programmeurs et les testeurs. Les informations doivent être collectées après l'application de la méthode par ces praticiens. Le projet doit être suivi dès la conception jusqu'à la maintenance. Des

mesures de la productivité et de la qualité du produit doivent être prises. Tout ceci nous permet d'évaluer l'applicabilité de la méthode aux différents domaines d'applications. Cependant, une étude ayant ces caractéristiques pourrait prendre plusieurs années avec la coopération de milliers de personnes. Ce n'est évidemment pas un contexte dont nous pouvions jouir pour notre étude

5.2 Discussion

Dans cette section, nous présentons les résultats d'évaluation de quatre méthodes formelles en question. Ces résultats montrent les forces et les faiblesses de ces méthodes pour le cas d'étude de la facturation de commandes. Notons que ces résultats s'appliquent seulement aux problèmes similaires à celui de la facturation.

5.2.1 La méthode relationnelle inductive

1. **Couverture** – Il est techniquement possible d'écrire une spécification complète du système de la facturation, mais cette tâche s'est avérée très difficile. Notre spécification ne couvre que les séquences d'entrées contenant des symboles valides (i.e., des entrées qui ne produisent pas de message d'erreur en sortie lorsqu'ils sont traités), car cela simplifiait grandement la rédaction de la spécification. Une des difficultés du cas d'étude consiste à exprimer les contraintes d'ordonnancement entre les entrées et à exprimer le comportement lorsque plusieurs commandes font partie de la séquence d'entrée. Comme cela est typique de plusieurs spécifications de système d'information, cela suggère que la méthode relationnelle est peu applicable, dans l'état actuel de nos connaissances. Il sera nécessaire d'étudier d'autres styles de spécification inductive pour améliorer cette approche.
2. **Élicitation** – Avec cette approche, il est très facile de décrire les comportements de base grâce aux axiomes de base. Par comportement de base, nous entendons les

séquences d'entrées qui contiennent une seule commande, un seul produit, etc. Par contre, il est extrêmement difficile de spécifier le comportement d'une séquence où il y a plusieurs instances d'une commande ou d'un produit, ou s'il y a des entrées "invalides" dans la séquence d'entrée.

3. **Évolution** – La spécification du deuxième cas d'étude nous a amené à faire plusieurs changements à la spécification du premier cas. Cela illustre qu'il est difficile d'écrire des spécifications partielles de ce type. Nous avons dû modifier la signature des opérations : dans le premier cas, la facturation comporte une entrée qui donne l'état courant du stock ; dans le deuxième cas, il faut gérer les stocks, donc il fallait modifier la signature de l'opération facturation et adapter les axiomes qui calculent sa sortie. Dans une spécification de type machine à états comme B, ce problème ne se pose pas, car les stocks peuvent être modélisés dès le premier cas par une variable d'état.
4. **Facilité de validation** – La spécification est assez volumineuse, ce qui rend sa validation laborieuse. De plus, il est difficile de valider la spécification par inspection. Il n'est pas facile de se convaincre de la validité d'un axiome de réduction ou de permutation. Par contre, les axiomes de base sont très faciles à valider. Un outil de simulation de la spécification serait fort utile.
5. **Facilité de compréhension/clarté** – Les axiomes de base sont très faciles à comprendre. Par contre, certains axiomes de réduction sont très cryptiques, surtout ceux qui décrivent comment plusieurs commandes interagissent entre elles.

5.2.2 La méthode boîte noire par entités

1. **Couverture** – L'expressivité de la notation est assez riche pour notre cas d'étude. Ceci est dû à :
   - l'utilisation du diagramme d'entité inspiré de la méthode JSD [32] qui impose

des contraintes sur l'ordonnancement des entrées ;

- la possibilité de définir des contraintes non traités par ce diagramme ;
- la définition de deux prédicats qui permettent d'extraire facilement des informations voulues.

Ces mécanismes ont éliminé les difficultés que nous avons eu dans la méthode relationnelle inductive afin d'exprimer tous les aspects du système.

2. **Élicitation** – Il est très facile de spécifier le comportement d'une séquence où il y a plusieurs instances d'une commande ou d'un produit. De plus, une modélisation en boîte noire par entités obéit à une démarche systématique. La modélisation du système de facturation suivant cette démarche nous a permis de poser des questions critiques et précises permettant de collecter des informations pertinentes.
3. **Évolution** – En spécifiant le deuxième cas, de nouvelles entrées ont été définies. L'ajout de ces entrées nous a obligé à restructurer la spécification du premier cas. De nouvelles entités, de nouveaux diagrammes d'entité et de nouvelles contraintes furent définies. Des modifications de certaines contraintes et des diagrammes d'entité définis dans le premier cas étaient nécessaires.
4. **Facilité de rédaction** – La méthode fait abstraction des cas d'erreurs, ce qui a réduit le volume de travail. L'expressivité de la notation a réduit la taille et a rendu la spécification assez simple.
5. **Facilité de validation** – La taille et la complexité de la spécification sont raisonnablement appropriées. La validation de la spécification par inspection fut assez facile. Par conséquent, le temps exigé ne fut pas très long.
6. **Facilité de compréhension/clarté** – Les diagrammes d'entité permettent de comprendre assez facilement le comportement du système au niveau des séquences d'événements attendus. Toutefois, la compréhension du calcul des sorties est plus ardue.

5.2.3 La méthode B

1. **Couverture** – En B, une variable représente une structure de donnée abstraite (ensemble, etc) et décrit un état (ensemble des valeurs d'une variable à un instant donné). Une variable peut être modifiée ou consultée par des opérations. Le concept de variable d'état permet de spécifier facilement tous les aspects du système de facturation. Il a suffit de définir des ensembles représentant les commandes et les produits du système et leurs opérations correspondantes.
2. **Élicitation** – La méthode B permet de décomposer la spécification globale d'un système en plusieurs machines abstraites. Une machine abstraite se compose des ensembles de base, des variables d'états (invariants) et des opérations. En définissant ces composants, des questions critiques sont soulevées. Ces questions furent suffisantes pour collecter toutes les informations pertinentes du système.
3. **Évolution** – En spécifiant le deuxième cas de notre cas d'étude, il a suffit d'ajouter de nouvelles opérations à la spécification du premier cas. Ceci est dû au concept de la décomposition modulaire utilisé en B. Notons qu'aucun changement de la spécification du premier cas était nécessaire.
4. **Facilité de rédaction** – Vu l'expressivité de la notation, la taille et la complexité de la spécification étaient raisonnablement appropriées. De plus, l'Atelier-B nous a permis d'écrire la spécification avec un temps raisonnablement approprié.
5. **Facilité de validation** – Un ensemble d'outils logiciels, en particulier l'Atelier-B avec lequel nous avons travaillé, permet une assistance dans la réalisation des développements en B. Ces outils permettent, entre autres, la production automatique de code exécutable, la génération des obligations de preuve et la démonstration automatique de certaines entre elles. Par exemple, l'Atelier-B nous a permis :
   - de détecter les erreurs syntaxique dans la spécification ;
   - de démontrer 94 % des obligations de preuve de notre spécification.

Tout ceci nous a permis de valider la spécification avec un temps raisonnablement approprié.

6. **Facilité de compréhension/clarté** – La méthode B permet de décomposer la spécification globale d'un système en plusieurs machines. Cette décomposition rend le document de spécification bien structuré. La spécification en B est raisonnablement facile à lire, à analyser et à comprendre. Cependant, le lecteur doit être familier avec les notations utilisées (logique de prédicats, théorie des ensembles, etc).
7. **Automatisation** – L'Atelier-B permet de simuler la spécification et de visualiser le comportement du système.

5.2.4 La méthode des assertions de traces

1. **Couverture** – Comme dans la méthode relationnelle inductive, nous avons eu des difficultés à exprimer les contraintes d'ordonnancement entre les programmes d'accès. Cependant, la possibilité d'imposer certaines contraintes sur l'ordonnancement des appels de programme d'accès facilite légèrement la modélisation du cas d'étude. La méthode des assertions repose sur la décomposition du système en modules. Cette décomposition amène à modéliser les liens entre modules. Cependant, cette méthode ne définit pas comment modéliser ces liens. Une des difficultés de la spécification par la méthode des assertions vient de là. Dans une spécification en B, ce problème ne se pose pas, car il est possible de définir des clauses permettant d'exprimer les liens entre machines (INCLUDES, SEES, USES, etc).
2. **Élicitation** – Il est difficile de spécifier le comportement d'une trace (séquence d'appels des programmes d'accès) ayant plusieurs instances d'une commande ou d'un produit. Cependant, il est possible de définir des fonctions récursives qui calculent les sorties d'une trace. Cette possibilité rend la spécification du comportement d'une trace moins difficile.

Durant notre modélisation, la notation nous a permis de poser des questions qui se concentrent généralement sur la décomposition du système en modules, l'énumération de programmes d'accès, l'identification des programmes ICT. De plus, la définition des fonctions auxiliaires, afin de calculer les équivalences de traces et leurs valeurs de sorties, nous a permis de poser des questions sur les contraintes d'activation du système.

3. **Évolution** – Le concept d'encapsulation de données utilisé dans cette méthode permet de réutiliser les mêmes modules définis dans la spécification du premier cas. En modélisant le deuxième cas du problème de facturation, il a suffit d'ajouter des nouveaux programmes afin de répondre aux nouvelles exigences. Cependant, l'ajout de nouveaux programmes d'accès nous a obligé à faire des changements majeurs des traces canoniques définies dans le premier cas. Ces changements nous ont obligé à restructurer les équivalences de traces et les valeurs de sorties définies dans la spécification du premier cas.
4. **Facilité de validation** – Un outil, *Table Tool System* (TTS) [1], a été développé pour la validation de la spécification et pour vérifier si une implantation satisfait la spécification. L'outil TTS permet de manipuler ou interpréter des groupes d'expressions tabulaires (table des équivalences de traces, valeurs de sortie, etc). Cependant, l'outil TTS ne nous étant pas disponible, nous avons dû procéder par inspection. Pour cela, le temps exigé pour la validation n'était pas raisonnablement approprié. La disponibilité de cet outil aurait peut-être réduit le temps exigé pour la validation.
5. **Facilité de compréhension/clarté** – La spécification en assertions de traces se présente comme des expressions tabulaires. Cette présentation rend la spécification facile à lire et à comprendre. Cependant, il est difficile de comprendre la notation, car la méthode des assertions de traces emploie des abréviations qui ne sont pas conformes à la syntaxe classique de la logique de premier ordre. De plus, la définition

des fonctions auxiliaires, présentées dans une section autre que la section où elles se présentent, rendent la spécification moins compréhensible. Aussi, il est plus difficile de s'assurer de la cohérence entre les modules (il n'est pas évident d'identifier les liens entre les différents modules du système).

6. **Automatisation** – L'outil TTS peut produire du code pour évaluer une expression tabulaire en utilisant le modèle sémantique général. Le générateur de code d'évaluation fait partie du TTS et produit un programme en C pour chaque expression définie dans l'entrée. Ce programme permet de visualiser le comportement du système en produisant des valeurs de sortie pour chaque entrée. Cependant, la production de ces valeurs de sortie est limitée, parce que le code généré produit des valeurs de sortie pour des données d'entrée seulement si l'expression de spécification définit une fonction.

5.2.5 Points communs

Dans cette section, nous présentons quelques résultats d'évaluation qui sont communs entre les méthodes formelles en question.

- **Relationnelle inductive et assertions de traces**
  - **Navigation** – La présentation tabulaire de la spécification nous a permis de chercher facilement les informations voulues et de localiser les informations pertinentes dans le document de spécification.
  - **Facilité de rédaction** – La faible expressivité de ces notations rend les spécifications plus longues et assez complexes. Par conséquent nous avons eu besoin de plus de temps pour décrire le système par ces deux méthodes.
- **Relationnelle inductive et boîte noire par entités**
  - **Automatisation** – Ces deux méthodes manquent d'outils pour exécuter la spécification afin de visualiser le comportement du système.

- **Boîte noire par entités et B**
  - **Navigation** – La spécification en boîte noire par entités ou en B est généralement bien structurée. Cependant, la manque d'un guide (comme la table des matières dans le langage naturel) rend la navigation et la recherche des informations voulues assez difficiles.
- **Relationnelle inductive, boîte noire par entités, B et assertions de traces**
  - **Facilité d'apprentissage** – Durant notre traitement du cas d'étude, nous avons remarqué la simplicité de l'application de ces méthodes. Il nous semblait être capable de continuer à les étudier sans assistance. De plus, on était assez confortable avec les notations dans le cas où une modification du système devait être établie. Cependant, l'utilisateur doit être familier avec les différentes notations mathématiques utilisées dans ces méthodes.

5.2.6 Résumé de l'évaluation

Les tableaux 32 et 33 présentent un résumé de notre évaluation pour les quatre méthodes formelles en question. Pour chaque critère, nous avons donné une évaluation en termes de trois niveaux (Faible, Moyen, Fort).

Critères	Relationnelle inductive	Boîte noire par entités	langage B	Assertions de traces
Couverture	Faible	Fort	Fort	Faible
Élicitation	Moyen	Fort	Fort	Moyen
Évolution	Faible	Moyen	Fort	Moyen
Navigation	Fort	Moyen	Moyen	Fort
Facilité de rédaction	Faible	Fort	Fort	Faible
Facilité d'apprentissage	Moyen	Moyen	Moyen	Moyen

TAB. 32 – *Résultats d'évaluation pour la documentation de spécifications*

Critères	Relationnelle inductive	Boîte noire par entités	langage B	Assertions de traces
Facilité de validation	Faible	Moyen	Fort	Faible
Facilité de compréhension /clarté	Faible	Moyen	Moyen	Moyen
Automatisation	Faible	Faible	Fort	Faible

TAB. 33 – *Résultats d'évaluation pour la validation de spécifications*

5.3 Leçons tirées

Dans cette section, nous présentons quelques leçons tirées de la modélisation du problème de la facturation.

- *Les inspections de spécifications sont inestimables*

  De mauvaises structures et des erreurs peuvent être éliminées de la spécification par l'inspection et la discussion avec d'autres personnes familières avec le projet. Durant la modélisation du cas de la facturation, plusieurs inspections informelles ont été faites et ont eu comme conséquence des révisions importantes de la spécification.

- *La validation de complétude est indispensable*

  La complétude de la spécification est essentielle à son succès comme document de référence au sujet du système. Etant donné la taille de l'espace d'états, il est essentiel d'avoir des outils qui automatisent une partie de la validation de la complétude. Toutefois, cette tâche nécessitera toujours une inspection par les analystes et les clients.

- *En spécifiant en B, il est préférable d'élaborer une première architecture de la spécification par une méthode boîte noire*

  La méthode B repose sur une construction modulaire du système ainsi que sur les variables d'état et les opérations. Cependant, elle ne permet pas d'indiquer le

comportement observable du système. Durant la modélisation du problème de la facturation avec la méthode B, nous avons remarqué qu'il est préférable d'élaborer une première architecture par une méthode boîte noire, notamment la méthode boîte noire par entités. Ceci nous permet de comprendre mieux ce que le système doit faire et de décrire plus abstraitement ses exigences.

Conclusion

Dans ce document, nous avons modélisé le problème de la facturation sous quatre méthodes formelles. Pour chaque méthode nous avons défini un processus en insistant sur les étapes nécessitant des choix de modélisation et les conséquences de ces choix. Ce processus a permis de faciliter la modélisation et d'éviter les oublis. La variété de niveaux de description (boîte noire, machine à états) fait naître des questions différentes. Ces dernières ont apporté des contributions à l'éclaircissement de l'énoncé en soulevant des problèmes spécifiques. De plus, nous avons défini une approche afin d'évaluer l'applicabilité des méthodes en question. Les deux étapes principales de cette approche sont : la dérivation des critères et l'évaluation des spécifications. Pour la dérivation des critères, nous nous sommes basés sur l'utilisation de la pratique en vigueur de la modélisation et sur le contexte d'utilisation du document de spécification dans les différentes phases du cycle de développement.

Les résultats de notre évaluation permettent aux développeurs de choisir facilement la méthode la plus appropriée afin de spécifier des problèmes similaires. Dans ces résultats, nous avons montré les avantages et les limitations des quatre méthodes.

- La méthode relationnelle inductive utilise la notion d'induction pour décrire le comportement observable du système. Cette notion ne permet pas d'exprimer facilement les contraintes d'ordonnancement entre les séquences d'entrée pour un problème comme la facturation de commande. Par contre, d'autres études ont montré qu'il était facile de décrire le comportement du système téléphonique avec cette

méthode. Ce qui montre l'importance des études d'évaluation.

- La spécification graphique utilisée dans la méthode boîte noire par entités facilite la lecture, la compréhension et la validation par inspection de la spécification. La méthode boîte noire par entités utilise le concept d'extraction des instances d'une séquence pour décrire le comportement observable et spécifier seulement les cas valides. Ceci réduit l'effort exigé de la part de l'analyste. Cependant, l'absence d'un outil de visualisation du comportement du système rend son utilisation moins intéressante.
- Un avantage de la modélisation en B est qu'aucune modification était nécessaire à la spécification du premier cas en spécifiant le deuxième cas, ce qui n'était pas le cas pour les autres méthodes formelles. De plus, le concept de variable d'état permet de spécifier facilement le système. L'Atelier-B, avec lequel nous avons travaillé, permet une assistance dans la modélisation en B, une vérification syntaxique de la spécification, une génération des obligations de preuve et une génération de code exécutable. Cependant, en B, il n'est pas évident d'identifier le comportement observable du système.
- La méthode des assertions de traces repose sur une construction modulaire du système. L'avantage de ce type de modélisation est que la modification d'un module entraîne peu de modifications sur les autres modules. Une spécification par cette méthode est un automate, où les états sont représentés par des traces. Ceci facilite la validation de la complétude de la spécification. L'utilisation des abréviations qui ne sont pas conformes à la syntaxe classique de la logique du premier ordre rend la spécification obscure (si le lecteur n'est pas familier avec la notation).

Le résultat de ce travail reste subjectif. Une voie intéressante est la définition de critères et de mesures pour une évaluation objective. Pour ce faire, on doit appliquer les méthodes en question à une variété de domaines d'applications. Le développement doit être suivi dès la conception jusqu'à la maintenance. Des mesures doivent être fixées

afin de rendre l'évaluation plus objective (le nombre de fautes détectées, le temps de la rédaction, de la validation et de la vérification, la taille, etc). Cependant, une étude qui a ces caractéristiques aura besoin de la coopération de plusieurs personnes et pourrait prendre plusieurs années. Ce qui n'était pas faisable dans le contexte de notre étude.

Annexe A

La spécification avec la méthode relationnelle inductive

A.1 1^{er} cas

A.1.1 L'espace d'entrée et de sortie

$$I \triangleq \{ \textbf{Créer\_cde} \} \times Id\_COMMANDE \cup$$
$$\{ \textbf{Ajouter\_ligne} \} \times Id\_COMMANDE \times Id\_LIGNE \times Id\_PRODUIT \times \mathbb{N} \cup$$
$$\{ \textbf{Facturer\_cde} \} \times Id\_COMMANDE \times QUANTITÉ\_EN\_STOCK \cup$$
$$\{ \textbf{État\_cde} \} \times Id\_COMMANDE$$

$$O \triangleq \{ \text{INEXISTANTE, EN\_ATTENTE, FACTURÉE, OK, ERREUR} \}$$

A.1.2 Les axiomes de base

- Créer une commande :

$$\text{B1 - } \frac{}{\langle \textbf{Créer\_cde}, cde \rangle \lhd \text{R} \rhd \text{OK}}$$

- Afficher l'état d'une commande :

$$\text{B2 - } \frac{}{\langle \textbf{État\_cde}, cde \rangle \lhd \text{R} \rhd \text{INEXISTANTE}}$$

A.1.3 Les axiomes de réduction

- Créer plusieurs commandes :

$$\text{R1 - } \frac{\begin{array}{c} cde \neq cde' \wedge \\ x.\langle \textbf{Créer\_cde}, cde \rangle \lhd \text{R} \rhd \text{OK} \wedge \\ x.\langle \textbf{Créer\_cde}, cde' \rangle \lhd \text{R} \rhd \text{OK} \end{array}}{x.\langle \textbf{Créer\_cde}, cde \rangle.\langle \textbf{Créer\_cde}, cde' \rangle \lhd \text{R} \rhd \text{OK}}$$

- Ajouter ligne de commande (plusieurs commandes) :

$$\text{R2 - } \frac{\begin{array}{c} cde \neq cde' \wedge \\ x.\langle \textbf{Créer\_cde}, cde' \rangle \lhd \text{R} \rhd \text{OK} \wedge \\ x.\langle \textbf{Ajouter\_ligne}, cde, l, p, q \rangle \lhd \text{R} \rhd \text{OK} \end{array}}{x.\langle \textbf{Ajouter\_ligne}, cde, l, p, q \rangle.\langle \textbf{Créer\_cde}, cde' \rangle \lhd R \rhd \text{OK}}$$

- Facturer commande :

$$\text{R3 - } \frac{\begin{array}{c} cde \neq cde' \wedge \\ x.\langle \textbf{Créer\_cde}, cde' \rangle \lhd \text{R} \rhd \text{OK} \wedge \\ x.\langle \textbf{Facturer\_cde}, cde, Q \rangle \lhd \text{R} \rhd \text{OK} \end{array}}{x.\langle \textbf{Facturer\_cde}, cde, Q \rangle.\langle \textbf{Créer\_cde}, cde' \rangle \lhd R \rhd \text{OK}}$$

- État commande :

$$\text{R4 - } \frac{x \neq \varepsilon \ \wedge x.x' \lhd \text{R} \rhd y}{x.\langle \textbf{État\_cde}, cde \rangle.x' \lhd \text{R} \rhd y}$$

- Ajouter ligne de commande (une commande) :

$$\text{R5 - } \frac{x.\langle \textbf{Créer\_cde}, cde\rangle \lhd \mathrm{R} \rhd \textsc{ok}}{x.\langle \textbf{Créer\_cde}, cde\rangle.\langle \textbf{Ajouter\_ligne}, cde, l, p, q\rangle \lhd \mathrm{R} \rhd \textsc{ok}}$$

- Ajouter plusieurs lignes :

$$\text{R6 - } \frac{\begin{array}{c} p \neq p' \ \wedge \ l \neq l' \ \wedge \\ x.\langle \textbf{Ajouter\_ligne}, cde, l, p, q\rangle \lhd \mathrm{R} \rhd \textsc{ok} \ \wedge \\ x.\langle \textbf{Ajouter\_ligne}, cde, l', p', q'\rangle \lhd \mathrm{R} \rhd \textsc{ok} \end{array}}{\begin{array}{c} x.\langle \textbf{Ajouter\_ligne}, cde, l, p, q\rangle. \\ \langle \textbf{Ajouter\_ligne}, cde, l', p', q'\rangle \lhd \mathrm{R} \rhd \textsc{ok} \end{array}}$$

- Facturer une commande avec une seule ligne :

$$\text{R7 - } \frac{\begin{array}{c} Q(p) \geq q \ \wedge \\ x.\langle \textbf{Créer\_cde}, cde\rangle \lhd \mathrm{R} \rhd \textsc{ok} \end{array}}{\begin{array}{c} x.\langle \textbf{Créer\_cde}, cde\rangle.\langle \textbf{Ajouter\_ligne}, cde, l, p, q\rangle. \\ \langle \textbf{Facturer\_cde}, cde, Q\rangle \lhd \mathrm{R} \rhd \textsc{ok} \end{array}}$$

- Facturer commande avec plusieurs lignes :

$$\text{R8 - } \frac{\begin{array}{c} Q(p) \geq q \ \wedge \\ x.\langle \textbf{Ajouter\_ligne}, cde, l, p, q\rangle \lhd \mathrm{R} \rhd \textsc{ok} \ \wedge \\ x.\langle \textbf{Facturer\_cde}, cde, Q\rangle \lhd \mathrm{R} \rhd \textsc{ok} \end{array}}{x.\langle \textbf{Ajouter\_ligne}, cde, l, p, q\rangle.\langle \textbf{Facturer\_cde}, cde, Q\rangle \lhd \mathrm{R} \rhd \textsc{ok}}$$

- État commande en attente (commande vide) :

$$\text{R9 - } \frac{x.\langle \textbf{Créer\_cde}, cde\rangle \lhd \mathrm{R} \rhd \textsc{ok}}{x.\langle \textbf{Créer\_cde}, cde\rangle.\langle \textbf{État\_cde}, cde\rangle \lhd \mathrm{R} \rhd \textsc{en\_attente}}$$

- État commande en attente (commande non vide) :

$$\text{R10 - } \frac{x.\langle \textbf{Ajouter\_ligne}, cde, l, p, q\rangle \lhd \mathrm{R} \rhd \textsc{ok}}{x.\langle \textbf{Ajouter\_ligne}, cde, l, p, q\rangle.\langle \textbf{État\_cde}, cde\rangle \lhd \mathrm{R} \rhd \textsc{en\_attente}}$$

- État commande facturée :

$$\text{R11 - } \frac{x.\langle \textbf{Facturer\_cde}, cde, Q\rangle \lhd \mathrm{R} \rhd \textsc{ok}}{x.\langle \textbf{Facturer\_cde}, cde, Q\rangle.\langle \textbf{État\_cde}, cde\rangle \lhd \mathrm{R} \rhd \textsc{facturée}}$$

A.1.4 Les axiomes de permutation

$$\frac{p \wedge x' \neq \varepsilon \ \wedge \ x.e_1.e_2.x' \lhd R \rhd s}{x.e_2.e_1.x' \lhd R \rhd s}$$

où e_1 et e_2 correspondent aux deux dernières colonnes du tableau suivant.

No	Condition	Entrée1	Entrée2
P1	$cde \neq cde'$	⟨**Créer_cde**, cde⟩	⟨**Créer_cde**, cde'⟩
P2	$cde \neq cde'$	⟨**Créer_cde**, cde⟩	⟨**État_cde**, cde'⟩
P3	$cde \neq cde'$	⟨**Créer_cde**, cde⟩	⟨**Facturer_cde**, cde', Q⟩
P4	$cde \neq cde'$	⟨**Créer_cde**, cde⟩	⟨**Ajouter_ligne**, cde', l', p', q'⟩
P5	$p \neq p'$	⟨**Ajouter_ligne**, $cde,\ l,\ p,\ q$⟩	⟨**Ajouter_ligne**, cde, l', p', q'⟩
P6	$cde \neq cde'$	⟨**Ajouter_ligne**, $cde,\ l,\ p,\ q$⟩	⟨**Facturer_cde**, cde', Q⟩
P7	$cde \neq cde'$	⟨**Ajouter_ligne**, $cde,\ l,\ p,\ q$ ⟩	⟨**Ajouter_ligne**, cde', l', p', q'⟩
P8	$cde \neq cde'$	⟨**Facturer_cde**, $cde,\ Q$ ⟩	⟨**Ajouter_ligne**, cde', l', p', q'⟩
P9	$cde \neq cde'$	⟨**Facturer_cde**, cde', Q⟩	⟨**Ajouter_ligne**, $cde,\ l,\ p,\ q$⟩
P10	$cde \neq cde'$	⟨**Facturer_cde**, $cde,\ Q$⟩	⟨**Créer_cde**, cde'⟩
P11	$cde \neq cde'$	⟨**Facturer_cde**, $cde,\ Q$⟩	⟨**État_cde**, cde'⟩
P12		⟨**Ajouter_ligne**, $cde,\ l,\ p,\ q$⟩	⟨ **État_cde**, cde⟩
P13	$cde \neq cde'$	⟨**Ajouter_ligne**, $cde,\ l,\ p,\ q$⟩	⟨**Créer_cde**, cde'⟩
P14	$cde \neq cde'$	⟨**Facturer_cde**, $cde,\ Q$⟩	⟨**Facturer_cde**, cde', Q⟩
P15		⟨**État_cde**, cde⟩	⟨**État_cde**, cde'⟩

A.2 2^e cas

A.2.1 L'espace d'entrée et de sortie

$$I \triangleq \{ \textbf{Créer\_cde} \} \times Id\_COMMANDE \cup$$
$$\{ \textbf{Ajouter\_ligne} \} \times Id\_COMMANDE \times Id\_LIGNE \times Id\_PRODUIT \times \mathbb{N} \cup$$
$$\{ \textbf{Annuler\_ligne} \} \times Id\_COMMANDE \times Id\_LIGNE \cup$$
$$\{ \textbf{Annuler\_cde} \} \times Id\_COMMANDE \cup$$
$$\{ \textbf{Facturer\_cde} \} \times Id\_COMMANDE \cup$$
$$\{ \textbf{État\_cde} \} \times Id\_COMMANDE \cup$$
$$\{ \textbf{Ajouter\_produit} \} \times Id\_PRODUIT \cup$$
$$\{ \textbf{Ajouter\_stock} \} \times Id\_PRODUIT \times \mathbb{N} \cup$$
$$\{ \textbf{Afficher\_stock} \} \times Id\_PRODUIT$$

$$O \triangleq \{ \text{INEXISTANTE, EN\_ATTENTE, FACTURÉE, OK, ERREUR} \} \cup \mathbb{N}$$

A.2.2 Les axiomes de base

- Créer une commande :

B1 - $$\frac{}{\langle \textbf{Créer\_cde}, cde \rangle \lhd \mathrm{R} \rhd \text{OK}}$$

- État commande inexistante :

B2 - $$\frac{}{\langle \textbf{État\_cde}, cde \rangle \lhd \mathrm{R} \rhd \text{INEXISTANTE}}$$

- Ajouter nouveau produit :

B3 - $$\frac{}{\langle \textbf{Ajouter\_produit}, p \rangle \lhd \mathrm{R} \rhd \text{OK}}$$

- Afficher stock d'un produit :

B4 - $$\frac{}{\langle \textbf{Ajouter\_produit}, p \rangle . \langle \textbf{Afficher\_stock}, p \rangle \lhd \mathrm{R} \rhd 0}$$

A.2.3 Les axiomes de réduction

- Créer plusieurs commandes :

$$\text{R1 - } \frac{\begin{array}{c} cde \neq cde' \wedge \\ x.\langle \textbf{Créer\_cde}, cde\rangle \lhd \mathrm{R} \rhd \mathrm{OK} \wedge \\ x.\langle \textbf{Créer\_cde}, cde'\rangle \lhd \mathrm{R} \rhd \mathrm{OK} \end{array}}{x.\langle \textbf{Créer\_cde}, cde\rangle.\langle \textbf{Créer\_cde}, cde'\rangle \lhd \mathrm{R} \rhd \mathrm{OK}}$$

- Ajouter ligne de commande (plusieurs commandes) :

$$\text{R2 - } \frac{\begin{array}{c} cde \neq cde' \wedge \\ x.\langle \textbf{Créer\_cde}, cde'\rangle \lhd \mathrm{R} \rhd \mathrm{OK} \wedge \\ x.\langle \textbf{Ajouter\_ligne}, cde, l, p, q\rangle \lhd \mathrm{R} \rhd \mathrm{OK} \end{array}}{x.\langle \textbf{Ajouter\_ligne}, cde, l, p, q\rangle.\langle \textbf{Créer\_cde}, cde'\rangle \lhd R \rhd \mathrm{OK}}$$

- Facturer commande :

$$\text{R3 - } \frac{\begin{array}{c} cde \neq cde' \wedge \\ x.\langle \textbf{Créer\_cde}, cde'\rangle \lhd \mathrm{R} \rhd \mathrm{OK} \wedge \\ x.\langle \textbf{Facturer\_cde}, cde\rangle \lhd \mathrm{R} \rhd \mathrm{OK} \end{array}}{x.\langle \textbf{Facturer\_cde}, cde\rangle.\langle \textbf{Créer\_cde}, cde'\rangle \lhd R \rhd \mathrm{OK}}$$

- État commande :

$$\text{R4 - } \frac{x \neq \varepsilon \wedge x.x' \lhd \mathrm{R} \rhd y}{x.\langle \textbf{État\_cde}, cde\rangle.x' \lhd \mathrm{R} \rhd y}$$

- Ajouter ligne de commande (une commande) :

$$\text{R5 - } \frac{x.\langle \textbf{Créer\_cde}, cde\rangle \lhd \mathrm{R} \rhd \mathrm{OK}}{x.\langle \textbf{Créer\_cde}, cde\rangle.\langle \textbf{Ajouter\_ligne}, cde, l, p, q\rangle \lhd \mathrm{R} \rhd \mathrm{OK}}$$

- Ajouter plusieurs lignes :

$$\text{R6 - } \frac{\begin{array}{c} p \neq p' \ \wedge l \neq l' \wedge \\ x.\langle \textbf{Ajouter\_ligne}, cde, l, p, q\rangle \lhd \mathrm{R} \rhd \mathrm{OK} \wedge \\ x.\langle \textbf{Ajouter\_ligne}, cde, l', p', q'\rangle \lhd \mathrm{R} \rhd \mathrm{OK} \end{array}}{\begin{array}{c} x.\langle \textbf{Ajouter\_ligne}, cde, l, p, q\rangle. \\ \langle \textbf{Ajouter\_ligne}, cde, l', p', q'\rangle \lhd \mathrm{R} \rhd \mathrm{OK} \end{array}}$$

- État commande en attente (commande vide) :

$$\text{R7 - } \frac{x.\langle \textbf{Créer\_cde}, cde\rangle \lhd \mathrm{R} \rhd \textsc{ok}}{x.\langle \textbf{Créer\_cde}, cde\rangle.\langle \textbf{État\_cde}, cde\rangle \lhd \mathrm{R} \rhd \textsc{en\_attente}}$$

- État commande en attente (commande non vide) :

$$\text{R8 - } \frac{x.\langle \textbf{Ajouter\_ligne}, cde, l, p, q\rangle \lhd \mathrm{R} \rhd \textsc{ok}}{x.\langle \textbf{Ajouter\_ligne}, cde, l, p, q\rangle.\langle \textbf{État\_cde}, cde\rangle \lhd \mathrm{R} \rhd \textsc{en\_attente}}$$

- État commande facturée :

$$\text{R9 - } \frac{x.\langle \textbf{Facturer\_cde}, cde\rangle \lhd \mathrm{R} \rhd \textsc{ok}}{x.\langle \textbf{Facturer\_cde}, cde\rangle.\langle \textbf{État\_cde}, cde\rangle \lhd \mathrm{R} \rhd \textsc{facturée}}$$

- Annuler commande :

$$\text{R10 - } \frac{x \lhd R \rhd y}{\langle \textbf{Créer\_cde}, cde\rangle.\langle \textbf{Annuler\_cde}, cde\rangle.x \lhd \mathrm{R} \rhd y}$$

- Annuler ligne :

$$\text{R11 - } \frac{x.\langle \textbf{Créer\_cde}, cde\rangle \lhd \mathrm{R} \rhd y}{x.\langle \textbf{Créer\_cde}, cde\rangle.\langle \textbf{Ajouter\_ligne}, cde, l, p, q\rangle.\langle \textbf{Annuler\_ligne}, cde, l, p, q\rangle \lhd \mathrm{R} \rhd y}$$

- Créer plusieurs commandes avec ajout de stock :

$$\text{R12 - } \frac{\begin{array}{c} x.\langle \textbf{Créer\_cde}, cde\rangle \lhd \mathrm{R} \rhd \textsc{ok} \wedge \\ x.\langle \textbf{Ajouter\_stock}, p, q\rangle \lhd \mathrm{R} \rhd \textsc{ok} \end{array}}{x.\langle \textbf{Ajouter\_stock}, p, q\rangle.\langle \textbf{Créer\_cde}, cde\rangle \lhd \mathrm{R} \rhd \textsc{ok}}$$

- Ajouter stock d'un produit :

$$\text{R13 - } \frac{x.\langle \textbf{Ajouter\_produit}, p\rangle \lhd \mathrm{R} \rhd \textsc{ok}}{x.\langle \textbf{Ajouter\_produit}, p\rangle.\langle \textbf{Ajouter\_stock}, p, q\rangle \lhd \mathrm{R} \rhd \textsc{ok}}$$

- Ajouter plusieurs stocks d'un produit :

$$\text{R14 - } \frac{\begin{array}{c} q = q' + q'' \wedge \\ x.\langle \textbf{Ajouter\_stock}, p, q\rangle.x' \lhd \mathrm{R} \rhd y \end{array}}{x.\langle \textbf{Ajouter\_stock}, p, q'\rangle.\langle \textbf{Ajouter\_stock}, p, q''\rangle.x' \lhd \mathrm{R} \rhd y}$$

- Ajouter plusieurs stocks d'un produit :

$$\text{R15 - } \frac{\begin{array}{c} q = q' + q'' \wedge x' \neq \varepsilon \wedge \\ x.\langle \textbf{Ajouter\_stock}, p, q' \rangle . \\ \langle \textbf{Ajouter\_stock}, p, q'' \rangle . x' \lhd \mathrm{R} \rhd y \end{array}}{x.\langle \textbf{Ajouter\_stock}, p, q \rangle . x' \lhd \mathrm{R} \rhd y}$$

- Pour facturer une commande avec une ligne de commande :

$$\text{R16 - } \frac{\begin{array}{c} q \geq q' \wedge \\ x.\langle \textbf{Créer\_cde}, cde \rangle . \langle \textbf{Ajouter\_ligne}, cde, l, p, q' \rangle . \\ \langle \textbf{Ajouter\_stock}, p, q \rangle \lhd \mathrm{R} \rhd \mathrm{ok} \end{array}}{\begin{array}{c} x.\langle \textbf{Créer\_cde}, cde \rangle . \langle \textbf{Ajouter\_ligne}, cde, l, p, q' \rangle . \\ \langle \textbf{Ajouter\_stock}, p, q \rangle . \langle \textbf{Facturer\_cde}, cde \rangle \lhd \mathrm{R} \rhd \mathrm{ok} \end{array}}$$

- Facturer commande avec plusieurs lignes :

$$\text{R17 - } \frac{\begin{array}{c} q \geq q' \wedge \\ x.\langle \textbf{Facturer\_cde}, cde \rangle \lhd \mathrm{R} \rhd \mathrm{OK} \wedge \\ x.\langle \textbf{Ajouter\_ligne}, cde, l, p, q' \rangle . \langle \textbf{Ajouter\_stock}, p, q \rangle \lhd \mathrm{R} \rhd \mathrm{OK} \end{array}}{\begin{array}{c} x.\langle \textbf{Ajouter\_ligne}, cde, l, p, q' \rangle . \\ \langle \textbf{Ajouter\_stock}, p, q \rangle . \langle \textbf{Facturer\_cde}, cde \rangle \lhd \mathrm{R} \rhd \mathrm{OK} \end{array}}$$

- Ajouter stock :

$$\text{R18 - } \frac{x.\langle \textbf{Ajouter\_produit}, p \rangle \lhd \mathrm{R} \rhd \mathrm{OK}}{x.\langle \textbf{Ajouter\_produit}, p \rangle . \langle \textbf{Ajouter\_stock}, p, q \rangle \lhd \mathrm{R} \rhd \mathrm{OK}}$$

- Afficher stock :

$$\text{R19 - } \frac{x.\langle \textbf{Afficher\_stock}, p \rangle \lhd \mathrm{R} \rhd y}{x.\langle \textbf{Ajouter\_stock}, p, q \rangle . \langle \textbf{Afficher\_stock}, p \rangle \lhd \mathrm{R} \rhd y + q}$$

- Facturer commande avec une ou plusieurs lignes (ajouter stock) :

$$\text{R20 - } \frac{\begin{array}{c} q \geq q' \wedge \\ x.\langle \textbf{Ajouter\_ligne}, cde, l, p, q' \rangle . \\ \langle \textbf{Ajouter\_stock}, p, q \rangle \lhd \mathrm{R} \rhd \mathrm{OK} \end{array}}{\begin{array}{c} x.\langle \textbf{Créer\_cde}, cde \rangle . \langle \textbf{Ajouter\_ligne}, cde, l, p, q' \rangle . \\ \langle \textbf{Ajouter\_stock}, p, q \rangle . \langle \textbf{Facturer\_cde}, cde \rangle \lhd \mathrm{R} \rhd \mathrm{OK} \end{array}}$$

- Ajouter plusieurs produits :

$$\text{R21 - } \frac{\begin{array}{c} p \neq p' \wedge \\ x.\langle \textbf{Ajouter\_produit}, p \rangle \lhd \mathrm{R} \rhd \mathrm{OK} \wedge \\ x.\langle \textbf{Ajouter\_produit}, p' \rangle \lhd \mathrm{R} \rhd \mathrm{OK} \end{array}}{x.\langle \textbf{Ajouter\_produit}, p \rangle . \langle \textbf{Ajouter\_produit}, p' \rangle \lhd \mathrm{R} \rhd \mathrm{OK}}$$

A.2.4 Les axiomes de permutation

P0 - Permutation vers la gauche :

$$\frac{x.x' \langle \textbf{Ajouter\_stock}, p, q \rangle . x'' \lhd \mathrm{R} \rhd s}{x.\langle \textbf{Ajouter\_stock}, p, q \rangle . x'.x'' \lhd \mathrm{R} \rhd s}$$

No	Condition	Entrée1	Entrée2
P1	$cde \neq cde'$	⟨**Créer_cde**, cde⟩	⟨**Créer_cde**, cde'⟩
P2	$cde \neq cde'$	⟨**Créer_cde**, cde⟩	⟨**État_cde**, cde'⟩
P3	$cde \neq cde'$	⟨**Créer_cde**, cde⟩	⟨**Facturer_cde**, cde'⟩
P4	$p \neq p'$	⟨**Ajouter_ligne**, $cde,\ l,\ p,\ q$⟩	⟨**Ajouter_ligne**, cde, l', p', q'⟩
P5	$cde \neq cde'$	⟨**Ajouter_ligne**, $cde,\ l,\ p,\ q$⟩	⟨**Facturer_cde**, cde'⟩
P6	$cde \neq cde'$	⟨**Ajouter_ligne**, $cde,\ l,\ p,\ q$⟩	⟨**Ajouter_ligne**, cde', l', p', q'⟩
P7	$cde \neq cde'$	⟨**Facturer_cde**, cde'⟩	⟨**Ajouter_ligne**, $cde,\ l,\ p,\ q$⟩
P8	$cde \neq cde'$	⟨**Facturer_cde**, cde⟩	⟨**État_cde**, cde'⟩
P9		⟨**Ajouter_ligne**, $cde,\ l,\ p,\ q$⟩	⟨**État_cde**, cde ⟩
P10	$cde \neq cde'$	⟨**Ajouter_ligne**, $cde,\ l,\ p,\ q$⟩	⟨**Créer_cde**, cde'⟩
P11	$cde \neq cde'$	⟨**Facturer_cde**, cde⟩	⟨**Facturer_cde**, cde'⟩
P12		⟨**État_cde**,cde⟩	⟨**État_cde**, cde'⟩
P13	$cde \neq cde'$	⟨**Créer_cde**, cde⟩	⟨**Ajouter_ligne**, cde', l', p', q'⟩
P14	$cde \neq cde'$	⟨**Ajouter_ligne**, cde', l', p', q'⟩	⟨**Créer_cde**, cde⟩
P15	$p \neq p'$	⟨**Ajouter_produit**, p⟩	⟨**Ajouter_produit**, p'⟩
P16	$p \neq p'$	⟨**Ajouter_produit**, p ⟩	⟨**Afficher_stock**, p'⟩
P17	$p \neq p'$	⟨ **Ajouter_produit**, p⟩	⟨**Ajouter_stock**, p', q⟩
P18		⟨**Ajouter_stock**, $p,\ q$⟩	⟨**Ajouter_stock**, p', q'⟩
P19		⟨**Ajouter_produit**, p⟩	⟨**Créer_cde**, cde⟩
P20		⟨**Afficher_stock**, p⟩	⟨**État_cde**, cde⟩

Annexe B

La spécification avec la méthode boîte noire par entités

B.1 1^{er} cas

B.1.1 Les espaces d'entrée et de sortie

$I_1 \triangleq$ { **Créer_cde** } × Id_COMMANDE	$O_1 \triangleq$ { OK }
$I_2 \triangleq$ { **Ajouter_ligne**} × Id_COMMANDE × Id_LIGNE × Id_PRODUIT × QUANTITÉ_CDÉE	$O_2 \triangleq$ { OK }
$I_3 \triangleq$ { **Facturer_cde** } × Id_COMMANDE	$O_3 \triangleq$ { OK }
$I_4 \triangleq$ { **État_cde** } × Id_COMMANDE	$O_4 \triangleq$ { INEXISTANTE, EN_ATTENTE, FACTURÉE }

TAB. 34 – *Les espaces d'entrée et de sortie*

Les deux ensembles I et O sont définis par l'union des espaces d'entrée et de sortie (voir tableau 34) comme suit :

$$I \triangleq \bigcup_{i=1}^{4} I_i \text{ et } O \triangleq \bigcup_{i=1}^{4} O_i$$

B.1.2 La définition des entités et la description de leurs comportements individuels

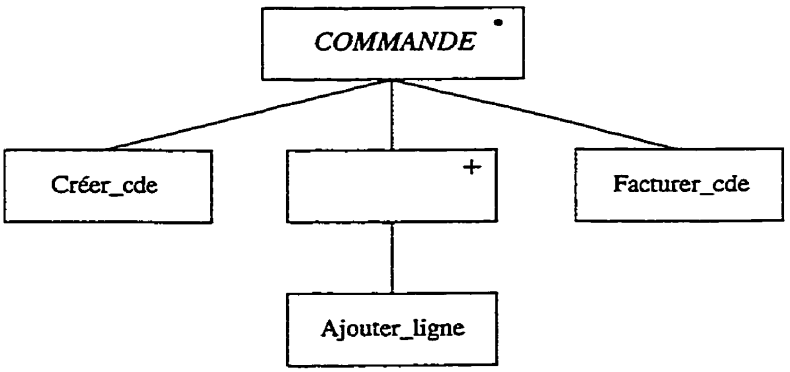


FIG. 14 – *Le diagramme de structure de l'entité COMMANDE*

$$
\begin{aligned}
COMMANDE(cde\_id) &= \langle \textbf{Créer\_cde}, cde\_id \rangle . (\langle \textbf{Ajouter\_ligne}, cde\_id, cde\_ligne, \_, \_ \rangle)^{+} . \\
&\quad \langle \textbf{Facturer\_cde}, cde\_id \rangle \\
REQUETE &= \langle \textbf{État\_cde}, \_ \rangle^{*}
\end{aligned}
$$

où $COMMANDE(cde\_id)$ correspond à la figure 14.

B.1.3 Les contraintes

1. $Produit\_unique(x) \Leftrightarrow$

   $(\forall\ cde, p\_id : instance\_de(cde, x, prefix \circ COMMANDE)$

   $\Rightarrow$

   $\#\ (cde \downarrow \{\ \langle \textbf{Ajouter\_ligne}, \_, \_, p\_id, \_ \rangle\ \}) \leq 1\ )$

2. $Quantité\_suffisante(x) \Leftrightarrow$

   $(\forall\ cde\_id, p\_id, q : instance\_de\_clé(cde, x, COMMANDE, cde\_id) \wedge$

   $\langle \textbf{Ajouter\_ligne}, cde\_id, \_, p\_id, q \rangle \preceq cde$

   $\Rightarrow$

   $Stock(x, p\_id) \geq q\ )$

3. *Stock*(x, *p_id*) =

let

$L = \{\ i \mid \exists\ cde\_id\colon x[i] = \langle$**Ajouter_ligne**, *cde_id*, _, *p_id*, _$\rangle \wedge$

instance_de_clé(_, x, *COMMANDE*, *cde_id*) $\}$

in

return *Quantité_stock*(*p_id*) - $\Sigma_{i \in L}$ *QUNATITÉ_CDÉE*($x[i]$)),

où *Quantité_stock* = *Id_PRODUIT* $\longrightarrow \mathbb{N}$ et *QUANTITÉ_CDÉE* est un attribut de l'entrée **Ajouter_ligne**.

B.1.4 La définition de la relation d'entrée-sortie

1. $\neg$ ($\exists$ *cde*: *instance_de_clé*(*cde*, x, *prefix* $\circ$ *COMMANDE*, *cde_id*))

   $\Rightarrow$

   $x \vdash \langle$**État_cde**, *cde_id*$\rangle \triangleleft$ *FACTURATION* $\triangleright$ INEXISTANTE

2. *instance_de_clé*(*cde*, x, *prefix* $\circ$ *COMMANDE*, *cde_id*) $\wedge$

   label(*last*(*cde_id*)) $\neq$ **Facturer_cde**

   $\Rightarrow$

   $x \vdash \langle$**État_cde**, *cde_id*$\rangle \triangleleft$ *FACTURATION* $\triangleright$ EN_ATTENTE

3. *instance_de_clé*(*cde*, x, *COMMANDE*, *cde_id*)

   $\Rightarrow$

   $x \vdash \langle$**État_cde**, *cde_id*$\rangle \triangleleft$ *FACTURATION* $\triangleright$ FACTURÉE

$I_1 \triangleq$ { **Créer_cde** } × Id_COMMANDE	$O_1 \triangleq$ { OK }
$I_2 \triangleq$ { **Ajouter_ligne**} × Id_COMMANDE × Id_LIGNE × Id_PRODUIT × QUANTITÉ_CDÉE	$O_2 \triangleq$ { OK }
$I_3 \triangleq$ { **Facturer_cde** } × Id_COMMANDE	$O_3 \triangleq$ { OK }
$I_4 \triangleq$ { **État_cde** } × Id_COMMANDE	$O_4 \triangleq$ { INEXISTANTE, EN_ATTENTE, FACTURÉE }
$I_5 \triangleq$ { **Annuler_cde** } × Id_COMMANDE	$O_5 \triangleq$ { OK }
$I_6 \triangleq$ { **Annuler_ligne** } × Id_COMMANDE × Id_LIGNE	$O_6 \triangleq$ { OK }
$I_7 \triangleq$ { **Ajouter_produit** } × Id_PRODUIT	$O_7 \triangleq$ { OK }
$I_8 \triangleq$ { **Ajouter_stock** } × Id_PRODUIT × *QUANTITÉE*	$O_8 \triangleq$ { OK }
$I_9 \triangleq$ { **Afficher_quantité_produit** } × Id_PRODUIT	$O_9 \triangleq \mathbb{N}$

TAB. 35 – *Le nouvel espace d'entrée-sortie*

B.2 2^e Cas

B.2.1 Les espaces d'entrée et de sortie

Les ensembles I et O sont définis par l'union des espaces d'entrée et de sortie (voir tableau 35) comme suit :

$$I \triangleq \bigcup_{i=1}^{9} I_i \text{ et } O \triangleq \bigcup_{i=1}^{9} O_i$$

B.2.2 La définition des entités et la description de leurs comportements individuels

Les représentations formelles (sous forme des séquences d'entrée) des figures 15, 16 et 17 (les figures 16 et 17 sont dans la page 138) sont les suivantes.

$$
\begin{aligned}
\textit{COMMANDE}(\textit{cde\_id}) &= (\textit{Commande\_annulée}(\textit{cde-id}))^* \,.\, \textit{Commande\_active}(\textit{cde\_id}) \\
\textit{Commande\_active}(\textit{cde\_id}) &= \langle \textbf{Créer\_cde}, \textit{cde\_id}\rangle.(|L \in F(\text{ID\_LIGNE})(|||_{l\_id \in L}\ (\textit{LIGNE\_A}(\textit{cde\_id}, \textit{Lid})))). \\
&\quad \langle \textbf{Facturer\_cde}, \textit{cde\_id}\rangle \\
\textit{Commande-annulée}(\textit{cde\_id}) &= \langle \textbf{Créer\_cde}, \textit{cde\_id}\rangle.(|L \in F(\text{ID\_LIGNE})(|||_{l\_id \in L}\ (\textit{LIGNE\_D}(\textit{cde\_id}, \textit{Lid})))). \\
&\quad \langle \textbf{Annuler\_cde}, \textit{cde\_id}\rangle \\
\textit{PRODUIT}(\textit{p\_id}) &= \langle \textbf{Ajouter\_produit}, \textit{p-id}\ \rangle.(\langle \textbf{Ajouter-stock}, \textit{p-id}, \_\rangle \mid \langle \textbf{Ajouter\_ligne}, \_, \_, \textit{p\_id}, \_\rangle)^* \\
\textit{LIGNE}(\textit{cde\_id}, \textit{Lid}) &= (\textit{Ligne\_annulée}(\textit{cde\_id}, \textit{Lid}\ ))^*.\langle \textbf{Ajouter\_ligne}, \textit{cde\_id}, \textit{Lid}, \_, \_\rangle \\
\textit{LIGNE\_A}(\textit{cde\_id}, \textit{Lid}) &= (\textit{Ligne\_annulée}(\textit{cde\_id}, \textit{Lid}\ ))^*.(\textit{Ligne\_active}(\textit{cde\_id}, \textit{Lid})) \\
\textit{LIGNE\_D}(\textit{cde\_id}, \textit{Lid}) &= (\textit{Ligne\_annulée}(\textit{cde\_id}, \textit{Lid}))^* \\
\textit{Ligne\_active}(\textit{cde\_id}, \textit{Lid}) &= \langle \textbf{Ajouter\_ligne}, \textit{cde\_id}, \textit{Lid}, \_, \_\rangle \\
\textit{Ligne\_annulée}(\textit{cde\_id}, \textit{Lid}) &= \langle \textbf{Ajouter\_ligne}, \textit{cde\_id}, \textit{Lid}, \_\ \_\rangle.\langle \textbf{Annuler\_ligne}, \textit{cde\_id}, \textit{Lid}\rangle \\
\textit{REQUETE} &= (\langle \textbf{État\_cde}, \_\rangle \mid \langle \textbf{Afficher\_quantité\_produit}, \_\rangle)^*
\end{aligned}
$$

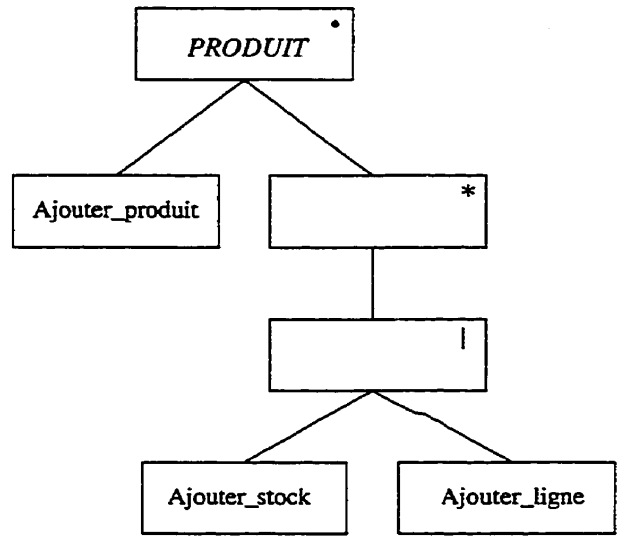


FIG. 15 – *Le diagramme de structure de l'entité PRODUIT*

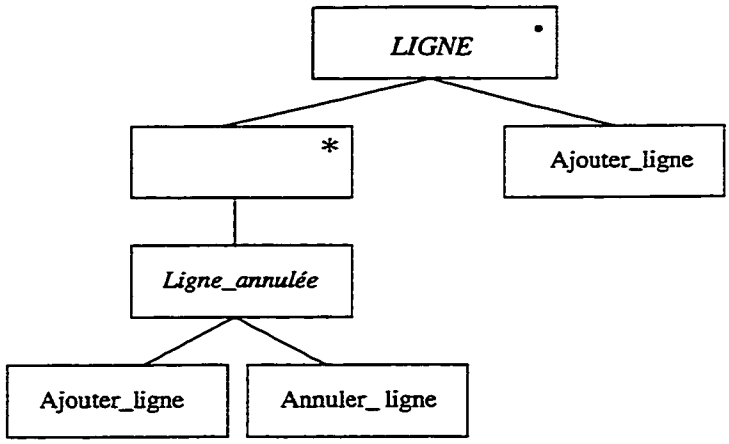


FIG. 16 – *Le diagramme de structure de l'entité LIGNE*

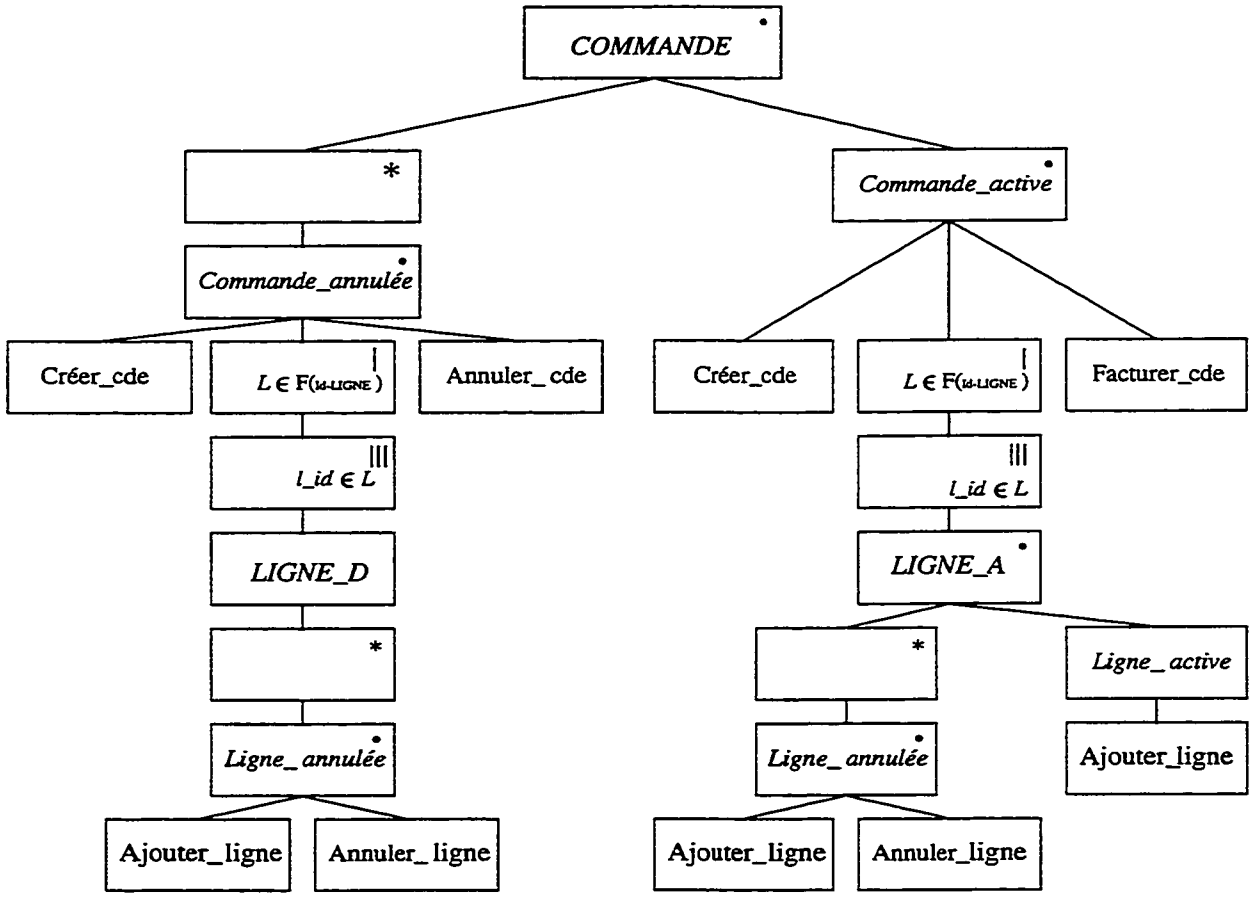


FIG. 17 – *Le diagramme de structure de l'entité COMMANDE*

B.2.3 Les contraintes

1. *Produit_unique* $(x) \Leftrightarrow$

   $(\forall$ *cde, p_id*: *instance_de* (*cde, x, prefix* $\circ$ *COMMANDE*)

   $\Rightarrow$

   (# $(cde \downarrow \{ \langle$**Ajouter_ligne**, _, *l_id*, _, _$\rangle\})$ - # $(cde \downarrow \{ \langle$**Annuler_ligne**, _, *l_id*$\rangle\})$

   $) \leq 1$)

2. *Quantité_suffisante*$(x) \Leftrightarrow$

   $(\forall$ *cde_id, p_id, q,*: *instance_de_clé*$(l,$ *x, LIGNE,*$(cde\_id,$ *l_id*$))$ $\wedge$

   $last(l) = \langle$**Ajouter_ligne**, *l_id*, _, *p_id, q* $\rangle$

   $\Rightarrow$

   Stock$(x,$ *p_id* $) \geq q)$

   avec *Stock*$(x,$ *p_id*$) =$

   if $\exists p, a, L$:

   instance_de_clé$(p,$ *x, PRODUIT, p_id*$)$

   $a = p \downarrow\downarrow \{$ **Ajouter_stock** $\}$

   $L = \{ c \mid \exists$ *l*: *instance_de_clé*$(l,$ *x, prefix* $\circ$ *LIGNE, l_id* $)$ $\wedge$

   $last(l) = \langle$**Ajouter_ligne**, *cde_id, l_id, p_id,* _$\rangle$ $\wedge$

   instance_de_clé(-, *x, COMMANDE, cde_id* $)\}$

   then

   return $(\Sigma_{i=1}^{\#a}$ *QUANTITÉ*$(a\ [i])$ - $\Sigma_{c \in L}$ *QUANTITÉ_CDÉE*(c),

   où *QUANTITÉ* et *QUANTITÉ_CDÉE* sont respectivement les attributs des entrées **Ajouter_stock** et **Ajouter_ligne**.

B.2.4 La définition de la relation d'entrée-sortie

1. $\neg$ ($\exists$ *cde*: *instance_de_clé*(*cde*, *x*, *prefix* $\circ$ *COMMANDE*, *cde_id*))

   $\Rightarrow$

   $x \vdash \langle$**État_cde**, *cde_id*$\rangle \triangleleft$ *FACTURATION* $\triangleright$ INEXISTANTE

2. *instance_de_clé*(*cde*, *x*, *prefix* $\circ$ *COMMANDE*, *cde_id*) $\wedge$

   label(*last*(*cde_id*)) $\neq$ **Facturer_cde**

   $\Rightarrow$

   $x \vdash \langle$**État_cde**, *cde_id*$\rangle \triangleleft$ *FACTURATION* $\triangleright$ EN_ATTENTE

3. *instance_de_clé*(*cde*, *x*, *COMMANDE*, *cde_id*)

   $\Rightarrow$

   $x \vdash \langle$**État_cde**, *cde_id*$\rangle \triangleleft$ *FACTURATION* $\triangleright$ FACTURÉE

4. $x \vdash \langle$**Ajouter_produit**, *p_id*$\rangle \triangleleft$ *FACTURATION* $\triangleright$ OK

5. $x \vdash \langle$**Afficher_quantité_produit**, *p_id*$\rangle \triangleleft$ *FACTURATION* $\triangleright$ *Stock*(*x*, *p_id*)

Annexe C

La spécification avec la méthode B

C.1 1^{er} cas

MACHINE
Produit1
SETS
$PRODUIT$
VARIABLES
$Produit$, $Quantité\_en\_stock$
INVARIANT
$Produit \subseteq PRODUIT \wedge$
$Quantité\_en\_stock \in Produit \longrightarrow \text{NAT}$
INITIALISATION
$Produit := \emptyset \;||$
$Quantité\_en\_stock := \emptyset$
OPERATIONS
Supprimer_stock ($stock$) =
PRE
$stock \in Produit \leftrightarrow \text{NAT} \wedge$
$\forall\, pp.\ (pp \in \mathbf{dom}(stock) \Rightarrow$
$Quantité\_en\_stock(pp) \geq stock(pp)$
THEN
$Quantité\_en\_stock := Quantité\_en\_stock \Leftarrow$
$(\lambda\, pp.\ (pp \in \mathbf{dom}(stock) \mid$
$Quantité\_en\_stock(pp) - stock(pp)))$
END ;

TAB. 36 – *Machine* **Produit1**

MACHINE

Facturation1

SETS

$COMMANDE$;

$LIGNE$;

$ÉTAT = \{$ INEXISTANTE EN_ATTENTE, FACTURÉE $\}$;

$RÉPONSE = \{$ OK, ERREUR $\}$;

INCLUDES

Produit1

VARIABLES

Ligne, Produit_réf, Quantité_commandée,
Commande, Commande_de, État

INVARIANT

$Ligne \subseteq LIGNE \wedge$
$Commande \subseteq COMMANDE \wedge$
$Commande\_de \in Ligne \longrightarrow Commande \wedge$
$Produit\_réf \in Ligne \longrightarrow Produit \wedge$
$Quantité\_commandée \in Ligne \longrightarrow \text{NAT} \wedge$
$État \in Commande \longrightarrow ÉTAT \wedge$
$(Commande\_de \otimes Produit\_réf) \in (Ligne \rightarrowtail (Produit \times Commande))$

INITIALISATION

$Ligne := \emptyset \parallel$
$Commande := \emptyset \parallel$
$Commande\_de := \emptyset \parallel$
$Produit\_réf := \emptyset \parallel$
$Quantité\_commandée := \emptyset \parallel$
$État := \emptyset$

OPERATIONS

$réponse \longleftarrow$ **Facturer_cde**$(cc) =$

PRE

$cc \in COMMANDE$

THEN

IF

$État(cc) = \text{EN\_ATTENTE} \wedge$
$\mathbf{card}(Commande\_de^{-1}[\{ cc \}] > 0 \wedge$
$\forall ll.\ (ll \in Ligne \wedge Commande\_de(ll) = cc \Rightarrow$
$Quantité\_en\_stock(produit\_réf(ll)) \geq Quantité\_commandée(ll)$

THEN

Supprimer_stock((*produit_réf* ⊗ *Quantité_commandée*)
[*Commande_de*$^{-1}$[{*cc*}]]) ||
État(*cc*) := FACTURÉE ||
réponse := OK
ELSE
réponse := ERREUR
END
END ;
réponse ⟵ **État_cde**(*cc*) =
PRE
cc ∈ *Commande*
THEN
réponse := *État*(*cc*)
END ;
END

TAB. 37 – *Machine* **Facturation1**

C.2 2^e Cas

MACHINE
Produit2
SETS
PRODUIT
VARIABLES
Produit, Quantité_en_stock
INVARIANT
$Produit \subseteq PRODUIT \wedge$
$Quantité\_en\_stock \in Produit \longrightarrow \text{NAT}$
INITIALISATION
$Produit := \emptyset \parallel$
$Quantité\_en\_stock := \emptyset$
OPERATIONS
Supprimer_stock (*stock*) =
PRE
$stock \in Produit \nrightarrow \text{NAT} \wedge$
$\forall pp.\ (pp \in \mathbf{dom}(stock) \Rightarrow$
$Quantité\_en\_stock(pp) \geq stock(pp)$
THEN
$Quantité\_en\_stock := Quantité\_en\_stock \Leftarrow$
$(\lambda pp.\ (pp \in \mathbf{dom}(stock) \mid$
$Quantité\_en\_stock(pp) - stock(pp)))$
END ;
Ajouter_stock(*stock*) =
PRE
$stock \in Produit \nrightarrow \text{NAT} \wedge$
$(\forall pp.\ (\ pp \in \mathbf{dom}(stock) \Rightarrow$
$Quantité\_en\_stock + stock(pp) \leq \text{MAXINT}))$
THEN
$Quantité\_en\_stock := Quantité\_en\_stock \Leftarrow$
$(\lambda pp.(\ pp \in \mathbf{dom}(stock) \mid$
$Quantité\_en\_stock(pp) + stock(pp)))$
END
END

TAB. 38 – *Machine* **Produit2**

MACHINE

Facturation2

SETS

$COMMANDE$;

$LIGNE$;

$ÉTAT = \{$ INEXISTANTE, EN_ATTENTE, FACTURÉE $\}$;

DEFINITIONS

$items(cc) = Commande\_de^{-1}\ [\{\ cc\ \}]$

INCLUDES

Produit2

VARIABLES

*les mêmes variables que la machine **Facturation1***

INVARIANT

*les mêmes invariants que la machine **Facturation1***

INITIALISATION

*la même initialisation que la machine **Facturation1***

OPERATIONS

Créer_cde=

PRE

$Commande \neq COMMANDE$

THEN

ANY cc WHERE

$cc \in COMMANDE$ - $Commande$

THEN

$Commande := Commande \cup \{cc\ \} \parallel$

$État(cc) :=$ EN_ATTENTE

END

END;

réponse $\longleftarrow$ **Ajouter_ligne** $(cc,\ pp,\ qq\ ) =$

PRE

$qq \in$ NAT $\wedge$

$cc \in COMMANDE \wedge$

$pp \in PRODUIT$

THEN

IF

$(pp,\ cc) \notin \mathbf{ran}(Commande\_de \otimes Produit\_réf) \wedge$

$Ligne \neq LIGNE \wedge$

$État(cc\ ) =$ EN_ATTENTE

```
      THEN
         ANY ll WHERE
            ll ∈ LIGNE - Ligne
         THEN
            Produit_réf (ll ) := pp ||
            Commande_de (ll ) := cc ||
            Ligne := Ligne ∪ {ll } ||
            Quatité_commandée (ll ) := qq ||
            réponse := OK
         END
      ELSE
         réponse := ERREUR
      END
END ;
réponse ⟵ Annuler_cde(cc) =
PRE
      cc ∈ COMMANDE
THEN
      IF
         État(cc) = EN_ATTENTE
      THEN
         Commande := Commande - {cc } ||
         État := {cc} ⩤ État ||
         Ligne := Ligne - items(cc) ||
         Commande_de :=items(cc) ⩤ Commande_de ||
         Produit_réf := items(cc) ⩤ Produit_réf ||
         Quatité_commandée := items(cc) ⩤ Quatité_commandée ||
         réponse := OK
      ELSE
         réponse := ERREUR
      END
réponse ⟵ Annuler_ligne(ll) =
PRE
      ll ∈ LIGNE
THEN
      IF
         État (Commande_de(cc)) = EN_ATTENTE
      THEN
         Ligne := Ligne - {ll } ||
```

$Commande\_de := \{ll\} \lhd Commande\_de \parallel$
$Produit\_réf := \{ll\} \lhd Produit\_réf \parallel$
$Quantité\_commandée := \{ll\} \lhd Quantité\_commandée \parallel$
$réponse :=$ OK
ELSE
$réponse :=$ ERREUR
END
END ;
END ;
$réponse \longleftarrow$ **Facturer_cde**$(cc) =$
PRE
$cc \in COMMANDE$
THEN
IF
$État(cc) =$ EN_ATTENTE $\wedge$
card$(Commande\_de^{-1}[\{ cc \}] > 0 \wedge$
$\forall ll.\ (\ ll \in Ligne \wedge Commande\_de(ll) = cc \Rightarrow$
$Quantité\_en\_stock(produit\_réf(ll)) \geq Quantité\_commandée(ll)$
THEN
Supprimer_stock$((produit\_réf \otimes Quantité\_commandée)$
$[Commande\_de^{-1}[\{cc\}]]) \parallel$
$État(cc) :=$ FACTURÉE $\parallel$
$réponse :=$ OK
ELSE
$réponse :=$ ERREUR
END
END ;
$réponse \longleftarrow$ **État_cde**$(cc) =$
PRE
$cc \in Commande$
THEN
$réponse := État(cc)$
END ;
END

TAB. 39 – *Machine* **Facturation2**

Annexe D

La spécification avec la méthode des assertions de traces

D.1 1^{er} cas

D.1.1 La syntaxe

Nom de programme	1^{er} argument	2^{e} argument	Valeur
État_cde	$< COMMANDE >$		E
Facturation_cde	$< COMMANDE >$	$<$ *Booléen* $>$	*Réponse*
Facturer_cde	$< COMMANDE >$		($COMMANDE$, *Quantité_Cdée*) ou *Réponse*

TAB. 40 – *La syntaxe des programmes d'accès du module COMMANDE*

Nom de programme	1^{er} argument	2^{e} argument	Valeur
Supprimer_stock	$< COMMANDE >$	$<$ *Quantité_Cdée* $>$	($COMMANDE$, *Booléen*)

TAB. 41 – *La syntaxe des programmes d'accès du module PRODUIT*

D.1.2 Les traces canoniques

- La trace canonique du module *COMMANDE* est définie comme suit :

$$T_{C_c} = \_ \vee T_{C_F}$$

où T_{C_F} est définie comme suit :

$$T_{C_F} = [\textbf{Facturation\_cde}(cde_i, \text{ vrai})]_{i=1}^{n},$$

où $cde_i \in COMMANDE$

- La trace canonique du module *PRODUIT* est définie comme suit :

$$T_{C_p} = \_ \vee T_{C_S}$$

où T_{C_S} est définie comme suit :

$$T_{C_S} = [\textbf{Supprimer\_stock}(cde_i, \textit{Quantité\_Cdée})]_{i=1}^{n}$$

où $cde_i \in COMMANDE$

D.1.3 Les équivalences de traces

T.**État_cde**(*cde*) $=_r$

Patron de trace	Trace équivalente
vrai	T

TAB. 42 – *Trace équivalente à* T étendue par **État_cde**

T.**Facturation_cde**(*cde*, *Quantité_suffisante*) $=_r$

Condition	Patron de trace	Trace équivalente
vrai	$T = X$.**Facturation_cde**(*cde*, vrai).Y	$T \wedge$ % COMMANDE_FACTURÉE %
Quantité_suffisante = vrai	$\neg\exists X, Y$: $T = X$.**Facturation_cde**(*cde*, vrai).Y	T.**Facturation_cde**(*cde*, vrai)
Quantité_suffisante = faux	$\neg\exists X, Y$: $T = X$.**Facturation_cde**(*cde*, vrai).Y	$T \wedge$ % STOCK_INSUFFISANT %

TAB. 43 – *Trace équivalente à* T étendue par **Facturation_cde**

T.**Facturer_cde**(*cde*) $=_r$

Patron de trace	Trace équivalente
vrai	T

TAB. 44 – *Trace équivalente à* T étendue par **Facturer_cde**

T.**Supprimer_stock**(*cde*, *Quantité_Cdée*) $=_\tau$

Condition	Patron de trace	Trace équivalente
Quantité_suffisante(T, *Quantité_Cdée*) = vrai	vrai	T. **Supprimer_stock**(*cde*, *Quantité_Cdée*)
Quantité_suffisante(T, *Quantité_Cdée*) = faux	vrai	$T \wedge$ % QUANTITÉ_INSUFFISANTE %

TAB. 45 – *Trace équivalente à* T étendue par **Supprimer_stock**

D.1.4 Les valeurs de sortie

O (T, **État_cde**(*cde*)) =

Condition	Patron de trace	Valeur	Port
$cde \notin$ *COMMANDE_CRÉÉE*	vrai	INEXISTANTE	CE
$cde \in$ *COMMANDE_CRÉÉE*	$\neg\exists X, Y$: $T = X$.**Facturation_cde**(*cde*, vrai).Y	EN_ATTENTE	CE
$cde \in$ *COMMANDE_CRÉÉE*	$T = X$.**Facturation_cde**(*cde*, vrai).Y	FACTURÉE	CE

TAB. 46 – *Valeurs de sortie de* T étendue par **État_cde**

$O(T$,**Facturation_cde**(*cde*, *Quantité_suffisante*)) =

Condition	Patron de trace	Sortie	Port
vrai	$T = X$.**Facturation_cde**(*cde*, vrai).Y	ÉCHEC	CE
Quantité_suffisante = vrai	$\neg\exists X, Y$: $T = X$.**Facturation_cde**(*cde*, vrai).Y	OK	CE
Quantité_suffisante = faux	$\neg\exists X, Y$: $T = X$.**Facturation_cde**(*cde*, vrai).Y	ÉCHEC	CE

TAB. 47 – *Valeurs de sortie de* T étendue par **Facturation_cde**

$O(T$,**Facturer_cde**(*cde*)) =

Condition	Patron de trace	Sortie	Port
cde $\notin$ *COMMANDE_CRÉÉE*	vrai	ÉCHEC	CE
cde $\in$ *COMMANDE_CRÉÉE*	$\neg\exists X, Y$: $T = X$.**Facturation_cde**(*cde*, vrai).Y	(*cde*, *Ligne*(*cde*))	CP
cde $\in$ *COMMANDE_CRÉÉE*	$T = X$.**Facturation_cde**(*cde*, vrai).Y	ÉCHEC	CE

TAB. 48 – *Valeurs de sortie de* T étendue par **Facturer_cde**

$O(T$,**Supprimer_stock**(*cde*), *Qté_Cdée*) =

Condition	Patron de trace	Valeur	Port
Quantité_suffisante(T, *Quantité_Cdée*) = vrai	vrai	(*cde*, vrai)	PC
Quantité_suffisante(T, *Quantité_Cdée*) = faux	vrai	(*cde*, faux)	PC

TAB. 49 – *Valeurs de sortie de* T étendue par **Supprimer_stock**

D.1.5 Le dictionnaire

1. *COMMANDE* est l'ensemble de toutes les commandes du système ;
2. $E = \{$ INEXSISTANTE, EN_ATTENTE, FACTURÉE $\}$;
3. $Ligne \in COMMANDE \longrightarrow Quantité\_Cdée$;
4. *PRODUIT* est l'ensemble de tous les produits du système ;
5. $Quantité\_Cdée = PRODUIT \nrightarrow \mathbb{N}$;
6. $Quantité\_en\_stock \in T_{C_p} \times PRODUIT \longrightarrow \mathbb{N}$;

   $Quantité\_en\_stock(T.\textbf{Supprimer\_stock}(cde,\ Qc),\ p) =$

   If $p \in \textbf{dom}(Qc)$

   then

   return $Quantité\_en\_stock(T,\ p)$ - $Qc(p)$

   else

   return $Quantité\_en\_stock(T,\ p)$;

   $Quantité\_en\_stock(\_,\ p) = Stock\_initial(p)$;
7. $Stock\_initial \in PRODUIT \longrightarrow \mathbb{N}$ est la quantité initiale en stock du système ;
8. $Quantité\_suffisante \in T_{C_p} \times Quantité\_Cdée \longrightarrow Booléen$

   $Quantité\_suffisante(T,\ Qc)$

   $\Leftrightarrow$

   $\forall\ cde,\ p$: $(cde,\ p) \in \textbf{dom}(Qc)$

   $\Rightarrow$

   $Quantité\_en\_stock(T,\ p) \geq Qc(p)$;
9. $Réponse = \{$ OK, ÉCHEC $\}$.

D.2 2^e Cas

D.2.1 La syntaxe

Nom de programme	1^{er} arg	2^e arg	3^e arg	Valeur
Ajouter_ligne	$< COMMANDE >$	$< PRODUIT >$	$< \mathbb{N} >$	*Réponse*
Annuler_cde	$< COMMANDE >$			*Réponse*
Annuler_ligne	$< COMMANDE >$	$< PRODUIT >$		*Réponse*
Créer_cde	$< COMMANDE >$			*Réponse*
État_cde	$< COMMANDE >$			*E*
Facturation_cde	$< COMMANDE >$	$<$ *Booléen* $>$		*Réponse*
Facturer_cde	$< COMMANDE >$			(*COMMANDE*, *Quantité_Cdée*) ou *Réponse*

TAB. 50 – *La syntaxe des programmes d'accès du module COMMANDE*

Nom de programme	1^{er} argument	2^e argument	Valeur
Ajouter_produit	$< PRODUIT >$		*Réponse*
Ajouter_stock	$< PRODUIT >$	$< \mathbb{N} >$	*Réponse*
Supprimer_stock	$< COMMANDE >$	$<$ *Quantité_Cdée* $>$	(*Char*, *Booléen*)

TAB. 51 – *La syntaxe des programmes d'accès du module PRODUIT*

D.2.2 Les traces canoniques

- La trace canonique du module *COMMANDE* est définie comme suit :

$$T_{C_c} = T_C.T_I.T_F,$$

1. $T_C = \_ \vee T_{C'}$, où $T_{C'} = [\textbf{Créer\_cde}(cde_i)]_{i=1}^{n}$
   où n est le nombre maximal de commandes du système.
2. $T_I = \_ \vee T_{I'}$, où $T_{I'} = [\textbf{Ajouter\_ligne}(cde_i, p_j, q_j)]_{i=1,\ j=1}^{n,m}$
   où m est le nombre maximal de lignes dans la commande cde_i.
3. $T_{C_F} = \_ \vee T_{C_{F'}}$, où $T_{C_{F'}} = [\textbf{Facturation\_cde}(cde_i, \text{vrai})]_{i=1}^{n}$.

- La trace canonique du module *PRODUIT* est définie comme suit :

$$T_{C_p} = T_{C_{Ap}}.T_{C_{As}}$$

1. $T_{C_{Ap}} = \_ \vee T_{C_{Ap'}}$, où $T_{C_{Ap'}} = [\textbf{Ajouter\_produit}(p_i)]_{i=1}^{n}$;
2. $T_{C_{As}} = \_ \vee T_{C_{As'}}$, où $T_{C_{As'}} = [\textbf{Ajouter\_stock}(p_i, q_j)]_{i=1,\ j=1}^{n,m}$.

D.2.3 Les équivalences de traces

$T.\textbf{Facturer\_cde}(cde) =_\tau$

Patron de trace	Trace équivalente
vrai	T

T.**Ajouter_ligne**(cde, p, q) $=_r$

Condition	Patron de trace	Trace équivalente
$cde \notin COMMANDE(T)$	vrai	$T \wedge$ $T \wedge$ % COMMANDE_NON_TROUVÉE %
vrai	$T = X$.**Facturation_cde**(cde,vrai).Y	$T \wedge$ % COMMANDE_FACTURÉE %
vrai	$T = X$.**Ajouter_ligne**(cde, p, q')	$T \wedge$ % LIGNE_AJOUTÉE %
vrai	$\neg\exists X, Y : T = X$.**Ajouter_ligne**(cde, p, q).$Y \vee$ $T = X$.**Facturation_cde**(cde,vrai).Y	T.**Ajouter_ligne**(cde, p, q)

TAB. 52 – *Trace équivalente à* T étendue par **Ajouter_ligne**

T.**Ajouter_produit**(p) $=_r$

Condition	Trace équivalente
$p \in PRODUIT(T)$	$T \wedge$ % PRODUIT_AJOUTÉ %
$p \notin PRODUIT(T)$	T.**Ajouter_produit**(p)

TAB. 53 – *Trace équivalente à* T étendue par **Ajouter_produit**

T.**Ajouter_stock**(p, q) $=_r$

Condition	Trace équivalente
$p \notin PRODUIT(T)$	$T \wedge$ % PRODUIT_NON_TROUVÉ %
$p \in PRODUIT(T)$	T.**Ajouter_stock**(p, q)

TAB. 54 – *Trace équivalente à* T étendue par **Ajouter_stock**

T.**Annuler_cde**(cde) $=_r$

Condition	Patron de trace	Trace équivalente
$cde \notin COMMANDE(T)$	vrai	$T \wedge$ % COMMANDE_NON_TROUVÉE %
$cde \in COMMANDE(T)$	$T = X$.**Facturation_cde**(cde, vrai).Y	$T \wedge$ % COMMANDE_FACTURÉE %
$cde \in COMMANDE(T)$	$\neg\exists X, Y$: $T = X$.**Facturation_cde**(cde, vrai).$Y \wedge$ $T = X$.**Créer_cde**(cde).Y	$X.Y$

TAB. 55 – *Trace équivalente à* T étendue par **Annuler_cde**

T.**Annuler_ligne**(cde, p, q) $=_r$

Condition	Patron de trace	Trace équivalente
$cde \notin COMMANDE(T)$	vrai	$T \wedge$ % COMMANDE_NON_TROUVÉE %
$cde \in COMMANDE(T)$	$T = X$.**Facturation_cde**(cde, vrai).Y	$T \wedge$ % COMMANDE_FACTURÉE %
$cde \in COMMANDE(T)$	$\neg\exists X, Y : T = X.$ **Ajouter_ligne**(cde, p, q).Y	$T \wedge$ % LIGNE_NON_TROUVÉ %
$cde \in COMMANDE(T)$	$T = X.$ **Ajouter_ligne**(cde, p, q).Y	$X.Y$

TAB. 56 – *Trace équivalente à* T étendue par **Annuler_ligne**

T.**Créer_cde**(cde) $=_r$

Condition	Patron de trace	Trace équivalente
$cde \in COMMANDE(T)$	vrai	$T \wedge$ % COMMANDE_CRÉÉE %
$cde \notin COMMANDE(T)$	vrai	T.**Créer_cde**(cde)

TAB. 57 – *Trace équivalente à* T étendue par **Créer_cde**

T.**État_cde**(cde) $=_r$

Patron de trace	Trace équivalente
vrai	T

TAB. 58 – *Trace équivalente à* T étendue par **État_cde**

T.**Facturation_cde**(cde, *Quantité_suffisante*) $=_r$

Condition	Patron de trace	Trace équivalente
vrai	$T = X$.**Facturation_cde**(cde, vrai).Y	$T \wedge$ % COMMANDE_FACTURÉE %
Quantité_suffisante = vrai	$\neg\exists X, Y$: $T = X$.**Facturation_cde**(cde, vrai).Y	T.**Facturation_cde**(cde, vrai)
Quantité_suffisante = faux	$\neg\exists X, Y$: $T = X$.**Facturation_cde**(cde, vrai).Y	$T \wedge$ % STOCK_INSUFFISANT %

TAB. 59 – *Trace équivalente à* T étendue par **Facturation_cde**

T.**Supprimer_stock**(cde, *Quantité_Cdée*) $=_r$

Condition	Patron de trace	Trace équivalente
Quantité_suffisante(T, *Quantité_Cdée*) = vrai	vrai	T. **Supprimer_stock**(cde, *Quantité_Cdée*)
Quantité_suffisante(T, *Quantité_Cdée*) = faux	vrai	$T \wedge$ % QUANTITÉ_INSUFFISANTE %

D.2.4 Les valeurs de sortie

$O(T,$ **Ajouter_ligne**$(cde,\ p,\ q)) =$

Condition	Patron de trace	Valeur	Port
$cde \notin COMMANDE(T)$	vrai	ÉCHEC	CE
vrai	$T = X.$**Facturation_cde**$(cde,$vrai$).Y$	ÉCHEC	CE
vrai	$T = X.$**Ajouter_ligne**$(cde,\ p,\ q').Y$	ÉCHEC	CE
$cde \in COMMANDE(T)$	$\neg\exists X, Y : T = X.$**Ajouter_ligne**$(cde,\ p,\ q).Y$	OK OK	CE CE

TAB. 60 – *Valeurs de sortie de* T étendue par **Ajouter_ligne**

$O(T,$**Ajouter_produit**$(p)) =$

Condition	Valeur	Port
$p \in PRODUIT(T)$	ÉCHEC	CE
$p \notin PRODUIT(T)$	OK	CE

TAB. 61 – *Valeurs de sortie de* T étendue par **Ajouter_produit**

$O(T,$ **Ajouter_stock**$(p,\ q)) =$

Condition	Valeur	Port
$p \notin PRODUIT(T)$	ÉCHEC	CE
$p \in PRODUIT(T)$	OK	CE

TAB. 62 – *Valeurs de sortie de* T étendue par **Ajouter_stock**

$O(T,$ **Annuler_cde**$(cde)) =$

Condition	Patron de trace	Valeur	Port
$cde \notin COMMANDE(T)$	vrai	ÉCHEC	CE
$cde \in COMMANDE(T)$	$T = X.$**Facturation_cde**$(cde,$ vrai$).Y$	ÉCHEC	CE
$cde \in COMMANDE(T)$	$\neg\exists X, Y :$ $T = X.$**Facturation_cde**$(cde,$ vrai$).Y \wedge$ $T = X.$**Créer_cde**$(cde).Y$	OK	CE

TAB. 63 – *Valeurs de sortie de* T étendue par **Annuler_cde**

$O(T,$ **Annuler_ligne**$(cde,\ p,\ q)) =$

Condition	Patron de trace	Valeur	Port
$cde \notin COMMANDE(T)$	vrai	ÉCHEC	CE
$cde \in COMMANDE(T)$	$T = X.$ **Facturation_cde**$(cde,$ vrai$).Y$	ÉCHEC	CE
$cde \in COMMANDE(T)$	$\neg\exists X, Y : T = X.$ **Ajouter_ligne**$(cde,\ p,\ q).Y$	ÉCHEC	CE
$cde \in COMMANDE(T)$	$T = X.$ **Ajouter_ligne**$(cde,\ p,\ q).Y$	OK	CE

TAB. 64 – *Valeurs de sortie de* T étendue par **Annuler_ligne**

$O(T,$ **Créer_cde**$(cde)) =$

Condition	Patron de trace	Valeur	Port
$cde \in COMMANDE(T)$	vrai	ÉCHEC	CE
$cde \notin COMMANDE(T)$	vrai	OK	CE

TAB. 65 – *Valeurs de sortie de* T étendue par **Créer_cde**

O (T, **État_cde**(cde)) =

Condition	Patron de trace	Valeur	Port
$cde \notin COMMANDE(T)$	vrai	INEXISTANTE	CE
$cde \in COMMANDE(T)$	$\neg\exists X, Y$: $T = X$.**Facturation_cde**(cde, vrai).Y	EN_ATTENTE	CE
$cde \in COMMANDE(T)$	$T = X$.**Facturation_cde**(cde, vrai).Y	FACTURÉE	CE

TAB. 66 – *Valeurs de sortie de T* étendue par **État_cde**

$O(T$,**Facturation_cde**(cde, *Quantité_suffisante*)) =

Condition	Patron de trace	Sortie	Port
vrai	$T = X$.**Facturation_cde**(cde, vrai).Y	ÉCHEC	CE
Quantité_suffisante = vrai	$\neg\exists X, Y$: $T = X$.**Facturation_cde**(cde, vrai).Y	OK	CE
Quantité_suffisante = faux	$\neg\exists X, Y$: $T = X$.**Facturation_cde**(cde, vrai).Y	ÉCHEC	CE

TAB. 67 – *Valeurs de sortie de T* étendue par **Facturation_cde**

$O(T$,**Facturer_cde**(cde)) =

Condition	Patron de trace	Sortie	Port
$cde \notin COMMANDE(T)$	vrai	ÉCHEC	CE
$cde \in COMMANDE(T)$	$\neg\exists X, Y$: $T = X$.**Facturation_cde**(cde, vrai).Y	(cde, *Ligne*(T, (cde))	CP
$cde \in COMMANDE(T)$	$T = X$.**Facturation_cde**(cde, vrai).Y	ÉCHEC	CE

TAB. 68 – *Valeurs de sortie de T* étendue par **Facturer_cde**

$O(T$,**Supprimer_stock**(cde), *Quantité_cdée*$) =$

Condition	Patron de trace	Valeur	Port
Quantité_suffisante$(T$, *Quantité_Cdée*$)$ = vrai	vrai	(*cde*, vrai)	PC
Quantité_suffisante$(T$, *Quantité_Cdée*$)$ = faux	vrai	(*cde*, faux)	PC

TAB. 69 – *Valeurs de sortie de* T étendue par **Supprimer_stock**

D.2.5 Le dictionnaire

1. *COMMANDE*$(T.E) =$

   if $E =$ **Créer_cde**(cde)

   then

   if $cde \notin$ *COMMANDE*(T)

   then

   return *COMMANDE*$(T) \cup \{$ *cde* $\}$

   else

   return *COMMANDE*(T)

   endif

   endif

   endif

   COMMANDE$(\_) = \emptyset$;

2. $E = \{$ INEXSISTANTE, EN_ATTENTE, FACTURÉE $\}$;

3. *Ligne* $\in T_{C_c} \times$ *COMMANDE* $\longrightarrow$ *Quantité_Cdée*;

   Ligne$(T.E,$ *cde*$) =$

   Case of:

   $E =$ **Ajouter_ligne**(cde, p, q)

   then

   if $(p, q) \notin$ *Ligne*$(T,$ *cde*$)$

then

return *Ligne*(T, *cde*) $\cup$ { (p, q) }

else

return *Ligne*(T, *cde*)

endif

endcase

Ligne(_, cde) $= \emptyset$;

4. *PRODUIT*($T.E$) =

if $E =$ **Ajouter_produit**(p)

then

return *PRODUIT*(T) $\cup$ { *cde* }

else

return *PRODUIT*(T)

endif

PRODUIT(_) $= \emptyset$;

5. *Quantité_Cdée* $=$ *PRODUIT* $\rightarrowtail\!\!\!\!\!\!+\;\; \mathbb{N}$;

6. *Quantité_en_stock* $\in T_{C_p} \times$ *PRODUIT* $\longrightarrow \mathbb{N}$;

Quantité_en_stock($T.E$, p) =

Case of:

$E =$ **Supprimer_stock**(*cde*, Qc)

then

if $p \in$ **dom**(Qc)

then

return *Quantité_en_stock*(T, p) - $Qc(p)$

else

return *Quantité_en_stock*(T, p)

endif

$E =$ **Ajouter_stock**$(p,\ q)$

then

if $p \in PRODUIT(T)$

then

return $Quantité\_en\_stock(T,\ p) + q$

else

return $Quantité\_en\_stock(T,\ p)$

endif

$E =$ **Ajouter_produit**(p)

then

return 0

endcase

$Quantité\_en\_stock(\_,\ p) = Stock\_initial(p)$;

7. $Quantité\_suffisante \in T_{C_p} \times Quantité\_Cdée \longrightarrow Booléen$

   $Quantité\_suffisante(T,\ Qc)$

   $\Leftrightarrow$

   $\forall\ cde,\ p$: $(cde,\ p) \in \mathbf{dom}(Qc)$

   $\Rightarrow$

   $Quantité\_en\_stock(T,\ p) \geq Qc(p)$;

8. $Réponse = \{$ OK, ÉCHEC $\}$;

9. $Stock\_initial \in PRODUIT \longrightarrow \mathbb{N}$ est la quantité initiale en stock du système.

Bibliographie

[1] R. Abraham. Evaluating Generalized Tabular Expressions in Software Documentation. Report No 346, Communications Research Laboratory CRL, Février 1997.

[2] J. R. Abrial, M. K. O. Lee, D. S. Neilson, P. N. Scharbach, and I. H. Sorensen. The B-method. In S. Prehn and W. J. Toetenel, éditeurs, *VDM'91 : Formal Software Development Methods*, pages 398-405, Noordwijkerhout, The Netherlands, Octobre 1991. Volume 552 of Lecture Notes in Computer Science, Springer-Verlag. Volume 2 : Tutorials.

[3] J. R. Abrial, M. K. O. Lee, D. S. Neilson, P. N. Scharbach, and I. H. Sorensen. *The B-method.* Springer-Verlag, 1991.

[4] G. H. Archinoff, R. J. Hohendorf, A. Wassyng, B. Quigley, and M. R. Borsch. Verification of the Shutdown System Software at the Darlington Nuclear Generating Station. In *International Conference on Control and Instrumentation in Nuclear Installations*, Glasgow, UK, Mai 1990. The Institution of Nuclear Engineers.

[5] G. Barrett. Formal methods applied to floating-point number system. *IEEE Transactions on Software Engineering*, pages 611-621, 5(15), Mai 1989.

[6] W. Bartussek and D. L. Parnas, éditeurs. Using Assertions About Traces to Write Abstract Specifications for Software Modules, *Proceedings of 2nd Conference of European Coorporation in Informatics*, Venice, 1978, Volume 65 of Lecture Notes in Computer Science, Springer-Verlag, pages 211-236, 1978.

[7] P. Behm, P. Desforges, and J. M. Meynadier. METEOR: An Industrial Success in Formal Development. Volume 1393 of Lecture Notes in Computer Science, Springer-Verlag. B'98: Recent Advances in the Development and Use of the B-Method (Second International B Conference), page 26, Montpellier, France, Avril 1998.

[8] B. W. Boehm. *Software Engineering Economics.* Prentice-Hall, 1981.

[9] B. W. Boehm. A spiral model of software development and enhancement. *Computer*, pages 61-72, 5(21), Mai 1988.

[10] J. P. Bowen and M. G. Hinchey. Seven more myths of formal methods. *IEEE Software*, pages 34-41, 4(12), Juillet 1995.

[11] J. P. Bowen and M. G. Hinchey. Ten commandments of formal methods. *IEEE Computer*, pages 56-63, 4(28), Avril 1995.

[12] M. Bush. Improving software quality: The use of formal inspections at the Jet Propulsion Laboratory. In *12th Int. Conf. On Software Engineering*, pages 196-199, Nice, France, Mars 1990. IEEE Computer Society.

[13] C. Choppy. Spécifications algébriques: validation et prototypage. Habilitation à diriger des recherches, Université Paris-Sud, Orsay, 1994.

[14] E. M. Clark and J. M. Wing. Formal Methods: State of the Art and Future Directions. ACM Workshop on Strategic Directions in Computing Research–Group Report: Formal Methods, pages 14-15, Juin 1996, Cambridge, MA, USA.

[15] D. Craigen, S. Gerhart, and T. Ralston. An International Survey of Industrial Applicayions of Formal Methods. Volume 1: Purpose, approach, analysis and conclusions; Volume 2: Case studies. Technical Report NIST GCR 93/626, National Institute of Standards and Technology, Gaithersburg, MD, Avril 1993.

[16] A. M. Davis. A comparison of techniques for the specification of external system behavior. *Communications of the ACM*, pages 1098–1115, 31(9), 1988.

[17] N. Delisle and D. Garlan. A formal specification of an oscilloscope. *IEEE Software*, 5(7), pages 29-36, Septembre 1990.

[18] E. W. Dijkstra. *Discipline Programming.* Prentice-Hall, 1976.

[19] Groupe STERIA en coopération avec J. R. Abrial et A. Gec, éditeurs. *Le langage B: Manuel de Référence version 3.0*, 1997.

[20] M. A. Ardis *et al.* A Framework for evaluating specification methods for reactive systems: experience report. *IEEE Transactions on Software Engineering*, 22(6), pages 378-389, Juin 1996.

[21] K. L. Heninger *et al.* Software Requirements for the A-7E Aircraft. Report 3876, Naval Research Laboratory NRL, Novembre 1978.

[22] R. E. Fairley. *Software Engineering Concepts.* McGraw-Hill, 1985.

[23] M. Frappier, A. Mili, and J. Desharnais. Defining and detection feature interactions. *IFIP TC 2 Working Conference on Algorithmic Languages and Calculi*, R.S. Bird and L.G.L.T. Meertens, eds., Chapman and Hall, 1997.

[24] M. Frappier and R. St-Denis. A specification method for cleanroom's black box description. *31st Hawaii International Conference on System Sciences, IEEE Computer Society Press*, 1998.

[25] J. V. Guttag, E. Horowitz, and D. Musser. The Design of Data Type Specifications. Current Trends in Programming Methodology IV, R. T. Yeh(ed.), Prentice-Hall, 1978.

[26] H. Habrias, C. Attiogbé, and M. Allemand, éditeurs. *International Workshop on: Comparing Systems Specification Techniques*, page XIII, 1998.

[27] A. Hall. Seven myths of formal methods. *IEEE Software*, 5(7), pages 11-19, Septembre 1990.

[28] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[29] D. Hoffman and P. Stooper. *Fundamentals of Software Design and Verification.* McGraw-Hill, 1994.

[30] I. Houston and S. King. CICS Project Report: Experiences and Results from the Use of Z in IBM. In S. Prehn and W. J. Toetenel, [39], pages 588-596, 1991.

[31] ISO/AFNOR. *Dictionnaire de génie logiciel.* AFNOR, 1997.

[32] M. Jackson. *System Development.* Prentice-Hall, 1983.

[33] C. B. Jones. The Search for Tractable Ways of Reasonings About Programs. Technical Report, UMCS-92-4-4, Department of Computer Science, University of Manchester, Manchester, UK, Mars 1992.

[34] B. Marre et F. Schlienger M.-C. Gaudel and G. Bernot. *Précis de génie logiciel.* Masson, 1996.

[35] R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[36] J. F. Monin. *Comprendre les méthodes formelles, panorama et outils logiques.* Masson, 1996.

[37] D. L. Parnas. Some Theorems We Should Prove. In J. J. Joyce and C.-J. H. Seger, éditeurs, *Higher Order Logic Theorem Proving and its Applications (6th International Workshop HUG 93)*, pages 155-162, Vancouver, Canada, Août 1993. Number 780 in Lecture Notes in Computer Science, Spring-Verlag.

[38] D. L. Parnas, éditeur. Information Distributions Aspects of Design Methodology, *Proceedings of IFIP Congress* 1971, pages 26-30, 1972.

[39] S. Perhn and W. J. Toetenel, éditeurs. *VDM'91: Formal Software Development Methods.* Volume 551 of Lecture Notes in Computer Science, Noordwijkerhout, The Netherlands, Octobre 1991. Springer-Verlag. Volume1 : Conference Contributions.

[40] S. L. Pfleeger and L. Hatton. Investigating the influence of formal methods. *Computer*, 30(2), pages 33-43, Février 1997.

[41] W. W. Royce. Managing the Development of Large Software Systems. *In Proceedings WESCON*, Août, 1970.

[42] J. Rushby. Formal Methods and the Certification of Critical Systems. Technical Report CSL-93-7, SRI International, Décembre 1993.

[43] H. Saiedian. An invitation to formal methods. *IEEE Computer*, 4(29), pages 16-30, Avril 1996.

[44] F. Stuart. Software Requirements : A Tutorial. Technical Report MR/5546–95-7775. Naval Research Laboratories NRL, Novembre 1995.

[45] Y. Wang and D. L. Parnas. Specifying and Simulating the Externally Observable Behavior of Modules. Report/Manuscript 292, TRIO CRL, McMaster University, Ontario, Canada, Août 1994.