Université de Sherbrooke
Conservatoire National des Arts et Métiers

EXAMEN DE SPÉCIALITÉ

EB^4 : Vers une méthode combinée de spécification formelle des systèmes d'information

Frédéric GERVAIS

présenté et soutenu le 30 Juin 2004
devant le jury composé de :

Marc FRAPPIER (Université de Sherbrooke)	*co-directeur*
Régine LALEAU (Université Paris 12)	*co-directrice*
Luc LAVOIE (Université de Sherbrooke)	*président*
Anatol SLISSENKO (Université Paris 12)	*membre*

Département d'Informatique
Groupe de Recherche en Ingénierie du Logiciel
Université de Sherbrooke
SHERBROOKE, Québec, J1K 2R1 (CANADA)

—

Laboratoire CEDRIC
Institut d'Informatique d'Entreprise
18 Allée Jean Rostand
91025 ÉVRY CEDEX (FRANCE)

À mon grand-père, Quirino

Table des matières

Chapitre 1

Introduction

> “Les erreurs ...
>
> Pour un architecte, c’est dramatique, car il devra vivre avec toute sa vie et ses œuvres resteront le témoignage de son incompétence même après sa mort.
>
> Pour un médecin, c’est angoissant, car une erreur médicale peut briser une carrière à moins qu’il n’enterre lui-même son ancien patient.
>
> Pour un informaticien, ce n’est pas grave, car personne ne s’en rendra compte.”
>
> — Eric B. Motta Jr.

1.1 Motivation

Un système d’information (SI) est un système informatisé qui rassemble l’ensemble des informations présentes au sein d’une organisation, sa mémoire, et les activités qui permettent de les manipuler [Lal02]. Un tel système est caractérisé par des structures de données complexes et par des opérations impliquant des volumes de données importants. De plus, les données peuvent être modifiées et/ou accédées par de nombreux utilisateurs en concurrence.

Les méthodes de spécifications formelles sont utilisées en génie logiciel pour raisonner sur des modèles mathématiques. L’intérêt est de pouvoir prouver ou vérifier des propriétés sur ces modèles. Malgré les coûts supplémentaires liés au travail d’analyse et de conception en spécification formelle, l’utilisation de telles méthodes est de plus en plus justifiée pour des logiciels qui impliquent des données ou des conditions de sécurité critiques, car elles permettent d’assurer leur bon fonctionnement et d’éviter ainsi des risques d’erreur.

Une des difficultés actuelles est de spécifier formellement le comportement des systèmes d’information. Un des paradigmes les plus couramment utilisés est celui des transitions d’état. Une spécification consiste alors en un espace d’états défini par des variables d’état et en des opérations qui définissent les transitions d’état. À la réception d’un événement externe au système d’information, une opération est appelée afin de calculer le nouvel état du SI et éventuellement de produire une sortie. Plusieurs langages existent pour décrire des spécifications

basées sur les transitions d'état, comme les diagrammes états-transitions, les machines à états ou bien les langages de l'approche ensembliste.

Certaines propriétés, comme les contraintes d'ordonnancement des événements, sont toutefois difficiles à vérifier en utilisant une approche basée sur les transitions d'état. Les algèbres de processus par exemple permettent plus facilement de spécifier ce type de propriétés, mais ne sont pas bien adaptées pour prendre en compte les caractéristiques des SI. Les approches basées sur les événements et les approches basées sur les transitions d'état semblent donc complémentaires pour modéliser le comportement dans les SI.

1.2 Deux approches complémentaires : B et EB^3

Deux méthodes nous semblent plus particulièrement adaptées pour spécifier formellement des systèmes d'information : B et EB^3.

La méthode B [Abr96] est à la fois un langage et une méthode de spécifications formelles. Elle a l'avantage de couvrir toutes les phases du cycle de conception, depuis l'analyse des besoins jusqu'à l'implémentation finale. Elle est de plus très bien outillée. Ces raisons nous ont donc conduit à considérer B plutôt que d'autres langages du même type comme VDM ou Z, qui ne couvrent pas tous ces aspects. Le langage B est basé sur la notion de machine abstraite qui est fondée sur les notions d'état et de propriétés d'invariance. Les outils associés à la méthode permettent d'une part de vérifier la correction des machines spécifiées et d'autre part de prouver des propriétés sur les spécifications obtenues. Enfin, la méthode B a été utilisée dans [Ngu98] pour spécifier des systèmes d'information à partir de diagrammes UML traduits ensuite en des spécifications B.

La méthode EB^3 [FSD03] a été spécialement développée pour la spécification des systèmes d'information. Elle s'inspire des approches basées sur les événements comme CSP, CCS ou LOTOS tout en simplifiant leur sémantique et leur utilisation. Le langage EB^3 est de plus orienté sur les traces d'événements ce qui rend les propriétés d'ordonnancement ou de sûreté plus explicites. La méthode est également outillée avec un interpréteur [FF02] d'expressions EB^3 qui permet de générer automatiquement des systèmes d'information à partir d'une spécification EB^3 valide. Enfin, le langage, qui est issu du monde académique, est en perpétuelle évolution. Une version simplifiée du langage, appelée eb^{3}web [NXD03], a été récemment introduite pour générer automatiquement des interfaces de systèmes d'information sur internet.

La méthode B est en outre utilisée dans l'industrie, ce qui rend son attractivité encore plus grande. Si la méthode B permet de bien prendre en compte les structures de données d'un système d'information, elle est en revanche inadéquate pour considérer des propriétés temporelles ou d'ordonnancement des opérations. Si la méthode EB^3 est intéressante pour décrire et vérifier ce type de propriétés, il est en revanche plus difficile de prendre en compte les actions et les modifications d'un état particulier, ainsi que les propriétés d'invariance de l'espace d'états. Les deux méthodes semblent donc être complémentaires.

1.3 Raffinement

Le raffinement constitue un des principaux atouts de la méthode B. Il permet en effet aux concepteurs de développer par étapes successives le projet jusqu'à l'implémentation finale. La méthode permet en outre de valider chaque étape par la preuve de certaines conditions.

Il existe principalement deux moyens pour spécifier un système complexe avec une méthode formelle : soit le processus de développement est compositionnel, c'est-à-dire que le système est spécifié petit à petit, par morceaux que l'on compose ensuite entre eux, et ainsi de suite ; soit la méthode supporte l'activité de raffinement qui consiste à dériver progressivement la spécification, tout en préservant la correction vis-à-vis des spécifications précédentes.

Dans le but d'intégrer les approches EB^3 et B, il nous semble indispensable de doter la méthode EB^3 d'une relation de raffinement, comme c'est déjà le cas pour B. Le raffinement d'EB^3 doit tenir compte des caractéristiques du langage et nous aider à le rendre compatible avec B. Dans ce but, nous avons réalisé une étude des nombreuses notions de raffinement existant dans la littérature.

1.4 Combinaisons de spécifications formelles

Dans [FL03], Frappier et Laleau montrent comment prouver une propriété d'ordonnancement décrite en EB^3 sur des opérations d'une machine B. Notre objectif est d'utiliser ces deux méthodes formelles de manière plus intégrée afin de développer des spécifications des systèmes d'information qui modélisent les propriétés dynamiques.

Dans ce but, nous nous sommes intéressés à une série d'exemples de combinaisons de spécifications formelles afin d'analyser les avantages, les défauts et les contraintes liés à ce type d'approches. Afin de les comparer, nous avons utilisé un petit exemple de référence dont les opérations sont assez caractéristiques des systèmes d'information.

La combinaison de spécifications formelles peut apporter de nombreux avantages. Les modèles sont plus riches et permettent de mieux représenter les aspects statiques et dynamiques des systèmes. Les possibilités de vérification sont nombreuses et si la sémantique est une unification des sémantiques des langages utilisés, les relations de raffinement peuvent également être unifiées.

Le caractère formel peut aussi avoir ses inconvénients. L'intégration de plusieurs méthodes formelles est en effet difficile. Les spécifications peuvent d'une part être contradictoires ou redondantes. Elles sont d'autre part difficiles à analyser et à comprendre, car la définition sémantique de l'intégration pose souvent problème.

Le principal problème à résoudre concerne le niveau d'intégration d'une approche par rapport à l'autre. Si deux langages sont peu intégrés, les possibilités d'analyse et de vérification seront limitées. Si, au contraire, les deux langages sont unifiés pour en créer un nouveau, la principale difficulté sera la définition d'une nouvelle sémantique.

Le problème dépend en fait des propriétés et des caractéristiques souhaitées pour la combinaison.

1.5 Organisation du rapport

Dans un premier temps, nous présenterons la problématique et les motivations de notre sujet (chapitre 2). Les définitions et les principaux langages utilisés dans la suite du rapport seront détaillés dans le chapitre 3. Le chapitre 4 sera consacré à l'étude du raffinement. Puis nous présenterons et comparerons les principaux exemples concernant une approche possible pour modéliser le comportement dans les SI : la combinaison de méthodes de spécifications formelles (chapitre 5). Notre proposition, ainsi que les problèmes qui en découlent, sera présentée dans le chapitre 6. Enfin, nous conclurons ce rapport avec les perspectives de nos travaux, dans le chapitre 7.

Chapitre 2

Problématique : la spécification formelle des systèmes d'information

> "Les problèmes de la vie réelle, ce sont ceux qui restent une fois que vous avez retiré toutes les solutions connues."
>
> — Edsger W. Dijkstra

L'objectif de ce chapitre est de présenter et de situer notre problème. Notre souhait est de modéliser le comportement dans les systèmes d'information en utilisant des langages formels de spécification. Dans un premier temps, nous présentons les principales caractéristiques d'un système d'information, puis nous aborderons le problème de la spécification et de la vérification des propriétés dans les systèmes d'information. Nous présenterons enfin deux approches de spécifications formelles des SI qui nous semblent intéressantes : UML-B et EB^3. Nous verrons que chacune de ces approches permet de spécifier formellement certains aspects, mais qu'elles ne sont pas nécessairement complètes.

2.1 Introduction sur les systèmes d'information

Un *système d'information* (SI) est un système informatisé qui rassemble l'ensemble des informations présentes au sein d'une organisation, sa mémoire, et les activités qui permettent de les manipuler [Lal02]. Il est caractérisé par :

1. l'utilisation de nombreuses données,
2. des relations complexes entre les structures de données,
3. des utilisateurs hétérogènes qui peuvent agir en concurrence,
4. des opérations impliquant plusieurs structures de données, utilisant un volume important de données, tout en préservant l'intégrité des données modifiées.

Une conséquence importante est le choix d'une structure de donnée adaptée dans le processus de modélisation des SI. En revanche, les algorithmes des opérations ne sont pas complexes.

Un système d'information est généralement constitué de trois parties :

- une interface graphique,
- une base de données,
- des transactions.

Si le concept d'interface graphique est assez courant, il est en revanche difficile de définir avec précision les notions de base de données et de transaction.

Selon Gardarin [Gar99], une *base de données* est un ensemble de données interrogeables modélisant les objets d'une partie du monde réel et qui sert de support à une application informatique. Par conséquent, la base de données n'a d'intérêt que si elle peut être interrogée et modifiée. Une *transaction* est une application sur la base de données qui permet d'y faire des interrogations et/ou des mises à jour. Elle vérifie en général les propriétés appelées ACID (*Atomicity, Consistency, Isolation, Durability*) [Elm04] :

- une transaction est atomique, car elle est exécutée dans son ensemble ou pas du tout,
- la base de données doit rester cohérente après l'exécution d'une transaction,
- chaque transaction est indépendante et n'interfère pas avec les autres transactions,
- et les modifications d'une transaction qui a été exécutée perdurent même en cas de panne de la base de données.

Les méthodes de conception des systèmes d'information peuvent être classées en trois catégories : les méthodes fonctionnelles, les méthodes systémiques et les méthodes orientées objets. Ces méthodes se fondent sur des vues et des concepts différents pour modéliser le système d'information.

Méthodes fonctionnelles. Les méthodes fonctionnelles s'appuient sur les techniques de décomposition cartésienne et sur les représentations des flots de données. L'approche est dite fonctionnelle, car elle identifie un système d'information à une fonction globale de gestion qui est ensuite décomposée en des fonctions plus détaillées, et ainsi de suite ...

Le modèle du système d'information est représenté à l'aide de diagrammes de flots de données [YC79] et de diagrammes de structure. Les méthodes de conception les plus connues appartenant à cette catégorie sont la méthodologie Gane et Sarson [GS79] et SADT [MM88].

Ces méthodes de la première génération (années 60) ne reposent pas sur des fondements théoriques et les modélisations ne sont pas formelles.

Méthodes systémiques. Les méthodes systémiques se focalisent sur la modélisation des données. Les modèles permettent de représenter à la fois les informations du système et les relations entre elles.

Il existe plusieurs niveaux d'abstraction dans ces méthodes, depuis l'analyse du système, en passant par les niveaux conceptuel, logique et physique. La modélisation la plus abstraite des données et des traitements sur ces données est réalisée par des schémas conceptuels de données et de traitements.

La modélisation conceptuelle des données repose généralement sur le modèle relationnel et sur le modèle entité-association. Les modèles de traitement dépendent des méthodes de conception. Parmi les plus connues, on peut citer : Merise [TRC83], Remora [RFB88], Axial [Pel86], ...

Les modèles utilisés dans ces approches comme les entités-associations ne sont pas formels. Certaines méthodes ont toutefois étendu les concepts pour les rendre formels.

Méthodes orientés objet. Les méthodes orientées objet modélisent les systèmes d'information autour d'entités appelées des classes qui représentent un état et qui définissent des opérations (ou méthodes). Les classes sont reliées entre elles par des associations.

Une conception orientée objet est constituée de trois modèles. Le modèle statique permet de représenter les classes et leurs associations. Le modèle dynamique représente le comportement de chaque type d'objet. Enfin, le modèle d'interaction permet de représenter les flux de messages entre objets. Les approches appartenant à cette catégorie sont par exemple : OOD [Boo94], OOSE [Jac94], OMT [RBP$^+$91], ou UML [Mul97].

Les notations graphiques utilisées dans ces approches constituent à la fois un avantage et un inconvénient. Elles permettent une meilleure compréhension des modèles mais leur sémantique n'est pas précise. Les langages graphiques sont en effet considérés comme *semi*-formels.

Une notation est dite *formelle* si elle est fondée sur des bases mathématiques. Un *langage formel* a une syntaxe et une sémantique formelles. Une spécification écrite dans un langage formel est appelée spécification formelle. Si un langage a seulement une syntaxe formelle, il est dit *semi-formel* [Lal02]. Une *méthode formelle* est une méthode de développement de logiciels qui s'appuie sur des techniques et sur des langages formels.

S'il existe des méthodes pour concevoir les bases de données en s'appuyant sur des langages formels (voir section 2.3), la spécification des applications du comportement fonctionnel du SI reste en revanche dans la plupart des cas informelle.

Pour clore cette introduction générale, on peut citer les livres de Gardarin [Gar99], Ullman [Ull88] et Elmasri [Elm04] comme principales références concernant les fondements et les principes des bases de données.

2.2 Un problème lié à la modélisation des SI : la spécification du comportement

La modélisation du comportement reste un défi important dans l'ingénierie des systèmes d'information. Le comportement fonctionnel d'un SI est essentiellement défini par des transactions sur les bases de données : transaction d'interrogation pour interroger la base de données et transaction de mise à jour pour modifier cette base. La cohérence des données est garantie par des contraintes d'intégrité de deux types : les contraintes statiques, qui imposent des restrictions sur les données de la base, et des contraintes dynamiques, qui imposent des restrictions sur le cycle de vie des données.

Dans les bases de données actives [WC96], le comportement peut être modélisé par des règles de transformation sur les structures de données. Ces règles actives, qui sont des extensions des déclencheurs (*triggers*), permettent de décrire les réactions du système lorsqu'un événement se produit. Elles sont de la forme événement-condition-action (ECA). Lorsqu'un événement E survient, si la con-

dition C est vérifiée, alors l'action A est exécutée. Cette approche reste toutefois limitée, car elle est difficile à mettre en œuvre sur l'ensemble des transactions d'une base de données.

En outre, la plupart des méthodes de conception actuelles des SI ne considèrent pas la définition des transactions au niveau de l'analyse, mais plutôt dans les phases successives du développement des SI.

2.2.1 Quels types de propriétés ?

On peut distinguer plusieurs types de propriétés à spécifier et/ou à vérifier sur les systèmes d'information.

Les propriétés *statiques* permettent d'assurer la cohérence du système et l'intégrité des informations : elles s'expriment par l'intermédiaire de contraintes d'intégrité dans les bases de données. Dans les approches formelles basées sur les transitions d'état, les contraintes d'intégrité sont spécifiées sont la forme d'invariant ou bien de garde sur les opérations.

Une propriété est dite *dynamique* si elle traite de l'occurrence et de l'ordonnancement des événements. Cette définition comprend aussi bien les propriétés de sécurité et de vivacité, que les propriétés du type "a suivi d'un nombre arbitraire de b et suivi d'un c".

Une propriété de *sûreté* est une propriété de la forme : "quelque chose de mauvais n'arrive jamais". Dans le cas de méthodes formelles basées sur les transitions d'état, un invariant constitue une propriété de sûreté pour le système. Dans le cas de spécifications exprimées sous forme de traces d'événements, une trace valide du système est une trace qui respecte les propriétés de sûreté.

Une propriété de *vivacité* est une propriété de la forme : "quelque chose de bien arrivera nécessairement". Avec des langages de type algèbre de processus, ce type de propriété se vérifie en analysant les traces. Dans un langage formel basé sur les transitions d'état, la vivacité est plus difficile à vérifier, car elle demande une analyse des traces valides ou des séquences de transitions d'état qui ne sont pas naturelles dans ce type de langages.

Enfin, une propriété de type ordonnancement demande également un certain effort d'analyse dans les approches basées sur les états contrairement aux approches basées sur les événements. Par exemple, la propriété "a, suivi d'un nombre fini de b, suivi de c" s'exprime facilement avec un langage comme EB^3 (sa syntaxe sera présentée plus en détail dans la section 2.3.2) :

$$a \;.\; b^* \;.\; c$$

Dans le cas d'un langage basé sur les transitions d'état, les opérations sont généralement décrites à l'aide de modifications sur des variables d'état. Pour s'assurer que des opérations a, b et c satisfont la contrainte d'ordonnancement définie ci-dessus, il faut vérifier que, si n est le nombre d'exécutions de b, alors :

- l'état après avoir exécuté a correspond à un état dans lequel il est possible d'exécuter b,
- les états après avoir exécuté respectivement une, deux, ..., $n-1$ fois b correspondent à des états dans lesquels il est encore possible d'exécuter b,
- l'état après avoir exécuté n fois b correspond à un état dans lequel il est possible d'exécuter c.

La vérification d'une propriété d'ordonnancement est donc moins naturelle dans un langage de type transitions d'état que dans un langage basé sur les événements.

Enfin, il faut bien cerner le mode d'expression des propriétés du comportement d'un système. À la limite, tous les formalismes permettent d'exprimer une propriété dynamique, car on peut considérer la propriété comme étant un système, et la spécifier avec ce formalisme. Les langages basés sur les événements (par exemple, les logiques temporelles, les algèbres de processus, les automates, les expressions régulières et les grammaires) sont souvent bien adaptés pour spécifier de manière explicite les propriétés dynamiques. Toutefois, certaines propriétés dynamiques sont plus faciles à spécifier avec un invariant sur l'espace d'états. Par exemple, spécifier que deux emprunteurs ne peuvent emprunter un livre en même temps s'exprime plus facilement par un invariant (du genre la variable emprunteur est une fonction de livre vers membre) que par une expression de processus. En fait, on pourrait considérer un invariant sur l'espace d'état comme un cas particulier de propriété dynamique.

Pour définir la notion de propriété de manière plus précise, on peut utiliser la notion de système de transitions sur un espace d'état défini par des variables. Dans la suite, les systèmes de transitions étiquetés (ou LTS) nous serviront à représenter et/ou expliquer nos exemples.

2.2.2 Vers une méthode de spécification formelle

Les principales techniques utilisées [AMF00] pour modéliser le comportement dans les SI, comme les réseaux de Petri, les modèles entités-associations étendus ou les approches orientées objet, sont limitées. En outre, elles sont généralement appliquées une fois que le système est défini, pour vérifier que les propriétés voulues sont satisfaites. Pour assurer que ces propriétés restent vraies dans le temps, certaines méthodes de conception proposent aussi la définition de règles de déclenchement (*triggers*) qui empêchent ou modifient les demandes de mises à jour qui pourraient violer les contraintes d'intégrité. Toutes ces stratégies sont toutefois plutôt défensives, ce qui limite les possibilités du système.

Dans le cas des réseaux de Petri, le comportement du système ne peut être que partiellement décrit et simulé. Un réseau de Petri [Pet81] est un graphe biparti composé d'une part de places qui représentent les variables logiques du système, et d'autre part de transitions qui définissent les transitions ou les actions du système. Dans la modélisation d'une règle de type ECA (voir l'introduction du chapitre), les places du réseau peuvent être utilisées pour représenter les événements et les conditions, tandis que les transitions représentent les actions. La complexité des réseaux de Petri limite néanmoins leur utilisation et cette technique est plutôt utilisée pour modéliser une partie seulement des transactions.

Le modèle entités-associations permet de modéliser les contraintes d'intégrité d'un système d'information. Dans certaines approches, il est étendu afin de représenter des transitions ou des règles actives. Par exemple, le modèle ER^2 (entités-associations-événements-règles) [Tan92] permet de considérer les événements et les règles, qui sont représentés comme des objets. Un réseau de Petri coloré [Jen96], c'est-à-dire un réseau de Petri dont les places sont paramétrées, est utilisé pour modéliser les flots de contrôle entre processus. Si ce modèle permet de représenter facilement les relations entre règles et objets, il ne

permet pas de représenter des règles autres que celles concernant les opérations sur les données.

Dans les méthodes orientées objet, les transactions sont modélisées grâce à des diagrammes états-transitions. Ils décrivent le comportement en termes d'états et de transitions sur ces états. Les effets d'une règle de transformation sur les objets ne sont pas toujours visibles avec cette approche. De plus, elles utilisent un formalisme graphique, semi-formel.

En conclusion, les approches actuelles ne privilégient pas l'aspect formel pour spécifier les propriétés dynamiques. De plus, il est souvent difficile d'intégrer rapidement les aspects statiques et dynamiques lors du processus de développement des SI. Les spécifications concernant les propriétés dynamiques peuvent en effet être en contradiction avec les structures de données. Comme les aspects statiques sont déjà spécifiés, les contraintes dynamiques sont dépendantes de la modélisation de la statique. Dans le cas d'approches non formelles ou semi-formelles, cette intégration tardive des aspects dynamiques peut être une source non négligeable d'erreurs.

2.3 Deux exemples de spécification formelle des systèmes d'information

Pour mieux comprendre les intérêts de l'utilisation des approches formelles pour spécifier les systèmes d'information, nous présentons maintenant deux exemples de méthodes qui ont été développées dans cette optique : UML-B et EB^3.

L'approche UML-B permet de formaliser les descriptions semi-formelles des diagrammes de classes UML en utilisant le langage de spécification formel B. Les propriétés dynamiques du SI modélisé ne sont pas facilement exprimables par cette approche.

Le langage EB^3, qui a été créé pour spécifier des systèmes d'information, permet de bien spécifier les traces des actions des différentes entités du système. Les propriétés concernant les états et les données sont en revanche plus difficiles à vérifier.

2.3.1 UML-B [Ngu98]

Cette approche permet de spécifier un système d'information à partir d'un diagramme de classes UML qui est ensuite traduit en B pour rendre la spécification formelle. Le langage B sera présenté en détail dans le chapitre 3. La spécification obtenue peut ensuite être raffinée pour obtenir du code SQL.

Le langage UML. Le langage UML [Mul97] est un langage graphique orienté objet, issu de l'unification de plusieurs méthodes. La description graphique de UML représente un sérieux avantage, car les diagrammes sont plus faciles à appréhender pour le concepteur, et c'est la raison pour laquelle les approches graphiques sont couramment utilisées dans l'industrie. Toutefois, un langage comme UML est considéré comme *semi*-formel, car la sémantique des notations graphiques utilisées n'est pas précise.

Le langage UML offre plusieurs sortes de diagrammes afin de couvrir les diffrentes étapes du processus de développement d'un système, depuis l'analyse

des besoins jusqu'à l'implémentation. Dans le cas des systèmes d'information, les diagrammes suivants sont utilisés :

- le diagramme de classes représente les aspects statiques et structurels,
- les diagrammes d'états-transitions permettent de décrire le comportement des objets d'une classe donnée,
- enfin, les diagrammes de collaborations représentent les fonctionnalités du système.

Si la description d'un système selon plusieurs vues facilite la compréhension du fonctionnement pour l'utilisateur, elle peut également rendre la spécification globale incohérente. Comme les diagrammes ne sont pas formels, il n'est pas possible de faire des vérifications. L'utilisation de règles de traduction pour définir chaque notation graphique en une notation mathématique permet d'obtenir une spécification formelle du système et de réaliser des preuves ou des vérifications de propriétés.

La méthode UML-B. La spécification d'un système d'information est réalisée de la manière suivante :

- La première étape consiste à décrire les aspects statiques du système d'information avec un diagramme de classes UML. Les contraintes qui ne peuvent pas être exprimées de façon précise avec les notations graphiques sont spécifiées en B.
- Les règles de traduction définies dans [Ngu98, Mam02] permettent ensuite de générer des spécifications B à partir du diagramme de classes établi lors de la première étape.
- L'étape suivante consiste à décrire les transactions du système à l'aide des diagrammes d'états-transitions et de collaborations. Les expressions B de la première étape sont alors utilisées pour annoter ces nouveaux diagrammes et ainsi assurer leur traduction automatique. Cette étape peut nécessiter l'ajout de nouveaux attributs dans le diagramme de classes et par conséquent la réitération des deux premières étapes du processus.
- Les spécifications B correspondant aux diagrammes d'états-transitions et de collaborations sont ensuite générées automatiquement par un outil.
- La méthode permet de générer des obligations de preuves afin d'assurer la satisfaction des propriétés d'invariance (propriétés statiques).

Conclusion. La méthode de spécification UML-B permet ainsi d'obtenir une spécification formelle en B d'un système d'information. Seules les propriétés fonctionnelles du système sont effectivement prises en compte par cette méthode, car l'utilisation de B ne permet pas la spécification de propriétés dynamiques, telles que les contraintes temporelles ou les propriétés de vivacité.

2.3.2 EB^3 [FSD03]

"Entity-Based Black-Box" (EB^3) [FSD03] est à la fois un langage formel et une méthode de spécification dédiés à la modélisation des systèmes d'information.

Le langage EB^3. Le langage EB^3 est un langage de spécification formel dédié à la modélisation des systèmes d'information, qui est fondé sur la notion de trace,

sur les algèbres de processus et sur la notion d'entité de la méthode JSD [Jac83].

Afin de décrire les principales fonctionnalités des systèmes d'information, la syntaxe du langage EB^3 emprunte les principaux opérateurs de CSP [Hoa85], CCS [Mil89] et LOTOS [BB87], mais leur sémantique a été simplifiée et des propriétés comme l'action interne ou le non-déterminisme n'ont pas été prises en compte en EB^3.

Le langage EB^3 permet de décrire des *expressions de processus*. Les notions classiques de classe et d'objet des modèles orientés objet sont appelées respectivement en EB^3 des *types d'entité* et des *entités*. Les méthodes de chaque type d'entité constituent les événements ou *actions* du système d'information. Les expressions de processus *élémentaires* du langage sont les actions définies dans le système et forment l'alphabet Σ du système.

Les opérateurs utilisés pour composer les expressions de processus sont les suivants. La concaténation (.) permet de concaténer au sens des listes deux expressions de processus. Par exemple, la concaténation :

$$a.b$$

est la séquence des actions a et b. L'opérateur $\Rightarrow$ permet d'exprimer une garde sur une expression de processus. La clôture de Kleene sur une action a est dénotée par a^* et désigne un nombre arbitraire fini d'exécutions de a.

Les opérateurs de choix (|), de composition parallèle (||) et d'entrelacement (|||) sont les mêmes qu'en CSP. Si P_1 et P_2 sont deux expressions de processus et si Δ est un ensemble d'actions de l'alphabet, $P_1 \mid [\Delta] \mid P_2$ est la composition parallèle paramétrée de P_1 et P_2 avec synchronisation sur les actions de Δ. Cet opérateur est emprunté à LOTOS.

Enfin, les opérateurs |, | [Δ] | et ||| peuvent être quantifiés. Par exemple, l'expression de processus :

$$\mid x : \mathbb{N} : P(x)$$

représente :

$$P(0) \mid P(1) \mid ...$$

La méthode. La spécification d'un système d'information avec EB^3 est réalisée de la manière suivante :

- La première étape consiste à spécifier un diagramme de classes UML définissant les types d'entités (noms désignant les classes en EB^3), les associations et les attributs du modèle de données métier (*business model*).
- Il faut ensuite définir un espace d'entrées-sorties. Cette étape passe par la définition des types de données et des signatures des actions du système.
- Les comportements des types d'entités et des associations définis dans le diagramme de classes de la première étape sont spécifiés à l'aide d'expressions de processus. Chaque type d'entité (respectivement association) est en effet décrit en EB^3 par une expression de processus définissant l'ensemble de ses traces d'actions valides.
- L'exécution de chaque action en entrée peut générer une réponse du système. Des fonctions sont dans un premier temps utilisées pour spécifier les valeurs de sortie pour les différents attributs de chaque type d'entité du système.

– La dernière étape consiste dans un second temps à définir des règles d'entrées-sorties pour spécifier les sorties du système en fonction des traces d'entrée valides.

Outils et applications. Afin de générer automatiquement des systèmes d'information à partir de spécifications EB^3 valides, un interpréteur, appelé EB^{3}PAI, a été développé [FF02]. Plus récemment, les systèmes de transitions utilisés par l'interpréteur ont été étendus [FF03] afin d'optimiser l'interprétation des spécifications.

Bien qu'issu du monde académique, le langage EB^3 a aussi été utilisé dans le monde industriel pour spécifier des systèmes d'information. Il a l'avantage d'être en constante évolution afin de prendre en considération les évolutions du marché. Une variante d'EB^3, appelée eb^{3}web [NXD03], a été récemment définie pour spécifier formellement des interfaces de systèmes d'information sur internet.

Conclusion. La méthode EB^3 est donc formelle, orientée objet et modulaire, elle est de plus basée sur les traces d'événements, ce qui permet de prendre en compte les principales spécificités des systèmes d'information. Comme elle est basée sur les traces, elle ne permet pas de bien représenter les conséquences d'une action sur un état particulier du système. D'autre part, EB^3 ne dispose pas d'un outil de preuve puissant comme dans la méthode B. Les propriétés fonctionnelles sont plus difficiles à vérifier avec cette approche.

2.4 Problématique

Il existe des exemples de méthodes formelles (voir section 2.3) pour spécifier certaines propriétés (statiques ou dynamiques) des systèmes d'information, mais elles n'intègrent pas de manière suffisante les différents aspects. Les approches en question ne permettent pas en effet d'exprimer à la fois les propriétés statiques et dynamiques des systèmes.

Une difficulté consiste aujourd'hui à modéliser de manière formelle le comportement du SI, de préférence en relation avec les propriétés statiques du système. Une autre difficulté réside dans la vérification : le comportement du système doit en effet respecter les propriétés voulues. La modélisation du comportement dans les SI à l'aide de langages formels reste donc un axe de recherche à explorer.

Compte tenu de l'importance des informations d'une organisation ou d'une entreprise et des difficultés actuelles liées à la sécurité et à l'intégrité de ces données, nous nous posons le problème de la spécification par des méthodes formelles des propriétés statiques **et** dynamiques des systèmes d'information.

En particulier, nous souhaitons répondre aux questions suivantes :

– Comment peut-on spécifier, de manière progressive, avec une méthode formelle et outillée des systèmes d'information ?
– Comment peut-on tenir compte à la fois des problèmes d'intégrité des données et du comportement du système ?

2.5 Notre proposition

Intégration de la dynamique et de la statique. Une piste pour résoudre notre problème concerne l'intégration des propriétés dynamiques dès l'analyse et la conception des premières structures de données du système. La modélisation des propriétés statiques devrait en effet être réalisée en fonction des propriétés dynamiques. Cette "inter-dépendance" entre la statique et la dynamique n'est possible que si la méthode de spécification permet de prendre en compte les deux aspects à la fois et si les modifications réalisées sur l'un peuvent facilement être prises en compte dans l'autre.

Comme un unique langage formel permettrait difficilement de prendre en compte à la fois les aspects statiques et dynamiques d'un système d'information, nous pensons que deux méthodes comme B et EB^3 pourraient être complémentaires pour résoudre ce problème.

L'exemple des deux approches de spécification formelle, UML-B et EB^3 (section 2.3), nous montre en effet que les langages formels ne sont pas toujours adaptés pour prendre en compte à la fois les propriétés statiques et dynamiques d'un système. D'une part, les approches de spécification basées sur les transitions d'état, comme B, permettent de décrire facilement certaines propriétés spécifiques au SI, comme les relations complexes entre les larges structures de données du système, mais elles rendent aussi la vérification des propriétés d'ordonnancement des événements difficile. D'autre part, si la spécification des contraintes d'ordonnancement est plus facile en utilisant des spécifications basées sur les événements comme EB^3, les caractéristiques des SI sont au contraire plus difficiles à prendre en compte. Ces deux approches de spécification, qui ne sont pas efficaces utilisées séparemment, semblent donc complémentaires pour modéliser le comportement des SI.

Pourquoi B et EB^3 ? Le choix des méthodes B et EB^3 est en grande partie justifié par l'existence-même des méthodes de spécification présentées dans ce chapitre qui sont formelles et dédiées aux systèmes d'information. Le choix de B par rapport à des approches comme VDM ou Z est justifié par une méthode de spécification complète et outillée qui couvre tout le processus de conception du système. La méthode UML-B montre également que B est adapté pour formaliser des diagrammes semi-formels UML qui sont couramment utilisés dans les méthodes de conception industrielles des SI. La méthode EB^3 est historiquement une méthode dédiée à la spécification formelle des SI puisqu'elle a été créée dans ce but.

Pour ces raisons, chaque méthode peut être considérée, indépendamment l'une de l'autre, comme une bonne approche pour concevoir de manière formelle des SI. Les outils associés à B et la grande souplesse qui entoure les possibilités d'évolution d'EB^3 sont deux autres atouts indéniables. B et EB^3 semblent donc être des candidats naturels à une intégration.

L'utilisation de méthodes formelles est enfin importante, car elle permet de vérifier des propriétés. Le principal défaut de B est la difficulté d'exprimer des propriétés dynamiques ou temporelles. La méthode dispose pourtant d'un prouveur efficace, avec une base de règles importante. Le langage EB^3 est basé sur les traces, ce qui permet d'exprimer facilement des propriétés d'ordonnancement sur les actions du système.

Une méthode globale et progressive. L'approche consistant à combiner des spécifications formelles de type transitions d'état et de type événements semble enfin apporter une solution au problème de la modélisation du comportement dans les SI. Il existe de nombreux exemples dans la littérature, dans des domaines d'application autres que les systèmes d'information, de combinaisons de spécifications formelles dans le but d'intégrer les aspects statiques et dynamiques dans le processus de spécification. L'intégration d'un langage basé sur les états comme B et d'un langage basé sur les événements comme EB^3 semble donc être une approche raisonnable pour spécifier le comportement des systèmes d'information.

L'aspect intuitif des expressions de traces d'EB^3 constitue un atout dans les relations spécificateur - client. Tout comme les diagrammes UML de l'appproche UML-B, les traces s'interprètent assez facilement même pour un non spécialiste. Pour considérer le plus vite possible les aspects dynamiques, nous pensons que la spécification du système doit débuter par la partie EB^3. Ensuite, la partie B permet grâce aux outils et aux obligations de preuve de spécifier et de vérifier des contraintes d'intégrité sur le système. Notre souhait est en effet de créer une méthode globale et progressive de spécification formelle, dédiée aux systèmes d'information, qui intègrerait les langages B et EB^3. Nous l'avons baptisée EB^4.

Au cours de notre étude sur les principaux langages de spécification formels, nous avons remarqué que le raffinement constituait une activité de plus en plus importante dans les méthodes formelles. Pour développer des systèmes complexes, il existe principalement deux techniques : soit on construit progressivement le modèle par petits composants qu'on associe les uns aux autres ; soit on part d'une spécification très abstraite qu'on dérive ensuite par étapes successives. Nous nous sommes penchés sur la deuxième option. La méthode B est en effet basée sur le raffinement. Comme nous pensons la combiner avec EB^3, il semblait naturel de lui définir aussi une notion de raffinement pour spécifier des systèmes complexes. L'étude sur le raffinement est présentée dans le chapitre 4.

Dans le but d'intégrer les approches B et EB^3, nous avons étudié plusieurs exemples de combinaisons de spécifications formelles dans la littérature afin d'en analyser les avantages et les défauts. Ce travail nous a permis d'estimer le niveau d'intégration nécessaire à notre projet. Cette revue de littérature nous a également servi à situer nos futurs travaux par rapport à l'existant, dans un domaine un peu plus général que l'application à la conception des systèmes d'information. Cette étude est l'objet du chapitre 5.

La méthode que nous proposons s'appuie d'une part sur des activités de raffinement en EB^3 et en B pour spécifier progressivement le système et d'autre part sur une intégration des deux approches pour considérer globalement les aspects statiques et dynamiques. Notre proposition, ainsi que les problèmes et les effets qui en découlent, est présentée dans le chapitre 6.

Pour améliorer la compréhension et l'assimilation des nombreuses notions utilisées dans la suite de ce rapport, le chapitre suivant est consacré à un rappel des principales définitions et des langages de spécifications formels.

Chapitre 3

Définitions générales

> "L'ouïe de l'oie de Louis a ouï. Ah oui ? Et qu'a ouï l'ouïe de l'oie de Louis ? Elle a ouï ce que toute oie oit ! "
>
> — Raymond Devos

Ce chapitre est une introduction aux définitions et aux principaux langages utilisés dans la suite de ce rapport.

3.1 Introduction sur les langages formels

Un langage de spécification est dit *formel* lorsque sa syntaxe et ses notations sont rigoureusement définies par une sémantique précise, c'est-à-dire avec des modèles mathématiques. Une méthode de spécification est dite *formelle* lorsqu'elle utilise un ou plusieurs langages de spécification formels.

3.1.1 Langages basés sur les états et langages basés sur les événements

Il est possible de classer les méthodes de spécification formelles en deux groupes :

- les approches basées sur les états,
- et celles basées sur les événements.

Les méthodes basées sur les états représentent le système à travers deux modèles complémentaires : la partie statique permet de décrire les entités constituant le système et leurs états, tandis que la partie dynamique modélise les changements d'états que le système peut effectuer par l'intermédiaire d'opérations ou d'actions. Des propriétés d'invariance sont souvent définies sur le système pour assurer la cohérence du système. Les langages s'appuyant sur les états sont par exemple : Statechart [Har87], Esterel [BG92], ASM [Gur93], Action Systems [BKS83], VDM [Jon90], Z [Spi92], Object-Z [Smi00] et B [Abr96]. Nous illustrerons dans la section 3.2 les approches Action Systems, Z, Object-Z et B.

Les approches basées sur les événements représentent le système à travers des processus ou des agents, qui sont des entités indépendantes qui communiquent entre elles ou avec l'extérieur du système. Ces méthodes permettent de modéliser le comportement du système à l'aide de séquences ou d'arbres d'événements. Parmi les exemples de formalismes basés sur les événements, on peut

citer : les réseaux de Petri [Pet81], LOTOS [BB87], CCS [Mil89], CSP [Hoa85] et EB^3 [FSD03]. Dans la section 3.3, nous présenterons CCS et CSP.

Il existe également des approches différentes ou plus hybrides. Il y a par exemple les approches algébriques comme CASL [ABK$^+$02] ou Larch [GH93], ou bien des méthodes combinant plusieurs aspects comme RAISE [Geo91] ou LOTOS [BB87]. Dans la suite, nous nous concentrerons sur les méthodes basées sur les états et sur les événements.

3.1.2 Sémantique

La *sémantique* d'un langage est la définition de ce que les expressions de ce langage signifient. Elle permet en effet d'interpréter dans un modèle les symboles de la syntaxe utilisée.

Une approche possible consiste à définir le sens des expressions d'un langage en décrivant comment elles seraient exécutées. Cette approche de la sémantique est dite opérationnelle. Les systèmes de transitions étiquetés permettent ainsi de représenter les états et les transitions d'états possibles d'un système.

Les langages de type algèbre de processus comme CSP sont plutôt représentés par l'observation de leur comportement. Cette approche de la sémantique est alors dite dénotationnelle. Elle permet de relier un programme à son comportement.

Système de transitions étiqueté. Les systèmes de transitions étiquetés ou *labelled transition systems* (LTS) [Mil90] permettent de modéliser le comportement de systèmes basés sur les états lors de leur exécution.

Le système de transitions étiqueté $(A, S, \longrightarrow, R)$ d'un système P est défini par la donnée de quatre éléments :

- A est l'alphabet, c'est-à-dire l'ensemble des événements possibles du système,
- S est l'ensemble des états possibles,
- $\longrightarrow$ est la relation de transition d'états, avec $\longrightarrow \subseteq S \times A \times S$,
- et R est l'ensemble des états initiaux, avec $R \subseteq S$ et $R \neq \varnothing$.

La relation $\longrightarrow$ est définie par un ensemble de règles d'inférence, de la forme :

$$\frac{H_1 \ldots H_n}{G} \text{ CONDITION}$$

où les H_i sont des hypothèses et G est la conclusion. Ces règles permettent de définir le comportement de chaque opérateur de la syntaxe du langage.

Le meilleur moyen de représenter un LTS est un diagramme. La figure 3.1 est un exemple de LTS. Les cercles représentent les états du système tandis que que les flèches modélisent les transitions d'états. L'état initial est marqué par le symbole “>”. Dans cet exemple, l'alphabet du LTS est :

$$A = \{B, C, D, E, F, G, H, I, J, K, L\}$$

L'ensemble des états possibles est :

$$S = \{q_0, q_1, q_2, q_3, q_4, q_5\}$$

L'ensemble des états initiaux est :

$$R = \{q_0\}$$

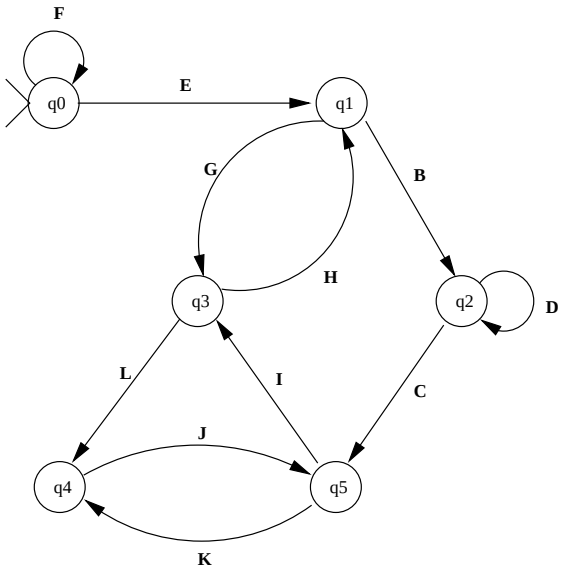


FIG. 3.1 – Exemple de LTS

Dans l'exemple de la figure 2.1, on a bien :

$$(q_0, E, q_1) \in \longrightarrow$$

La transition (q_0, E, q_1) est aussi dénotée par :

$$q_0 \xrightarrow{E} q_1$$

Chaque transition $q \xrightarrow{action} r$ de $\longrightarrow$ est représentée par une flèche dans le diagramme.

Comme le LTS permet de représenter aisément le comportement d'un système, il sera utilisé dans les chapitres suivants pour améliorer la compréhension du lecteur.

Sémantique dénotationnelle. Les modèles sémantiques présentés dans ce paragraphe constituent les modèles de base [Ros97] usuellement utilisés dans la communauté des algèbres de processus pour représenter les langages à processus comme CSP. La sémantique dénotationnelle d'un processus est un ensemble d'observations.

Le modèle le plus simple est celui des traces : un processus P est représenté par $traces(P)$, l'ensemble de toutes les séquences d'événements possibles que ce processus peut exécuter. Le *modèle des traces* identifie un processus avec l'ensemble de ses traces.

Le *modèle des échecs stables* identifie un processus avec l'ensemble de ses traces et l'ensemble de ses échecs stables. Un *échec stable* du processus P est une paire trace/refus (tr, X) où tr est une séquence d'événements que P peut exécuter en rejoignant un état stable et X est l'ensemble des événements que P peut refuser d'exécuter à partir de cet état stable. L'ensemble des échecs

stables d'un processus P est dénoté par $failures(P)$. Si tr est une trace de P telle que $(tr, \Sigma) \in failures(P)$, où Σ est l'ensemble de tous les événements, alors P atteint un état dans lequel aucun événement n'est exécutable et dans ce cas, P est bloqué. Un processus est dit *libre de blocage* s'il n'existe aucun ensemble des échecs stables de la sorte.

Un autre modèle permet de prendre en compte les divergences d'un processus. Un *état divergent* est un état du processus dans lequel il est possible d'exécuter une infinité d'événements internes[1]. Une *divergence* du processus P est une séquence d'événements tr telle que P atteint un état divergent après avoir exécuté tr. Un processus est dit *libre de divergence* si l'ensemble de ses divergences est vide. Un *échec* du processus P est une paire trace/refus. Le *modèle des échecs-divergences* identifie un processus avec l'ensemble de ses traces, l'ensemble de ses divergences et l'ensemble de ses échecs.

Les deux sections suivantes présentent des langages qui seront utilisés dans la suite du rapport.

3.2 Langages basés sur les états

Quatre approches basées sur les états sont présentés dans les paragraphes suivants : Action Systems, Z, Object-Z et B.

3.2.1 Action Systems

Les Action Systems sont basés sur les systèmes de transitions (qui sont des LTS sans étiquette). Plusieurs versions existent, comme celles de Back [BKS83] ou UNITY [MC88].

Un système d'actions est composé de :
- un espace d'états, défini par des variables d'états,
- des actions qui définissent une initialisation et des transitions sur ces variables d'états.

Dans l'approche Action Systems de Back [BKS83], les actions sont décrites par le langage de commandes gardées de Dijkstra [Dij76].

Le langage de Dijkstra est associé à un calcul de plus faible précondition. Dans ce langage, les programmes agissent sur des variables, sont séquentiels et terminent. Le langage de commandes gardées de Dijkstra comprend notamment des affectations de variables, des compositions séquentielles, des conditionnelles de type **IF THEN ELSE END** et des boucles **DO END**. Le comportement d'un programme est spécifié avec une condition (appelée *précondition*) sur les valeurs des variables avant l'exécution du programme et une condition sur ces valeurs après l'exécution du programme (*postcondition*).

Afin d'assurer la terminaison d'un programme, Dijkstra a défini une sémantique de plus faible précondition qui introduit l'opérateur wp. Si S est une commande et $post$ est une postcondition de S, alors :

$$wp(S, post)$$

représente tous les états initiaux à partir desquels il est assuré d'atteindre la postcondition $post$ en exécutant S. Ainsi S satisfait les conditions $(pre, post)$

[1] Les événements internes sont des événements qu'un processus peut exécuter mais qui ne sont pas observables par d'autres processus.

si :

$$pre \Rightarrow wp(S, post)$$

Plusieurs travaux sont basés sur le langage de commandes gardées de Dijkstra. Le calcul de raffinement introduit par Back [BvW98] est notamment une extension du calcul de plus faible précondition de Dijkstra. Les raffinements de données de Z et B sont également inspirés des travaux de Dijkstra et de Back.

Les actions dans les Action Systems de Back sont de la forme :

$$g \longrightarrow com$$

où g est une garde représentant des contraintes sur les variables d'états et *com* est une commande ou un appel de programme.

Une action $g \longrightarrow com$ est exécutable dans tout état satisfaisant la garde g. Un *système d'actions* commence par exécuter l'initialisation. Ensuite, les actions du programme sont analysées par l'évaluation de leur garde et une action est choisie parmi les actions exécutables pour être exécutée. Le système termine lorsqu'il atteint un état dans lequel plus aucune action n'est exécutable. Le système peut diverger si l'initialisation ou bien une des actions échoue.

Si le système termine, le programme est de la forme :

$$I\ ;\ \textbf{do}\ A_0 \Box A_1 \Box ...\ \textbf{od}$$

où I désigne la commande d'initialisation et les A_i représentent les actions du système. Le symbole $\Box$ est un choix non déterministe des actions. L'expression **DO OD** est un pseudo-langage pour désigner une boucle de type **WHILE** true **DO END**.

Une contrainte des Action Systems de Back est la terminaison du système uniquement lorsqu'il n'y a plus d'action exécutable. Une variante possible est fournie par UNITY [MC88] : les actions sont déterministes, n'échouent pas et sont toujours exécutables. Le système peut donc terminer à tout moment. Les propriétés des systèmes d'action sont souvent exprimées à l'aide de la logique temporelle [Pnu77].

Les trois langages décrits dans les trois sections suivantes sont assez proches. Le langage Z est basé sur la théorie des ensembles et sur la logique du premier ordre. [Spi92] est un manuel de référence complet sur le langage Z. Le langage Object-Z [Smi00] est une version orientée objet de Z. Enfin, B [Abr96] est à la fois un langage et une méthode puisqu'il permet de spécifier un système depuis son analyse jusqu'à son implémentation. Il dispose en outre de nombreux outils pour assister l'utilisateur.

3.2.2 Langage Z

Le *schéma* est la notion de base des spécifications Z. Un schéma est une boîte contenant des descriptions utilisant les notations Z. Les schémas sont utilisés pour décrire les états d'un système, les états initiaux ou bien les opérations.

Statique. La partie statique permet de définir les états et les relations d'invariant qui sont préservées lors des transitions d'états. Elle est décrite en Z sous la forme d'un schéma d'état.

Par exemple, le *Birthday Book* est un exemple connu de système qui permet de retenir les dates d'anniversaire [Spi92]. Les types de base de ce système sont :

$[NAME, DATE]$. Le schéma *BirthdayBook*, qui permet de définir les aspects statiques du système, est dénoté en Z par :

BirthdayBook

$$known : \mathbb{P}\, NAME$$
$$birthday : NAME \pfun DATE$$
$$known = \mathrm{DOM}\, birthday$$

La première partie du schéma correspond à la déclaration des variables *known* et *birthday*. La seconde partie est la définition de l'invariant. La variable *known* est ici égale au domaine de la variable *birthday*.

Dynamique. Les aspects dynamiques concernent les opérations, les relations entre les entrées et les sorties, et les changements d'états.

Dans l'exemple précédent, l'opération *AddBirthday* permet de rajouter une date d'anniversaire dans le système :

AddBirthday

$$\Delta BirthdayBook$$
$$n? : NAME$$
$$d? : DATE$$
$$n? \notin known$$
$$birthday' = birthday \cup \{n? \mapsto d?\}$$

La notation Δ indique que l'état du schéma *BirthdayBook* sera modifié par cette opération. La notation ? signifie que les variables $n?$ et $d?$ sont des paramètres d'entrée de l'opération. Le prédicat $n? \notin known$ est la précondition de l'opération. La notation $birthday'$ indique un changement d'état de la variable *birthday* par exécution de l'opération. En l'occurrence, l'opération a pour effet de rajouter un élément à la variable *birthday*. Un paramètre de sortie d'une opération est dénoté en Z par le nom de la variable suivi d'un !.

Pour initialiser, un schéma *InitBirthdayBook* est défini :

InitBirthdayBook

$$BirthdayBook$$
$$known = \varnothing$$

La variable *known* est ici initialisée par l'ensemble vide.

Raffinement de données. Le *raffinement* permet de remplacer les types de données abstraits d'une spécification Z par des types de données plus concrets. Le raffinement d'un schéma d'état est, en outre, accompagné des raffinements des opérations qui modifient l'état du schéma raffiné. Les opérations sont par conséquent à nouveau spécifiées en utilisant les nouveaux types de données définis dans le raffinement du schéma du système.

Par exemple, *BirthdayBook* est raffiné par le schéma :

*BirthdayBook*1

$$names : \mathbb{N}_1 \rightarrow NAME$$
$$dates : \mathbb{N}_1 \rightarrow DATE$$
$$hwm : \mathbb{N}_1$$

$$\forall i, j : 1..hwm \bullet i \neq j \Rightarrow names(i) \neq names(j)$$

Ce schéma définit les nouvelles variables *names*, *dates* et *hwm*. Cette nouvelle spécification est plus concrète, car les variables sont désormais représentées par des tableaux de données.

Le schéma *Abs* définit la *relation d'abstraction* entre les variables abstraites (*known* et *birthday*) et les variables concrètes (*names*, *dates* et *hwm*) :

Abs

$$BirthdayBook$$
$$BirthdayBook1$$

$$known = \{i : 1..hwm \bullet names(i)\}$$
$$\forall i : 1..hwm \bullet$$
$$\quad birthday(names(i)) = dates(i)$$

La variable abstraite *known* est remplacée dans le raffinement par un ensemble de noms. La variable concrète *hwm* représente le nombre de noms disponibles dans le carnet et *names* est un tableau de noms. La variable abstraite *birthday* permet de relier les éléments du tableau de noms *names* aux éléments du tableau de dates *dates*.

Les opérations Z sont raffinées en spécifiant les opérations définies dans les schémas abstraits avec les types de données concrets. Par exemple, l'opération *AddBirthday* est raffinée par :

*AddBirthday*1

$$\Delta BirthdayBook1$$
$$n? : NAME$$
$$d? : DATE$$

$$\forall i : a..hwm \bullet n? \neq names(i)$$
$$hwm' = hwm{+}1$$
$$names' = names \oplus \{hwm' \mapsto n?\}$$
$$dates' = dates \oplus \{hwm' \mapsto d?\}$$

L'opération a les mêmes paramètres d'entrée et de sortie que dans la spécification abstraite. L'ajout d'une date d'anniversaire dans le carnet est désormais spécifié en incrémentant la variable *hwm* de 1, et en surchargeant les variables *names* et *dates* pour compléter les tableaux de données par le nouveau nom et la nouvelle date respectivement.

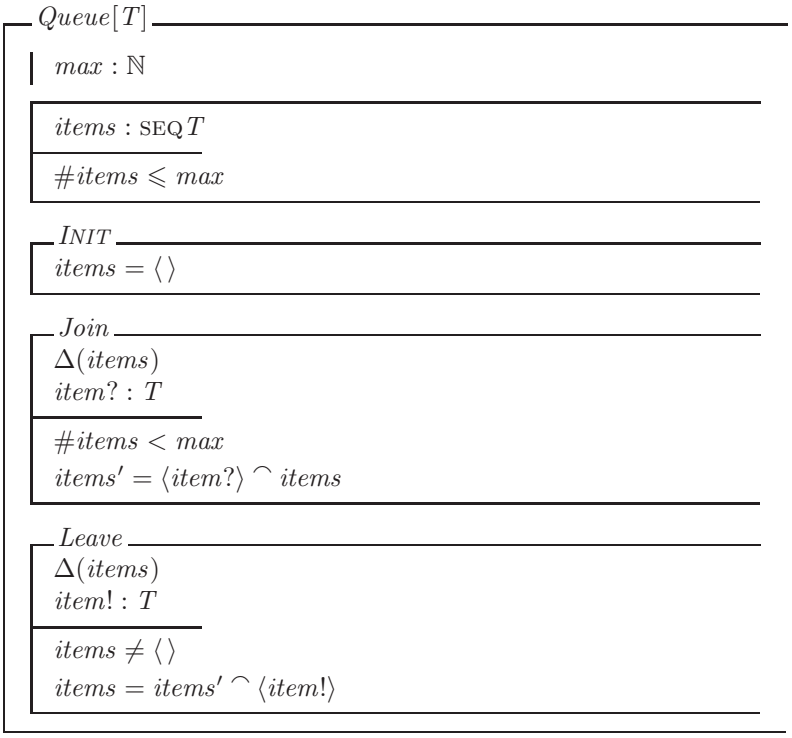
3.2.3 Object-Z

Object-Z est une extension du langage Z qui permet de spécifier des systèmes dans un style orienté objet. Dans une spécification Z, il est difficile de déterminer

les conséquences des opérations sur un schéma d'état donné, car les schémas d'opération peuvent agir sur les états de plusieurs schémas d'état. La notion de classe est introduite pour regrouper dans un même schéma toutes les opérations la concernant.

Notion de classe. Dans Object-Z, la structure appelée *classe* permet de décrire à la fois un schéma d'état et des schémas d'opérations qui agissent sur cet état. Une classe Object-Z est représentée par une boîte contenant :
- la liste des classes héritées,
- des définitions de types,
- des définitions de constantes,
- un schéma d'état,
- un schéma d'état initial,
- et des schémas d'opération.

Le schéma d'état ne porte généralement pas de nom dans une classe Object-Z.



Par exemple, la classe *Queue[T]* définit une file d'attente de type FIFO. Elle est représentée comme une séquence d'éléments dont le type est défini en paramètre de la classe. La file d'attente a une capacité maximale *max*. Deux opérations sont définies, *Join* et *Leave*. Elles permettent respectivement d'ajouter et de retirer un élément de la file. Contrairement aux schémas d'opération en Z, une Δ-liste des variables modifiées est spécifiée pour chaque opération. Enfin, la classe *Queue[T]* n'hérite pas d'autres classes.

Héritage et instanciation. L'*héritage* permet à une classe de considérer les définitions d'une autre classe, en particulier les définitions de type, de constante

et les schémas. L'*instanciation* permet de renommer les variables, les types et les constantes d'une classe.

Les deux mécanismes sont généralement liés. Par exemple, la définition d'un canal de transmission de messages avec pertes (lossy channel system) peut être considérée comme l'instanciation et l'héritage de la classe *Queue[T]*. Si *MSG* est l'ensemble de tous les messages possibles, alors :

MsgChannel

$Queue[MSG][msgs/items, msg?/item?, msg!/item!]$

Lose

$\Delta(msgs)$

$msgs' = tail\ msgs$

est la classe Object-Z définissant un canal de transmission de messages avec pertes. Les variables *items*, *item*? et *item*! sont renommées par *msgs*, *msg*? et *msg*! respectivement. Une nouvelle opération, *Lose*, est en outre définie.

3.2.4 Langage B

Le langage B est un langage de spécification formel, basé sur la notion de machine abstraite. Les fondements théoriques de la méthode sont spécifiés dans le B-Book [Abr96].

Machine abstraite. Une *machine abstraite* représente un état spécifié par une partie statique (à l'aide de variables d'état et des propriétés d'invariance) et une partie dynamique (à l'aide d'opérations). Le langage pour la description de la statique repose sur la théorie des ensembles et sur la logique du premier ordre. Les variables sont ainsi typées par des ensembles et les invariants sont spécifiés à l'aide de conjonctions de prédicats du premier ordre. L'état de la machine abstraite ne peut être modifié que par des opérations. Le langage permettant d'exprimer la partie dynamique est un langage de substitutions généralisées. Il permet de décrire les opérations qui font évoluer l'état du système modélisé. Lors des phases initiales de spécification, le langage est abstrait : les instructions des opérations utilisent des préconditions et de l'indéterminisme. Les différentes clauses d'une machine abstraite B sont présentées dans le tableau 3.1.

Exemple_Machine est un exemple de machine abstraite spécifiée avec le langage B :

MACHINE *Exemple_Machine*

VARIABLES x

INVARIANT
$x \in 0 .. 20$

INITIALISATION
$x := 10$

TAB. 3.1 – Clauses d'une machine abstraite en B

Clauses	Description
MACHINE	Nom et paramètres de la machine
CONSTRAINTS	Définition des propriétés des paramètres de la machine
SETS	Liste des ensembles abstraits et définition des ensembles énumérés
CONSTANTS	Liste des constantes de la machine
PROPERTIES	Définition des propriétés des constantes et des ensembles
VARIABLES	Liste des variables d'état de la machine
INVARIANT	Définition des types et des propriétés des variables
DEFINITIONS	Liste d'abréviations pour les prédicats, les expressions ou les substitutions
INITIALISATION	Initialisation des variables d'état
OPERATIONS	Liste des opérations de la machine

OPERATIONS
change =
pre
$x + 2 \leq 20 \wedge$
$x - 2 \geq 0$
then
choice
$x := x + 2$
or
$x := x - 2$
end
end

On remarque que, dans le corps de l'opération abstraite *change*, la substitution est spécifiée à l'aide de la commande **choice** qui n'est pas déterministe. Dans ce cas, la variable abstraite x peut être substituée par $x+2$ ou par $x-2$. La précondition (**pre**) permet de faire respecter l'invariant (voir la clause **INVARIANT**) si l'opération *change* est exécutée.

Raffinement. Une machine abstraite B est ensuite raffinée. Cette phase permet de passer d'une structure abstraite à une structure proche du code. Le raffinement B se fait en plusieurs étapes successives. Les préconditions des opérations deviennent alors de plus en plus larges et les instructions de plus en plus déterministes. Les machines issues du raffinement ne contiennent alors ni précondition, ni indéterminisme. Par exemple,

REFINEMENT *Exemple_Raffinement*

REFINES *Exemple_Machine*

VARIABLES y

INVARIANT
$y \in 0 .. 10 \wedge$
$x = 2 \times y$

INITIALISATION
$y := 5$

OPERATIONS
change =
begin
$y := y + 1$
end

On a introduit dans cette machine une variable concrète y. L'invariant $x = 2 \times y$ permet de relier cette nouvelle variable y avec la variable abstraite x de la machine abstraite. Cet invariant est appelé un invariant de *collage*. Cette nouvelle machine est plus concrète que *Exemple_Machine*, puisque l'opération *change* est désormais déterministe et sans précondition.

Outil. À la différence des approches précédentes, B possède un outil très puissant qui couvre toutes les phases de la méthode de conception. L'Atelier B [Cle], commercialisé par la société Clearsy, est en effet un environnement permettant de gérer des projets en langage B. Il offre différentes fonctionnalités :

- automatisation de certaines tâches (vérification syntaxique, génération automatique de théorèmes à démontrer, traduction de B vers C, C++, ...),
- aide à la preuve pour démontrer automatiquement des théorèmes,
- aide au développement.

Plus précisément, le prouveur de l'Atelier B permet de vérifier quatre points importants des spécifications B :

- au niveau de la machine : la dynamique doit respecter la statique,
- au niveau de l'initialisation : l'initialisation (clause **INITIALISATION**) établit l'invariant (clause **INVARIANT**),
- au niveau des opérations : chaque opération (clause **OPERATIONS**) doit préserver les propriétés d'invariance (clause **INVARIANT**),
- l'Atelier B permet enfin de prouver la correction du raffinement par rapport au modèle initial.

Le prouveur B est un outil de preuve interactif. Dans un premier temps, il permet de générer les obligations de preuve (de la forme : hypothèses $\Rightarrow$ conclusion) qui sont classées selon deux catégories : les obligations dont la preuve est évidente (notamment dans les cas où la conclusion fait partie des hypothèses) et les autres. L'Atelier B dispose alors d'un prouveur automatique de force variable. La force est un compromis entre l'efficacité et la rapidité du prouveur. Si, malgré l'exécution du prouveur automatique, il reste encore des obligations à prouver, l'utilisateur doit les prouver interactivement en utilisant les tactiques du prouveur.

Bien plus qu'un langage, B est une méthode de spécification complète. L'Atelier B peut être considéré comme faisant partie intégrante de la méthode B. Intéressons-nous désormais aux approches basées sur les événements.

3.3 Langages basés sur les événements

Dans cette section, nous présentons les langages CCS et CSP.

3.3.1 CCS

CCS (Calculus of Communicating Systems) a été défini par R. Milner [Mil80]. CCS est fondé sur l'observation d'*agents* qui représentent une partie unitaire d'un système concurrent à modéliser.

Agent. La notion d'agent est vague et peut désigner des parties plus ou moins atomiques du système. Un agent peut donc être composé de plusieurs sous-agents. Les deux notions fondamentales de CCS sont concurrence et communication. La *concurrence* est caractérisée en CCS par l'indépendance des *actions* d'un agent par rapport aux autres agents du système. La *communication* des agents en CCS est de deux types : les actions peuvent agir à l'intérieur d'un agent ou bien interagir avec ses agents voisins. Le comportement d'un système est défini en CCS par l'observation de ses actions.

Un agent contient deux *ports* qui permettent de communiquer avec l'extérieur : un port d'entrée et un port de sortie. Une action de label a entrant dans un agent est dénotée simplement par a, tandis qu'une action sortante de même label est dénotée par $\bar{a}$. Les actions a et $\bar{a}$ sont dites *complémentaires*.

Comportement d'un agent. Un agent est défini par une expression spécifiant les actions possibles et le comportement final de l'agent. La syntaxe du langage CCS est la suivante. La préfixation (.) permet d'associer une action à un agent résultant de l'exécution de cette action. Par exemple,

$$A = \alpha.A'$$

représente l'agent A qui accepte une action α et se comporte ensuite comme l'agent A'. Les actions en CCS sont indéterministes et récursives.

La sommation (+) permet de représenter deux comportements possibles. Par exemple, l'action C définie par :

$$C = A+B$$

se comporte soit comme l'agent A, soit comme l'agent B.

La composition de deux agents A et B est dénotée par $A \mid B$. Dans ce cas, les actions complémentaires sont synchronisées et deviennent des actions *internes* du résultat de la composition. Par exemple, si les agents A et B sont définis par :

$$\begin{aligned} A &= a.A'+\bar{c}.A'' \\ B &= b.B'+c.B'' \end{aligned}$$

alors, les actions a et b demeurent indépendantes dans la composition $A \mid B$. Par exemple, la transition d'état suivante est possible :

$$A \mid B \xrightarrow{a} A' \mid B$$

La composition des actions complémentaires c et $\bar{c}$ est une action interne de l'agent $A \mid B$, elle est dénotée en CCS par τ :

$$A \mid B \xrightarrow{\tau} A'' \mid B''$$

Restriction et instanciation. L'exemple de composition précédent n'impose aucune restriction sur les actions c et $\bar{c}$: elles peuvent donc être exécutées dans la composition. Dans ce cas, il est possible d'agir sur la transition :

$$A \mid B \xrightarrow{c} A'' \mid B$$

Le symbole de restriction ($\setminus$) permet d'éviter ce type d'action. Dans ce cas, il n'est possible d'exécuter sur l'agent :

$$(A \mid B) \setminus \{c\}$$

que les actions autres que c et $\bar{c}$.

Il est enfin possible en CCS d'instancier un agent en utilisant des renommages de la forme *Agent*[*NewName*/*OldName*] où l'action *OldName* est instanciée par *NewName* dans l'agent *Agent*.

3.3.2 CSP

Le langage CSP (Communicating Sequential Processes) [Hoa85] est une notation utilisée pour décrire des systèmes concurrents. Le langage est supporté par quelques outils, comme FDR [For97], qui permettent d'analyser et de vérifier les spécifications en cours ou existantes. CSP a été inventé par C.A.R. Hoare et développé à l'Université de Oxford dans les années 80.

Syntaxe de CSP

En CSP, les *processus* sont des entités, indépendantes les unes des autres, mais qui peuvent communiquer entre elles. Un processus peut exécuter des *événements* (ou *actions*). Les événements permettent de décrire le comportement des processus. L'ensemble des événements que le processus P peut exécuter est appelé son *alphabet* (ou *interface*) et est dénoté par $\alpha(P)$. Le comportement le plus simple d'un processus est de ne rien faire : un tel processus est dénoté par $STOP$.

Préfixe. Le *préfixe* ($\rightarrow$) permet de définir un processus en explicitant les événements qu'il peut exécuter. Si a est un événement et P un processus, alors :

$$a \rightarrow P$$

est le processus qui peut exécuter a et se comporte ensuite comme le processus P. L'opérateur de préfixe est toujours utilisé sous cette forme avec un événement à la gauche de $\rightarrow$ et un processus à droite. Il est possible d'enchaîner les préfixes.

Dans ce cas, les parenthèses sont implicites et portent sur les processus à droite de la flèche (associativité à droite). Par exemple,

$$a \rightarrow b \rightarrow Q$$

correspond à l'expression de processus :

$$a \rightarrow (b \rightarrow Q)$$

Récursivité. Les définitions *récursives* permettent de décrire en CSP des processus qui agissent sans fin. Par exemple,

$$Alternative = On \rightarrow Off \rightarrow Alternative$$

est un processus défini de manière récursive. Il exécute les événements *On* et *Off* en alternance. Les processus peuvent ainsi être définis par mutuelle récursion. Par exemple,

$$\begin{aligned} Light\_Off &= On \rightarrow Light\_On \\ Light\_On &= Off \rightarrow Light\_Off \end{aligned}$$

Opérateurs de choix. Il existe plusieurs opérateurs de choix en CSP. Le plus simple, dénoté par $|$, agit sur les processus de la forme $a \rightarrow P$. Le processus :

$$a \rightarrow P \mid b \rightarrow Q$$

peut soit exécuter l'événement a et se comporter ensuite comme le processus P, soit exécuter l'événement b et se comporter ensuite comme le processus Q. Les préfixes sont nécessairement distincts ($a \neq b$). Il est possible d'utiliser $|$ avec plus de deux choix :

$$a \rightarrow P \mid b \rightarrow Q \mid ... \mid z \rightarrow R$$

L'opérateur $|$ se restreint aux processus de la forme $a \rightarrow P$. Il existe en CSP deux autres opérateurs de choix sur les processus : choix externe ($\Box$) et choix interne ($\sqcap$). Le processus $P \Box Q$ ($P \sqcap Q$ respectivement) peut exécuter tout événement que P ou Q peut exécuter. Après l'exécution de ce premier événement, le comportement de $P \Box Q$ ($P \sqcap Q$ respectivement) est soit celui de P, soit celui de Q, selon sur quel processus agissait le premier événement. Le choix est dit *externe*, si le choix du premier événement dépend d'un autre processus. Le choix est dit *interne*, si le choix est non déterministe. Pour définir les choix internes, CSP introduit la notion d'*événement interne*, dénotée par τ, qui représente un événement du processus qui n'est pas observable par les autres processus du système considéré.

Événements cachés. Il est possible à partir d'un processus P de *cacher* certains événements de son alphabet. Si A est un ensemble d'événements et si P est un processus, alors :

$$P \setminus A$$

est le processus dont les événements appartenant à A deviennent des événements internes du processus. Par exemple, pour cacher l'événement a du processus P dans la transition :

$$P \xrightarrow{a} P'$$

on définit le processus $P \setminus \{a\}$ et la transition $\xrightarrow{a}$ devient :

$$P \setminus \{a\} \xrightarrow{\tau} P'$$

Composition parallèle. Pour décrire plusieurs processus en concurrence, CSP introduit l'opérateur de *composition parallèle* entre processus. Si A et B sont les alphabets des processus P et Q respectivement, alors :

$$P_A||_B Q$$

est la composition parallèle de P et de Q. Dans ce cas, P peut uniquement exécuter les événements de A, Q ceux de B, et tous les événements de l'intersection de A et de B sont exécutés par P et Q en synchronisation. Par exemple, supposons que les processus P et Q sont définis par :

$$\begin{aligned} P &= a \to c \to P' \\ Q &= b \to c \to Q' \end{aligned}$$

avec $A = \{a, c\}$ et $B = \{b, c\}$. Dans ce cas, les transitions

$$P_A||_B Q \xrightarrow{a} (c \to P')_A||_B Q$$

et

$$P_A||_B Q \xrightarrow{b} P_A||_B (c \to Q')$$

sont possibles. Dans le premier cas, il est ensuite possible d'agir sur Q :

$$(c \to P')_A||_B Q \xrightarrow{b} (c \to P')_A||_B (c \to Q')$$

Il est à présent possible d'exécuter c qui demande la synchronisation des processus :

$$(c \to P')_A||_B (c \to Q') \xrightarrow{c} P'_A||_B Q'$$

Par abus de notation, les alphabets des processus ne sont parfois pas précisés dans la composition parallèle. Par exemple, la composition parallèle des processus P et Q peut être dénotée par :

$$P \,||\, Q$$

Entrelacement. L'opérateur d'*entrelacement* ($|||$) est similaire à l'opérateur de composition parallèle, mais sans synchronisation. On note :

$$P \;|||\; Q$$

l'entrelacement des processus P et Q. Les événements de P sont alors exécutés indépendamment de l'exécution des événements de Q. Par exemple, si P et Q sont définis par :

$$\begin{aligned} P &= a \rightarrow b \rightarrow P' \\ Q &= c \rightarrow Q' \end{aligned}$$

Les séquences d'événements possibles pour $P \,|||\, Q$ sont : (a, b, c), (a, c, b) et (c, a, b). Ces séquences sont appelées les *traces* du processus $P \,|||\, Q$.

Entrées et sorties. Un événement en CSP peut être représenté par un message passant par un canal. Les événements sont donc de la forme $c.v$, où c est le nom d'un *canal* et v est la *valeur* du message passant par le canal c. Le *type* d'un canal est l'ensemble des valeurs possibles pouvant passer par ce canal.

Les valeurs des messages peuvent être exprimées en CSP en termes d'*entrées* ou de *sorties* des canaux dans les processus avec préfixe. Le processus :

$$c\,!v \rightarrow P$$

est un processus dont la valeur v est une sortie du canal c et qui se comporte ensuite comme le processus P. Dans cette expression de processus, v doit vérifier que $v \in T$, avec T le type du canal c.

De même,

$$c\,?x : T \rightarrow P(x)$$

est un processus dont le paramètre x de type T est une entrée du canal c et qui se comporte ensuite comme le processus paramétré $P(x)$.

Quantification. Les opérateurs de choix interne et externe, de composition parallèle et d'entrelacement ont une forme plus générale pour considérer des combinaisons d'un nombre quelconque de processus. Si I est un ensemble fini d'indices, alors :

$$\sqcap_{i \in I} P_i$$

définit le choix interne entre les processus P_i, avec $i \in I$. De même,

$$\Box_{i \in I} P_i$$

est le choix externe entre tous les processus P_i, avec $i \in I$. Si A_i est l'interface de P_i, pour chaque $i \in I$, alors :

$$||_{A_i}^{i \in I}\ P_i$$

est la composition parallèle de tous les P_i. L'entrelacement de plusieurs processus P_i, avec $i \in I$, est dénoté par :

$$|||_{i \in I}\ P_i$$

Dans la plupart des méthodes formelles étudiées, le raffinement constitue une activité à part entière du processus de conception. Après ce bref rappel concernant les langages de spécifications formels, intéressons-nous maintenant au raffinement.

Chapitre 4

Raffinement

> "... le programme concret implémente (ou raffine) le programme abstrait correctement lorsque toute utilisation du programme concret ne conduit pas à une observation qui n'est pas aussi une observation du programme abstrait."
>
> — Paul Gardiner et Carroll Morgan

Le raffinement constitue une activité de plus en plus importante dans les méthodes de spécification formelle. Nous allons résumer dans ce chapitre les principales approches du raffinement.

4.1 Introduction sur le raffinement

Le raffinement est une approche de plus en plus répandue pour construire progressivement des programmes corrects : il consiste à dériver par étapes successives une spécification initiale en vérifiant que chaque transformation du programme préserve bien sa correction vis-à-vis de la spécification précédente.

Il existe selon les langages utilisés de nombreuses notions de raffinement qui ne sont pas toujours équivalentes. Si le raffinement est invariablement une préservation de la correction, celui-ci s'exprime et se vérifie de différentes manières.

La méthode la plus courante est la recherche d'une relation de simulation de la spécification concrète par sa spécification abstraite. Les approches étudiées proposent la vérification de conditions suffisantes sur la relation afin d'assurer la correction du raffinement. Selon les sémantiques considérées, les conditions ne s'expriment pas de la même façon.

4.2 Raffinement : préservation de la correction

La notion de raffinement a été introduite par Wirth [Wir71] et par Dijkstra [Dij76] dans les années 1970, puis formalisée par Back [Bac78, Bac88] dans les années 1980. Plusieurs travaux ont ensuite développé cette notion, en particulier Abadi et Lamport [AL88], Back [BvW98], Morgan [Mor88, Mor90], Morris [Mor87] et Abrial [Abr96]. Le raffinement est un moyen de construire de manière progressive des programmes corrects.

4.2.1 Correction des programmes

L'intérêt des méthodes formelles est la possibilité de vérifier de manière rigoureuse des propriétés sur un modèle formel. Pour assurer qu'un programme respecte certaines propriétés, on lui associe une spécification formelle sur laquelle il est possible de raisonner mathématiquement.

Définition et notation. Dans la logique de Hoare [Hoa69], un programme S est associé à une précondition P et une postcondition Q. La précondition P permet de caractériser les états possibles avant l'exécution de S, tandis que la postcondition Q est vérifiée par les états possibles après l'exécution de S.

Un programme S est *totalement correct* vis-à-vis de sa spécification P, Q si, à partir de tout état initial vérifiant la précondition P, le programme termine et fait passer le système dans un état satisfaisant la postcondition Q. Si la terminaison n'est pas assurée, la correction est dite *partielle*. Un programme S totalement correct par rapport à sa spécification P, Q est dénoté dans la suite par $P < S > Q$.

Notion de contrat. Back [BvW98] introduit la notion de *contrat* pour interpréter la correction des programmes. Un contrat est une généralisation des programmes et des spécifications. Il est défini par un langage sur les instructions *inst*. L'instruction la plus simple est une affectaction de valeur. Les instructions sont composées entre elles par des séquences et des choix. Toute instruction peut être associée à une assertion *cond* et à une hypothèse H. Un cas typique de contrat C est de la forme :

$$[H_1]inst_1^j\{cond_1\}\ ;\ ...\ ;\ [H_i]inst_i^j\{cond_i\}\ ;\ ...\ ;\ [H_n]inst_n^j\{cond_n\}$$

Les indices j représentent les entités concernées par l'instruction. Les entités agissent sur le système en exécutant les instructions qui leur sont associées dans le contrat. Une entité peut être un utilisateur, le système ou bien l'environnement : ce sont les *acteurs* du contrat.

Le contrat peut être vu comme une règle du jeu pour les acteurs. Comme le choix est un opérateur sur les instructions, la règle du jeu encadre et limite les options de chaque acteur. Une instruction *inst* n'est exécutable que si l'assertion *cond* qui lui est associée est vérifiée. Les choix d'un acteur peuvent donc conduire un autre acteur à se retrouver bloqué si toutes les assertions deviennent fausses. Les assertions correspondent à la notion de garde. Les hypothèses H représentent les assomptions du contrat. On ne s'intéresse pas dans le contrat à ce qui se passe si les hypothèses ne sont pas vérifiées : cela correspond à la notion de précondition. La correction et le raffinement sont interprétés dans cette sémantique des jeux.

Un ange contre un démon. Les acteurs considérés sont un ange et un démon. L'état initial est σ. Le but est d'arriver à l'état q à partir de σ en respectant le contrat C. La correction est associée au point de vue de l'ange.

Les choix du démon risquent de bloquer l'ange et le contrat ne sera alors pas respecté. Si l'ange parvient à l'état q tout en respectant le contrat, alors le démon aura perdu. La difficulté consiste donc à trouver une stratégie gagnante pour l'ange quels que soient les choix du démon.

Pour respecter son contrat, l'ange doit faire ses choix de manière à :

– faire échouer le démon quelles que soient ses options : dans ce cas, le démon se retrouvera bloqué et l'ange aura gagné,
– ou bien sortir des termes du contrat : si les hypothèses ne sont plus vraies, le contrat n'a plus besoin d'être respecté et dans ce cas, l'ange aura gagné.

La correction d'un programme correspond donc à l'existence d'une stratégie gagnante de l'ange pour faire passer le système de l'état σ à l'état q. Pour préserver la correction, le raffinement correspond à une augmentation des chances de réussite de l'ange et une diminution de celles du démon. Le raffinement se traduit donc par une augmentation des stratégies gagnantes de l'ange et par une diminution des stratégies gagnantes du démon.

L'idée d'un jeu entre un ange et un démon a d'abord été proposée par Hintikka [Hin72] puis développée par Moschovakis [Mos72], Aczel [Acz75] et Back [BvW89].

4.2.2 Sémantique relationnelle des programmes

La notion de contrat est plus générale qu'un programme défini comme une séquence d'opérations. Une sémantique courante pour représenter les programmes consiste à utiliser le *calcul relationnel* [Tar41]. Elle fut notamment décrite par de Bakker en 1980 [dB80] et Mili en 1983 [Mil83]. Les langages VDM [Jon90] et Z [Spi92], par exemple, utilisent cette vue relationnelle.

Représentation des programmes. Un programme est alors défini par trois sortes de relations :
– une initialisation,
– une séquence d'opérations,
– une finalisation.

Les relations gg représentant les programmes sont définies sur un monde global G :

$$gg : G \leftrightarrow G$$

Un programme S défini sur le monde T est une composition relationnelle de la forme :

$$S = ti\ ;\ top_1\ ;\ ...\ ;\ top_n\ ;\ tf$$

avec une relation d'initialisation $ti : G \leftrightarrow T$, des opérations $top_j : T \leftrightarrow T$ avec $j \in \{1, .., n\}$ et une relation de finalisation $tf : T \leftrightarrow G$.

Choix angélique et démoniaque. Dans le cadre de cette sémantique, il est possible d'interpréter le choix de deux manières distinctes : soit on considère que les choix dans le programme dépendent de l'ange et dans ce cas le choix est dit *angélique* ; soit le choix considéré est *démoniaque*, dans le cas contraire.

Dans le premier cas, un programme S arrive à la postcondition Q à partir d'un état initial σ (noté $\sigma\{|\ S\ |\}Q$) s'il existe un état σ' tel que :
– l'exécution de S permet de passer de l'état σ à l'état σ' (ce qui est noté : $\sigma S \sigma'$),
– et l'état σ' satisfait la postcondition Q (noté $Q.\sigma'$).

Il existe donc une stratégie gagnante pour l'ange, et on suppose qu'il choisira toujours cette exécution afin de satisfaire la spécification.

Dans le cas du choix démoniaque, $\sigma\{|\ S\ |\}Q$ est vraie si pour tout état σ' satisfaisant $\sigma S \sigma'$, on a : $Q.\sigma'$. Le démon gagne donc quels que soient ses choix.

La sémantique de la correction et du raffinement dépend donc de l'interprétation du choix. S est correct vis-à-vis de la précondition P et de la postcondition Q (noté $P < S > Q$) si pour tout état σ satisfaisant P,

$$\sigma\{|\ S\ |\}Q$$

Si le choix est interprété de manière angélique, alors :

$$\forall\, \sigma : P.\sigma, (\exists\, \sigma' : \sigma S \sigma' \wedge Q.\sigma')$$

Cette définition correspond à la correction totale : l'existence de σ' assure la terminaison du programme S. Si le choix est démoniaque, alors :

$$\forall\, \sigma : P.\sigma, (\forall\, \sigma' : \sigma S \sigma' \Rightarrow Q.\sigma')$$

La correction n'est alors que partielle, car l'existence d'un état final n'est pas assuré.

Le raffinement dépend également de l'interprétation du choix. Si elle est angélique, alors l'ange augmente ses chances de gagner. En termes de relation, cela se traduit par le fait que $S \subseteq S'$ si S est raffiné par S'. Si le choix est démoniaque, alors $S' \subseteq S$ quand S est raffiné par S'.

4.3 Raffinement : un problème de sémantique

L'expression $P < S > Q$ contient trois paramètres : S, P et Q. Le problème est différent selon que S est connu ou pas. La correction de programmes suppose que le programme S existe déjà. Si S n'est pas connu, le problème consiste à dériver progressivement un programme correct simple. On fixe $P < S > Q$ et on cherche S' tel que :

$$\forall\, P, Q, (P < S > Q \Rightarrow P < S' > Q)$$

Autrement dit, toute spécification satisfaite par S est aussi satisfaite par S'. On peut donc remplacer S par S', tout en continuant à satisfaire la spécification initiale. S' est un raffinement de S, ce qui est dénoté par :

$$S \sqsubseteq S'$$

La relation $\sqsubseteq$ doit vérifier trois propriétés importantes : elle est réflexive, transitive et monotone. Un programme est ainsi le raffinement de lui-même par réflexivité. La propriété de transitivité permet de raffiner un programme par étapes successives. Enfin, la monotonie de la relation sert à raffiner les parties d'un programme séparément. La dérivation est ainsi réalisée de manière progressive.

Il est toutefois difficile d'analyser et de comparer des expressions de la forme $P < S > Q$. Plusieurs sémantiques ont permis de fixer quelques paramètres afin de simplifier les vérifications. La correction et le raffinement dépendent en effet de la sémantique utilisée pour interpréter $P < S > Q$.

4.3.1 Sémantique des transformateurs de prédicats

Pour vérifier la correction des programmes, Dijkstra [Dij76] a introduit la notion de "plus faible précondition", notée *wp*, qui s'appuie sur la logique de Hoare. L'opérateur *wp* permet de calculer la plus faible précondition qui garantisse la terminaison d'un programme S et qui laisse le système dans un état qui satisfait la postcondition Q. Pour un programme S et une postcondition Q, $wp(S, Q)$ est la plus faible précondition P telle que $P < S > Q$.

La correction et le raffinement sont maintenant exprimés dans cette sémantique. Un programme S de précondition P et de postcondition Q est alors correct si :

$$P \Rightarrow wp(S, Q)$$

Dans la sémantique *wp*, le raffinement se traduit par :

$$S \sqsubseteq S' \Leftrightarrow (\forall\, Q, wp(S, Q) \Rightarrow wp(S', Q))$$

À partir d'un programme S et d'une postcondition Q, l'opérateur *wp* permet de revenir sur les préconditions possibles. La sémantique *wp* est dite *backward*.

De nombreux raffinements dans les méthodes formelles sont basés sur la sémantique *wp*, en particulier les raffinements de Z [DW96] et B [Abr96]. Les raffinements dans les Actions Systems [BvW98] et dans CSP [Hoa85] ont été reliés au raffinement des *wp*.

4.3.2 Sémantique des jeux

La notion de contrat présentée dans la section 4.2.1 est associée à des jeux, autrement dit les choix de chaque joueur, dans la sémantique du jeu. On note $gm(C)$ l'ensemble des jeux du contrat C.

Les notions de correction et de raffinement de la sémantique *wp* sont interprétées par la sémantique des jeux de la manière suivante : un programme S est correct s'il existe une stratégie gagnante pour l'ange. On note $ws(J, Q)$ le prédicat qui indique si le jeu J est une stratégie gagnante sous le contrat C pour atteindre un état qui satisfait la postcondition Q.

Back définit dans [BvW89] une sémantique opérationnelle des jeux avec des règles d'inférence.

4.3.3 Sémantique du choix

La sémantique *wp* permet de passer des postconditions aux préconditions. Elle est dite *backward*. Une sémantique *forward* est la sémantique du choix. Elle permet de considérer les ensembles de postconditions du programme S en fonction d'une précondition P. Ces ensembles sont dénotés par $ch(S, P)$. Un programme S de précondition P et de postcondition Q est correct si :

$$Q \in ch(S, P)$$

Intuitivement, on associe à tout état initial un ensemble de postconditions possibles pour l'état final. Une postcondition peut être vue comme un prédicat sur les états ou bien comme un ensemble d'états. Les ensembles d'ensembles

$ch(S, P)$ tels que définis par Back [BvW98] sont fermés par le haut. En particulier, pour tous prédicats Q, Q',

$$(Q \in ch(S, P) \wedge Q \subseteq Q') \Rightarrow Q' \in ch(S, P)$$

Le retrait d'éléments dans un ensemble d'états revient à ajouter de nouveaux ensembles dans l'ensemble d'ensembles. En termes de postcondition, le rajout d'un choix possible de postconditions revient à renforcer les postconditions dans ces choix. Le raffinement se traduit dans cette sémantique par :

$$S \sqsubseteq S' \Leftrightarrow (\forall\, P, ch(S, P) \subseteq ch(S', P))$$

Ces trois premières sémantiques, ainsi que la notion de contrat, sont détaillées dans le livre de Back [BvW98]. La figure 4.1 représente les liens entre les différentes sémantiques.

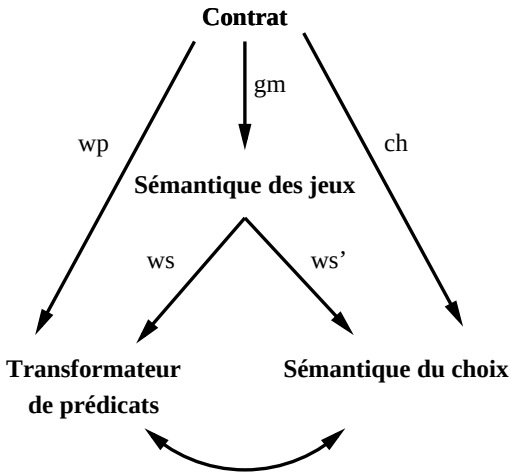


FIG. 4.1 – Liens entre les sémantiques wp, des jeux et du choix

En particulier, le lien entre la sémantique des jeux et les transformateurs de prédicats *wp* est l'existence de stratégies gagnantes pour l'ange :

$$wp(S, Q) = ws(gm(C), Q)$$

De même, il est possible de définir un opérateur ws' de manière à relier la sémantique des jeux à la sémantique du choix :

$$ch(S, P) = ws'(gm(C), P)$$

Enfin, pour tous P, Q,

$$(P \Rightarrow wp(S, Q)) \Leftrightarrow (Q \in ch(S, P))$$

4.3.4 Sémantique relationnelle

Dans la sémantique relationnelle (voir section 4.2.2), le raffinement s'exprime au niveau des types de données. Un type de donnée $\mathcal{P}$ est défini comme la donnée

de $(P, pi, pf, \{pop_j\})$. Un programme sur $\mathcal{P}$ est défini comme une séquence de la forme :

$$pi\ ;\ pop_1\ ;\ ...\ ;\ pop_n\ ;\ pf \subseteq gg_{\mathcal{P}}$$

où $gg_{\mathcal{P}}$ désigne l'ensemble des programmes sur le type de donnée $\mathcal{P}$.

Un type de donnée $\mathcal{A}$ est raffiné par un type de donnée $\mathcal{C}$ si l'ensemble des programmes possibles de $\mathcal{C}$ est inclus dans l'ensemble de tous les programmes possibles de $\mathcal{A}$. Autrement dit :

$$\mathcal{A} \sqsubseteq \mathcal{C} \Leftrightarrow gg_{\mathcal{C}} \subseteq gg_{\mathcal{A}}$$

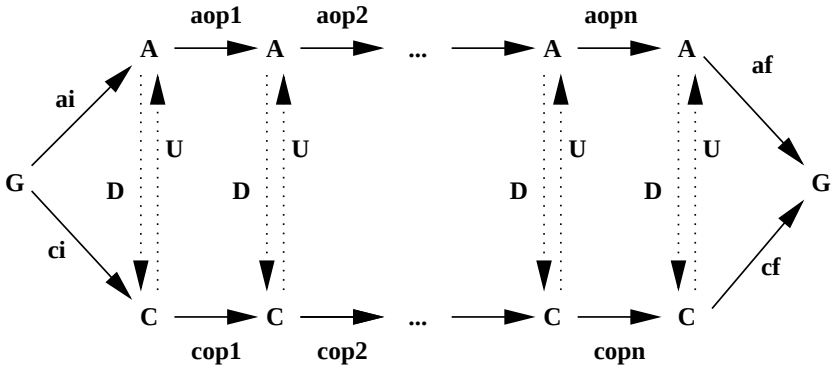


FIG. 4.2 – Relations de simulation

Afin de prouver ce raffinement sans raisonner sur l'espace de tous les programmes, He et al. montrent que deux ensembles de règles sont suffisants pour prouver le raffinement [HHS86]. La figure 4.2 représente les relations d'un programme à travers le monde abstrait $\mathcal{A}$ et à travers le monde concret $\mathcal{C}$. Les règles de He et al. s'appliquent si les séquences d'opérations dans $\mathcal{A}$ et dans $\mathcal{C}$ sont les mêmes, dans le sens où on définit une bijection entre les opérations de $\mathcal{A}$ et les opérations de $\mathcal{C}$. Une relation de simulation permet de relier les types de données $\mathcal{A}$ et $\mathcal{C}$.

Une relation de simulation *forward* ou *downward* (voir sens de la relation sur la figure 4.2) est une relation $D : A \leftrightarrow C$ telle que :

$$\begin{aligned} ci &\subseteq ai\ ;\ D \\ D\ ;\ cf &\subseteq af \\ D\ ;\ copi &\subseteq aopi\ ;\ D \end{aligned}$$

Une relation de simulation *backward* ou *upward* est une relation $U : C \leftrightarrow A$ telle que :

$$\begin{aligned} ci\ ;\ U &\subseteq ai \\ cf &\subseteq U\ ;\ af \\ copi\ ;\ U &\subseteq U\ ;\ aopi \end{aligned}$$

Les conditions sur les relations de simulation sont appelées respectivement les règles *forward* et *backward*. Les règles de He et al. sont toutefois limitées,

car elles imposent la totalité des relations. Plusieurs travaux, comme [BDW99], ont permis ensuite de considérer des cas plus généraux, avec des relations non totales. Le livre de de Roever [dRE98] fait le point sur les différentes techniques de simulation d'un point de vue théorique.

4.3.5 Sémantique opérationnelle : LTS

Un système de transition étiqueté (ou LTS) est généralement défini par la donnée de :

- un ensemble d'états E,
- un ensemble d'états initiaux I, avec $I \subseteq E$,
- un ensemble d'étiquettes de transitions (ou actions) L,
- et une relation de transition $T \subseteq E \times L \times E$.

Dans le cadre de programmes utilisant des variables, le LTS est augmenté d'une fonction l qui associe à chaque état s de E des affectations de valeurs aux variables V du système, sous la forme d'une conjonction de propositions d'états correspondant aux différentes variables. Un chemin entre deux états s et s' de E est une séquence finie de transitions $a_1, ..., a_n$ de T telles qu'il existe des états intermédiaires $s_1, ..., s_{n-1}$ de E vérifiant :

$$s \xrightarrow{a_1} s_1 \xrightarrow{a_2} s_2 \xrightarrow{a_3} ... \xrightarrow{a_{n-1}} s_{n-1} \xrightarrow{a_n} s'$$

Une définition possible du raffinement au niveau des LTS a été donnée dans [BJK00, Dar02]. Soient $ST_1 = (E_1, I_1, L_1, T_1, l_1)$ et $ST_2 = (E_2, I_2, L_2, T_2, l_2)$ deux LTS définis comme ci-dessus. Pour faire le lien entre les états abstraits de E_1 et les états concrets de E_2, on utilise un invariant de collage $I_{1,2}$ et une relation de collage $\rho : E_2 \times E_1$ telle que :

$$s_2 \ \rho \ s_1 \Leftrightarrow ((l_2(s_2) \wedge I_{1,2}) \Rightarrow l_1(s_1))$$

Pour vérifier que le LTS ST_1 est raffiné par ST_2, il suffit de montrer que les conditions suivantes sont satisfaites : elles permettent de prouver que les nouvelles transitions introduites par T_2 ne contredisent pas le comportement de ST_1.

1. Le raffinement strict des transitions consiste à vérifier que pour chaque transition de T_2, il existe une transition correspondante dans le LTS abstrait :

$$(s_2 \ \rho \ s_1 \wedge s_2 \xrightarrow{a} s_2' \in T_2) \Rightarrow \exists \, s_1'(s_1 \xrightarrow{a} s_1' \in T_1 \wedge s_2' \ \rho \ s_1')$$

2. Le bégaiement de transition est l'introduction d'une transition muette τ entre deux états concrets correspondant au même état abstrait :

$$(s_2 \ \rho \ s_1 \wedge s_2 \xrightarrow{\tau} s_2' \in T_2) \Rightarrow s_2' \ \rho \ s_1$$

3. Un état qui n'a pas de transition avec un autre état est un état qui bloque dans le LTS. Il s'agit d'un blocage du système de transitions. Pour ne pas introduire de nouveaux blocages pendant le raffinement, on vérifie que chaque état qui bloque dans ST_2 (noté $\nrightarrow_2$) correspond à un état qui bloque dans ST_1 :

$$(s_2 \ \rho \ s_1 \wedge s_2 \nrightarrow_2) \Rightarrow s_1 \nrightarrow_1$$

4. Pour éviter que les nouvelles transitions τ ne prennent indéfiniment le contrôle, on vérifie la non τ-divergence. Si un état σ_1 de E_1 est collé à plusieurs états concrets, les k états sont distingués par la notation (σ_2, i), $i \in \{1, ..., k\}$.

   $\forall\, \sigma_2, k\ (\sigma_2 \in \Sigma(T_2) \wedge k \geqslant 0 \Rightarrow \exists\, a, k'(a \in L_1 \wedge k' \geqslant k \wedge (\sigma_2, k'-1) \xrightarrow{a} (\sigma_2, k') \in T_2))$

5. Le LTS concret doit préserver le non-déterminisme externe de T_1 :

   $(s_1 \xrightarrow{a} s_1' \wedge s_2\ \rho\ s_1) \Rightarrow \exists\, s_2', s_2'', s_1''(s_2'\ \rho\ s_1 \wedge s_2' \xrightarrow{a} s_2'' \in T_2 \wedge s_1 \xrightarrow{a} s_1'' \in T_1 \wedge s_2''\ \rho\ s_1'')$

6. Enfin, pour tout état initial concret, il existe un état initial abstrait qui lui est collé :

   $$\forall\, s_2 \in I_2\ \exists\, s_1 \in I_1 :\ s_2\ \rho\ s_1$$

Il existe toutefois d'autres notions de raffinement des LTS (voir relations de Josephs [Jos88] en section 4.4.3).

4.3.6 Sémantique dénotationnelle : traces-divergences

Pour finir cette revue des sémantiques, nous rappelons la sémantique dénotationnelle des algèbres de processus comme CSP [Hoa85] (voir chapitre 3). Elle repose sur l'observation du comportement des processus. Les trois principaux modèles sémantiques [Ros97] sont les traces, les échecs stables et les traces-divergences.

Le modèle des traces associe à chaque processus les séquences finies d'événements admises par ce processus. Les traces du processus P sont dénotées par *traces*(P). Ce modèle permet donc de représenter les comportements possibles des processus sous forme de traces.

Le modèle des échecs stables associe à chaque processus P les couples de la forme (t, E), où t est une trace finie admise par P et E est l'ensemble des événements que le processus ne peut pas exécuter après avoir exécuté les événements de t. L'ensemble de ces couples est noté *failures(P)*. Ce modèle permet de caractériser les blocages de P. En effet, si E est égal à l'ensemble des événements exécutables par P, alors P se retrouve bloqué.

Enfin, le modèle des échecs-divergences associe à chaque processus P l'ensemble de ses échecs stables et l'ensemble de ses divergences. Un processus P n'est divergent que s'il se retrouve dans un état dans lequel les seuls événements possibles sont les événements internes. Cet état est dit divergent. L'ensemble des divergences de P, noté *divergences(P)*, est l'ensemble des traces t telles que le processus se retrouve dans un état divergent après avoir exécuté t. Si le processus est déterministe, alors *divergences(P)* est vide.

Le raffinement consiste alors à calculer et à comparer les modèles sémantiques de deux processus. Le raffinement dépend donc du modèle considéré. Par exemple, dans le cas du modèle des échecs-divergences, si P et Q sont deux processus, alors Q raffine P si :

$$\begin{aligned} \mathit{failures}(Q) &\subseteq \mathit{failures}(P) \\ \mathit{divergences}(Q) &\subseteq \mathit{divergences}(P) \end{aligned}$$

Dans cet exemple, il n'est pas utile de comparer *traces*(P) et *traces*(Q), car par définition des échecs stables :

$$\mathit{failures}(Q) \subseteq \mathit{failures}(P) \Rightarrow \mathit{traces}(Q) \subseteq \mathit{traces}(P)$$

4.3.7 Conclusion

La notion de raffinement dépend donc de la sémantique considérée. Chaque sémantique permet d'interpréter la notion de correction d'une certaine manière et la vérification du raffinement en dépend. Dans l'expression $P < S > Q$, la sémantique fixe un ou plusieurs paramètres pour interpréter les autres.

Relier ces différentes notions de raffinement est difficile, car cela revient à les comparer dans une sémantique commune. Comme chaque sémantique constitue une vue particulière des aspects syntaxiques, cette comparaison n'est pas aisée.

De Roever [dRE98] a comparé les relations de simulations utilisées pour montrer le raffinement dans les langages orientés-modèles comme VDM et Z. Il définit les simulations de manière rigoureuse et compare les méthodes avec des correspondances de Galois. Back [BvW98] introduit son calcul du raffinement à l'aide de la notion générale de contrat et des trois sémantiques décrites ci-dessus : *wp*, *ch* et *gm*. Les fondements mathématiques du calcul du raffinement de Back reposent notamment sur la théorie des catégories.

Concernant les approches basées sur les états et celles basées sur les événements, plusieurs travaux ont permis de relier les sémantiques *wp* et relationnelle avec les modèles sémantiques de CSP : Josephs [Jos88] et Hoare [HHS86] ont défini les premières relations de simulation tandis que, plus récemment, Bolton [BD02] et Boiten [DB03] ont affiné les équivalences entre les sémantiques du raffinement.

4.4 Raffinement : applications dans les méthodes formelles

Les relations de raffinement présentées dans la section précédente sont souvent difficiles à vérifier dans le cas d'exemples précis. Les méthodes formelles utilisent généralement des relations de simulation ou des outils pour assurer la correction du raffinement.

La simulation est une technique qui consiste d'une part à expliciter une relation entre un programme et sa dérivation et d'autre part à vérifier que cette relation satisfait certaines propriétés. Ces conditions sont suffisantes pour assurer la propriété de raffinement. Dans la section précédente, nous avons donné des exemples de simulation dans la sémantique relationnelle et avec les LTS. Nous en présenterons trois autres dans les paragraphes suivants avec les langages B, Z et CSP. Une autre technique consiste à vérifier le raffinement par *model-checking*, comme dans le cas de CSP.

Nous présentons dans cette section les applications du raffinement dans cinq exemples de méthodes formelles : B, B événementiel, CSP, Z et Object-Z.

4.4.1 Langage B

Le langage B [Abr96] utilise un langage de substitutions généralisées pour modifier les états du système modélisé. Ce langage qui permet de relier les états avant et après l'exécution d'une opération s'appuie sur une notion de transformateur de prédicats qui est proche de celle de Dijkstra (*wp*). Dans la sémantique de B, l'équivalent de *wp* est dénoté par *str*.

Une substitution S est vue comme une relation. La sémantique de B s'appuie sur le calcul relationnel. Pour assurer la correction des programmes en B, plusieurs relations et prédicats sur les substitutions sont définies. Soit S une substitution quelconque, $pre(S)$ est l'ensemble de précondition, autrement dit l'ensemble des états pour lesquels la précondition de S est satisfaite. La relation $rel(S)$ relie les états avant et les états après l'exécution de S, elle exprime donc la dynamique de la substitution. Les définitions de ces relations sont détaillées dans [Abr96].

Le transformateur *str* vérifie en particulier :

$$str(S)(p) = pre(S) \cap \overline{rel(S)^{-1}[\overline{p}]}$$

$str(S)(p)$ permet de caractériser les états qui assurent la terminaison de S ($pre(S)$) mais qui ne risquent pas d'aboutir aux états de $\overline{p}$ par une des relations dynamiques de $rel(S)$ ($\overline{rel(S)^{-1}[\overline{p}]}$).

Le raffinement B est défini par :

$$S \sqsubseteq S' \Leftrightarrow (\forall\, a(a \subseteq s \Rightarrow str(S)(a) \subseteq str(S')(a)))$$

En utilisant la propriété exprimant *str* en fonction de *pre* et de *rel*, la relation de raffinement se traduit alors par :

$$S \sqsubseteq S' \Leftrightarrow (pre(S) \subseteq pre(S') \wedge rel(S') \subseteq rel(S))$$

Autrement dit, le raffinement affaiblit les préconditions et retire de l'indéterminisme dans les opérations.

Le raffinement B est vérifié à l'aide d'une relation de simulation *backward*. Par le théorème 11.2.4 du B-Book [Abr96], s'il existe une relation totale $v \in c \leftrightarrow b$ entre l'ensemble concret c de la machine N et l'ensemble abstrait b de la machine M telle que, pour chaque opération,

$$\begin{array}{lcl} v^{-1}[pre(T)] & \subseteq & pre(U) \\ v^{-1}\,;\, rel(U) & \subseteq & rel(T)\,;\, v^{-1} \end{array}$$

où T est la substitution abstraite de l'opération et U sa substitution concrète, alors N est bien un raffinement de M.

Les conditions suffisantes de la simulation sont ensuites traduites en termes d'obligations de preuve sur les invariants et sur les préconditions des opérations. Soient A une machine abstraite B d'invariant I et d'initialisation *Init* et C un raffinement de A dont l'initialisation est la substitution $Init'$. L'invariant de collage J de la spécification C se déduit de la relation de simulation v présentée ci-dessus : il relie les variables d'état concrètes de C avec les variables abstraites de la machine A. Les obligations de preuve du raffinement sont :

- L'initialisation du raffinement ne doit pas contredire l'initialisation de la machine raffinée :
$$[Init']\neg[Init]\neg J$$
- La machine abstraite et son raffinement contiennent les mêmes opérations : seules leurs substitutions et leurs préconditions diffèrent. Pour chaque opération, les invariants et la précondition abstraite P doivent d'une part

établir la précondition concrète P' et d'autre part éviter que la substitution concrète S' de l'opération n'empêche la substitution abstraite S d'établir l'invariant de collage J :

$$I \wedge J \wedge P \Rightarrow P' \wedge [S']\neg[S]\neg J$$

Le raffinement utilisé en B repose donc sur un concept analogue à celui exprimé avec le transformateur de prédicats *wp*, mais il se restreint de plus à des formes particulières de spécifications regroupées au sein de machines. Le raffinement de S par S' s'exprime sous la forme d'une inclusion de $str(S)$ dans $str(S')$. Pour simplifier la vérification d'une telle propriété, le raffinement est prouvé en B à l'aide d'obligations de preuve suffisantes.

4.4.2 B événementiel

Le B événementiel [AM98] est une évolution du langage B pour l'adapter à la spécification de systèmes complexes constitués de plusieurs composants.

Les principales différences avec B sont d'une part la considération d'un système fermé pour représenter l'ensemble des composants dans un seul modèle et d'autre part la définition du comportement sous la forme d'événements et non par des opérations comme en B. L'objectif est de prendre en compte l'ensemble du système.

Un événement est défini en B événementiel par une garde, ie. une condition bloquante qui assure la cohérence du système en cas d'exécution de l'événement, et d'une action exprimée à l'aide du langage de substitutions généralisées comme en B. Un événement est de la forme générale :

any $x, y, ...$ **where**
  $P(x, y, ..., v, w, ...)$
then
  $S(x, y, ..., v, w, ...)$
end

avec $x, y, ...$ des variables locales et $v, w, ...$ des constantes ou des variables d'état du système d'événements. Dans cet exemple, P est la garde et S est l'action. Lorsqu'aucune variable locale n'est définie, l'expression d'un événement se simplifie par :

select $P(v, w, ...)$
then $S(v, w, ...)$
end

Le raffinement en B événementiel permet de raffiner les structures de données comme en B, mais aussi de rajouter des détails avec la définition de nouveaux événements. Le raffinement des états s'exprime comme en B à l'aide d'un invariant de collage. Le raffinement au niveau des événements se traduit par un renforcement des gardes et par la préservation de l'invariant de collage. Soit un événement abstrait de la forme :

any x **where** $P(x, v)$

then $v := E(x, v)$
end

qui est raffiné par l'événement concret :

any y **where** $Q(y, w)$
then $w := F(y, w)$
end

avec comme invariant abstrait $I(v)$ et comme invariant de collage $J(v, w)$. Dans ce cas, l'obligation de preuve est :

$$I(v) \wedge J(v, w) \wedge Q(y, w) \Rightarrow \exists\, x(P(x, v) \wedge J(E(x, v), F(y, w)))$$

Les nouveaux événements permettent de détailler le comportement du système. L'ajout d'un nouvel événement correspond en fait à un raffinement d'un événement qui ne fait rien au niveau abstrait. Ainsi, pour un nouvel événement de la même forme que l'événement concret décrit ci-dessus, l'obligation de preuve du raffinement est la suivante :

$$I(v) \wedge J(v, w) \wedge Q(y, w) \Rightarrow J(v, F(y, w))$$

De plus, les nouveaux événements ne doivent pas prendre le monopole du contrôle : un variant $V(w)$ est défini dans ce but et l'obligation de preuve correspondante est :

$$I(v) \wedge J(v, w) \wedge Q(y, w) \Rightarrow V(F(y, w)) < V(w)$$

Enfin, une dernière contrainte exprimée par les obligations de preuve du raffinement en B événementiel permet d'éviter un plus grand nombre de blocages au niveau concret qu'au niveau abstrait. Pour chaque événement abstrait de la forme décrite ci-dessus, il faut prouver que :

$$I(v) \wedge J(v, w) \wedge P(x, v) \Rightarrow Q_1 \vee ... \vee Q_n$$

où les Q_i sont les gardes des événements du système concret.

Le raffinement en B événementiel est donc plus complexe qu'en B classique car il permet de rajouter de nouveaux événements. Cette possibilité impose la vérification d'un plus grand nombre d'obligations de preuve pour éviter des contradictions avec le comportement du système raffiné.

4.4.3 CSP

Le langage CSP [Hoa85] décrit le comportement d'un système sous la forme de processus communiquant les uns avec les autres. La sémantique de CSP repose sur l'observation des effets des processus (voir section 4.3.6) : modèles des traces, des échecs stables et des traces-divergences.

Comparer les processus dans ces modèles peut se révéler complexe à mettre en œuvre, car le calcul et la comparaison de tels ensembles sont souvent difficiles. Il existe toutefois des outils comme FDR [For97] qui permettent d'analyser les traces, les échecs et les divergences d'un processus. Ils sont cependant limités par la complexité des modèles des expressions de processus. Comme pour le

model-checking, l'analyse d'un processus avec FDR peut échouer à cause d'une explosion du nombre d'états.

Une autre approche consiste à se ramener à une sémantique opérationnelle des processus sous la forme de LTS. Josephs a défini deux relations de simulation cohérentes par rapport au raffinement CSP [Jos88]. Autrement dit, si une des deux relations présentées ci-après est vérifiée, elle est suffisante pour assurer le raffinement au sens CSP. Pour obtenir une condition nécessaire du raffinement, il faut considérer les deux relations de simulation conjointement.

Pour définir ses relations de simulation, Josephs impose comme contrainte que les deux processus aient le même alphabet, c'est-à-dire les mêmes ensembles d'événements pour chacun des processus. Un LTS est défini par la donnée de l'alphabet, des états possibles, de la relation de transition et des états initiaux. Soient $(A, S_1, \longrightarrow_1, R_1)$ et $(A, S_2, \longrightarrow_2, R_2)$ les LTS des processus P_1 et P_2 respectivement avec le même alphabet A. Par convention, l'ensemble des prochains événements du processus P à partir d'un état σ est défini par :

$$next_P(\sigma) = \{e \in A \mid \exists\, \sigma' \in S \bullet \sigma \xrightarrow{e} \sigma'\}$$

Si les deux processus ont les mêmes espaces d'états (ie. $S = S_1 = S_2$), alors P_1 est raffiné par P_2 si :

1. À chaque état, les processus peuvent s'engager sur les mêmes événements :

$$\forall\, \sigma \in S, next_{P_1}(\sigma) = next_{P_2}(\sigma)$$

2. Chaque transition de P_2 est aussi une transition de P_1 :

$$\longrightarrow_2 \subseteq \longrightarrow_1$$

3. Chaque état initial de P_2 est un état initial de P_1 :

$$R_2 \subseteq R_1$$

Josephs définit deux relations de simulation pour des processus dont les espaces d'état sont différents. P_1 est dit une *simulation vers le bas* de P_2 s'il existe une relation $D \subseteq S_1 \times S_2$ telle que :

1. $\forall\, \sigma_1 \in S_1, \sigma_2 \in S_2 \bullet \sigma_1 D \sigma_2 \Rightarrow next_{P_1}(\sigma_1) = next_{P_2}(\sigma_2)$
2. $\forall\, \sigma_1 \in S_1, \sigma_2, \sigma_2' \in S_2, e \in A \bullet \sigma_1 D \sigma_2 \wedge \sigma_2 \xrightarrow{e}_2 \sigma_2'$
   $\Rightarrow (\exists\, \sigma_1' \in S_1 \bullet \sigma_1 \xrightarrow{e}_1 \sigma_1' \wedge \sigma_1' D \sigma_2')$
3. $\forall\, \sigma_2 \in R_2 \bullet \exists\, \sigma_1 \in R_1 \bullet \sigma_1 D \sigma_2$

Avec cette règle *forward*, le processus P_1 peut simuler le comportement de P_2 tant que les deux processus sont dans des états correspondants.

Un processus P_1 est dit une *simulation vers le haut* de P_2 s'il existe une relation $U \subseteq S_2 \times S_1$ telle que :

1. $\forall\, \sigma_2 \in S_2 \bullet \exists\, \sigma_1 \in S_1 \bullet \sigma_2 U \sigma_1 \Rightarrow next_{P_1}(\sigma_1) \subseteq next_{P_2}(\sigma_2)$
2. $\forall\, \sigma_1' \in S_1, \sigma_2, \sigma_2' \in S_2, e \in A \bullet \sigma_2' U \sigma_1' \wedge \sigma_2 \xrightarrow{e}_2 \sigma_2'$
   $\Rightarrow (\exists\, \sigma_1 \in S_1 \bullet \sigma_1 \xrightarrow{e}_1 \sigma_1' \wedge \sigma_2 U \sigma_1)$
3. $\forall\, \sigma_1 \in S_1, \sigma_2 \in R_2 \bullet \sigma_2 U \sigma_1 \Rightarrow \sigma_1 \in R_1$

Avec cette relation, si P_2 atteint un certain état, alors P_1 est capable de "retracer" la séquence d'événements afin de trouver un état correspondant à partir duquel P_1 peut simuler le comportement de P_2 : il s'agit d'une simulation *backward*.

Josephs montre que si l'une des deux relations de simulation est vérifiée, alors le processus P_2 est un raffinement de P_1.

4.4.4 Z et Object-Z

Les règles de simulation présentées dans la section 4.3.4 sont adaptées aux langages Z [Spi92] et Object-Z [Smi00]. Il existe plusieurs règles de simulation possibles selon les interprétations de la totalité ou non des relations : voir travaux de Bolton, Davies et Woodcock [BDW99, Bol02, BD02] et Smith et Derrick [SD01].

Lorsqu'une relation n'est pas totale, cela signifie que certains états ne sont pas considérés par les programmes. Il existe alors deux interprétations possibles. Si la sémantique considérée est *bloquante*, alors les opérations ne peuvent pas être exécutées en dehors du domaine de la relation : les conditions associées aux états du domaine de la relation sont appelées des *gardes*. Si l'interprétation de la relation partielle est *non bloquante*, alors les opérations peuvent être exécutées en dehors du domaine, mais le résultat n'est pas garanti : en particulier, le système peut alors diverger. Ces conditions sont couramment appelées les *préconditions* des opérations.

Pour revenir à des relations de simulation totales comme définies par He et al. [HHS86], les relations partielles définies dans les sémantiques de Z et d'Object-Z sont rendues totales à l'aide du rajout de nouveaux éléments symbolisant l'échec, la réussite ou l'élément indéfini selon la sémantique considérée. Les relations classiques de simulation utilisées pour raffiner des schémas Z et des classes Object-Z sont présentées dans [DB01]. On se restreint dans la suite à une stratégie bloquante pour les relations partielles. Cela correspond à la sémantique d'Object-Z ou bien à des opérations gardées de Z.

Soient $A = (AState, AInit, \{AOp_i\}_{i \in I})$ et $C = (CState, CInit, \{COp_i\}_{i \in I})$ deux types de données Z. Les initialisations et les opérations relient les états avant et les états après exécution. Par convention, *State* dénote un état avant alors que *State*$'$ est un état après. La notation **pre** *Op* désigne le domaine de la relation *Op*. La relation R définie sur *AState* et *CState* est une simulation vers le bas de A vers C si :

1. $\forall\, CState' \bullet CInit \Rightarrow \exists\, AState' \bullet AInit \land R'$
2. $\forall\, i \in I, \forall\, AState, CState \bullet R \Rightarrow (\mathbf{pre}\ AOp_i \Leftrightarrow \mathbf{pre}\ COp_i)$
3. $\forall\, i \in I, \forall\, AState, CState, CState' \bullet R \land COp_i \Rightarrow R' \land AOp_i$

Concernant la simulation *backward*, T est une relation vers le haut de A vers C si :

1. $\forall\, CState \bullet \exists\, AState \bullet T$
2. $\forall\, AState', CState' \bullet CInit \land T' \Rightarrow AInit$
3. $\forall\, i \in I, \forall\, CState \bullet \exists\, AState \bullet T \land (\mathbf{pre}\ AOp_i \Rightarrow \mathbf{pre}\ COp_i)$
4. $\forall\, i \in I, \forall\, AState', CState, CState' \bullet (T' \land COp_i) \Rightarrow \exists\, AState \bullet T \land AOp_i$

Les travaux de Bolton [Bol02] ont montré que les relations de simulation définies pour Z et Object-Z par [DB01] n'étaient pas cohérentes par rapport au modèle des échecs-divergences de CSP (voir section 4.3.6) comme c'est le cas pour les règles de Josephs [Jos88] et de He et al. [HHS86]. Les relations de simulation ci-dessus sont valides et complètes par rapport au modèle des échecs singletons défini par Bolton dans [Bol02]. Ce modèle est un intermédiaire entre le modèle des traces-divergences et celui des échecs-divergences. Contrairement aux échecs stables, les échecs singletons considèrent au plus un événement comme échec, et non pas un ensemble d'événements comme dans la définition. La différence vient du fait que la sémantique relationnelle permet de prendre en compte l'exécutabilité des opérations de manière individuelle alors que le modèle des échecs stables caractérise les ensembles d'événements.

4.5 Conclusion

Les relations de simulation utilisées pour prouver le raffinement expriment des propriétés différentes selon les langages formels. Ces différences dépendent de la sémantique et de l'expressivité des langages.

4.5.1 Analyse et comparaison

Les tableaux 4.1 et 4.2 résument les principales caractéristiques de chacune des vérifications de raffinement présentées. Les abréviations utilisées dans les

Tab. 4.1 – Comparaison des approches - partie 1

Méthode	Langage B	B évén.	Z et Object-Z
Sémantique	wp	wp	relationnel
Ajout op. ou évén.	non	oui	non
Diversité des raffinements	1 relation	1 relation	2 types : 2 relations ind. 2 relations coll.
Oblig. de preuve	sim. back. [Abr96]	sim. back. [Abr00]	simulations [DB01, Bol02]
Outil de vérif.	Atelier B [Cle] B-Toolkit [B-C]	Atelier B [Cle]	

tableaux sont : sim. back. pour relation de simulation *backward*, sim. pour relations de simulation, m.c. pour *model-checking*, relations ind. pour les relations de simulation cohérentes par rapport au modèle des échecs singletons et relations coll. pour les celles qui sont cohérentes par rapport au modèle des échecs stables.

Sémantique. Le raffinement et sa vérification dépendent en premier lieu de la sémantique considérée. Les trois sémantiques présentées dans [BvW98], la sémantique relationnelle, la sémantique opérationnelle des LTS et la sémantique

Tab. 4.2 – Comparaison des approches - partie 2

Méthode	CSP	
Sémantique	traces, échecs et divergences	LTS
Ajout op. ou évén.	non	oui
Diversité des raffinements	3 modèles et 2 relations	2 relations
Oblig. de preuve	m.c. [Ros97] et sim. [HHS86]	simulation [Jos88]
Outil de vérif.	traces-div. : FDR [For97]	

dénotationnelle n'ont pas la même abstraction et la même vision des programmes. Les sémantiques *wp* et *ch* (sections 4.3.1 et 4.3.3) sont plus abstraites que la sémantique des jeux (section 4.3.2), dans le sens qu'il est possible d'exprimer des cas particuliers de *wp* dans la sémantique des jeux, tandis que la réciproque est fausse. D'autre part, les sémantiques *wp* et *ch* sont plus abstraites que la sémantique relationnelle, car elles prennent en compte simultanément les choix angéliques et démoniaques, alors qu'une seule interprétation à la fois n'est possible pour les relations.

Concernant les liens entre les approches basées sur les états et les approches événementielles, la sémantique des jeux qui associe à un contrat une séquence d'instructions semble analogue au modèle des traces de CSP (section 3.6) qui associe à un processus une séquence d'événements. Le modèle des échecs-divergences permet en outre de caractériser les blocages et les divergences du processus. Ce modèle est plus expressif que la sémantique *wp* (voir section 4.4.4) : la sémantique des transformateurs de prédicats correspond en fait au modèle des échecs singletons et des divergences de Bolton [Bol02]. Toutefois, la sémantique *wp* permet de représenter les opérations qui sortent des préconditions, cela n'est pas possible avec les LTS ou le modèle des échecs-divergences. Enfin, Derrick et Boiten montrent qu'il est possible de faire correspondre une sémantique au modèle des échecs-divergences en caractérisant les finalisations des programmes dans la sémantique relationnelle [DB03].

Simulations et outils. Les relations de simulation permettent de prouver le raffinement entre deux systèmes à l'aide de conditions suffisantes. Il existe deux principales relations : *forward* et *backward*. Toutes les méthodes formelles ne proposent pas les deux. Dans les cas de B et B événementiel, le raffinement est prouvé à l'aide d'obligations de preuve correspondant à une simulation *backward*. La disponibilité d'outils a également son importance : une méthode n'utilisant qu'un seul type de relation avec un bon outil peut s'avérer dans la pratique plus efficace et plus utile qu'une méthode sans outil.

4.5.2 Plusieurs notions de raffinement

Nous avons présenté quatre formes principales de raffinement : *wp*, séquences d'opérations, LTS et échecs-divergences. Le raffinement de séquences d'opérations (comme les types de données ou les machines B) permet de réduire le non déterminisme et d'augmenter les préconditions. Cette notion de raffinement est prouvée à l'aide de conditions suffisantes, grâce à des relations de simulation. Les algèbres de processus proposent des relations de raffinement plus ou moins fines selon les modèles sémantiques. Le raffinement peut alors être prouvé par *model-checking*. Il existe également des relations de simulation exprimées à l'aide de LTS pour prouver le raffinement en CSP. Le raffinement s'adapte donc aux caractéristiques et aux objectifs de chaque méthode et il constitue un critère important dans la sélection d'un langage de spécification formel.

Du raffinement B ... Dans le cas particulier du langage B, le raffinement permet d'une part d'élargir les préconditions des opérations et d'autre part de réduire le non-déterminisme. En outre, le raffinement se fait opération par opération. Ces propriétés sont dues au fait que les spécifications en B sont regroupées sous forme de machines.

... au raffinement EB^3. Dans le but d'intégrer les langages B et EB^3, il nous semble important que la méthode EB^3 soit dotée d'une relation de raffinement. Une première solution possible consiste à définir ce raffinement en fonction de celui de B. Nous avons constaté dans la revue de littérature que le raffinement dépend des caractéristiques du langage. Par conséquent, cette solution n'est pas suffisante si nous souhaitons tenir compte de toute l'expressivité du langage EB^3, comme par exemple la spécification des traces admissibles.

Une autre solution consiste à créer une notion de raffinement *ad hoc*, en s'inspirant des autres relations existant dans les langages proches d'EB^3, comme CSP. Notre objectif est aussi de définir une relation de raffinement qui s'adapte à notre proposition, la méthode EB^4 : le raffinement EB^3 doit pour cela être orienté de manière à faciliter la combinaison des approches B et EB^3.

Maintenant que nous avons présenté les principaux langages formels et la notion de raffinement, intéressons-nous désormais aux combinaisons de spécifications formelles.

Chapitre 5

Combinaison de spécifications formelles pour modéliser le comportement

> "Vos lacunes en mathématiques ne doivent pas vous inquiéter ... Je peux vous assurer que les miennes sont encore plus importantes ! "
>
> — Albert Einstein

Dans ce chapitre, nous allons étudier des exemples d'utilisation de combinaisons de spécifications formelles pour modéliser des systèmes et pour vérifier des propriétés.

Afin de comparer les différentes approches, nous avons utilisé un exemple de référence. Il s'agit d'une machine qui crée et détruit des produits. La machine de production, qui est dans l'état marche ou arrêt, ne peut produire que si elle est en service. Le LTS de ce système est représenté par la figure 5.1.

Une des propriétés dynamiques importantes que les modèles doivent respecter concerne la séquence des productions et destructions : un produit créé doit en effet être supprimé avant que la machine ne puisse créer un nouveau produit. Notre but est de décrire cette forte contrainte d'ordonnancement avec les différentes approches pour comparer leurs avantages et leurs inconvénients.

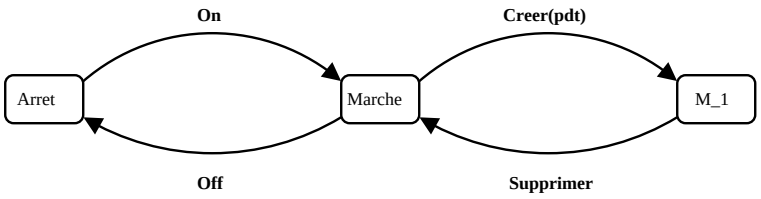


FIG. 5.1 – LTS de l'exemple

5.1 csp2B [But99]

Ce premier exemple de combinaison utilise les langages CSP [Hoa85] et B [Abr96]. Des descriptions de type CSP sont associées à des spécifications B standard pour contraindre l'ordonnancement des opérations définies dans la machine B. Un outil, appelé csp2B, permet de traduire ces descriptions CSP en des machines B. L'intérêt est de pouvoir ensuite utiliser les outils de la méthode B pour valider et pour vérifier des propriétés sur le système.

5.1.1 Syntaxe

Les machines CSP, qui permettent de définir les processus dans un formalisme proche des machines abstraites B, ont pour but :

- d'une part, de définir des processus et d'être traduites en des machines B par l'outil csp2B,
- et d'autre part, de contraindre l'ordonnancement des opérations d'une machine B.

Les processus CSP sont décrits sous la forme d'une machine, qui comporte deux clauses : la clause **ALPHABET** indique le nom des événements de la machine et la clause **PROCESS** décrit le comportement des processus de la machine.

La syntaxe utilisée pour décrire les processus est celle du langage CSP (voir paragraphe 3.3.2) avec quelques restrictions : $\rightarrow$ (préfixe), $\Box$ (choix externe), $||$ (composition parallèle), $|||$ (entrelacement), $STOP$ (processus de blocage). Les deux premiers opérateurs sont utilisés sans restriction. La composition parallèle est utilisée au niveau le plus à l'extérieur : l'outil csp2B n'accepte les compositions parallèles que de deux séquences de processus. L'entrelacement n'est accepté que pour des instances multiples de processus similaires. Tout appel récursif de processus est préfixé par un événement. Le langage csp2B accepte également des processus de la forme **IF** c **THEN** P_1 **ELSE** P_2, avec c une condition et P_1 et P_2 des processus.

Par exemple, la machine *VendingMachine* décrit le comportement d'un distributeur de boissons chaudes :

MACHINE *VendingMachine*
ALPHABET *Coin*, *Tea*, *Coffee*
PROCESS $VM = AwaitCoin$
WHERE
$AwaitCoin = Coin \rightarrow DeliverDrink$
$DeliverDrink = Tea \rightarrow AwaitCoin \Box Coffee \rightarrow AwaitCoin$
END

Le processus *VM* défini par la machine *VendingMachine* peut être représenté par le LTS de la figure 5.2. Les états q_0 et q_1 représentent les états *AwaitCoin* et *DeliverDrink* respectivement.

Le langage est toutefois limité, car il ne propose pas de quantification et l'utilisation des opérateurs est restrictif. Dans le cadre des systèmes d'information, l'approche csp2B ne convient pas car il n'est pas possible de considérer un grand nombre d'utilisateurs en concurrence, avec notamment des entrelacements et des compositions parallèles complexes.

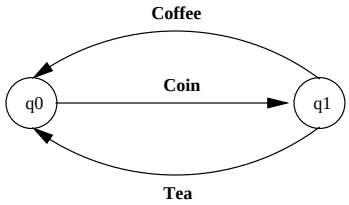


FIG. 5.2 – LTS du processus *VM*

Une machine CSP peut enfin être utilisée pour contraindre l'ordre d'exécution des opérations d'une machine B standard. Dans ce cas, **CONJOINS** est une clause dans la description CSP qui permet d'indiquer le nom de la machine B contrainte, qui est spécifiée de manière classique en B. Il n'y a en effet aucune restriction sur le langage B dans cette approche. Cette possibilité est la propriété est la plus intéressante de csp2B, car elle permet de relier une description CSP avec une machine B.

5.1.2 Exemple

L'approche csp2B nécessite donc la définition de deux machines : une machine B et une machine CSP.

La machine B permet de spécifier les structures de données et les opérations du système qu'on veut modéliser. Concernant notre exemple de référence (voir figure 5.1), on définit une première variable d'état, *Produit*, qui représente les produits créés. Une autre variable, P, est définie pour garder en mémoire le dernier produit créé par la machine de production.

La machine B spécifiant le système, avec les opérations de création et de suppression de produit, est :

MACHINE *Exemple_Act*
SETS *PRODUITS*
VARIABLES *Produit*, *P*
INVARIANT $Produit \subseteq PRODUITS \wedge P \in PRODUITS$
INITIALISATION $Produit := \varnothing \parallel P :\in PRODUITS$
OPERATIONS
$Creer\_Act(pdt) =$
pre $pdt \in PRODUITS - Produit$
then
$Produit := Produit \cup \{pdt\} \parallel$
$P := pdt$
end ;
$Supprimer\_Act =$
pre $P \in Produit$
then $Produit := Produit - \{P\}$
end
END

Dans l'invariant, l'expression :

$$Produit \subseteq PRODUITS$$

indique que la variable d'état *Produit* est incluse dans l'ensemble abstrait *PRODUITS* qui représente l'ensemble de tous les produits qu'il est possible de créer. La notation $:\in$ dans l'initialisation :

$$P :\in PRODUITS$$

signifie qu'un élément arbitraire de l'ensemble *PRODUITS* est choisi pour initialiser la variable P.

Les opérations de la machine *Exemple_Act* sont spécifiées de manière classique en B. Par exemple, l'opération *Creer_Act* a pour précondition que le produit *pdt* à créer appartient bien aux produits qui ne sont pas encore créés. Les substitutions de cette opération permettent d'une part d'ajouter le produit dans les produits créés et d'autre part de garder en mémoire ce produit dans la variable P.

La machine CSP suivante, *Exemple*, contrôle les opérations de la machine B *Exemple_Act*, qui est indiquée dans la clause **CONJOINS**. Par convention, lorsqu'un événement *Op* fait référence à une opération B, cette dernière est spécifiée sous le nom *Op_Act* dans la machine B contrainte.

MACHINE *Exemple*
CONJOINS *Exemple_Act*
SETS *PRODUITS*
ALPHABET *On*, *Off*, *Creer*(*pdt : PRODUITS*), *Supprimer*
PROCESS $EX = Arret$
CONSTRAINS *Creer*, *Supprimer*
WHERE
$Arret = On \rightarrow Marche$
$Marche = (Creer?pdt \rightarrow Supprimer \rightarrow Marche) \Box (Off \rightarrow Arret)$
END

Le processus *EX* défini dans la machine *Exemple* permet ainsi de contraindre les opérations *Creer_Act* et *Supprimer_Act* de la machine B. **CONSTRAINS** est une clause utilisée pour indiquer les événements contraints par le processus. L'accès aux variables d'état définies dans la machine B contrainte est limité à une simple lecture dans la machine CSP. La variable *Produit* de la machine *Exemple_Act* pourrait en effet être utilisée comme paramètre de l'état *Marche* afin de représenter les produits créés. Cette spécification est ici inutile, car la valeur de la variable *Produit* est toujours l'ensemble vide lorsque la machine se trouve dans cet état.

Le processus *EX* définit deux états intermédiaires, *Arret* et *Marche*. À partir de l'état *Arret*, il n'est possible d'exécuter que l'action *On*, qui n'est pas contrainte par la machine CSP.

Ensuite, le processus se trouve à l'état *Marche*. De cet état, le processus ne peut exécuter que deux actions. Soit il crée un produit qu'il devra ensuite supprimer avant de retourner à l'état *Marche*, soit il exécute *Off* et il se retrouve à l'état *Arret*.

L'action *Creer* est définie dans la partie CSP de la spécification comme une communication par canal (?), car elle correspond à une opération B avec un paramètre d'entrée : *Creer_Act*. Le type du paramètre est défini dans la clause **ALPHABET**.

Le système de notre exemple de référence est donc modélisé par une machine B décrivant l'état et les opérations et par une machine CSP qui décrit les contraintes d'ordonnancement sur les opérations de la machine B. On remarquera que le LTS du processus *EX* respecte celui de la figure 5.1. L'approche csp2B permet donc de bien représenter ce type d'exemple.

5.1.3 Outil csp2B

L'outil csp2B permet de générer une machine B à partir d'une machine CSP qui vérifie la syntaxe décrite dans le paragraphe 5.1.1.

La traduction de CSP vers B est effectuée de la manière suivante. Les états possibles d'un processus CSP sont représentés par un ensemble énuméré en B. L'état d'un processus est donné par une variable d'état dans la machine B. Les événements sont traduits en des opérations gardées de la forme **select then end**. Chaque opérateur du langage est ensuite traduit en B. Par exemple, la composition parallèle de processus devient une composition parallèle des substitutions correspondant aux traductions des processus. Les autres opérateurs sont détaillés dans [But99].

Dans notre exemple, la machine CSP *Exemple* est traduite en B par :

MACHINE *Exemple*
INCLUDES *Exemple_Act*
SETS $PRODUITS\ ;\ EXState = \{Arret, Marche, Marche\_1\}$
VARIABLES *EX*
INVARIANT $EX \in EXState$
INITIALISATION $EX := Arret$
OPERATIONS
$On =$
select $EX = Arret$
then
$EX := Marche$
end ;
$Off =$
select $EX = Marche$
then
$EX := Arret$
end ;
$Creer(pdt) =$
pre
$pdt \in PRODUITS - Produit \wedge pdt \in PRODUITS$
then
select $EX = Marche$
then $EX := Marche\_1$
end
||
select $EX = Marche$

```
  then Creer_Act(pdt)
  end
end;
Supprimer =
pre
  P ∈ Produit
then
  select EX = Marche_1
  then EX := Marche
  end
  ||
  select EX = Marche_1
  then Supprimer_Act
  end
end
END
```

Dans cette machine, la variable d'état *EX* correspond aux différents états du processus décrit dans la machine CSP : *Arret*, *Marche* et *Marche_1*. Ce dernier est un état implicite généré automatiquement par l'outil csp2B. Il correspond à l'analyse de la séquence du processus suivante :

$$Creer?pdt \rightarrow Supprimer \rightarrow Marche$$

Lorsque l'action *Creer* a été exécutée, le processus se retrouve dans l'état implicite *Marche_1*.

Chaque événement de l'alphabet est traduit en une opération B gardée de la forme **select then end**, ce qui permet de s'assurer de la bonne exécution des opérations de la machine. Un événement du processus ne peut en effet s'exécuter que si le processus se trouve dans l'état requis.

La clause **CONJOINS**, qui permet de contraindre l'ordonnancement des opérations d'une machine B, se traduit par une inclusion de cette machine. De plus, chaque événement de la machine CSP est traduit en B de la manière suivante. L'opération a des préconditions si l'opération de la machine contrainte en a ou bien si l'événement CSP correspondant est une communication avec un paramètre d'entrée. Dans ce cas, les préconditions de l'opération sont la conjonction de toutes ces conditions.

La garde de l'opération est la même que dans le cas sans **CONJOINS**. Le corps de l'opération est la composition parallèle du corps de l'opération tel qu'il aurait été généré sans la présence de la clause **CONJOINS** et d'un appel sous la forme **select then end** de l'opération correspondante de la machine B contrainte. Par exemple, le corps de l'opération *Creer(pdt)* est :

```
select EX = Marche
then EX := Marche_1
end
||
select EX = Marche
then Creer_Act(pdt)
end
```

La première substitution correspond à une traduction de l'événement *Creer* qui fait passer l'état du système de *Marche* à *Marche_1*. L'autre substitution est simplement un appel gardé de l'opération B correspondant à l'événement *Creer*. La forme présentée correspond à une génération automatique de l'outil, mais l'opération pourrait être simplifiée par :

Creer(*pdt*) =
pre *pdt* ∈ *PRODUITS* − *Produit*
then
 select *EX* = *Marche*
 then
  Creer_Act(*pdt*) ||
  EX := *Marche_1*
 end
end

5.1.4 Sémantique de la traduction

La traduction de CSP vers B est justifiée par une approche opérationnelle de la sémantique. La machine CSP et la machine B générée sont ainsi représentées par deux systèmes de transitions étiquetés (LTS) qui sont équivalents. Ces LTS sont similaires au LTS de la figure 5.1. Par convention, on indice les éléments du LTS (voir paragraphe 3.1.2) par *CSP* pour la machine CSP et par *csp2B* pour la machine B obtenue par traduction. Dans l'exemple précédent, on a bien que :

$$A_{CSP} = A_{csp2B} = \{On, Off, Creer, Supprimer\}$$

Les états possibles du système et les états initiaux sont :

$$\begin{aligned} S_{CSP} = S_{csp2B} &= \{Arret, Marche, Marche\_1\} \\ R_{CSP} = R_{csp2B} &= \{Arret\} \end{aligned}$$

Enfin les relations de transition définies par le processus *EX* dans la machine CSP sont équivalentes aux relations de transition définies par les opérations gardées de la machine B générée.

Cette équivalence est garantie par une normalisation par transformation syntaxique des équations du processus CSP par l'outil csp2B afin de déterminer les différentes actions (implicites et explicites) exécutées par le processus. Ces actions sont ensuites traduites en B par des opérations gardées en utilisant des règles de traductions. Par conséquent, $\longrightarrow_{CSP}$ et $\longrightarrow_{csp2B}$ sont équivalentes et les LTS $(A_{CSP}, S_{CSP}, \longrightarrow_{CSP}, R_{CSP})$ et $(A_{csp2B}, S_{csp2B}, \longrightarrow_{csp2B}, R_{csp2B})$ le sont aussi.

5.1.5 Vérification

Comme les machines CSP se traduisent en des machines B grâce à csp2B, il est possible de vérifier certaines propriétés sur le processus en utilisant la clause **ASSERTIONS** de la machine B obtenue. La difficulté consiste à exprimer

ces propriétés en B afin de pouvoir utiliser les outils supportant la méthode B, comme l'Atelier B.

Par exemple, il est difficile de vérifier avec l'approche csp2B que les actions *Creer* et *Supprimer* ne sont exécutables que si l'état du système est préalablement passé à l'état *Marche*, car cette propriété s'exprime mal avec des prédicats du langage B.

L'utilisation de B constitue donc à la fois un avantage et un inconvénient. Les propriétés concernant les états sont assez faciles à exprimer en B, car une variable d'état est définie dans la machine B pour représenter les états du système. Les ordonnancements des événements sont en revanche difficiles à exprimer en B, car les actions sont traduites par des opérations et il n'est pas possible d'exprimer des prédicats sur les opérations en B. Il est toutefois possible d'utiliser les substitutions de ces opérations.

5.1.6 Raffinement

Le raffinement csp2B d'une machine CSP est défini grâce à un invariant d'abstraction déclaré dans la clause **INVARIANT** du raffinement csp2B. L'invariant d'abstraction permet d'exprimer les états de contrôle concrets du raffinement csp2B en fonction des états de contrôle abstraits de la machine CSP raffinée.

La figure 5.3 représente une méthode de raffinement possible en utilisant l'outil csp2B. Comme les variables d'état en B correspondent aux états des pro-

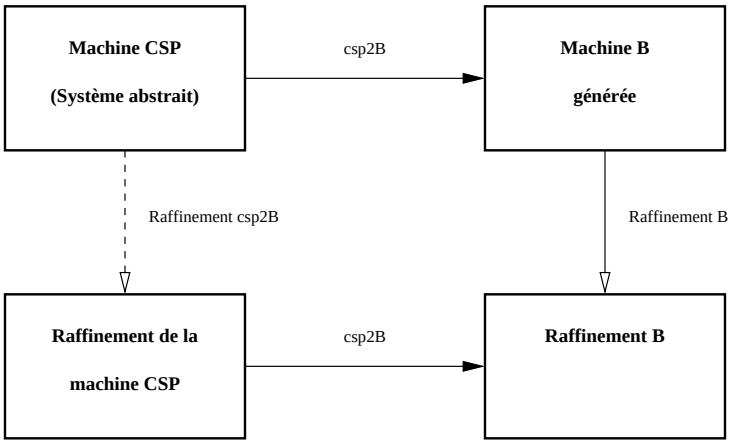


FIG. 5.3 – Méthode de raffinement csp2B

cessus en CSP, cette notion de raffinement csp2B est analogue au raffinement usuellement défini en B. Une traduction de la machine CSP avec l'outil csp2B peut donc être utilisée pour expliciter les états implicites des processus et le raffinement classique en B de la machine obtenue peut servir à définir l'invariant du raffinement csp2B. Cet invariant est ensuite inclus dans l'invariant du raffinement csp2B comme invariant d'abstraction. Un exemple de raffinement est donné dans [But99].

Le raffinement csp2B est donc défini en fonction du raffinement B. L'approche ne propose donc pas de nouvelle notion de raffinement.

5.1.7 Conclusion

Cette approche propose une définition B à des processus utilisant un sous-ensemble de la syntaxe CSP. Une sémantique opérationnelle basée sur les LTS est définie pour les processus CSP et pour les machines B correspondantes afin de justifier la traduction de CSP vers B. Un outil, csp2B, qui s'appuie sur cette sémantique est ainsi proposé aux utilisateurs afin de traduire des descriptions de type CSP en des machines B.

L'avantage de cette approche est de s'appuyer sur la méthode B, ce qui permet de profiter des outils existants pour le processus de vérification. D'autre part, la notation CSP acceptée qui est assez riche se traduit facilement en B. En revanche, les descriptions CSP ne supportent pas le choix non déterministe interne, les événements cachés et les compositions parallèles et les entrelacements arbitraires. Concernant la combinaison de spécifications formelles, l'outil csp2B devient intéressant lorsqu'on utilise la clause **CONJOINS** qui permet à une machine CSP de contrôler l'ordonnancement des opérations d'une machine B.

Outre les restrictions liées au sous-langage CSP utilisé, cette méthode ne permet pas de préserver le double point de vue des spécifications car tous les mécanismes sont définis en fonction de la traduction B. Par exemple, la notion de raffinement csp2B est définie par le raffinement B. Une méthode basée sur la traduction en B est en effet proposée.

Les propriétés sur les ordonnancements sont difficiles à exprimer en B, ce qui rend la vérification de ces propriétés difficile avec l'approche csp2B. Comme l'approche est fondée sur l'utilisation de la méthode B pour représenter les processus, la vérification des propriétés est donc limitée.

Si pour ces raisons, l'approche csp2B ne convient pas pour modéliser le comportement des systèmes d'information, elle est revanche très intéressante du point de vue des outils. Grâce à la combinaison des outils csp2B et Atelier B, il est possible d'exprimer certains types d'expressions CSP en B. Les machines obtenues peuvent ensuite être analysées avec l'Atelier B.

5.2 CSP || B [ST02]

Cette approche utilise une combinaison des langages CSP [Hoa85] et B [Abr96]. Un processus CSP est défini sous la forme d'une boucle récursive afin de contrôler les opérations d'une machine B. Il agit ainsi comme un contrôleur et ses événements permettent de guider les opérations de la machine B. Cette approche s'applique également à un ensemble de machines B en concurrence, chaque machine étant contrôlée par un processus CSP.

5.2.1 Syntaxe

Le système est défini dans cette approche à l'aide d'un processus de type CSP appelé *contrôleur d'exécution*, décrit par un sous-langage de CSP, associé à une machine B classique, dont les opérations sont contrôlées par le processus.

Le langage décrivant les processus est le suivant :

$$P \quad ::= \quad a \rightarrow P \mid c\,?x\langle E(x)\rangle \rightarrow P \mid d\,!v\{E(v)\} \rightarrow P \mid e\,\mathbf{?}v\,\mathbf{!}x\{E(x)\} \rightarrow P \mid P_1 \,\Box P_2 \mid P_1 \sqcap P_2 \mid \sqcap_{x|E(x)} P \mid \textit{if } b \textit{ then } P_1 \textit{ else } P_2 \textit{ end}$$

où a est un événement, c est un canal de communication acceptant des entrées, d est un canal de communication acceptant des sorties, e est un canal de machine, x représente une variable de données, v une valeur de données, $E(x)$ est un prédicat sur x, b est une expression booléenne et p est une expression de processus.

Ce langage reprend la plupart des opérateurs couramment utilisés en CSP : $\rightarrow$ (préfixe), $\Box$ (choix externe), $\sqcap$ (choix interne). L'entrée d'une valeur x le long d'un canal c est dénotée par $c?x$. De même, la sortie d'une valeur v le long d'un canal d est notée $d!v$. Le prédicat $E(x)$ permet de représenter la garde ou l'assertion d'un événement.

En revanche, les opérateurs de composition parallèle et d'entrelacement ne sont pas pris en compte au niveau des contrôleurs d'exécution. La concurrence est en effet considérée à un niveau supérieur, entre les contrôleurs d'exécution.

Le langage permet de distinguer les communications impliquant uniquement des processus CSP (! et ?) des communications entre machines B et contrôleurs CSP (**!** et **?**). La figure 5.4 représente graphiquement les interactions possibles entre machines B et processus CSP. Une interaction entre un contrôleur CSP

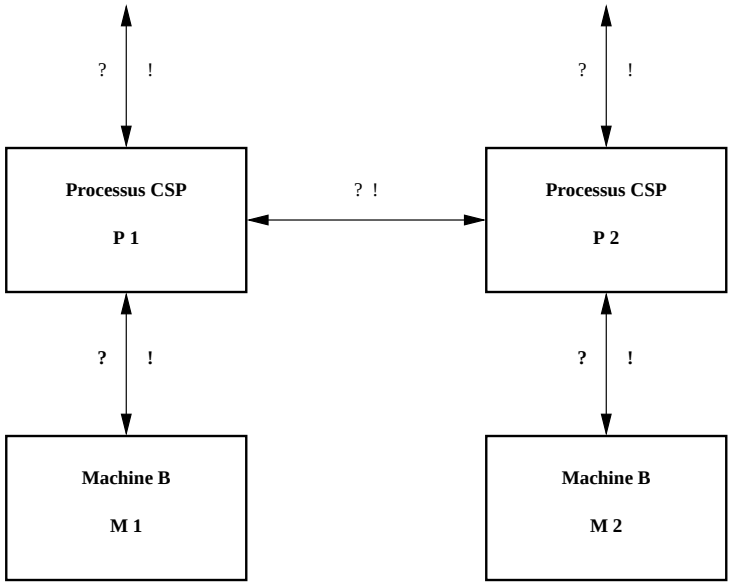


FIG. 5.4 – Interactions entre machines B, processus CSP et le reste de l'environnement

et sa machine B correspondante est représentée par $e\,\mathbf{?}v\,\mathbf{!}x\{E(x)\} \rightarrow P$. Le canal e de machine est utilisé pour communiquer avec la machine B à travers l'opération correspondante $x \longleftarrow e(v)$. Un canal de machine est donc identifié dans cette approche avec une opération B. Les communications entre deux processus CSP ne sont pas distinguées des communications avec l'environnement

extérieur. Enfin, la communication entre deux machines B est donc définie par l'intermédiaire de la communication de leurs contrôleurs d'exécution respectifs.

Un système S est décrit dans cette approche par une machine M spécifiée en B et par un contrôleur d'exécution P. Le système est intuitivement considéré comme la "composition parallèle" de P et de M. Pour cette raison, S est dénoté par :

$$S = P \mid\mid M$$

Dans le cas d'un système défini par plusieurs machines B en concurrence, chaque machine M_i est contrôlée par un processus P_i. Le système est alors dénoté par :

$$||_i \ (P_i \mid\mid M_i)$$

5.2.2 Exemple

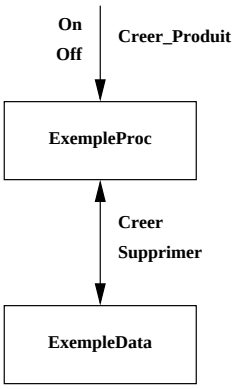


FIG. 5.5 – Lien entre la machine B et le processus CSP

Pour décrire l'exemple de la figure 5.1, le système spécifié en utilisant cette approche est de la forme représentée par la figure 5.5. Les liens entre la machine *ExempleData* et le processus *ExempleProc* sont les suivants : les événements *On* et *Off* sont appelés par l'environnement extérieur. La création et la suppression de produits sont également commandées depuis l'extérieur, notamment pour déterminer les produits à créer ou à supprimer. Le processus permet de spécifier l'ordonnancement des opérations de la machine B qui est la même, à renommages près, que dans l'approche csp2B :

MACHINE *ExempleData*
SETS *PRODUITS*
VARIABLES *Produit*, *P*
INVARIANT $Produit \subseteq PRODUITS \wedge P \in PRODUITS$
INITIALISATION $Produit := \varnothing \parallel P :\in PRODUITS$
OPERATIONS
$Creer(pdt) =$
pre $pdt \in PRODUITS - Produit$
then $Produit := Produit \cup \{pdt\}$
end ;

Supprimer =
pre $P \in Produit$
then $Produit := Produit - \{P\}$
end
END

Le processus CSP qui permet de contrôler la machine *ExempleData* est :

$$\begin{aligned} ExempleProc &= Arret(\varnothing) \\ Arret(PC) &= On \rightarrow Marche(PC) \\ Marche(PC) &= (Creer\_Produit\,?pdt < pdt \in PRODUITS - PC > \\ &\quad \rightarrow Creer\,\mathbf{?}pdt \rightarrow Supprimer \rightarrow Marche(PC)) \\ &\quad \Box(Off \rightarrow Arret(PC)) \end{aligned}$$

Ce processus a pour paramètre PC qui représente l'ensemble des produits créés. L'état initial du processus *ExempleProc* est *Arret* avec $PC = \varnothing$. À l'état *Marche*, le contrôleur décrit les séquences des événements possibles en précisant à la fois l'action venant de l'extérieur (*Creer_Produit*) et les actions correspondant aux opérations B (*Creer* et *Supprimer*). Les gardes indiquées entre crochets permettent de spécifier le typage attendu par le processus des entrées des canaux de communication.

Pour appeler l'opération *Creer* dans la machine B, l'état du système doit être *Marche*, puis l'événement *Creer_Produit* doit être exécuté avec une valeur de *pdt* bien typée. Ensuite, l'opération *Supprimer* est exécutée et l'état du système est de nouveau *Marche*.

Les gardes et les assertions $E(x)$ et $E(v)$ introduites dans le langage permettent de décomposer le système en plusieurs sous-systèmes qu'il est possible de vérifier indépendamment. Les gardes bloquent les entrées non désirées alors que les assertions divergent si des communications sont inattendues. Les gardes sur les entrées d'un contrôleur sont utilisées pour décrire les entrées attendues du reste du système et par conséquent pour caractériser l'environnement d'un processus. De même, les assertions sur les sorties d'une machine B vers son processus associé permettent de caractériser les sorties attendues de ce processus.

Dans la pratique, ces prédicats proviennent des machines B (préconditions des opérations et typage des paramètres de sortie) et sont par conséquent redondants. Ils sont toutefois nécessaires pour déterminer formellement les entrées et les sorties des opérations dans le processus de contrôle.

5.2.3 Cohérence et définition de CSP || B

La description de systèmes de communication combinés utilisant des machines B associées à des contrôleurs d'exécution n'est possible que si les descriptions B et CSP sont cohérentes, dans le sens qu'elles ne sont pas contradictoires. Dans ce but, le processus de spécification d'un système combiné doit permettre de vérifier que les processus CSP des contrôleurs n'introduisent ni divergence, ni blocage lors de l'exécution des opérations B correspondantes. Une méthode de vérification est présentée afin d'assurer cette notion de cohérence. Elle consiste en deux étapes :

1. vérifier que chaque machine M_i contrôlée est libre de divergence : $P_i \parallel M_i$,
2. vérifier que la combinaison des contrôleurs est libre de blocage : $||_i\ P_i$.

Ces deux étapes sont suffisantes pour assurer la cohérence du système combiné dans son ensemble [ST02].

Les opérations B considérées sont de la forme classique **pre then end**. De telles opérations sont exécutables à tout moment, même lorsque les préconditions ne sont pas vérifiées. L'exécution d'opérations B n'entraîne donc pas de blocage. En revanche, si une opération est appelée alors que ses préconditions ne sont pas vérifiées, le comportement de l'opération est inattendu et peut entraîner une divergence du système.

Par conséquent, pour chaque combinaison $P_i \parallel M_i$, le contrôleur P_i n'introduit pas de divergence dans la machine M_i si, à chaque appel d'opération de M_i dans P_i, les préconditions sont vérifiées. Cette propriété est assurée grâce à un invariant dans chaque boucle récursive du contrôleur P_i, noté CLI, qui vérifie, pour tout appel récursif $S(p)$ dans P_i :

$$CLI \wedge I \Rightarrow [\{BBODY_{S(p)}\}]CLI$$

où la séquence d'opérations $\{BBODY_{S(p)}\}$ est issue de la traduction en B de l'expression de processus associée à $S(p)$ et I est l'invariant de la machine M_i. Cette propriété signifie qu'à chaque appel récursif du processus, la séquence d'opérations B correspondantes préserve l'invariant de boucle CLI.

La traduction des expressions de processus en B est définie par des règles de traduction décrites dans [TS99]. Chaque sous-séquence du processus se terminant par un appel récursif est traitée. Pour le processus *ExempleProc*, on a par exemple :

$$\begin{aligned} \{On \rightarrow Marche(PC)\} &= On\ ;\{Marche(PC)\} \\ &= On\ ; c := 0 \end{aligned}$$

et :

$$\{Marche(PC)\} = \textbf{Choice } \{P_1\} \textbf{ or } \{P_2\} \textbf{ end}$$

où :

$$\begin{aligned} \{P_1\} &= \{Creer\_Produit\ ?pdt < pdt \in PRODUITS{-}PC > \rightarrow Creer\ \textbf{?}pdt \\ &\quad \rightarrow Supprimer \rightarrow Marche(PC)\} \\ &= \textbf{select } pdt \in PRODUITS{-}PC \textbf{ then } Creer(pdt)\ ;\ Supprimer \textbf{ end}\ ;\ \{Marche(PC)\} \\ &= \textbf{select } pdt \in PRODUITS{-}PC \textbf{ then } Creer(pdt)\ ;\ Supprimer \textbf{ end}\ ;\ c := 1 \\ \{P_2\} &= \{Off \rightarrow Arret(PC)\} \\ &= Off\ ;\ c := 0 \end{aligned}$$

Un appel récursif est interprété comme une affectation de valeur à une variable de contrôle de la machine B. Bien que non spécifiées dans les machines

B, les variables de contrôle permettent de vérifier l'invariant *CLI* à chaque appel récursif de la boucle plutôt qu'à chaque appel d'opération. La valeur de la variable de contrôle c correspond ici à l'état 0 (*Arret*) ou à l'état 1 (*Marche*).

Un invariant *CLI* qui vérifie la propriété suivante :

$$CLI \wedge I \Rightarrow [\{BBODY_{S(p)}\}]CLI$$

pour l'exemple précédent est :

$$c = n \wedge c \in 0..1$$

La spécification de l'invariant CLI_i dans le processus *ExempleProc* permet de prouver [TS99] la liberté de divergence de la combinaison :

$$ExempleProc \parallel ExempleData$$

La détermination de cet invariant n'est cependant pas immédiate et demande une certaine analyse. L'utilisation d'outils comme FDR [For97] peut s'avérer pratique.

Dans le cas de plusieurs combinaisons $P_i \parallel M_i$ en concurrence, la cohérence est vérifiée en analysant les blocages de la combinaison $||_i\ P(i)$. Schneider et Treharne montrent que si chaque $P_i \parallel M_i$ est libre de divergence, il suffit alors de vérifier que la combinaison $||_i\ P(i)$ est libre de blocage pour prouver la cohérence du système combiné dans son ensemble [ST02]. Ces propriétés sont possibles grâce aux gardes et aux assertions qui permettent d'analyser et de traiter les $P_i \parallel M_i$ et la combinaison $||_i\ P(i)$ indépendamment les uns des autres.

L'analyse des blocages de $||_i\ P(i)$ est un exercice difficile et indépendant du cadre des combinaisons de spécifications formelles. Le principal apport de cette méthode est la compositionnalité de la vérification. La possibilité de vérifier chaque système combiné $P_i \parallel M_i$ et ensuite la combinaison $||_i\ P(i)$ de manière indépendante permet de simplifier les analyses et les calculs.

5.2.4 Raffinement et vérification

Le raffinement n'est pas explicitement traité dans cette approche. Dans le cadre de la méthode B, il est toutefois possible de raffiner une machine M d'une combinaison $P \parallel M$. Une autre solution consiste à raffiner le processus P, ce qui est assez difficile en CSP (voir section 4.4.3). Il n'existe pas à notre connaissance de méthode de raffinement intégrée proposée dans le cadre de cette approche.

Concernant les vérifications, il est possible d'utiliser l'Atelier B pour la partie concernant la machine B et FDR pour la partie processus CSP. Les seules vérifications proposées dans cette approche concernent la cohérence des deux spécifications (voir section précédente).

5.2.5 Bilan

Cette approche permet de considérer la composition parallèle de machines B avec leur contrôleur décrit par un processus CSP et un invariant de boucle récursive. La cohérence du système de machines combinées est justifiée par l'absence de divergence pour chaque paire contrôleur-machine et par l'absence de

blocage dans la composition parallèle de tous les contrôleurs. L'utilisation d'un invariant permet d'exprimer simplement une condition qui assure l'absence de divergence dans un contrôleur. L'analyse des blocages de tous les processus est réalisée sur la composition de tous les processus. La compositionnalité de cette méthode permet de simplifier les analyses et d'utiliser des outils existants.

L'utilisation d'un riche langage CSP implique une analyse plus détaillée de la cohérence entre processus CSP et machine B. À la différence de l'outil csp2B qui restreint la syntaxe des descriptions CSP afin d'éviter des contradictions, cette approche permet de conserver de nombreux opérateurs CSP pour décrire le comportement des opérations B. On remarque cependant que ce langage n'autorise pas la quantification d'opérateurs.

Cette méthode utilise la spécification B classique des opérations. Concernant les opérations sous la forme **select then end** comme dans la section 5.1, l'analyse de la cohérence est différente. Avec des préconditions, une opération B est toujours exécutable tandis que dans le cas d'opérations gardées par **select**, un blocage est possible si la garde d'une opération appelée est fausse. Dans ce cas, de nouvelles conditions sont introduites pour assurer la liberté de blocage des combinaisons contrôleur-machine [TS00].

L'idée de considérer CSP || B est intéressante car elle permet de conserver les deux points de vue. La cohérence des spécifications n'est pas naturelle et demande un effort de vérification. Le langage utilisé pour CSP est assez riche pour utiliser des outils existants comme FDR pour analyser les divergences et les blocages des processus CSP. Cette automatisation reste néanmoins limitée à la partie CSP de la spécification. Il est de même possible d'utiliser des outils de la méthode B pour prouver la cohérence des machines. Il n'existe pas enfin de méthode de raffinement proposée dans le cadre de combinaisons CSP || B.

Si on se place dans le cadre des systèmes d'information, outre le fait que l'approche ne propose pas de méthodes intégrées de vérification ou de raffinement, CSP || B semble compliquer la modélisation plutôt que la simplifier. Il faudrait en effet modéliser chaque type d'acteur par une machine B et lui associer un contrôleur, ce qui aurait pour conséquence de dispatcher les propriétés fonctionnelles du SI sur un ensemble de machines. Ou bien chaque classe serait modélisée par une machine B et dans ce cas, il ne serait plus possible de représenter la concurrence entre les types d'acteur puisqu'un contrôleur d'exécution ne dispose pas d'opérateur comme la composition parallèle. Enfin, les vérifications souhaitées en SI portent plutôt sur l'ensemble du système que sur une simple paire de la forme $P \parallel M$.

Tout comme csp2B, l'approche CSP || B ne semble pas convenir à la modélisation des SI. Elle apporte toutefois des renseignements intéressants concernant la combinaison de deux approches telles que CSP et B. Elle nous montre d'une part que la combinaison de deux spécifications formelles est difficile car elle nécessite une vérification de la cohérence du système. D'autre part, l'intégration de deux approches implique la définition de nouvelles méthodes pour vérifier et raffiner des modèles.

5.3 CSP-OZ [Fis00]

L'idée de cette approche est de créer un nouveau langage, CSP-OZ, inspiré des langages CSP [Hoa85] et Object-Z [Smi00]. Les processus CSP permettent de

décrire les aspects dynamiques d'un système tandis qu'Object-Z est une notation orientée objet qui permet de décrire les structures de données d'un système. Le nouveau langage est basé sur l'identification d'une classe Object-Z avec un processus CSP.

5.3.1 Syntaxe

Le langage CSP-OZ est un langage de spécification formel destiné à décrire des systèmes de communication distribués. Il est une extension du langage CSP_Z, un dialecte CSP qui utilise la syntaxe Z pour décrire les expressions de processus et les données, qui intègre en outre des descriptions orientées objet issues du langage Object-Z. L'idée est d'associer une classe à un processus. Comme un processus est une séquence d'événements qui modifient l'état du système, les opérations d'une classe sont reliées aux événements d'un processus qui lui est associé.

CSP_Z Le langage CSP_Z est une extension du langage Z qui permet de décrire des processus CSP avec une syntaxe proche de Z. Le langage CSP_Z et sa sémantique sont présentés en détail dans [Fis00]. Une spécification Z est une séquence de paragraphes (ou déclarations) regroupés dans des schémas (voir section 3.2.2). CSP_Z étend la syntaxe de Z par l'ajout de deux nouveaux types de paragraphe, les canaux et les processus :

$$\mathit{Paragraph}_C \quad ::= \quad \mathit{Paragraph}_Z \mid \mathit{Channel} \mid \mathit{Process}$$

Le paragraphe *Channel* permet de définir les canaux de communication des processus et d'introduire les événements possibles. Un canal est spécifié par :

$$\mathit{Channel} \quad ::= \quad \text{chan } \mathit{Name} \; [\, : \mathit{Expr}]$$

où *Expr* est une expression Z qui déclare les types des paramètres d'entrée et de sortie.

Le paragraphe *Process* permet de spécifier les processus en les associant à des expressions du langage *Expr* :

$$\mathit{Process} \quad ::= \quad \mathit{DefProc} = \mathit{Expr}$$

où *DefProc* permet d'identifier le processus spécifié.

Ces déclarations sont appliquées sur des constructeurs Z standards, mais elles nécessitent également une extension de la grammaire des expressions pour décrire les processus. Parmi les expressions *Expr*, sont définies des expressions de processus *ProcExpr* de la forme :

$$\begin{aligned} \mathit{ProcExpr} \quad ::= \quad & \mathit{Stop} \mid \mathit{Skip} \mid \mathit{Diver} \mid \\ & \mathit{Expr} \rightarrow \mathit{ProcExpr} \mid \\ & \mathit{ProcExpr} \,\Box \mathit{ProcExpr} \mid \\ & \mathit{ProcExpr} \sqcap \mathit{ProcExpr} \mid \\ & \mathit{ProcExpr} \parallel \mathit{ProcExpr} \mid \\ & \vdots \end{aligned}$$

Les processus de base sont *Stop* (le processus de blocage), *Skip* (le processus qui termine) et *Diver* (le processus qui diverge).

Par exemple, un processus de la forme :

$$Alternative = On \rightarrow (Diver \,\Box Stop)$$

où *On* est défini par :

$$\texttt{chan}\ On$$

représente un processus qui exécute tout d'abord l'événement *On* puis se comporte comme *Diver* ou *Stop*.

Les expressions *ProcExpr* constituent le langage CSP admis dans la syntaxe de $\mathrm{CSP_Z}$. La plupart des opérateurs CSP sont en effet disponibles, comme le préfixe ($\rightarrow$), les choix externe ($\Box$) et interne ($\sqcap$) ou la composition parallèle ($||$). Outre les expressions de processus déjà présentées, $\mathrm{CSP_Z}$ autorise aussi la récursivité, le multipréfixe ou le préfixe de schémas Z.

CSP-OZ Le langage CSP-OZ étend le langage $\mathrm{CSP_Z}$ afin de rajouter la notion de classe dans la syntaxe. La grammaire CSP-OZ est donc de la forme :

$$Paragraph_O \quad ::= \quad Paragraph_C \mid Class_O \mid Class_C$$

où $Paragraph_C$ est la grammaire du langage $\mathrm{CSP_Z}$, $Class_C$ une classe CSP-OZ et $Class_O$ une classe Object-Z. De manière informelle, la classe CSP-OZ est la version orientée objet d'un processus, tandis que la classe Object-Z est utilisée pour décrire les aspects données d'un système. De plus, les classes Object-Z du langage CSP-OZ jouent un rôle dans l'instanciation des objets en CSP-OZ, puisque les classes peuvent hériter ou être des instances de classes Object-Z.

En résumé, dans la syntaxe CSP-OZ, les classes Object-Z $Class_O$ sont des boîtes contenant :

- une liste de visibilité : les items visibles depuis l'extérieur de la classe,
- une liste des classes dont la classe hérite,
- une liste de définitions locales,
- un schéma décrivant l'état de l'objet,
- un schéma initial spécifiant l'état initial,
- une liste des opérations qui peuvent modifier l'état de l'objet.

Une telle classe ne peut hériter que d'une autre classe Object-Z.

Les classes CSP-OZ $Class_C$ sont similaires aux classes Object-Z, mais :

- elles ne contiennent pas de liste de visibilité,
- elles contiennent en plus :
  - une interface,
  - une liste de processus $\mathrm{CSP_Z}$.

L'interface d'une classe CSP-OZ permet de décrire tous les liens de communications de la classe, comme les méthodes de la classe (dans une classe Object-Z, elles sont contenues dans la liste de visibilité) mais aussi les méthodes des autres objets que la classe peut utiliser. Elle contient donc l'alphabet qu'une classe CSP-OZ peut utiliser pour communiquer avec les autres classes. La liste de processus $\mathrm{CSP_Z}$ constitue une des particularités de ce langage, puisque la classe CSP-OZ intègre une partie CSP dans sa description. Elle permet de contraindre l'ordre d'exécution des méthodes de la classe.

Pour relier les opérations d'une classe Object-Z aux événements de processus d'une classe CSP-OZ, il existe en CSP-OZ plusieurs mots-clés. Si *Op* désigne une opération, alors :

– `enable_`*Op* définit la garde de l'opération,
– `effect_`*Op* est l'effet de l'opération *Op*,
– et `com_`*Op* est utilisé pour définir à la fois la garde et l'effet de *Op*.

Il est alors possible d'exprimer les caractéristiques d'une opération (garde, effet) en fonction des événements d'un processus CSP_Z.

5.3.2 Exemple

Si on reprend l'exemple de la figure 5.1, deux classes sont définies dans cette approche pour représenter le système : une classe Object-Z et une classe CSP-OZ.

La classe Object-Z permet de spécifier les types de données et les opérations du système. On considère que le système est représenté par une seule classe qui reçoit les commandes depuis l'extérieur. La classe définie représente les produits. La spécification est très proche des machines B définies dans les approches précédentes :

ExampleData

$$produit : \mathbb{P}\, PRODUITS$$
$$P : PRODUITS$$

Init

$$produit = \varnothing$$
$$\exists\, pdt : PRODUITS \bullet P = pdt$$

Creer

$$\Delta(produit, P)$$
$$pdt? : PRODUITS$$

$$pdt? \in PRODUITS - produit$$
$$produit' = produit \cup \{pdt?\}$$
$$P' = pdt?$$

Supprimer

$$\Delta(produit)$$

$$P \in produit$$
$$produit' = produit - \{P\}$$

Dans la classe *ExempleData*, l'état du système est défini par la variable *produit* qui représente l'ensemble des produits créés. Les opérations *Creer* et *Supprimer* agissent sur cette variable, d'où la notation Δ qui indique les changements d'états induits par ces opérations. La variable P sert à garder en mémoire le dernier produit créé.

Les classes CSP-OZ ont la particularité de pouvoir spécifier des processus avec le langage CSP_Z. La classe *ExempleProc* utilise les méthodes *Creer* et *Supprimer* et les canaux *On* et *Off*. Les méthodes correspondent aux opérations définies dans la classe *ExempleData*, tandis que les canaux sont des opérations, définies dans d'autres classes, dont on ne connaît que les interfaces.

ExempleProc

`method` *Creer*[*pdt*? : *PRODUITS*]
`method` *Supprimer*
`chan` *On*
`chan` *Off*

$Main = On \rightarrow Marche$
$Marche = (Creer?pdt \rightarrow Supprimer \rightarrow Marche) \Box$
$\quad (Off \rightarrow Main)$

Le processus *Main* défini dans cette classe est similaire aux approches précédentes. Dans notre exemple, les canaux n'ont pas de paramètre. La méthode *Creer* déclare le paramètre d'entrée de l'opération correspondante.

Pour relier ces deux classes, une nouvelle classe Object-Z, *Exemple*, qui hérite des classes *ExempleData* et *ExempleProc*, est définie :

Exemple

`method` *Creer*[*pdt*? : *PRODUITS*]
`method` *Supprimer*
`chan` *On*
`chan` *Off*
`inherit` *ExempleData*, *ExempleProc*

`com_`*Creer* = *Creer*
`com_`*Supprimer* = *Supprimer*

Les déclarations des méthodes et des canaux utilisés par cette classe sont rappelées. Les opérations *Creer* et *Supprimer* de la classe *ExempleData* sont liées aux événements *Creer* et *Supprimer* de la classe *ExempleProc* avec le mot-clé `com`. Cela signifie que ces opérations ont les mêmes gardes et les mêmes effets que les événements correspondants.

Les sémantiques de CSP_Z et d'Object-Z ont été étendues afin de tenir compte de la syntaxe particulière de CSP-OZ. Fischer montre que la sémantique d'une classe CSP-OZ est un processus CSP_Z. Ce processus est une composition parallèle de la partie CSP et d'un processus qui caractérise la sémantique de la partie Z [Fis00].

5.3.3 Raffinement et vérification

Le raffinement en CSP-OZ est basé sur les relations de raffinement Z et CSP. Fischer montre dans un premier temps que ces deux raffinements sont cohérents dans le cadre de CSP_Z. Il étend ensuite ces résultats en CSP-OZ afin de définir une relation de simulation entre classes CSP-OZ. Fischer n'a pas défini une nouvelle notion de raffinement qui permet à la fois de raffiner les données et les processus, mais il a montré que si les classes Object-Z sont raffinées au niveau des données selon les règles qu'il a définies, alors les processus associés respectent bien le raffinement de processus au sens CSP.

Par exemple, la classe CSP-OZ *Exemple* peut être raffinée par la classe de la figure 5.6. La classe *Exemple_1* est en effet un raffinement de la classe *Exemple*, car ses types de données sont plus concrets que dans la classe abstraite. Le processus associé à cette classe qui n'est autre que le processus de la classe

ExempleProc mais avec les opérations raffinées, est en fait un raffinement au sens CSP du processus original.

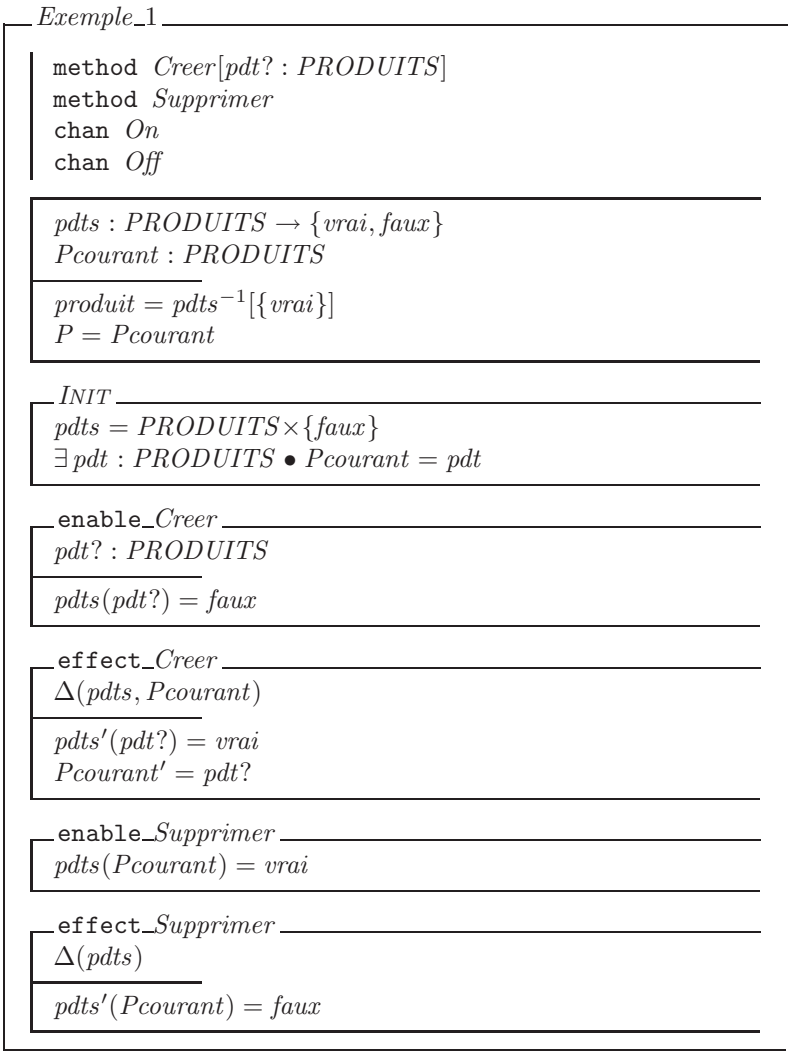


FIG. 5.6 – Raffinement de la classe *Exemple*

Concernant la vérification, le langage CSP-OZ est nouveau et dispose donc de peu d'outils. Comme la sémantique est bien définie, il est possible pour l'instant de faire des preuves à la main. De plus, si CSP-OZ accepte facilement des spécifications Z, Object-Z ou CSP, en faisant un minimum de changements, la réciproque n'est pas vraie puisque le langage CSP-OZ introduit des constructeurs inconnus en CSP et Object-Z. Il n'est donc pas possible d'utiliser les outils existant dans ces méthodes.

5.3.4 Bilan

CSP-OZ est un langage de spécification formel qui permet de décrire des systèmes à l'aide d'une combinaison des formalismes de CSP et Object-Z. Un langage intermédiaire, CSP_Z, permet de décrire des propriétés dynamiques à l'aide de processus utilisant à la fois les principaux opérateurs CSP et une syntaxe Z. CSP_Z et Object-Z sont deux sous-langages de CSP-OZ, ce qui permet d'intégrer des spécifications existantes dans le nouveau langage.

Comme tout nouveau langage, le principal défaut de CSP-OZ est l'introduction d'un nouveau formalisme pour décrire un système. Cela implique notamment la définition complète de la syntaxe et de la sémantique du langage. Le travail est ici plus important puisque CSP-OZ est basé sur un autre nouveau langage et que deux approches sont considérées pour la sémantique. Un utilisateur doit donc apprendre à utiliser CSP-OZ. Cette difficulté est atténuée par le fait que CSP-OZ reprend des expressions et des constructeurs connus en Z et Object-Z.

Un des avantages de CSP-OZ est l'utilisation d'un langage unique pour décrire à la fois les aspects statiques et dynamiques. Une classe CSP-OZ est en quelque sorte une classe Object-Z avec des spécifications de processus CSP_Z. Les communications entre objets sont limitées aux instanciations d'objets. Le problème de l'ordonnancement de plusieurs méthodes d'objets en concurrence n'est donc pas soulevé. Cette approche permet de contraindre l'ordonnancement des méthodes appelées ou appelant dans une classe CSP-OZ donnée. Les implications de ces appels sur d'autres classes ne sont donc pas prises en compte.

Cette étude nous montre que la création d'un nouveau langage intégrant deux approches comme CSP et Object-Z implique la définition d'une syntaxe et d'une sémantique nouvelles, sans oublier la création de nouveaux outils pour assister l'utilisateur. Ce type d'intégration rend également difficile la réutilisation de spécifications existantes. Dans un domaine comme les SI où il existe déjà de nombreuses méthodes de conception, l'apprentissage et l'utilisation d'un nouveau langage ressemblent plus à des défauts qu'à des avantages.

5.4 CSP et Object-Z [SD01]

Cette approche a pour but d'adopter un double point de vue pour décrire le même système : le modèle orienté objet et le modèle concurrent de type algèbre de processus. Les langages utilisés sont CSP [Hoa85] et Object-Z [Smi00]. Comme dans l'approche CSP-OZ, la notion de classe est associée à la notion de processus.

5.4.1 Syntaxe

La modélisation d'un système se présente sous la forme d'une double spécification avec :

- une partie Object-Z,
- et une partie CSP, où les classes Object-Z peuvent être utilisées.

Les syntaxes utilisées sont celles de CSP et Object-Z.

Cette présentation est justifiée par la définition d'une sémantique basée sur les échecs-divergences pour les classes Object-Z qui est proche de la sémantique

couramment utilisée pour représenter les processus CSP. Le lien entre classe Object-Z et processus CSP est défini à deux niveaux :

- entre les opérations et les événements,
- entre les classes et les processus.

Les opérations d'une classe Object-Z sont donc identifiées avec les événements d'un processus CSP. Trois types d'interactions entre opérations sont considérés pour relier les opérations définies dans des classes distinctes :

1. échange entre processus : un paramètre de sortie $x!$ est unifié avec un paramètre d'entrée $x?$ dans une opération synchronisée,
2. partage de valeurs d'entrée : un paramètre d'entrée $x?$ est unifié avec un autre paramètre d'entrée $y?$ dans une opération synchronisée,
3. coopération de processus dans la production d'un résultat : un paramètre de sortie $x!$ est unifié avec un autre paramètre de sortie $y!$ dans une opération synchronisée.

Enfin, une classe Object-Z est identifiée avec un processus CSP. Il est donc possible avec cette approche de composer des classes Object-Z avec quelques opérateurs du langage CSP. Les principaux opérateurs utilisés sont $||$ et $|||$. La séquence et le préfixe ne sont pas autorisés. Lorsque deux classes Object-Z sont synchronisées, Les opérations des classes sont synchronisées selon les trois liens décrits ci-dessus.

5.4.2 Exemple

L'exemple de la figure 5.1 est spécifié avec cette approche de la manière suivante. Deux classes Object-Z interagissent entre elles : une classe *Produit* et une classe *Utilisateur*. On définit en Z les types *PRODUITS* et *ETAT*, avec

$$ETAT = Arret \mid Marche \mid Marche\_1$$

qui sont utilisés dans les classes.

La classe *Utilisateur* permet de spécifier les demandes venant de l'utilisateur du système, la machine qui fabrique des objets. Elle est décrite dans la figure 5.7. Les opérations *On*, *Off*, *Creer* et *Supprimer* sont définies. Les deux premières permettent de mettre en service ou hors service la machine. L'opération *Creer* précise quel produit il faut créer à l'autre classe, *Produit*. Le paramètre *pdt* est en effet une sortie de l'opération. L'opération *Supprimer* ne fait que modifier la variable *etat*.

Dans la figure 5.8, la classe *Produit* décrit la classe Object-Z qui spécifie le système de la figure 5.1. Seules les opérations *Creer* et *Supprimer* sont définies. L'opération *Creer* permet de créer le produit *pdt* précisé en argument, tandis que *Supprimer* détruit le dernier produit créé. Cette classe est similaire aux descriptions B et Object-Z des approches précédentes.

Les opérations *Creer* et *Supprimer* dans une classe sont synchronisées avec les opérations homonymes dans l'autre classe. Par exemple, le paramètre d'entrée *pdt ?* de l'opération *Creer* de la classe *Produit* correspond à la sortie *pdt !* de l'opération *Creer* de la classe *Utilisateur*.

Pour spécifier l'ensemble du système, le processus CSP suivant est défini :

$$Systeme = Utilisateur \mid\mid Produit$$

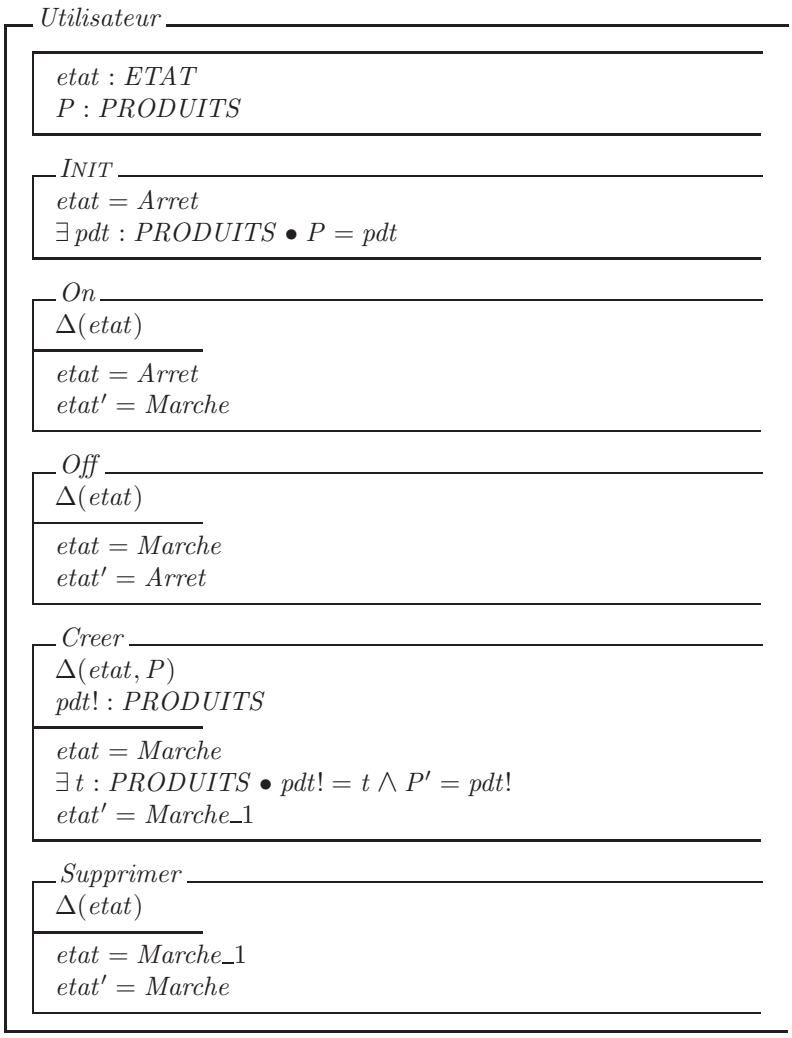


FIG. 5.7 – Classe Object-Z *Utilisateur*

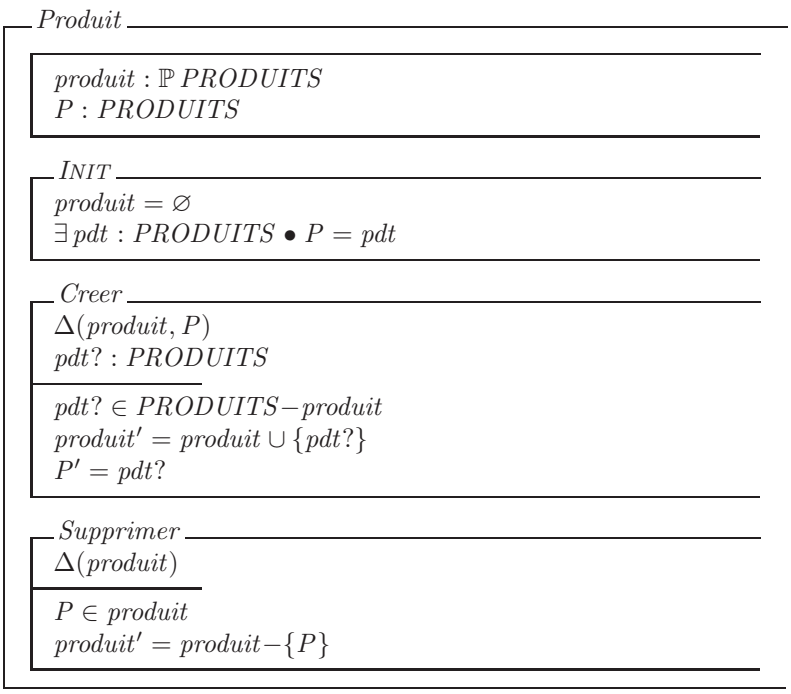


FIG. 5.8 – Classe Object-Z *Produit*

Les événements du processus *Systeme* ont pour effet l'exécution en synchronisation des opérations homonymes dans chaque classe.

Cette approche ne permet pas bien de spécifier le système considéré. Les ordonnancements entre opérations doivent en effet être spécifiés de manière plus précise dans les classes. Dans l'exemple présenté, l'exécution de l'événement *Supprimer* (qui correspond aux exécutions des opérations *Supprimer* dans les classes *Utilisateur* et *Produit*) n'est possible que si l'état du système est *Marche_1*. Par conséquent, il faut expliciter tous les états du système pour éviter qu'un produit ne soit supprimé avant qu'il ne soit créé. Or ce type de spécification est difficile dans un langage comme Object-Z, il faut donc pouvoir vérifier facilement ce type de propriétés pour pouvoir corriger rapidement ce type d'erreur. Dans notre cas, il suffit d'introduire un état intermédiaire *Marche_1* et changer les spécifications des opérations dans un style proche de celles de la machine B obtenue par traduction dans csp2B.

Par conséquent, les processus CSP de cette approche ont un contrôle limité des opérations des classes Object-Z. Cette approche permet de bien représenter plusieurs classes (et instances de classes) en concurrence, mais les propriétés d'ordonnancement sur les événements entre classes ou à l'intérieur d'une classe sont difficiles à exprimer.

5.4.3 Vérification de propriétés

Cette approche propose une méthode de vérification. Les propriétés dynamiques du système sont dans un premier temps spécifiées dans la description CSP en utilisant la notation **sat**. L'expression :

$$Processus \ \mathbf{sat} \ \phi$$

signifie que le processus *Processus* établit la propriété ϕ. Cette notation est définie dans la sémantique de CSP par :

$$Processus \ \mathbf{sat} \ \phi \Leftrightarrow \forall \, tr \bullet tr \in traces(Processus) \Rightarrow \phi$$

où $traces(Processus)$ désigne l'ensemble des traces possibles du processus *Processus*. La notation **sat** n'a ici aucun sens puisque les processus sont identifiés à des classes.

Par exemple, le système de la figure 5.1 doit vérifier que :

$$Systeme \ \mathbf{sat} \ On \rightarrow Creer \rightarrow Supprimer \rightarrow Marche$$

Cette propriété est réécrite en remplaçant la définition de *Systeme* :

$$(Utilisateur \parallel Produit) \ \mathbf{sat} \ On \rightarrow Creer \rightarrow Supprimer \rightarrow Marche$$

La seconde étape consiste à étendre les classes Object-Z avec des variables auxiliaires (par héritage Object-Z) afin de modéliser les termes des propriétés qui n'ont pas de sens en Object-Z.

Dans l'exemple, *Utilisateur* et *Produit* correspondent à des classes Object-Z et ne peuvent plus être simplifiées. La logique utilisée pour raisonner sur les classes est la logique W étendue au langage Object-Z [Smi95]. Cette logique est exprimée sous forme de séquents :

$$A :: d \mid \Psi \vdash \Phi$$

où A est une classe, d une liste de déclarations, Ψ et Φ des prédicats. Le séquent est valide si, étant donnés d et Ψ, au moins un des prédicats Φ est vrai dans la classe A.

Puisque Ψ et Φ sont des prédicats sur les termes de la classe, il n'est pas possible de vérifier directement avec cette logique les propriétés sur le processus CSP associé à la classe A. L'introduction de variables auxiliaires permet de représenter des termes de processus comme la séquence :

$$On \rightarrow Creer \rightarrow Supprimer \rightarrow Marche$$

dans la logique W. Il faut par exemple introduire des états supplémentaires et redéfinir les opérations. Ces ajouts de variables sont spécifiés à l'aide de l'héritage Object-Z.

Les nouvelles classes Object-Z ainsi obtenues incluent toutes les définitions des classes héritées mais sont aussi étendues de manière à vérifier les séquents correspondant aux propriétés CSP à vérifier.

Enfin, la dernière étape de la vérification est la démonstration que les classes étendues sont raffinées par les classes originales (voir section suivante pour le raffinement). Si tel est le cas, les classes originales vérifient aussi les propriétés voulues et la vérification est terminée.

La vérification de l'ordonnancement nous impose dans notre exemple l'introduction d'un état *Marche_1*. L'utilisation de classes Object-Z comme processus dans les expressions de processus de cette approche ne permet pas l'utilisation d'outil comme FDR pour vérifier des propriétés.

5.4.4 Raffinement

Concernant le raffinement, les relations de simulations de Josephs (voir chapitre 4) sont adaptées dans [SD01] au contexte des classes Object-Z.

Une classe Object-Z C est dite une *simulation vers le bas* de la classe A s'il existe une relation Abs telle que chaque opération abstraite AOp soit associée à une opération concrète COp de la manière suivante :

1. $\forall Astate, Cstate \bullet Abs \Rightarrow (\textbf{pre}\ AOp \Leftrightarrow \textbf{pre}\ COp)$
2. $\forall Astate, Cstate, Cstate' \bullet Abs \wedge COp \Rightarrow (\exists AState' \bullet Abs' \wedge AOp)$
3. $\forall Cinit \bullet \exists Ainit \bullet Abs$

où $Astate$ et $Cstate$ désignent les espaces d'états abstrait et concret respectivement et $Ainit$ et $Cinit$ les espaces d'états initiaux abstrait et concret respectivement.

De même, une classe Object-Z C est dite une *simulation vers le haut* de la classe A s'il existe une relation Abs telle que chaque opération abstraite AOp soit associée à une opération concrète COp de la manière suivante :

1. $\forall Cstate \bullet \exists Astate \bullet Abs \wedge \textbf{pre}\ AOp \Rightarrow \textbf{pre}\ COp$
2. $\forall Astate', Cstate, Cstate' \bullet COp \wedge Abs' \Rightarrow (\exists AState \bullet Abs \wedge AOp)$
3. $\forall Astate, Cinit \bullet Abs \Rightarrow Ainit$

En reprenant l'exemple de la figure 5.1, la classe *Produit* est raffinée par :

Produit_1

$pdts : PRODUITS \rightarrow \{vrai, faux\}$
$Pcourant : PRODUITS$

$produit = pdts^{-1}[\{vrai\}]$
$P = Pcourant$

INIT

$pdts = PRODUITS \times \{faux\}$
$\exists\, pdt : PRODUITS \bullet Pcourant = pdt$

Creer

$\Delta(pdts, Pcourant)$
$pdt? : PRODUITS$

$pdts(pdt?) = faux$
$pdts'(pdt?) = vrai$
$Pcourant' = pdt?$

Supprimer

$\Delta(pdts)$

$pdts(Pcourant) = vrai$
$pdts'(Pcourant) = faux$

La notion de raffinement définie dans cette approche permet uniquement de raffiner les classes Object-Z. Le raffinement de processus n'est pas considéré. Derrick et al. montrent cependant que cette relation de raffinement est cohérente avec le raffinement CSP. L'expression de processus entre les classes Object-Z reste donc valide après le raffinement de données des classes.

5.4.5 Bilan

Cette approche utilise les langages CSP et Object-Z. Elle permet d'identifier un processus CSP avec la séquence des opérations exécutées par une classe Object-Z. Le formalisme utilisé préserve un double point de vue entre les deux langages. La sémantique de la notation combinée est basée sur celle des processus CSP. Des relations de simulation, valides et complètes par rapport au raffinement CSP, sont définies afin de raffiner les classes Object-Z. L'utilisation de cette relation de raffinement permet enfin de vérifier des propriétés dynamiques sur le système modélisé.

Contrairement à l'approche CSP-OZ, les langages CSP et Object-Z sont séparés. Par conséquent, la spécification même d'un système qui assure la cohérence des deux parties est difficile. Le langage CSP n'est pas considéré dans son ensemble, ce qui limite l'expressivité des processus. Cette méthode a toutefois l'avantage de proposer un processus de vérification. L'utilisation des deux points de vue constitue cependant une lourdeur. La méthode de vérification avec l'utilisation de **sat** devient en effet difficile pour un système comprenant de nombreuses classes. Il faut de plus utiliser la logique W de Z étendue à Object-Z. Enfin, le raffinement n'est défini que pour les raffinements de données des

classes Object-Z et non sur les processus.

Concernant la concurrence, les classes interagissent entre elles par synchronisation de leurs opérations. Une première limite de cette approche est le type de communications concernées (échange, partage de valeurs, coopération). De plus, les processus mettent en concurrence les classes Object-Z, ce qui rend la spécification d'ordonnancement sur les opérations d'une même classe ou entre classes difficile. La méthode n'est donc pas appropriée pour spécifier le comportement dans les systèmes d'information.

Si la séparation des langages de spécification, comme dans le cas de cette approche, autorise l'utilisation des outils existants, elle ne permet pas de prendre en compte les deux points de vue à la fois. Il n'y a en effet aucun lien, autre que sémantique, entre les deux aspects. Par exemple, il est possible d'utiliser FDR pour analyser le processus spécifiant le système, mais les caractéristiques des classes ne sont pas prises en compte. De même, l'utilisation d'un outil sur les classes ne permettra pas de vérifier les propriétés liées aux processus. La méthode implique donc la définition de nouvelles méthodes de vérification qui intègre les deux points de vue.

5.5 PLTL et B événementiel [Dar02]

Dans cette approche, deux formalismes sont utilisés pour spécifier les propriétés temporelles des systèmes réactifs : PLTL [Pnu81] et B événementiel [AM98]. L'approche propose également une méthode de vérification de ces propriétés. Les techniques de preuve utilisées en B sont associées à des techniques de model-checking. La combinaison porte donc à la fois sur la spécification et sur la vérification.

5.5.1 Syntaxe

La spécification de propriétés temporelles en B est possible en utilisant les systèmes d'événements. L'idée est d'utiliser en plus la logique temporelle PLTL pour exprimer les propriétés du système.

Systèmes d'événements en B. La méthode B permet de décrire des systèmes "ouverts" dont les opérations sont appelées par l'environnement. Elle ne permet pas de tenir en compte un ensemble fermé, c'est-à-dire un système dont les événements ne dépendent pas de l'extérieur. Avec les systèmes d'événements présentés dans [AM98], la notion d'événement gardé est introduite afin de considérer des systèmes "fermés" dans lesquels les événements ne réagissent pas avec l'environnement.

Un système d'événements est décrit en B par une machine comprenant les clauses suivantes :

- **CONSTRAINTS** : prédicat,
- **SETS** : ensembles,
- **INVARIANT** : prédicat,
- **INITIALISATION** : substitution d'initialisation,
- **EVENTS** : événements.

Les propriétés temporelles du système modélisé sont alors exprimées sous forme de contraintes dynamiques, c'est-à-dire à l'aide d'invariants dynamiques et de modalités.

Un invariant dynamique est une formule de la forme :

DYNAMICS $\mathcal{P}(V, V')$

où $\mathcal{P}(V, V')$ est un prédicat qui permet de caractériser comment les variables V sont autorisées à évoluer vers V' lors de l'application d'un événement du système. Une modalité B est de la forme :

select P Leadsto Q [**while** $e_1, ..., e_n$]
invariant J
variant V
end

ou de la forme :

select P Until Q [**while** $e_1, ..., e_n$]
invariant J
variant V
end

La modalité P Leadsto Q signifie que si P est vraie, alors les séquences d'événements indiqués dans la clause **while** conduisent fatalement à Q. La modalité P Until Q signifie que P Leadsto Q est vérifiée et que P est vraie tant que Q ne l'est pas.

Les propriétés exprimées en B événementiel sont de la forme : "si P est vraie, alors Q sera fatalement vraie dans un futur état de l'exécution du système". L'expressivité des propriétés temporelles est donc limitée en B événementiel.

PLTL. La logique temporelle, introduite par [Pnu77], est un formalisme mieux adapté que le B événementiel pour exprimer des propriétés temporelles. La logique temporelle linéaire propositionnelle (PLTL) est ici considérée. Elle permet en effet d'exprimer des propriétés dynamiques comme les propriétés de sûreté ou de vivacité qui sont difficiles, voire impossibles, à exprimer en B. De plus, les invariants dynamiques et les modalités B peuvent toujours s'exprimer en PLTL.

Les opérateurs temporels de PLTL sont : $\bigcirc$ (état suivant), $\Diamond$ (fatalement), $\Box$ (toujours), Θ (état précédent), $\mathcal{U}$ (jusqu'à), $\mathcal{S}$ (depuis) et $\mathcal{W}$ (à moins que). Ce dernier opérateur est défini par : pour tous ϕ et ψ,

$$\phi \mathcal{W} \psi = (\Box \phi) \vee (\phi \mathcal{U} \psi)$$

Une sémantique opérationnelle de PLTL est définie dans [Dar02].

5.5.2 Exemple

Le système d'événements suivant représente l'exemple de la figure 5.1 :

EVENT SYSTEM *Exemple*
SETS *PRODUITS ; EXState* = {*Arret*, *Marche*, *Marche_1*}

VARIABLES $EX,\ P,\ Produit$
INVARIANT $EX \in EXState \wedge P \in PRODUITS$
$\wedge\ Produit \subseteq PRODUITS$
INITIALISATION $EX := Arret \parallel P :\in PRODUITS \parallel Produit := \varnothing$
EVENTS
On =
select $EX = Arret$
then
$EX := Marche$
end;
Off =
select $EX = Marche$
then
$EX := Arret$
end;
$Creer$ =
select $EX = Marche$
then
any xx **where** $xx \in PRODUITS - Produit$
then
$EX := Marche\_1 \parallel$
$Produit := Produit \cup \{xx\} \parallel$
$P := xx$
end
end;
$Supprimer$ =
select $EX = Marche\_1$
then
$EX := Marche \parallel$
$Produit := Produit - \{P\}$
end
END

Contrairement à l'approche B classique, il n'est pas possible d'exprimer précisément quels produits sont créés ou supprimés par les événements *Creer* et *Supprimer*. L'utilisateur ne peut qu'"observer" les changements d'états de la variable *Produit* et ne peut donc pas spécifier quels produits il souhaite ajouter ou supprimer. Notre exemple est toutefois bien représenté, car si un produit a été créé, la garde et la variable auxiliaire P assurent que ce produit doit être supprimé avant qu'un nouveau produit ne soit créé.

L'objectif de cette approche est d'utiliser PLTL pour exprimer des propriétés temporelles sur ce système d'événements B. Il est possible par exemple d'exprimer le fait que le système fonctionne tant qu'il n'est pas à l'arrêt et que le seul état possible après *Arret* est *Marche* :

$$\Box(((EX = Marche)\mathcal{U}(EX = Arret)) \wedge (EX = Arret \Rightarrow \bigcirc(EX) = Marche))$$

À la différence des autres approches, PLTL permet d'exprimer ici des propriétés complémentaires aux propriétés exprimées par le système d'événements B. Elle ne permet pas en revanche d'exprimer des ordonnancements sur les événements

du système. Les propriétés sont en effet décrites en fonction des états et des variables du système.

5.5.3 Raffinement et vérification

Plusieurs approches sont possibles concernant le raffinement de systèmes d'événements B dont les propriétés temporelles sont exprimées en PLTL, comme le raffinement B ou bien le raffinement LTL.

Un raffinement LTL agit sur les propriétés PLTL. Une propriété temporelle ϕ_1 est dite raffinée par ϕ_2 si toutes les exécutions qui satisfont ϕ_2 satisfont également ϕ_1, autrement dit :

$$\phi_2 \Rightarrow \phi_1$$

Par conséquent, si une spécification abstraite ϕ_1 est raffinée successivement par $\phi_2, ..., \phi_n$, alors :

$$\phi_n \Rightarrow \phi_{n-1} \Rightarrow ... \Rightarrow \phi_1$$

Si, de plus, ϕ_1 satisfait une propriété ψ, alors :

$$\phi_1 \Rightarrow \psi$$

et par conséquent :

$$\phi_n \Rightarrow ... \Rightarrow \phi_1 \Rightarrow \psi$$

Le raffinement LTL permet donc de préserver les propriétés temporelles du système. Si cette notion de raffinement est simple à exprimer, elle est en revanche difficile à calculer sur des propriétés complexes. Concernant notre exemple, les événements sont par exemple trop simples pour être raffinés en LTL.

Ces différents raffinements ont la propriété de préserver les propriétés temporelles du système d'événements raffiné. Il est possible de reformuler ces propriétés grâce à des schémas définis dans [Dar02]. L'idée est de combiner les méthodes de model-checking et de preuve afin de vérifier certaines propriétés temporelles dans des systèmes d'événements.

Reformulation. Lors d'un raffinement, il est possible d'exprimer à différents niveaux d'abstraction le fonctionnement d'un système. Comme les propriétés temporelles définies en PLTL sur le système sont préservées par raffinement, il est possible de reformuler ces propriétés à l'aide des nouveaux éléments introduits par le raffinement. La reformulation des propriétés est, par analogie au raffinement de systèmes, une technique permettant de réécrire les propriétés. La reformulation est décidée par l'utilisateur et l'intérêt est d'exprimer de nouvelles propriétés sur les nouveaux éléments introduits par le raffinement. Darlot a défini plusieurs schémas de reformulation. Chaque schéma de reformulation est associé à des conditions suffisantes permettant d'assurer que la propriété reformulée est vérifiée dans le système raffiné.

Méthode de vérification. La figure 5.9 est un résumé de la méthode proposée pour vérifier des propriétés PLTL sur des systèmes d'événements B.

La méthode de vérification consiste en quatre étapes.

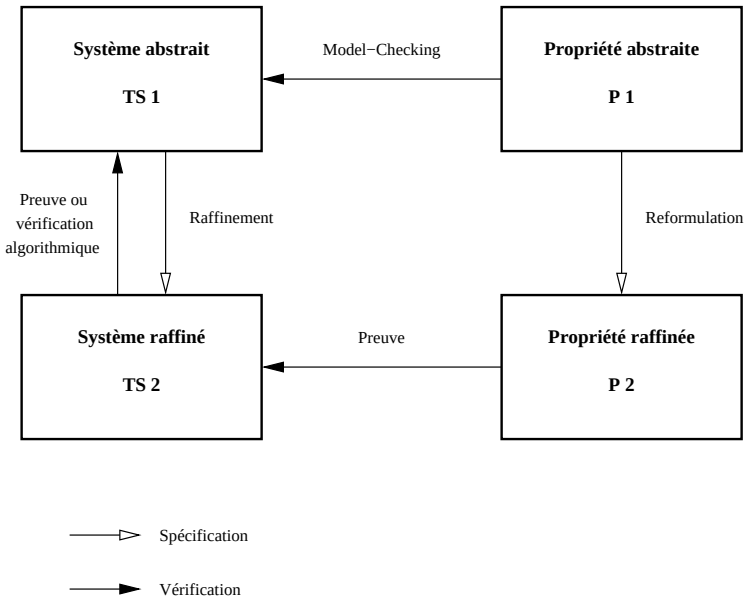


FIG. 5.9 – Méthode de vérification B-PLTL

1. Dans un premier temps, la propriété $P1$ est exprimée en PLTL sur le système abstrait $TS1$. Cette propriété est vérifiée par model-checking. Pour un système abstrait de petite taille, la vérification a de fortes chances d'aboutir (pas d'explosion combinatoire des états).
2. Ensuite, le système abstrait $TS1$ est raffiné en $TS2$. Cette étape est justifiée par preuve (technique usuelle en B) ou par un algorithme de vérification (sur les LTS). D'autre part, la propriété $P1$ est préservée par raffinement sur le système raffiné $TS2$.
3. La troisième étape consiste à reformuler la propriété temporelle $P1$ en une propriété $P2$.
4. Enfin, $P2$ est justifiée en prouvant les conditions suffisantes associées aux schémas de reformulation utilisés.

Cette méthode de reformulation des propriétés peut être considérée comme un raffinement des formules PLTL sur un système d'événements qui est raffiné.

5.5.4 Bilan

Cette approche repose sur l'utilisation combinée de la logique temporelle PLTL et du B événementiel. Ce dernier permet de spécifier des systèmes d'événements sur lesquels des propriétés temporelles sont exprimées à l'aide de PLTL. Cette approche se distingue des autres car elle concerne la spécification de propriétés temporelles sur des machines B.

Le raffinement des systèmes d'événements, comme le raffinement LTL, permet de préserver les propriétés temporelles. D'un point de vue sémantique, les systèmes d'événements sont modélisés par des LTS. PLTL est justifiée par une sémantique opérationnelle.

La préservation des propriétés temporelles par raffinement permet d'introduire la notion de reformulation de propriétés afin de dériver des propriétés de systèmes abstraits en de nouvelles propriétés sur les systèmes raffinés. Une méthode de vérification combinant raffinement et reformulation pour la partie spécification et preuve et model-checking pour la partie vérification est proposée afin d'automatiser au maximum les vérifications de propriétés temporelles dans le cas de système compliqué.

Contrairement aux autres approches qui combinent uniquement des langages de spécifications formelles, cette méthode combine également des méthodes de preuve et de model-checking pour vérifier les propriétés temporelles du système. Le processus manque actuellement d'outil.

L'utilisation du B événementiel peut sembler ambigü (pourquoi pas B tout simplement ?) puisque les propriétés sont exprimées en PLTL, mais elle permet de spécifier formellement les états et les événements au niveau des systèmes d'événements. Ces déclarations sont également prises en compte dans la sémantique au niveau des LTS.

Dans le cadre des systèmes d'information, la prise en compte de propriétés temporelles peut s'avérer utile. Toutefois, l'utilisation de système d'événements B ne convient pas : notre exemple a pu être bien représenté grâce aux états spécifiés dans la garde des événements. Dans le cas d'un événement qui ne modifierait pas l'état, l'observateur ne peut pas déterminer avec précision quels produits il souhaite créer ou supprimer. Cette approche ne permet pas enfin d'exprimer facilement des ordonnancements sur les événements du système.

5.6 Circus [WC02]

Circus est un langage créé à partir de Z [Spi92] et CSP [Hoa85], pour y intégrer le calcul du raffinement [BvW98]. Le but est de faciliter les utilisations de Z et CSP, tout en réutilisant les théories existantes. La notion de raffinement est complexe dans le cadre d'une méthode intégrée, car les raffinements au sens Z et CSP n'utilisent pas les mêmes modèles sémantiques. Le raffinement Z est par exemple fondé sur les transformations de prédicats, alors que CSP est représenté par le modèle des échecs-divergences. Pour pallier ce manque, Circus s'appuie sur une théorie unifiée [HJ98], qui permet d'interpréter les deux approches dans un modèle unique.

5.6.1 Syntaxe

Un programme Circus est structuré en termes de processus. Chaque processus est représenté par un état, décrit par un schéma Z, et par un comportement, décrit par des actions exprimées dans un style CSP. Les actions font référence aux Action Systems de Back, Morgan, etc ... (voir section 2.2.1). En Circus, les actions peuvent être décrites par les commandes gardées de Dijkstra [Dij76], mais aussi par des schémas Z, et sont composées avec des opérateurs issus de CSP.

Les programmes Circus sont construits comme les spécifications en Z. Ils sont composés de séquences de paragraphes regroupés dans des schémas. La syntaxe de Circus étend celle de Z afin de prendre en compte les définitions de canaux et de processus. Les canaux sont déclarés avec **channel**. Les processus

sont déclarés par :

process *Nom* = *ProcExpr*

où *Nom* est le le nom du processus et *ProcExpr* est une expression du processus du langage Circus.

Les expressions de processus en Circus sont de la forme :

begin *Paragraph** • *Action* **end**

où *Paragraph* désigne des paragraphes classiques de Z ou bien des déclarations d'action. La notation * indique que le processus peut déclarer un nombre arbitraire fini de paragraphes.

Les actions en Circus sont déclarées avec des schémas Z, des commandes gardées et des opérateurs CSP. Les actions de base sont *Stop* (l'action qui arrête l'exécution du processus), *Skip* (l'action qui ne fait rien) et *Chaos* (l'action qui diverge). Les actions gardées sont de la forme :

Predicate & *Action*

Les opérateurs sur les actions sont : les choix internes ($\sqcap$) et externes ($\Box$), la synchronisation paramétrée ($|\ [\Delta]\ |$, similaire à $||$, mais uniquement sur les actions de Δ), l'entrelacement ($|||$), la restriction ($\setminus$), la séquence ($;$), et les communications (de la forme $c\,?x \rightarrow A$ ou $c\,!x \rightarrow A$). Circus permet enfin d'utiliser les μ-expressions de Z pour définir des actions de manière récursive :

$\mu\, N \bullet Action$

où N est un identifiant. Cette expression n'est définie que si l'action est possible et, dans ce cas, elle prend en compte les valeurs des états dépendant de N.

Un processus de la forme :

begin $P_1, ..., P_n \bullet A$ **end**

exécute l'action A décrite par les expressions d'actions définies ci-dessus. Les paragraphes $P_1, ..., P_n$ servent à introduire les types et les opérations utilisées par les actions.

Les processus peuvent également être composés entre eux avec les opérateurs suivants : les choix internes ($\sqcap$) et externes ($\Box$), la synchronisation paramétrée ($|\ [\Delta]\ |$), l'entrelacement ($|||$), la restriction ($\setminus$) et la séquence ($;$). Les processus peuvent être indexés avec l'opérateur $\odot$. Ils peuvent enfin être renommés.

Contrairement à CSP, les opérateurs ne sont pas quantifiables et le préfixe des processus n'est pas pris en compte dans Circus.

5.6.2 Exemple

Notre exemple de référence peut être spécifié en Circus de la manière suivante :

ETATS = *Arret* | *Marche* | *Marche_1*
channel *In* : *PRODUITS*
process *Machine* $\triangleq$ **begin**

Etat

$etat : ETATS$
$produit : \mathbb{P}\, PRODUITS$
$P : PRODUITS$

EtatInit

$Etat$

$etat = Arret$
$produit = \varnothing$
$\exists\, pdt : PRODUITS \bullet P = pdt$

On

$\Delta Etat$

$etat = Arret$
$etat' = Marche$
$produit' = produit$
$P' = P$

Off

$\Delta Etat$

$etat = Marche$
$etat' = Arret$
$produit' = produit$
$P' = P$

Creer

$\Delta Etat$
$pdt? : PRODUITS$

$etat = Marche \land pdt? \in PRODUITS - produit$
$etat' = Marche\_1$
$produit' = produit \cup \{pdt?\}$
$P' = pdt?$

Supprimer

$\Delta Etat$

$etat = Marche\_1 \land P \in produit$
$etat' = Marche$
$produit' = produit - \{P\}$
$P' = P$

$CreerProduit \triangleq$
$In?pdt : PRODUITS - produit \rightarrow Creer$

•
EtatInit ; $(\mu\, X \bullet (On\ ;\ (\mu\, Y \bullet (CreerProduit\ ;\ Supprimer\ ;\ Y)) \Box (Off\ ;\ X)))$
end

Le processus qui spécifie le système s'appelle ici *Machine*. Les deux premiers schémas Z, *Etat* et *EtatInit*, définissent respectivement l'état et l'initialisation du système. Les schémas suivants représentent les opérations, dont les spécifications sont similaires aux approches précédentes. Le processus définit ensuite une action intermédiaire, *CreerProduit*, qui permet de contraindre l'opération *Creer*. Enfin, l'action principale du processus, qui est décrite juste après le symbole •, définit le comportement global du système modélisé. Elle utilise toutes les définitions et tous les paragraphes Z décrits auparavant.

Cette expression d'action signifie que l'initialisation du schéma *EtatInit* est dans un premier temps effectuée. Une première μ-expression permet ensuite de définir de manière récursive le comportement de l'action. Le processus exécute *On*. À partir de cet état, le processus *Systeme* a deux options : soit il exécute la deuxième μ-expression pour créer et supprimer des produits, soit il exécute *Off* et le système revient à l'état initial, prêt à exécuter de nouveau *On*. Dans le premier cas, la μ-expression permet d'exécuter de manière récursive la séquence *CreerProduit*, *Supprimer*.

Dans les systèmes d'actions, les actions exécutables sont déterminées en fonction de leur garde. Le choix entre les deux options dépend en effet de la garde de l'action *CreerProduit*. Si l'environnement propose à travers le canal *In* un produit *pdt* qui vérifie la garde de l'action, alors l'opération *Creer* sera exécutée. Sinon, le système choisira la seconde option.

Dans le cas de *Supprimer*, il peut y avoir un risque de divergence si les préconditions ne sont pas vérifiées. Toutefois, les contraintes de l'action *CreerProduit* assurent que l'opération *Creer* sera correctement exécutée et par conséquent, les préconditions de *Supprimer* seront satisfaites.

5.6.3 Théorie unifiée de programmation

Le langage Circus a été créé dans le but d'unifier les théories qui sont à la base de Z et CSP et ainsi faciliter le calcul de raffinement. La sémantique de Circus est inspirée de la théorie unifiée de Hoare et al. [HJ98].

Dans leur unification, Hoare et al. s'appuient sur la théorie des relations pour représenter les programmes, les spécifications et les conceptions. Ces travaux ont pour but d'apporter une base commune à toutes les branches de la programmation quel que soient les paradigmes utilisés. Par exemple, la séquence est représentée par une composition relationnelle et le non-déterminisme par une disjonction. Des concepts comme la correction ou le raffinement d'une spécification sont interprétés comme des inclusions de relations. Cette base permet ainsi de raisonner sur une représentation des langages avec les lois du calcul relationnel.

Dans cette approche, les théories des langages sont caractérisées par trois éléments : l'alphabet, la signature et les conditions *healthiness*. L'*alphabet* d'une théorie est l'ensemble des noms des observations externes d'un programme. La *signature* fournit une syntaxe pour dénoter les différents éléments de la théorie. Enfin, les conditions *healthiness* permettent de sélectionner, parmi la théorie, des éléments qui respectent certaines propriétés. Les éléments de la théorie

sont par exemple regroupés en programmes, conceptions et spécifications. Les programmes constituent un sous-ensemble des conceptions, et les conceptions forment un sous-ensemble des spécifications.

Afin d'unifier les théories de Z, CSP et des Action Systems, Woodcock et al. ont modélisé le langage Circus en s'appuyant sur la théorie unifiée des programmes [Woo02, WC02]. Le modèle choisi pour représenter un programme Circus est une spécification Z qui décrit les processus et les actions comme des relations. Le langage Z est donc utilisé comme méta-langage pour décrire la sémantique de Circus.

En modélisant ainsi le langage, une théorie du raffinement basée sur les relations de simulation a pu être définie à partir du modèle sémantique. Les interprétations des opérateurs empruntés à CSP et aux Action Systems ont pu ainsi être utilisées pour relier les différentes notions de raffinement en Circus.

5.6.4 Raffinement

Contrairement aux autres approches présentées dans ce chapitre, Circus propose une relation de raffinement qui intègre les différents aspects du langage : les actions, les processus et les données.

Il existe en effet plusieurs niveaux de raffinement. Parmi les définitions les plus courantes de cette notion, un raffinement doit préserver tous les comportements possibles que la spécification abstraite autorise (voir chapitre 4). Le niveau le plus simple du raffinement est donc une implication. Si un processus P satisfait une spécification S, dénoté par :

$$[P \Rightarrow S]$$

alors P est un raffinement de S. Les crochets signifient que la propriété est vraie pour toute quantification universelle sur toutes les observations et sur toutes les variables d'état.

Le raffinement d'actions est défini de la manière suivante. Soient A_1 et A_2 deux actions définies sur le même espace d'états, alors A_1 est raffinée par A_2 (notée $A_1 \sqsubseteq_{\mathcal{A}} A_2$) si et seulement si toute observation sur A_2 est possible sur A_1, autrement dit :

$$[A_2 \Rightarrow A_1]$$

Le raffinement de CSP est généralement défini sur les modèles sémantiques utilisés pour les algèbres de processus, comme par exemple le modèle des échecs-divergences. Le raffinement de processus est défini dans [CW02] en terme de raffinement d'actions. Dans la définition suivante, les notations $P.st$ et $P.act$ désignent respectivement l'état local et l'action principale du processus P. Le processus P est raffiné par le processus Q si :

$$(\exists\, P.st\,; P.st' \bullet P.act) \sqsubseteq_{\mathcal{A}} (\exists\, Q.st\,; Q.st' \bullet Q.act)$$

Les processus en Circus sont en effet décrits par une action principale sur les états du système. La notion de raffinement présentée ci-dessus signifie que si l'action principale correspondant au processus P est raffinée par une action définissant le processus Q, alors P est raffiné par Q.

Les relations de simulation sont enfin adaptées au contexte du langage Circus. Par exemple, une simulation *forward* entre les actions A et B des processus

P et Q, avec un état local L, est une relation R satisfaisant les deux propriétés suivantes :

1. $[\forall\, Q.st \bullet (\exists\, P.st \bullet R)]$
2. $[\forall\, P.st\ ; Q.st\ ; Q.st' \bullet R \wedge B \Rightarrow (\exists\, P.st' \bullet R' \wedge A)]$

Plus généralement, un processus P simule un processus Q si les actions principales de P et de Q vérifient une des relations de simulation. Enfin, ces relations sont suffisantes pour montrer le raffinement entre processus.

Woodcock et al. proposent dans [SWC02, CW02] des lois pour vérifier les relations de simulations sur chaque opérateur du langage. Une première série de lois permet de décomposer un processus décrit en Circus et une deuxième série de lois permet de prouver le raffinement sur les composants de la spécification.

La stratégie de raffinement [CSW03] proposée dans Circus, qui est la seule de cette revue de littérature à intégrer plusieurs relations de raffinement (CSP, Z et Action Systems), est toutefois difficile à utiliser. Elle ne propose en effet aucun outil et est actuellement au stade de l'expérimentation.

5.6.5 Bilan

Circus est un langage qui unifie les théories de Z, CSP et Action Systems pour fonder un nouveau calcul de raffinement. Le modèle de Circus, qui s'appuie sur la théorie unifiée de programmation de Hoare et al., représente les différents opérateurs de la syntaxe empruntés à Z, CSP et Action Systems dans une théorie du calcul relationnel. Cette approche a l'avantage de bien intégrer les langages CSP et Z. L'utilisation de Z comme méta-langage a permis d'utiliser des outils de Z pour vérifier le modèle du langage Circus.

Comme dans le cas du nouveau langage CSP-OZ, Circus propose une nouvelle syntaxe et une nouvelle sémantique. Il ne peut donc pas utiliser des outils existant en CSP ou en Z pour assister l'utilisateur. Même si Circus récupère une grande partie des opérateurs de ces langages, l'utilisateur devra s'habituer à la nouvelle syntaxe.

Concernant notre problème, à savoir la modélisation du comportement dans les systèmes d'information, l'exemple nous montre qu'il est difficile d'exprimer certaines séquences d'événements avec les actions Circus. Le processus spécifiant le système est en effet défini par une action principale. Comme les systèmes d'action peuvent bloquer, l'utilisateur doit faire attention aux gardes des actions.

L'utilisation d'une spécification basée sur les actions pourrait être intéressante s'il existait un moyen de vérifier si les expressions d'actions gardées sont exécutables. À notre connaissance, Circus ne propose pas encore d'outil permettant une telle analyse.

Une des caractéristiques les plus intéressantes de Circus concerne le raffinement. Le langage a été créé dans le but d'unifier les relations de raffinement existant en Z, CSP et Action Systems. Une fois de plus, en raison de la nouveauté du langage, la méthode proposée n'est pas outillée. Le raffinement est un processus qui implique fortement l'utilisateur, car il doit, à chaque étape, réécrire sa spécification afin de la rendre plus concrète et ce, en fonction de ses besoins. Ce processus est déjà difficile dans le cadre d'une approche simple, il devient donc très difficile dans le cadre d'une combinaison de plusieurs langages ! Les auteurs sont actuellement en train de créer des outils d'assistance. La possibilité de raffiner le processus du système avec son action principale en

un processus plus concret dont les schémas et les actions sont également raffinés sera alors un avantage indéniable de l'approche Circus.

5.7 EB^3-B [FL03]

Cet exemple, qui achève la revue des approches de combinaisons de spécifications formelles, est en réalité le point de départ de notre travail, c'est-à-dire utiliser EB^3 [FSD03] et B [Abr96] pour modéliser les systèmes d'information. Elle constitue avec [FL01] une des premières tentatives de combinaison des méthodes B et EB^3 présentées dans les chapitres 3 et 2, respectivement.

5.7.1 Principe

La méthode présentée dans [FL03] permet de vérifier l'ordonnancement des opérations spécifiées dans une machine B. Il n'y a aucune restriction sur les langages B et EB^3 pour appliquer cette approche.

Les propriétés statiques et les opérations du système sont modélisées avec le langage B, par la spécification d'une machine abstraite classique. Le langage EB^3 permet d'exprimer de manière explicite les séquences d'opérations B que l'utilisateur souhaite vérifier pour le système. Le système est donc uniquement modélisé par une machine B et les traces exprimées en EB^3 ne servent qu'à vérifier des propriétés d'ordonnancement sur les opérations de la machine. La vérification d'une telle propriété, spécifiée en EB^3, est réalisée à l'aide du raffinement B.

Il s'agit donc d'une spécification partielle des propriétés dynamiques. L'objectif à plus long terme est de spécifier et de vérifier divers types de propriétés (comme les propriétés de vivacité, de sûreté, etc ...) du système.

5.7.2 Exemple

Cette approche commence par la description du système avec une machine B classique. Si on reprend l'exemple de la figure 5.1, la machine décrivant le système est similaire aux approches précédentes utilisant B :

MACHINE *Produit*
SETS $PRODUITS ; \ ETATS = \{0, 1, 2\}$
VARIABLES *Produits*, *Etat*, P
INVARIANT $Produits \subseteq PRODUITS \wedge Etat \in ETATS$
$\wedge \ P \in PRODUITS$
INITIALISATION $Produits := \varnothing \parallel Etat := 0 \parallel P :\in PRODUITS$
OPERATIONS
$On =$
if $Etat = 0$
then
  $Etat := 1$
end ;
$Off =$
if $Etat = 1$
then

$Etat := 0$
end ;
$Creer(pdt) =$
pre $pdt \in PRODUITS - Produits$
then
if $Etat = 1$
then
$Produits := Produits \cup \{pdt\} \parallel$
$Etat := 2 \parallel$
$P := pdt$
end
end ;
$Supprimer =$
if $Etat = 2$
then
$Produits := Produits - \{P\} \parallel$
$Etat := 1$
end
END

On remarque que les opérations sont définies à l'aide d'expressions de la forme **if then end** pour garantir que les états sont bien ceux indiqués dans le LTS de la figure 5.1.

La méthode EB^3 classique s'appuie sur un diagramme de classes UML qui définit les classes (appelées types d'entité en EB^3), les associations et les attributs du système. Ces notions sont ici implicitement prises en compte par la description en B. La machine abstraite peut éventuellement être obtenue par génération automatique à partir de diagrammes UML, comme dans la méthode UML-B présentée dans le chapitre 3.

L'exemple considéré est ici modélisé par un type d'entité `Produit`, par les actions `On`, `Off`, `Creer` et `Supprimer`, et par une fonction `AfficheProduit` qui retourne, le cas échéant, le produit existant.

L'étape suivante consiste à définir l'espace des entrées-sorties en EB^3. Dans notre exemple, il y a un seul type de données, `PRODUITS`, et les événements du système ont pour signature :

```
On() : void
Off() : void
Creer(pid : PRODUITS) : void
Supprimer() : void
AfficheProduit() : PRODUITS
```

Les actions `On`, `Off` et `Supprimer` n'ont ni entrée, ni sortie. Seule l'action `Creer` a un paramètre d'entrée de type `PRODUITS`. `Creer` et `Supprimer` correspondent aux opérations B homonymes. L'action `AfficheProduit` permet d'afficher le produit existant.

Pour décrire le comportement du type d'entité `Produit`, on utilise une expression de processus EB^3 pour définir l'ensemble des traces d'actions valides :

```
Produit(pid : PRODUITS) =
   Creer(pid) . Supprimer
```

Le comportement global des entrées du système est spécifié dans `main` :

```
main =
(
   On .
   (
        ( | pid : PRODUITS : Produit(pid) )^*
        ||
        AfficheProduit^*
   ) .
   Off
)^*
```

Cette expression de processus décrit l'ensemble des traces d'entrée valides du système. Le système commence par exécuter `On`. Ensuite, chaque produit qui est créé est supprimé avant la création d'un nouveau produit. Le choix du produit (|) et la séquence de création-suppression (dans `Produit(pid)`) sont répétés un nombre fini de fois (`^*`). Ensuite le système termine avec `Off` et la machine peut être réactivée (le dernier `^*`).

Pour représenter les réponses du système, des fonctions récursives à la CAML sont utilisées. La fonction `ProduitCourant` indique quel est le produit existant en fonction de la trace courante du système :

```
ProduitCourant(trace : TRACE-VALIDE) : PRODUITS =
   match last(trace) with
      nil -> nil
      Creer(pid) -> pid
      Supprimer -> nil
      _ -> ProduitCourant(front(trace))
```

Enfin, la règle d'entrée-sortie `R1` permet d'invoquer la fonction récursive `ProduitCourant` à chaque fois que l'événement `AfficheProduit` est appelé :

```
Rule R1 :
Input AfficheProduit()
Output ProduitCourant(trace)
EndRule
```

Par conséquent, l'expression de processus `main` modélise bien le LTS présenté dans la figure 5.1. Pour vérifier qu'elle est cohérente avec le modèle décrit en B, la spécification EB^3 est dans un premier temps traduite en B, puis le raffinement permet de prouver l'ordonnancement des opérations.

5.7.3 Traduction EB^3 - B

La traduction d'une spécification EB^3 en B correspond à la définition d'une sémantique B pour chaque opérateur de la syntaxe du langage EB^3.

La machine B obtenue par traduction d'une spécification EB^3 vérifie les principes suivants :

1. Elle ne contient qu'une seule variable d'état t qui représente la trace courante du système.
2. L'ensemble des traces valides du système est spécifié en B à l'aide de l'ensemble $\tau(main)$.
3. L'invariant de la machine établit que la trace courante du système est toujours valide : $t \in \tau(main)$.
4. L'état initial de la machine est la trace vide.
5. Pour chaque action de la trace EB^3 à traduire, une opération B homonyme est spécifiée sous la forme **if then end**. La condition du **if** permet d'assurer que la nouvelle trace du système, une fois l'action exécutée, est valide. La substitution dans **then** ajoute effectivement l'action à la fin de la séquence d'actions définissant la trace du système.

Par exemple, la machine B suivante est la traduction de la spécification EB^3 présentée dans le paragraphe précédent :

MACHINE *TraductionEB3*
SEES ...
VARIABLES t
INVARIANT $t \in \tau(main)$
INITIALISATION $t := []$
OPERATIONS
$On =$
if $t \leftarrow On \in \tau(main)$
then
  $t := t \leftarrow On$
end ;
$Off =$
if $t \leftarrow Off \in \tau(main)$
then
  $t := t \leftarrow Off$
end ;
$Creer(pdt) =$
pre $pdt \in PRODUITS$
then
  if $t \leftarrow Creer(pdt) \in \tau(main)$
  then
    $t := t \leftarrow Creer(pdt)$
  end
end ;
$Supprimer =$
if $t \leftarrow Supprimer \in \tau(main)$
then
  $t := t \leftarrow Supprimer$
end
END

La clause **SEES** permet d'utiliser d'autres machines B qui contiennent les définitions de l'ensemble $\tau(main)$. La spécification en B de l'ensemble de toutes

les traces possibles qui sont valides constitue aujourd'hui encore un problème non résolu. Cette difficulté essentiellement technique ne remet tout de même pas en cause le raisonnement qui est à la base de cette approche.

La machine B obtenue est donc une traduction du LTS associé à l'expression de processus EB^3. Pour vérifier que la propriété d'ordonnancement est correcte, la technique de raffinement existant en B est utilisée.

5.7.4 Vérification

Pour prouver que la trace EB^3, traduite en la machine B *TraductionEB3*, est vérifiée par les opérations de la machine *Produit*, on prouve que cette dernière est un raffinement en B de la machine *TraductionEB3*. Un raffinement préserve en effet le comportement de la machine abstraite. En particulier, chaque opération a le même comportement observable dans le raffinement que dans la machine raffinée [Abr96].

La difficulté consiste à trouver d'une part les invariants de collage pour relier les variables concrètes *Produits* et *Etat* de la machine *Produit* et la variable abstraite t de la machine *TraductionEB3* et d'autre part à prouver les obligations de preuve définies dans [Abr96] qui assurent la correction du raffinement en B.

Concernant le premier point, la variable concrète *Produits* représente l'ensemble des produits existants. Or cet ensemble est retrouvé en EB^3 grâce à la fonction récursive `ProduitCourant`. Par conséquent,

$$Produits \quad = \quad \{\texttt{ProduitCourant(t)}\} \tag{5.1}$$

D'autre part, en analysant les LTS des deux machines, il est possible de trouver les correspondances entre les variables *Etat* et t :

$$Etat = 0 \quad \Leftrightarrow \quad t = [] \vee last(t) = Off \tag{5.2}$$
$$Etat = 1 \quad \Leftrightarrow \quad last(t) \in \{On, Supprimer\} \tag{5.3}$$
$$Etat = 2 \quad \Leftrightarrow \quad last(t) = Creer \tag{5.4}$$

où la fonction *last* retourne la dernière action de la trace. L'invariant de collage, dénoté par *IC*, est donc la conjonction des équations (5.1) à (5.4).

Les preuves de raffinement en B portent sur l'initialisation et sur chaque opération. L'obligation de preuve concernant l'initialisation est de la forme suivante :

$$[Produits := \varnothing \parallel Etat := 0 \parallel P :\in PRODUITS]$$
$$[t := []]$$
$$(IC \wedge Produits \subseteq PRODUITS \wedge Etat \in ETATS \wedge P \in PRODUITS)$$

Cette obligation de preuve assure que les substitutions d'initialisation des variables abstraites et concrètes permettent d'établir les invariants du raffinement (c'est-à-dire l'invariant de collage et les invariants de la machine concrète). En évaluant les substitutions, on obtient :

$$\begin{aligned} \varnothing \quad &= \quad \{ProduitCourant([])\} \wedge \\ 0 = 0 \quad &\Leftrightarrow \quad [] = [] \vee last([]) = Off \wedge \\ 0 = 1 \quad &\Leftrightarrow \quad last([]) \in \{On, Supprimer\} \wedge \end{aligned}$$

$$\begin{array}{rcl} 0 = 2 & \Leftrightarrow & last([]) = Creer \wedge \\ \varnothing & \subseteq & PRODUITS \wedge \\ 0 & \in & ETATS \wedge \\ P & \in & PRODUITS \end{array}$$

ce qui est vrai par définition de la fonction récursive *ProduitCourant*.

Les obligations de preuve concernant les opérations sont plus complexes. Il faut par exemple prouver pour l'opération *On* que, sous les hypothèses *IC*, sous les invariants de la machine concrète et si $t \in \tau(main)$, alors :

$$\begin{array}{l} [\textbf{if } Etat = 0 \textbf{ then } Etat := 1 \textbf{ end}] \\ [\textbf{if } t \leftarrow On \in \tau(main) \textbf{ then } t := t \leftarrow On \textbf{ end}] \\ (IC) \end{array}$$

Des stratégies de preuve sont indiquées dans [FL03]. Une technique consiste notamment à analyser le LTS associé à l'expression de processus EB^3 pour distinguer les différents cas possibles.

Une conséquence de cette vérification est la possibilité de modifier de manière progressive les modélisations obtenues en B et EB^3. Si on avait par exemple utilisé une machine B sans la variable *Etat*, les obligations de preuve du raffinement auraient été fausses. Il n'aurait en effet pas été possible de distinguer les différents états du système dans la machine B. L'utilisation d'une trace EB^3 permet donc de faire ressortir les erreurs au niveau de la machine B. D'un autre côté, la spécification EB^3 dépend de la modélisation en B car elle utilise les types d'entité implicitement définis dans les machines, et par conséquent leurs opérations.

5.7.5 Bilan

Cette approche repose sur la combinaison des méthodes B et EB^3. Elle permet de prouver grâce à la technique de raffinement existant en B qu'une trace d'opérations B est valide dans le système modélisé.

L'utilisation de B permet de profiter de tous les avantages de la méthode et notamment des outils associés. B a été de plus utilisé pour spécifier des systèmes d'information, notamment avec la méthode UML-B. Le langage EB^3 a été spécialement créé pour concevoir des systèmes d'information. Les deux approches semblent donc efficaces pour spécifier formellement des SI.

La modélisation B, qui repose sur les transitions d'états, permet de représenter aisément les structures de données et les opérations d'un système. Le langage EB^3 est plutôt dédié à la spécification du comportement, puisqu'il fait la distinction entre les entrées valides du système sous la forme de traces et les réponses avec l'aide de fonctions récursives sur la trace courante du système. Les deux vues semblent donc être complémentaires, même si EB^3 est à niveau plus abstrait que le langage B.

La vérification des propriétés d'ordonnancement repose sur cette dernière remarque. Les spécifications EB^3 sont traduites en B et la correction des propriétés dynamiques exprimées est prouvée en montrant que la machine correspondant aux traces EB^3 est raffinée au sens B par la machine B modélisant le système.

Une remarque importante concerne l'"inter-dépendance" des spécifications B et EB^3 : les preuves de raffinement permettent non seulement de prouver

les propriétés d'ordonnancement voulues mais aussi de modifier les éventuelles erreurs de la spécification B initiale. Cette interaction entre B et EB^3 n'est possible que si les deux méthodes autorisent une correction et une vérification rapides. Cela implique notamment l'utilisation d'outils comme l'Atelier B [Cle].

5.8 Étude des propriétés des combinaisons de spécifications formelles

Avant de faire une synthèse de l'état de l'art sur la combinaison de spécifications formelles, nous pouvons citer quelques autres approches similaires.

5.8.1 Travaux équivalents

Cette revue des approches de combinaisons de spécifications formelles n'est pas exhaustive. Il existe en effet d'autres approches similaires qui n'ont pas été détaillées faute de place. Elles sont toutefois assez proches des méthodes présentées et offrent à peu près les mêmes avantages et les mêmes défauts que les approches précédentes.

Abstract State Machines [Gur95]. Les Abstract State Machines (ASM), appelées aussi Evolving Algebras (EA), ont pour but de relier les spécifications de type algébrique et leur modèle sémantique. L'approche consiste à construire des machines (appelées aussi "e-algèbres") qui représentent les systèmes de manière à ce que la correction des spécifications soit établie par de simples observations et vérifications. Des outils permettent en outre de simuler les machines obtenues.

Les ASM sont une variante de la logique du premier ordre avec égalité. Les structures algébriques "classiques" sont généralement définies par la syntaxe du langage et par une algèbre qui modélise ce langage. Une signature est définie par la donnée des symboles de langage. Elle peut comprendre, suivant la logique considérée, des symboles de sortes, des noms de relations ou bien des noms de fonctions. Une algèbre A (qu'on appellera ici statique, par opposition aux e-algèbres) de signature S est un ensemble non vide X associé à une interpétation γ des symboles de S dans X.

Une e-algèbre permet en outre de changer la sémantique d'une fonction ou d'une relation par l'intermédiaire de règles sur les symboles du langage. Plusieurs opérateurs sont utilisés pour définir ces règles, comme la séquence, les boucles conditionnelles ou la récursivité. Elles peuvent enfin être gardées. À chaque étape de l'exécution de l'ASM, les gardes sont réévaluées afin d'exécuter les règles et de modifier la sémantique des symboles de relations et de fonctions concernées.

Cette approche se distingue des autres exemples de cette revue de littérature, car elle concerne d'une part les spécifications de type algébrique et elle n'est pas une combinaison de plusieurs approches de spécifications formelles. Elle permet d'intégrer les aspects dynamiques en modifiant la sémantique. Pour cette raison, cette approche ressemble à Circus où la sémantique est définie de manière à simplifier le calcul du raffinement.

Z + Petri Nets [PJ03]. Les réseaux de Petri [Pet81] sont des graphes bipartis composés de places et de transitions. Ce langage formel permet de décrire de manière graphique, mais rigoureuse, des systèmes concurrents. Dans cette approche, de nouveaux réseaux, appelés des réseaux d'activation concurrente, sont définis comme des réseaux de Petri classiques dans lesquels deux types de places sont distingués : les places classiques et les *ZPlaces*. Pour représenter le comportement du système, les réseaux de Petri associent aux places des jetons. Les transitions permettent de déterminer sous quelles conditions les jetons du système peuvent passer d'une place à une autre.

Dans l'approche Z + Petri Nets, le système est représenté d'une part par une spécification Z classique et d'autre part par un réseau d'activation concurrente dans lequel les ZPlaces sont associées à des opérations de la spécification Z. Quand une ZPlace contient un jeton, alors l'opération Z correspondante est activée et peut être exécutée.

Cette approche conserve le double point de vue entre les deux parties de la modélisation. Le seul lien entre spécification Z et réseaux de Petri est l'association d'une opération Z à une ZPlace. Par conséquent, l'utilisation des outils existants est possible dans chacune des deux parties. Pour éviter des redondances, les opérations Z ne spécifient aucune précondition concernant une variable représentant l'état global du système et les réseaux de Petri utilisés sont les plus simples et ne permettent pas, comme les réseaux de Petri de haut niveau ou les réseaux colorés, de spécifier des contraintes sur les données.

Les défauts de Z + Petri Nets sont, d'une part, que la séparation entre les deux parties de la spécification ne permet de vérifier le système dans son ensemble et que, d'autre part, l'analyse des réseaux de Petri devient difficile dès que le système est complexe.

Cet exemple est assez proche de la méthode avec CSP et Object-Z, où les deux parties de la spécification ne sont liées que par l'identification d'une classe Object-Z à un processus CSP.

ZCCS [GS97a]. Le langage CCS ne prévoit pas une syntaxe et une sémantique précise concernant le passage des valeurs en paramètre des agents (voir section 3.3.1). Cette approche propose d'utiliser la notation Z et une sémantique opérationnelle pour compléter CCS [Mil89].

Une spécification ZCCS est donc une spécification CCS dans laquelle les agents font appel à des données décrites par des schémas Z. La partie Z de la spécification permet de définir les ensembles abstraits, les variables et les constantes ainsi que les axiomes vérifiés par ces données qui seront utilisées dans la seconde partie de la spécification. Cette dernière est une séquence de déclarations d'agents, comme dans une description CCS standard, qui permet de modéliser le comportement du système. Les paramètres des agents sont exprimés avec la syntaxe Z pour décrire d'une part les prédicats et les types qu'ils doivent vérifier et d'autre part les effets attendus.

L'intégration est ici plus facile à réaliser que dans les approches avec CSP, car la sémantique associée à CCS est opérationnelle. La principale difficulté des approches CSP avec Z ou Object-Z réside dans la diversité des sémantiques utilisées pour les deux langages à intégrer. CSP utilise en effet des modèles de la sémantique dénotationnelle.

Par sa définition complète de la sémantique, ce travail s'apparente à celui de

CSP-OZ ou Circus. Comme dans le cadre de ces approches, ZCCS devra être complété par des outils. L'autre défaut de cet exemple concerne la très grande richesse du langage.

CSP et Z [BDW99]. Cette approche propose une méthode de comparaison des langages CSP et Z, en identifiant un processus CSP avec un type de donnée abstrait de Z. Le principal problème des combinaisons de spécifications formelles réside en effet dans le lien entre les langages utilisés.

Bolton propose dans cet article de comparer les types de données abstraits Z avec les processus CSP, en définissant une sémantique du comportement pour les types abstraits de données. Les opérations d'un type de donnée sont ainsi représentées dans le modèle sémantique comme des événements de communication. L'approche propose également des relations de simulation sur les données qui sont valides et complètes par rapport aux relations de raffinement CSP.

La démarche adoptée ici est similaire à celle de Fischer dans la définition du langage CSP-OZ. La différence vient des structures comparées : Bolton considère les types de données abstraits de Z, tandis que Fischer utilise les classes Object-Z. Toutefois, Bolton ne définit pas un nouveau langage, mais propose une nouvelle sémantique pour représenter le comportement des types de données Z en fonction des processus CSP. De ce point de vue, cet exemple ressemble aux approches comme csp2B ou ZCCS qui utilisent la sémantique d'un langage pour compléter le modèle de l'autre.

5.8.2 Synthèse des approches présentées

Pour conclure ce chapitre, nous allons faire la synthèse de toutes les approches étudiées. Les tableaux 5.1 et 5.2 résument les caractéristiques des différentes approches présentées.

Tab. 5.1 – Comparaison des méthodes de combinaison - partie 1

	csp2B [But99]	CSP et B [ST02]	CSP-OZ [Fis00]	CSP et OZ [SD01]
Paragraphe	5.1	5.2	5.3	5.4
Langages utilisés	CSP B	CSP B	CSP Object-Z	CSP Object-Z
But	outil de CSP vers B	processus CSP = contrôleur des op. B	nouveau langage	processus CSP = classe
Sémantique	opér. LTS	dénot. échecs stables échecs-div.	opér. et dénot.	dénot. échecs-div.
Raffinement	raff. B	non	raff. Z raff. CSP	raff. CSP rel. simul.
Outils	csp2B	non	non	non
Vérification	non	non	non	preuve raff.

TAB. 5.2 – Comparaison des méthodes de combinaison - partie 2

	PLTL et B [Dar02]	Circus [WC02]	EB^3-B [FL03]
Paragraphe	5.5	5.6	5.7
Langages utilisés	PLTL B évén.	CSP Z	EB^3 B
But	prop. PLTL sur des syst. évén. B	unifier théorie	traces EB^3 sur des syst. en B
Sémantique	opér. LTS	dénot.	opér. ELTS
Raffinement	raff. B raff. LTS	raff. unifié	raff. B
Outils	non	de Z	Atelier B
Vérification	preuve + M.C.	M.C.	preuve par raff.

Quel niveau d'intégration ? Le premier constat concerne la diversité des intégrations des méthodes. La combinaison de spécifications formelles est en effet réalisée selon plusieurs niveaux d'intégration :

1. création d'un nouveau langage : ce niveau d'intégration est le plus fort car il implique les définitions d'une nouvelle syntaxe et d'une nouvelle sémantique. Le nouveau langage emprunte les opérateurs qui semblent les plus utiles et la sémantique est unifiée. Cette approche a pour avantage de bien intégrer certaines notions comme le raffinement, mais elle a pour défaut l'impossibilité de réutiliser des spécifications ou des outils existants. Les exemples de notre revue de littérature qui entrent dans cette catégorie sont CSP-OZ et Circus.
2. définition de l'ensemble ou d'une partie des constituants d'un langage dans la sémantique de l'autre : ce niveau permet de conserver les notations existantes, mais le langage dont la sémantique est redéfinie perd souvent de sa richesse. S'il est possible avec cette approche de traduire ou d'interpréter les spécifications d'un langage dans l'autre, le résultat obtenu perd parfois de l'information à cause des restrictions de la sémantique du langage cible. Par exemple, dans le cas de csp2B, les expressions CSP traduites en B ne peuvent pas représenter autant de propriétés dynamiques que dans CSP, par restriction du langage B. Ce niveau est toutefois efficace pour vérifier des propriétés d'un modèle sur l'autre. Les approches concernées par ce niveau d'intégration sont csp2B, PLTL et B événementiel et EB^3-B.
3. juxtaposition des langages : ce niveau qui permet de conserver plusieurs points de vue entre les langages est le niveau d'intégration le plus faible, puisque le seul lien entre les sémantiques est une identification d'une structure d'un langage avec la structure de l'autre. Toutes les méthodes et toutes les approches existant dans chacun des langages sont réutilisables, mais il est difficile d'analyser le modèle dans son ensemble. Une solution

consiste par exemple à définir des règles de cohérence sur les spécifications. CSP et Object-Z et CSP || B entrent dans ce niveau d'intégration.

De plus, cette généralisation n'est pas restrictive : une approche comme ZCCS fait à la fois partie de la création d'un nouveau langage et de l'intégration de CCS dans Z. On remarque enfin que chaque niveau a ses avantages et ses inconvénients. Le choix du niveau d'intégration dépend en fait des caractéristiques voulues sur la modélisation. La vérification de propriétés sur le système modélisé semble par exemple plus immédiate dans le niveau intermédiaire que dans le plus bas niveau.

La complémentarité des informations joue également un rôle important dans la combinaison de spécifications formelles. Si les deux langages possèdent une capacité de représentation équivalente, les informations spécifiées dans les deux parties du modèle seront redondantes : il y aura alors un risque d'incohérence. Si, au contraire, l'"interface" entre les deux approches est réduite au minimum, le modèle risque d'être mal analysé.

Raffinement et vérification. Une autre remarque intéressante concerne le raffinement. Seule l'approche Circus propose une relation de raffinement intégrée des actions et des données des spécifications. Les autres exemples proposent dans le meilleur des cas un raffinement de données cohérent avec les autres notions de raffinement ou bien ne définissent aucune nouvelle relation. Par conséquent, une intégration de haut niveau ne suffit pas pour définir un raffinement intégré.

Concernant la vérification de propriétés, un des intérêts de la combinaison de spécifications formelles est la grande expressivité cumulée des deux approches utilisées. Les exemples nous montrent qu'il est difficile de vérifier sur une approche une propriété exprimée dans l'autre. Si le langage est nouveau, comme dans le cas de Circus, les propriétés sont difficiles à exprimer, car la syntaxe est nouvelle, et la nouvelle sémantique n'est supportée par aucun outil existant.

Dans le cas d'une interprétation d'un langage sur la sémantique de l'autre, les propriétés perdent souvent de leur expressivité lors de la traduction. Les résultats semblent toutefois intéressants. L'association d'une traduction d'une expression EB^3 en B avec un raffinement B permet par exemple de vérifier certains types de propriétés EB^3 sur des machines B.

Enfin, dans le cas d'une simple juxtaposition, la vérification demande un effort d'interprétation puisque la sémantique de la correspondance est limitée.

Et les SI ? Concernant notre problème, l'utilisation d'une combinaison semble être un bon moyen de vérifier des propriétés sur le comportement des systèmes d'information. L'exemple de l'approche EB^3-B est particulièrement intéressant, car il permet de vérifier l'ordonnancement des opérations B. Il est toutefois incomplet car il ne permet pas de vérifier des propriétés plus générales de la dynamique. En outre, on peut se demander si une intégration plus poussée ne serait pas intéressante.

Concernant les autres approches, elles n'offrent pas les mêmes possibilités et les mêmes avantages que EB^3-B. La définition d'un nouveau langage ne convient pas, car les approches sont déjà très nombreuses dans les systèmes d'information et l'apprentissage d'une nouvelle syntaxe serait difficile à faire admettre auprès des concepteurs de SI. Les langages de processus comme CSP ou CCS ont une capacité d'expressivité moins puissante que le langage EB^3 et la notion de trace

s'adapte bien au type de propriétés dynamiques que l'on souhaite vérifier sur les SI.

Pour ces raisons, nous pensons qu'une extension de l'exemple EB^3-B est une bonne piste pour représenter de manière formelle le comportement dans les systèmes d'information.

Conclusion. En conclusion, les approches de combinaisons de spécifications formelles semblent être un moyen d'enrichir l'expressivité d'un langage, mais demandent un effort d'analyse des besoins pour définir les liens suffisants entre les deux parties de la spécification. Il sera en effet inutile de créer un nouveau langage uniquement pour représenter un système avec deux vues complémentaires. L'utilisation d'une traduction sémantique d'un langage sur un autre semble être l'approche la plus efficace pour vérifier des propriétés. Dans le cas de modélisation des systèmes d'information, une intégration de la forme EB^3-B semble a priori suffisante.

Chapitre 6

EB^4 : vers une combinaison des approches EB^3 et B

"N'oublions pas que tout a commencé avec une souris."

— Walt Disney

Les états de l'art sur les combinaisons de spécifications formelles et sur le raffinement nous ont servi à situer notre problème par rapport à l'existant. Nous présentons dans ce chapitre les pistes de recherche qui nous semblent intéressantes pour spécifier de manière formelle les systèmes d'information.

6.1 Motivations et rappel du problème

L'expérience [Lal02] montre que les méthodes de conception actuelles des systèmes d'information n'intègrent pas la modélisation du comportement de manière formelle. Notre objectif est de définir une méthode de spécification des systèmes d'information qui soit formelle, outillée et adaptée pour prendre en compte le comportement du système.

S'il existe des approches de spécification formelle dans le domaine des SI, les aspects dynamiques ne sont pas considérés au premier plan mais sont plutôt intégrés lors des phases ultérieures du développement (par exemple, avec des règles actives : voir chapitre 2).

Cette approche ne nous semble pas intéressante dans le cas de systèmes d'information dont les transactions sont considérées comme critiques (consultation de bases de données sur internet, transactions sur des comptes bancaires, consultation de données confidentielles, etc ...). L'intérêt des méthodes formelles est la possibilité de vérifier des propriétés sur le modèle. Si les aspects dynamiques ne sont pas pris en compte dès les premières étapes du développement, le système d'information risque de ne pas correspondre aux attentes du client.

En utilisant des méthodes de spécifications formelles, il est possible de faire des vérifications, de corriger au plus vite les erreurs du modèle et de le modifier afin de le rendre compatible avec les exigences du client. Le comportement des systèmes d'information a donc tout intérêt à être modélisé de manière formelle.

Toutefois, il est difficile avec les approches formelles de représenter à la fois les aspects statiques et dynamiques d'un système. D'une part, un langage basé

sur les états comme Z [Spi92], Object-Z [Smi00] ou B [Abr96], permet de bien caractériser les structures de données et les effets des opérations sur les états, ainsi que les propriétés d'invariance sur les transitions d'états. D'autre part, un langage basé sur les événements, comme CSP [Hoa85], CCS [Mil89] ou EB^3 [FSD03], met en avant les comportements possibles d'un système, comme les propriétés de vivacité ou d'ordonnancement.

La complémentarité des informations et des propriétés modélisées par ces deux types d'approches nous incite donc à développer une approche de combinaisons de spécifications formelles pour spécifier au mieux les systèmes d'information. Cependant, une telle solution rend la conception plus complexe, car l'intégration de deux approches peut être une source facile d'erreurs et de contradictions (voir chapitre 5).

6.2 Proposition

Nous introduisons une ébauche de solution qui nous semble particulièrement intéressante pour spécifier de manière formelle les systèmes d'information. Notre objectif est d'exploiter et d'intégrer les avantages des langages de spécification formels EB^3 [FSD03] et B [Abr96] dans une méthode de spécification formelle dédiée aux systèmes d'information appelée EB^4.

Frappier et Laleau montrent qu'une propriété dynamique EB^3 caractérisant un ordonnancement des opérations d'une machine B peut être vérifiée en utilisant la notion de raffinement B [FL03]. La spécification EB^3 est en effet traduite en B et l'ordonnancement est vérifié si on peut prouver que la machine B est un raffinement, au sens B, de cette traduction.

Pour intégrer ces travaux dans EB^4, d'autres points doivent maintenant être étudiés, comme la traduction d'une spécification EB^3 en une machine B ou bien la définition d'une notion de raffinement en EB^3.

6.2.1 Présentation de l'approche EB^4

Notre but est de pouvoir décrire de manière formelle un système d'information dans son ensemble. La méthode proposée reprend le concept du paradigme du parachute [Abr00] pour compléter l'approche de combinaison EB^3-B présentée dans [FL03]. Il semble en effet difficile de spécifier en une seule étape un système complet. Plus les propriétés seront nombreuses, plus le système sera complexe à concevoir. Plutôt que de spécifier par composition comme dans la méthode EB^3, notre solution est de "raffiner" progressivement le modèle, comme il est d'usage en B (voir chapitre 4 sur le raffinement). Dans ce but, nous définirons une notion de raffinement en EB^3.

Le système est dans un premier temps modélisé du point de vue des événements, en décrivant avec EB^3 les traces des entrées possibles.

Au niveau le plus abstrait, le modèle décrit le système d'information de manière à pouvoir le raffiner par étapes successives. Le système d'information est alors défini par des acteurs qui consultent et modifient des informations. L'intérêt d'une telle approche est de pouvoir se concentrer sur un petit nombre de propriétés à la fois.

Ensuite, par raffinements successifs, de nouvelles entités et de nouveaux événements sont introduits progressivement afin de spécifier tous les éléments

composant le système. Grâce à notre nouvelle notion de raffinement en EB^3, on assurera ainsi que le système est globalement cohérent et qu'il vérifie les propriétés désirées.

Cette première phase, réalisée en EB^3, concerne principalement la spécification des événements et de leur ordonnancement.

Lorsque tous les événements voulus ont été introduits, la seconde phase consiste à traduire la spécification obtenue en B. Cette étape permet de basculer dans une approche basée sur les états du système. Elle a principalement deux intérêts. D'une part, les contraintes d'intégrité s'expriment plus facilement en B et il nous sera donc possible de compléter les spécifications obtenues. D'autre part, nous souhaitons récupérer les travaux sur UML-B-SQL [Mam02] qui permettent de dériver une spécification semi-formelle en une implémentation SQL, en passant par une traduction en B. Une nouvelle série de raffinements permettra alors de transformer progressivement la spécification B en une implémentation.

Le processus de conception sera alors complet depuis la spécification des propriétés dynamiques, en passant par les propriétés d'invariance jusqu'à l'implémentation finale. La méthode EB^4 doit pour cela préciser, à chaque étape de raffinement, quel type d'information il faut spécifier.

6.2.2 Problèmes et conséquences

La spécification en EB^3 de systèmes complexes est actuellement difficile. Deux problèmes se posent. D'une part, il est difficile de tenir compte de nombreuses propriétés sur de nombreux événements du premier coup. Une solution consiste à modéliser le système progressivement en ajustant de manière presque empirique le modèle. Cette approche n'est toutefois pas acceptable pour une méthode formelle. L'autre problème concerne le rajout d'une entité supplémentaire sur une spécification existante. Il est alors difficile de prouver la cohérence du nouvel ensemble.

Le raffinement a déjà montré qu'il était une approche convenable pour ce type de problèmes [Abr00]. L'idée est d'introduire progressivement les événements importants tout en respectant les propriétés déjà vérifiées. Il existe en CSP [Hoa85] un raffinement basé sur les traces-divergences. Les processus sont raffinés par restriction des traces admissibles, des blocages et des divergences. Comme le langage EB^3 reprend de nombreux opérateurs de CSP pour décrire des expressions de processus sous forme de traces, la définition d'un raffinement en EB^3 est une piste réaliste. Dans ce cas, il sera possible de définir des règles de raffinement pour aider le concepteur dans le processus de raffinement EB^3. Notre objectif est de proposer dans le cadre d'EB^3 des schémas de raffinement, avec des obligations de preuve associées, afin de valider chaque étape de raffinement.

La traduction de EB^3 vers B pose un autre problème. Elle n'a été testée que sur quelques exemples. Pour concrétiser la méthode EB^4, il est important de définir la traduction en B de tous les opérateurs du langage EB^3 et de prouver l'équivalence des deux spécifications. Cette étape permettra d'automatiser la traduction de n'importe quelle expression EB^3 syntaxiquement correcte.

Un autre problème est lié aux propriétés qu'on veut vérifier. Selon le type de propriétés, il faut également déterminer à quel niveau d'abstraction et dans quelle phase (partie EB^3 ou partie B) elles doivent être spécifiées. La méthode doit donc préciser à chaque étape les éléments à spécifier et les propriétés à

vérifier : par exemple, contraintes d'intégrité en B, propriétés d'ordonnancement en EB^3.

Notre principal argument contre la création d'un nouveau langage comme Circus est la difficulté d'adaptation et de récupération des travaux existants. Les questions qui se posent sont les suivantes. Dans quelle mesure est-il possible de réutiliser des spécifications existantes avec notre approche ? Comment peut-on relier cette démarche avec des travaux existants (comme UML-B-SQL [Mam02]) ? La réutilisation de spécifications est en effet un problème courant et difficile dans les méthodes formelles.

Enfin, une perspective particulièrement intéressante de la méthode EB^4 sera, à plus long terme, la prise en compte de tous les composants d'un système d'information. Modéliser de manière indépendante l'interface, le système de gestion de base de données et les transactions ne permet pas d'obtenir un modèle globalement cohérent. Une approche plus efficace consiste à modéliser le système à un niveau plus abstrait, de manière globale, afin de le décomposer. Tous les composants du système d'information seront ainsi spécifiés dans un seul modèle qui sera ensuite décomposé. L'ensemble des composants obtenus sera alors globalement cohérent. Cette approche apportera d'ailleurs une solution au problème de la réutilisation présenté ci-dessus. De plus, les différents composants obtenus par cette approche pourront être traités avec des travaux existants (comme eb3web [NXD03] pour l'interface, EB^3-B [FL03] et UML-B-SQL [Mam02] pour la base de données).

6.3 Exemple : une bibliothèque

Nous venons de présenter les problèmes que nous pose notre proposition et les perspectives qui semblent en découler. Pour mieux comprendre l'intérêt et les difficultés de cette approche, nous présentons un exemple de spécification : une bibliothèque. On s'intéresse ici à la spécification initiale (au plus haut niveau) du système, c'est-à-dire en amont des exemples EB^3-B [FL01, FL03]. L'exemple reprend le cas d'étude présenté dans [FFL03].

On souhaite introduire les principes décrits ci-dessus, et plus particulièrement les paradigmes du parachute et de la décomposition dans la partie EB^3 de la conception. Les premières étapes consistent à spécifier avec EB^3 les comportements possibles du système. L'innovation de ce processus réside dans la restriction progressive des traces possibles afin d'éliminer les comportements inattendus. Les étapes de raffinement présentées ci-dessous ne sont pour l'instant qu'intuitives, mais notre but, à terme, est de définir formellement des règles de transformation qui permettront de valider chaque étape.

Spécification abstraite. Au niveau le plus abstrait, une bibliothèque est simplement considérée comme un système gérant des emprunts de livres par des membres. À ce stade, on observe le système à un très haut niveau, il n'est donc pas possible d'observer les interdépendances entre les entités. On observe uniquement des entrelacements d'emprunts de livres par des membres :

```
main =
   ||| mId, lId : MEMBRES × LIVRES :
       Emprunts(mId,lId)^*
```

Premier raffinement. Dans l'expression de processus précédente, on observe deux entités : les membres et les livres. Pour préparer la décomposition en types d'entité, la trace principale du processus `main` est dans un premier temps dupliquée et décomposée en deux :

```
main =
   (||| mId, lId : MEMBRES × LIVRES :
       Emprunts(mId,lId)^*)
||
   (||| mId, lId : MEMBRES × LIVRES :
       Emprunts(mId,lId)^*)
```

Cette première forme de raffinement consiste à décomposer une trace principale en deux sous-traces identiques composées entre elles.

Deuxième raffinement. Dans cette étape, on souhaite détailler le comportement attendu de chaque entité. Dans ce but, on définit dans un premier temps les comportements isolés des entités.

Pour un membre donné, le scénario est le suivant : il s'inscrit à la bibliothèque, puis il fait des emprunts et, enfin, il peut résilier son abonnement :

```
inscrire(mId) . Emprunts(mId,lId)^* . resilier(mId)
```

Pour un livre, il est d'abord acheté par la bibliothèque, puis il est emprunté et il est enfin déréférencé :

```
acheter(lId) . Emprunts(mId,lId)^* . retirer(lId)
```

Par conséquent, le processus `main` est raffiné par :

```
main =
   (||| mId, lId : MEMBRES × LIVRES :
        inscrire(mId) . Emprunts(mId,lId)^* . resilier(mId))
||
   (||| mId, lId : MEMBRES × LIVRES :
       acheter(lId) . Emprunts(mId,lId)^* . retirer(lId))
```

On remarque dans ce cas une deuxième forme de raffinement possible qui consiste à rajouter de nouveaux événements concrets dans une trace.

Troisième raffinement. Les étapes précédentes ont permis d'amorcer la décomposition de la spécification abstraite. Les traces correspondant au comportement de chaque type d'entité sont extraites de l'expression de processus `main` et les types d'entité `Membre` et `Livre` sont maintenant définis séparément.

```
main =
   (||| mId : MEMBRES : Membre(mId)^*)
||
   (||| lId : LIVRES : Livre(lId)^*)
```

avec :

```
Membre(mId : MEMBRES) =
   inscrire(mId) .
   (
      (||| lId : LIVRES : Emprunts(mId,lId)^*)
   ||
      RequetesSurMembre(mId,trace)^*
   ).
   resilier(mId)


Livre(lId : LIVRES) =
   acheter(lId) .
   (
      (||| mId : MEMBRES : Emprunts(mId,lId)^*)
   ||
      RequetesSurLivre(lId,trace)^*
   ).
   retirer(lId)
```

Dans les définitions des types d'entité `Membre` et `Livre`, on remarque d'une part que les événements qui ne dépendaient pas du paramètre de quantification sont extraits des expressions d'entrelacement (|||). Dans les deux cas, seuls les événements `Emprunts(mId,lId)` sont entrelacés. D'autre part, des actions de requêtes sont rajoutées aux expressions de processus. Ces nouveaux événements, qui sont composés en parallèle avec les entrelacements, peuvent ainsi être appelés entre chaque emprunt. Il n'est pas nécessaire à cette étape de spécifier les fonctions récursives définissant les requêtes. Ces événements sont considérés ici comme des boîtes noires.

Cette étape introduit deux notions de raffinement intéressantes. La séparation et l'extraction des traces concernant les membres d'une part et les livres d'autre part complètent le processus de décomposition commencé aux étapes précédentes. La décomposition permet ainsi d'introduire progressivement les entités du système. Le rajout de `RequetesSurMembre` et de `RequetesSurLivre` à l'aide d'une composition parallèle est une autre forme de raffinement d'ajout de nouveaux événements.

Quatrième raffinement. L'étape suivante consiste à définir les comportements possibles pendant les emprunts. Il n'est pas toujours possible d'emprunter un livre, on peut alors le réserver. Un premier raffinement possible est de décomposer l'événement `Emprunts` en deux actions principales : `emprunter` et `reserver`.

```
Emprunts(mId,lId)^* = emprunter(mId,lId)^* ||| reserver(mId,lId)^*
```

Dans ce cas, le résultat de cette étape de raffinement est :

```
Membre(mId : MEMBRES) =
   inscrire(mId) .
   (
      (||| lId : LIVRES : emprunter(mId,lId)^* ||| reserver(mId,lId)^*)
   ||
      RequetesSurMembre(mId,trace)^*
   ).
   resilier(mId)


Livre(lId : LIVRES) =
   acheter(lId) .
   (
      (||| mId : MEMBRES : emprunter(mId,lId)^* ||| reserver(mId,lId)^*)
   ||
      RequetesSurLivre(lId,trace)^*
   ).
   retirer(lId)
```

Cinquième raffinement. Dans cette étape, les scénarios d'emprunt et de réservation sont détaillés. Le raffinement consiste à introduire de nouveaux événements autour des actions principales `reserver` et `emprunter` afin de spécifier les conditions d'emprunt et de réservation.

Par exemple, un emprunt n'est possible que si le livre est libre. Il est ensuite possible de le renouveler avant de le rendre :

```
(1) =
(estlibre(lId, trace) => emprunter(mId,lId)
                         .renouveler(mId,lId)^*
                         .rendre(mId,lId)
)^*
```

Une réservation est faite lorsque le livre demandé n'est pas libre. Il existe ensuite deux alternatives : ou bien la réservation est annulée, ou bien elle est consommée. Dans ce cas, le livre ne peut être pris que s'il est libre et si le membre est le premier de la liste d'attente. Il peut ensuite être renouvelé avant d'être rendu :

```
(2) =
(nonlibre(lId, trace)
    => reserver(mId,lId).
        (
           annuler(mId,lId)
        |
           estlibre(lId,trace) & estpremier(mId,lId,trace)
```

```
              => prendre(mId,lId).renouveler(mId,lId)^*
                  .rendre(mId,lId)
        )
)^*
```

Pendant cette étape de raffinement, chaque occurrence de l'expression de processus

```
emprunter(mId,lId)^* ||| reserver(mId,lId)^*
```

est remplacée par l'entrelacement des expressions `(1)` et `(2)`. On remarque une nouvelle forme de raffinement, avec la définition de garde sur les événements pour en restreindre l'exécution.

Sixième raffinement. Les étapes précédentes nous ont permis d'introduire les événements composant les associations `Emprunt` et `Reservation`. Ce nouveau pas dans le processus de spécification consiste à factoriser et réécrire les expressions de processus obtenues afin de pouvoir les décomposer en deux associations.

On remarque en effet que les traces d'emprunts et de réservations se superposent. Afin de décomposer les événements en deux groupes, on définit maintenant les emprunts et les réservations en parallèle dans les types d'entité. Il est également possible d'exprimer des contraintes supplémentaires. Par exemple, l'entrelacement est remplacé par un choix dans `Livre` pour indiquer qu'un livre ne peut être emprunté que par un seul membre à la fois.

```
Membre(mId : MEMBRES) =
   inscrire(mId) .
   (
      (||| lId : LIVRES : Emprunt(mId,lId)^*)
   ||
      (||| lId : LIVRES : Reservation(mId,lId)^*)
   ||
      RequetesSurMembre(mId,trace)^*
   ).
   resilier(mId)


Livre(lId : LIVRES) =
   acheter(lId) .
   (
      (| mId : MEMBRES : Emprunt(mId,lId)^*)
   ||
      (||| mId : MEMBRES : Reservation(mId,lId)^*)
   ||
      RequetesSurLivre(lId,trace)^*
   ).
   retirer(lId)
```

avec :

```
Emprunt(mId : MEMBRES,lId : LIVRES) =
   (
      estlibre(lId, trace) => emprunter(mId,lId)
   |
      estpremier(mId,lId,trace) => prendre(mId,lId)
   )
   .renouveler(mId,lId)^*
   .rendre(mId,lId)


Reservation(mId : MEMBRES,lId : LIVRES) =
   (nonlibre(lId, trace) => reserver(mId,lId).
   (
      annuler(mId,lId)
   |
      estpremier(mId,lId,trace) => prendre(mId,lId)
   )
```

Septième raffinement. Nous allons maintenant définir les requêtes sur les types d'entité. Pour un membre donné, la fonction récursive `NbrePrets` calcule le nombre de prêts en cours. L'emprunteur d'un livre est déterminé par la fonction `Emprunteur` :

```
Membre(mId : MEMBRES) =
   inscrire(mId) .
   (
      (||| lId : LIVRES : Emprunt(mId,lId)^*)
   ||
      (||| lId : LIVRES : Reservation(mId,lId)^*)
   ||
      NbrePrets(mId,trace)^*
   ).
   resilier(mId)


Livre(lId : LIVRES) =
   acheter(lId) .
   (
      (| mId : MEMBRES : Emprunt(mId,lId)^*)
   ||
      (||| mId : MEMBRES : Reservation(mId,lId)^*)
   ||
      Emprunteur(lId,trace)^*
   ).
   retirer(lId)
```

avec :

```
Nbreprets(mId : MEMBRES, trace : TRACE-VALIDE) : NAT =
```

```
   match last(trace) with
      nil -> 0
      emprunter(mId,_) -> Nbreprets(front(trace),mId) + 1
      prendre(mId,_) -> Nbreprets(front(trace),mId) + 1
      rendre(mId,_) -> Nbreprets(front(trace),mId) − 1
      _ -> Nbreprets(front(trace),mId)


Emprunteur(lId : LIVRES, trace : TRACE-VALIDE) : LIST of MEMBRES =
   match last(trace) with
      nil -> [ ]
      emprunter(mId,lId) -> [ mId ]
      prendre(mId,lId) -> [ mId ]
      rendre(_,lId) -> [ ]
      _ -> Emprunteur(front(trace),lId)
```

Derniers raffinements. Les raffinements suivants consistent à définir toutes les fonctions récursives sur la trace courante du système qui ont été jusqu'à présent considérées comme des boîtes noires. Par raffinement, on entend ici ajout de détails ou implémentation des fonctions.

Pour vérifier si un membre est le premier dans la liste de réservation, il faut introduire une nouvelle fonction récursive, `listereservation`, qui sera implémentée plus tard.

```
estpremier(mId : MEMBRES, lId : LIVRES, trace : TRACE-VALIDE) : BOOLEAN =
   match first(listereservation(lId,trace)) with
      mId -> true
      _ -> false


estlibre(mId : MEMBRES, lId : LIVRES, trace : TRACE-VALIDE) : BOOLEAN =
   (listereservation(lId,trace) = [ ])


nonlibre(mId : MEMBRES, lId : LIVRES, trace : TRACE-VALIDE) : BOOLEAN =
      (Emprunteur(lId,trace) /= [ ])
      and
      (Emprunteur(lId,trace) /= [ mId ])
   or
      (listereservation(lId,trace) /= [ ])
```

Il ne reste plus qu'à spécifier la fonction `listereservation` :

```
listereservation(lId : LIVRES, trace : TRACE-VALIDE) : LIST of MEMBRES =
   match last(trace) with
      nil -> [ ]
      reserver(mId,lId) -> mId :: listereservation(front(trace), lId)
      annuler(mId,lId) -> listereservation(front(trace),lId) − {mId}
      prendre(mId,lId) -> listereservation(front(trace),lId) − {mId}
      _ -> listereservation(front(trace),lId)
```

Traduction en B et implémentation. Lorsque toutes les entités et toutes les associations du système d'information sont définies en EB^3, il est possible de traduire le résultat obtenu en B (voir [FL03]). La démarche est ici différente de celle présentée dans [FL01], car la machine B n'est pas encore connue. Le but est de raffiner en B la machine obtenue par traduction pour raffiner les données jusqu'à l'implémentation finale.

Cette étape nécessite d'une part de déterminer à quel moment il est souhaitable de traduire en B et d'autre part de définir un lien entre cette dérivation et les approches existantes. Grâce aux outils de B et les travaux sur UML-B-SQL [Mam02], le système pourra alors être dérivé jusqu'à une implémentation finale.

Nous souhaitons aussi introduire au cours de cette étape la définition de nouvelles contraintes d'intégrité en B. Il n'est en effet pas possible d'utiliser le raffinement B pour rajouter par la suite de telles contraintes.

Concernant le niveau de précision en EB^3, on peut encore raffiner la spécification présentée ci-dessus. Par exemple, les traces d'un livre se terminent par `retirer`. Il est possible de raffiner encore le modèle en décomposant cet événement par les raisons possibles de retrait. Par exemple, `retirer` peut être défini par :

```
retirer(lId : LIVRES) =
      estbrule(lId, trace) => bruler(lId)
   |
      estvendu(lId, trace) => vendre(lId)
   |
      estvole(lId, trace) => voler(lId)
```

Conclusion. Pour mettre en œuvre cette méthode, il nous faut surtout définir des règles de raffinement formelles en EB^3. Intuitivement, on considère au niveau le plus abstrait un ensemble de traces admissibles très large et le but est de restreindre progressivement cet ensemble par étapes successives.

Quatre types de raffinement semblent plus particulièrement intéressants :
- ajout d'événements,
- décomposition d'un événement,
- décomposition d'une trace,
- ajout de gardes sur les événements.

Dans les approches de raffinement (voir chapitre 4), la décomposition d'un événement en une expression de processus n'a jamais été utilisée. Il s'agit donc à notre connaissance d'un point nouveau dans le domaine du raffinement.

Enfin, il faudra également analyser les conséquences sur les opérateurs du langage EB^3, afin de définir des obligations de preuve de correction du raffinement.

6.4 Points à étudier

À long terme, notre objectif est de définir la méthode EB^4 telle que présentée dans la section 6.2. Pour y parvenir, nous devons élucider au moins trois points :

1. le degré de combinaison entre EB^3 et B,
2. la définition du raffinement en EB^3,

3. la mise en œuvre de la méthode.

6.4.1 Combinaison EB^3-B

L'approche présentée dans [FL03] indique comment traduire une expression de processus EB^3 en une machine B (voir section 5.7.3). La technique de traduction demande toutefois quelques études supplémentaires.

Une première difficulté concerne l'analyse des états-transitions du système pour déterminer les conditions des opérations B. L'approche préconise l'emploi de systèmes de transitions étendus (ELTS) pour éviter d'une part les problèmes d'explosion des états liés aux quantifications des opérateurs et d'autre part l'analyse de systèmes trop larges. Cette solution n'a été testée jusqu'à présent que sur des exemples simples. L'analyse des systèmes de transitions et la généralisation de la traduction EB^3-B demandent quelques études supplémentaires.

Une autre difficulté concerne l'implémentation de l'ensemble des traces valides du système : $\tau(main)$ (voir section 5.7). La solution adoptée par [FL03] est la définition d'une machine annexe qui spécifie les traces valides. Toutefois, la manière de spécifier ce type d'information en B n'est pas précisée. Comme le langage B est basé sur les transitions d'état, la meilleure approche semble être la définition de règles d'inférence sur les expressions de processus. Cette solution demande encore quelques études et analyses.

Concernant le degré d'intégration entre EB^3 et B, nous pensons que la traduction d'une spécification EB^3 "aboutie" en une machine B suffit. Par "spécification aboutie", nous entendons une spécification qui est correcte et qui ne sera plus modifiée. Dans le cadre de notre proposition, cela signifie que toutes les propriétés que l'on souhaitait spécifier l'ont été au maximum de leurs possibilités avec le langage EB^3. Une conséquence importante de ce choix d'intégration est l'obligation de revenir sur la partie EB^3 et de refaire la traduction dans le cas où des erreurs d'ordonnancement seraient remarquées pendant la phase de travail sur la partie B. C'est la raison pour laquelle, la méthode EB^4 doit préciser à chaque étape quels types de propriétés doivent être spécifiés. Nous discuterons de ces problèmes dans la section 6.4.3. Le degré d'intégration dépend aussi de la définition que nous donnerons du raffinement EB^3. Il est toutefois possible que selon les résultats de nos études concernant ces autres problèmes, la définition de la traduction soit modifiée ou adaptée en conséquence. Pour éviter de rencontrer des problèmes de sémantique ou de création de nouveau langage (voir chapitre 5), nous souhaitons dans tous les cas nous en tenir à une traduction d'EB^3 vers B.

6.4.2 Raffinement EB^3

Nous avons constaté que la définition du raffinement était assez souple et dépendait en fait des caractéristiques de la méthode formelle associée. Dans notre cas, le raffinement EB^3 devra répondre à deux critères :

– permettre la conception de systèmes d'information,
– et faciliter le passage d'EB^3 à B.

Notre but est, d'une part, de fournir une définition formelle du raffinement en EB^3 et, d'autre part, d'aider le spécificateur à concevoir des systèmes d'information. Dans le petit exemple intuitif que nous avons présenté à la section 6.3,

nous avons déterminé quatre formes principales de dérivation que notre notion de raffinement doit supporter :

- ajout d'événements,
- décomposition d'un événement,
- décomposition d'une trace,
- ajout de gardes sur les événements.

Nous aurons besoin dans un premier temps d'étudier d'autres exemples de spécification de systèmes d'information pour compléter ou modifier la liste de caractéristiques du raffinement indiquée ci-dessus. Ce travail servira également à concevoir l'approche globale de la méthode EB^4.

Comme la sémantique du langage EB^3 est fondée sur des règles d'inférence sur les opérateurs, deux approches sont possibles pour définir le raffinement. D'un côté, on peut mettre en avant les traces et les échecs stables des systèmes modélisés, comme en CSP. Dans ce cas, on pourra s'inspirer des travaux de Hoare [Hoa85] et Roscoe [Ros97] pour définir le raffinement EB^3 à partir du modèle des échecs stables. D'un autre côté, la sémantique du langage ne s'y prête pas bien, puisqu'elle est opérationnelle et non dénotationnelle comme en CSP. L'alternative est une définition du raffinement sur la base des systèmes de transitions, comme les relations de Josephs [Jos88] ou le raffinement des LTS [BJK00]. Ce choix est délicat, car il aura des conséquences importantes sur les autres problèmes, en particulier concernant la traduction.

Une méthode formelle dotée d'une relation de raffinement met en jeu plusieurs niveaux de spécification et le langage doit être suffisamment riche pour supporter les différents niveaux d'abstraction. Dans le cas de la méthode B [Abr96], le niveau le plus abstrait est spécifié à l'aide d'un langage de substitutions généralisées. À ce stade de la conception, le langage B ne permet pas, par exemple, la spécification de boucles ou de séquences d'instructions. Tous ces concepts, proches du code, sont introduits dans les étapes de raffinement sucessives. Le dernier niveau de raffinement, appelé implémentation, est spécifié avec le langage B_0, qui est directement traduisible en du code.

Pour définir le raffinement en EB^3, nous devrons probablement considérer le langage à un niveau plus abstrait. Ce problème est très dépendant de celui de la traduction en B : nous devons pouvoir déterminer à quelle étape de raffinement la spécification EB^3 sera "aboutie" et prête à être traduite en B. Par analogie avec la méthode B, notre raffinement devra permettre de dériver une spécification EB^3 jusqu'à une "implémentation EB$^3_0$" directement traduisible en B. Une implémentation EB$^3_0$ correspond en fait à une spécification EB^3 classique, comme présentée dans le chapitre 2, mais orientée et définie de manière à faciliter le passage d'EB^3 vers B.

Outre la définition d'une relation de raffinement en EB^3, nous étudierons aussi la mise en œuvre de "schémas de dérivation" qui indiqueront comment et sous quelles conditions tranformer les spécifications. Ces schémas permettront ainsi d'orienter le raffinement dans le cadre précis de la conception des systèmes d'information avec EB^4.

6.4.3 La méthode EB^4

Une fois que les problèmes de la traduction et du raffinement seront résolus, nous devrons définir de manière précise les différentes étapes de la méthode EB^4.

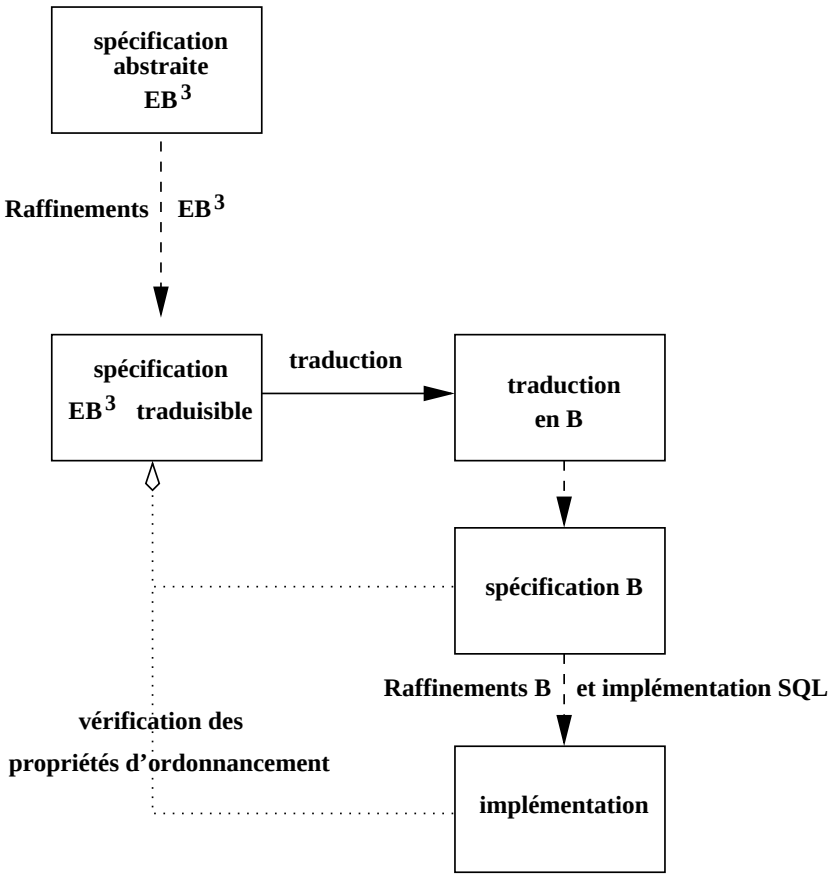


FIG. 6.1 – Résumé de l'approche EB^4

La figure 6.1 est un résumé de l'approche EB^4. Dans un premier temps, le système d'information est spécifié avec EB^3. Cette première étape permet de définir les scénarios et les entités du système d'information à l'aide des concepts de base du langage : les expressions de processus et les fonctions récursives. Grâce à la relation de raffinement EB^3, le système est construit progressivement jusqu'à l'obtention d'une spécification traduisible en B. Les problèmes concernant cette première étape ont été abordés dans les sections précédentes.

Concernant la partie B, deux problèmes se posent. D'une part, le premier pas de raffinement en B n'est pas aussi évident que dans [FL03]. Dans l'approche EB^3-B, la machine B était connue, alors que dans notre cas, elle ne l'est pas. Le passage de la machine B obtenue par traduction à une machine B spécifiée de manière plus lisible et plus proche des standards de la méthode UML-B demande une étude plus approfondie. Un des avantages est la possibilité de poursuivre le raffinement B jusqu'à l'obtention de code SQL, comme dans l'approche UML-B-SQL [Mam02].

D'autre part, si la traduction et le raffinement B nous assurent que les spécifications obtenues vérifient bien les propriétés définies dans la partie EB^3, il est souhaitable de pouvoir spécifier et vérifier avec B des propriétés d'intégrité sur le modèle. Ce type de contraintes s'exprime en effet plus facilement avec des propriétés d'invariance qu'avec des traces. Toutefois, l'ajout de nouvelles propriétés en B pourrait contredire le comportement déjà spécifié. Ce point reste à étudier.

Globalement, nous devrons aussi assurer que les deux étapes sont cohérentes, dans le sens que le travail réalisé dans la partie EB^3 ne soit pas contredit dans la partie B. Dans ce but, nous devrons prouver que les raffinements B et EB^3 sont compatibles et que la phase de traduction est correcte.

En conclusion, la solution proposée permettra de spécifier formellement des systèmes d'information en tenant compte à la fois des aspects statiques et dynamiques. Par son approche combinée, la méthode EB^4 sera une alternative à l'actuelle méthode EB^3 basée sur l'interprétation des expressions de processus. Il sera possible d'une part de dériver par raffinement les spécifications en EB^3 pour modéliser des systèmes complexes et d'autre part d'utiliser les mécanismes de la méthode B pour prouver des propriétés supplémentaires.

Chapitre 7

Conclusion et perspectives

> "C'est un exploit difficile, voire impossible, de changer les pneus d'un véhicule en mouvement."
>
> — Kolodzei

Notre objectif est de modéliser de manière formelle le comportement dans les systèmes d'information. Dans ce but, nous avons réalisé une étude des approches existant dans la littérature.

Les méthodes de conception actuelles proposent essentiellement des superpositions plutôt complexes de vues plus ou moins complémentaires des systèmes d'information (modèles des données, des flux de données, des transactions, etc ...) avec pour objectif d'obtenir à la fin une spécification cohérente. Pour assurer la cohérence de toutes ces vues complémentaires, les méthodes reposent essentiellement sur les capacités d'analyse et de synthèse du concepteur, puisque ces méthodes qui ne sont pas formelles n'autorisent aucune vérification.

En outre, les quelques approches formelles qui sont proposées dans la littérature pour spécifier les SI ne représentent pas bien les aspects à la fois statiques et dynamiques des systèmes. Une solution possible pour intégrer ces deux aspects semble donc être la combinaison de spécifications formelles.

7.1 Combinaisons de spécifications formelles.

Un langage basé sur les états comme Z, Object-Z ou B, permet d'une part de bien représenter les structures de données et les effets des opérations sur les états, et d'autre part de spécifier des propriétés d'invariance sur les états. Un langage basé sur les événements, comme CSP, CCS ou EB^3, met plutôt en avant le comportement attendu d'un système, comme les propriétés de vivacité ou les contraintes d'ordonnancement.

La complémentarité des informations et des propriétés modélisées par ces deux types d'approches nous a incité à étudier et à analyser les approches de combinaisons de spécifications formelles dans la littérature. À travers un exemple de référence plutôt simple mais qui finalement utilise les principaux types d'opérations d'un SI, nous avons comparé et analysé sept approches : l'outil csp2B, les systèmes combinés CSP || B, le nouveau langage CSP-OZ, la combinaison CSP avec Object-Z, la vérification de propriétés PLTL sur des systèmes d'événements B, le langage Circus et l'approche EB^3-B.

Si les combinaisons de spécifications formelles rendent la représentation des systèmes plus complète, elles la rendent également plus difficile à réaliser. La multiplication des représentations peut être une source de contradictions entre les différentes parties de la spécification. Les exemples de CSP || B et de CSP avec Object-Z nous montrent d'ailleurs que, même avec des méthodes formelles, le problème de la cohérence est difficile à résoudre. La redondance des informations est un autre problème qui est lié au précédent.

Le niveau d'intégration des approches est un problème délicat. La définition d'un nouveau langage inspiré de langages existants implique la définition d'une nouvelle syntaxe, d'une nouvelle sémantique, de nouveaux outils et d'une nouvelle méthodologie de travail pour le concepteur. D'un autre côté, la simple juxtaposition de deux langages rend l'analyse et la vérification du modèle global plus complexe. Le niveau intermédiaire consiste à interpréter un langage L_1 dans la sémantique d'un autre langage L_2. Il a pour avantage d'enrichir un des langages, mais il peut aussi faire perdre de l'information, car il n'est pas toujours possible d'exprimer tous les concepts de L_1 dans la sémantique de L_2.

En conclusion, la combinaison de spécifications formelles est une solution à double tranchant : elle peut apporter une solution pour compléter la modélisation, mais elle peut également apporter d'autres problèmes comme des inconsistances dans la spécification.

7.2 Raffinement

Une propriété importante des méthodes de spécification formelle est l'existence d'une relation de raffinement.

Pour concevoir des systèmes complexes, il est très utile de pouvoir rajouter des détails tout en s'assurant que la cohérence est maintenue. Le raffinement est un atout supplémentaire dans les méthodes formelles qui en sont dotées.

Parmi les relations existantes, on peut distinguer principalement quatre formes de raffinement. Le raffinement *wp* permet d'agir sur les pré- et post-conditions des opérations. Le raffinement de séquences d'opérations permet de considérer des pseudo-programmes ou des boîtes noires dont on observe les entrées et sorties. Les spécifications sont alors regroupées dans des schémas ou des machines abstraites. Une troisième forme de raffinement concerne les LTS. Enfin, la dernière relation de raffinement étudiée concerne les modèles des algèbres de processus avec les traces, échecs et divergences.

La diversité des relations de raffinement est le reflet des nombreuses caractéristiques des méthodes formelles. Suivant le type de système spécifié, le raffinement s'adapte pour permettre aux concepteurs de réaliser leur projet. Un raffinement dans Circus qui est plus adapté aux systèmes distribués permet d'introduire du parallélisme, tandis qu'un raffinement en B événementiel autorise le rajout d'événements.

En conclusion, le raffinement dépend des caractéristiques et des objectifs des méthodes formelles.

7.3 Perspectives : de EB^3-B vers EB^4 ?

Dans le cadre des SI, l'approche EB^3-B nous semble la mieux adaptée pour modéliser le comportement dans les systèmes d'information. Elle permet de vérifier une propriété exprimée par une trace EB^3 sur un système modélisé par une machine abstraite B.

7.3.1 But

Notre objectif est d'utiliser ces deux méthodes de spécification formelles de manière plus intégrée afin de développer des systèmes d'information en considérant dès le travail d'analyse les problèmes liés à la modélisation des propriétés dynamiques.

La spécification formelle a l'avantage de permettre au spécifieur de constater une erreur au plus tôt notamment grâce aux vérifications mathématiques sur le modèle. Cette interaction permet de corriger rapidement les erreurs et éviter ainsi un travail de programmation inutile.

De manière analogue, une interaction entre le modèle statique représenté en B et le modèle dynamique représenté en EB^3 peut compléter par étapes successives la modélisation formelle d'un système d'information qui soit à la fois cohérente et correcte.

Alors que les propriétés dynamiques d'un système d'information ne sont généralement considérées qu'une fois que les structures du système ont été déjà définies, l'intégration d'EB^3-B en une méthode de spécification baptisée EB^4 permettra d'interagir entre les modèles basés sur les états et sur les événements et de concevoir ainsi le système d'information avec l'aide de deux vues complémentaires.

7.3.2 Perspectives

Avec la méthode EB^4, le système d'information sera spécifié selon deux vues orthogonales, EB^3 et B. Les problèmes liés aux combinaisons de spécifications formelles (chapitre 5) nous incitent à éviter de créer un nouveau langage EB^4 qui serait fondé sur EB^3 et B.

Dans un premier temps, une nouvelle relation de raffinement définie pour EB^3 permettra de dériver progressivement la spécification afin de satisfaire toutes les propriétés voulues. La spécification sous forme de traces du langage EB^3 est assez intuitive pour faciliter les communications entre les concepteurs et le client. Comme la notion de raffinement sera spécifiquement dédiée à la conception des systèmes d'information, les techniques de vérification seront limitées à quelques "schémas" de dérivation qui seront associés à des obligations de preuve génériques. L'objectif est de valider chaque étape de raffinement avec des preuves ou des tests simples.

Une fois que le travail sur la partie EB^3 sera terminé, la méthode consistera à traduire l'expression de processus EB^3 en une machine B. Par traduction et par raffinement B, les spécifications obtenues vérifient les propriétés d'ordonnancement des opérations décrites dans la spécification EB^3 [FL03]. Cela suppose en particulier que le raffinement de la partie EB^3 soit orienté de manière à ce que la dernier niveau de spécification soit directement traduisible en B.

La suite du travail consistera à poursuivre l'effort de spécification en utilisant le raffinement B jusqu'à l'obtention de code SQL, comme dans l'approche UML-B-SQL [Mam02]. L'autre intérêt de l'utilisation de B sera la possiblité de vérifier des propriétés d'intégrité sur le modèle.

En conclusion, les perspectives concernant EB^4 semblent nombreuses. L'exploitation de la souplesse du langage EB^3 associée à l'utilisation des outils de la méthode B permettra de développer une méthode de conception des SI qui tiendra compte de la modélisation du comportement. La partie B permettra de décrire les aspects statiques du système, tandis qu'EB^3 servira à modéliser les aspects dynamiques. L'utilisation des méthodes formelles permettra, enfin, de spécifier progressivement le système d'information par des vérification successives, jusqu'à l'obtention d'un modèle cohérent qui vérifiera les propriétés souhaitées.

Annexe A

Sommaire des références bibliographiques

"Les paroles s'envolent, les écrits restent."

— Anonyme, proverbe latin

A.1 Livres, articles et manuels de référence sur les langages

A.1.1 Langages basés sur les états

La Méthode B :

- J.R. Abrial : *The B-Book : Assigning programs to meanings*, Cambridge University Press, 1996 [Abr96]
  Le "B-Book" est **le** livre de référence du langage B. Il introduit tous les fondements mathématiques et la syntaxe du langage, ainsi que les obligations de preuve qui rendent les spécifications B cohérentes. La notion de raffinement est également traitée. Pour tous les problèmes de raisonnement concernant la sémantique et les preuves en B, ce livre a été notre principale source de référence, comme il l'est d'ailleurs pour l'ensemble des membres de la communauté B.
- H. Habrias : *Spécification formelle avec B*, Lavoisier, Hermes Sciences, 2001 [Hab01]
  Ce livre présente la méthode B ainsi que de nombreux exemples d'application. Il nous a essentiellement servi à traiter des exemples.
- J.R. Abrial, L. Mussat : Introducing dynamic constraints in B, *In Proceedings of the Second Conference on the B Method*, LNCS, volume 1393, Springer-Verlag, 1998 [AM98]
  Dans cet article, de nouvelles notions ont été définies en B afin de considérer les événements et non plus les opérations d'un système. Il est donc à l'origine du B "événementiel". Ce document nous a servi à comprendre les systèmes d'événements B.
- J.R. Abrial : Guidelines to Formal Systems Studies, ClearSy, November 2000 [Abr00]

Cette référence est un manuel pour aider les utilisateurs de B, et plus généralement les concepteurs, à spécifier des systèmes complexes avec le paradigme du parachute. Nous nous sommes inspirés de cette approche pour élaborer notre sujet de thèse : la méthode EB^4 s'appuiera en effet sur le paradigme du parachute pour concevoir des systèmes d'information.

Le langage Z :

- J.M. Spivey : *The Z Notation : a Reference Manual*, Prentice-Hall, 1992 [Spi92]
  Ce manuel introduit la syntaxe du langage Z et est illustré par de nombreux exemples. Les notions d'obligations de preuve et de raffinement de données sont également présentées. Il nous a servi d'introduction au langage Z.
- J. Davies, J.C.P. Woodcock : *Using Z : Specification, Refinement, and Proof*, Prentice-Hall, 1996 [DW96]
  Ce livre, bien que plus récent, est assez proche du manuel de Spivey [Spi92]. Il contient de nombreux exemples. Il spécifie aussi les notions de raffinement et de relations de simulation en Z. Ce livre nous a servi de référence pour les problèmes de raffinement en Z et Object-Z.

Object-Z :

- G. Smith : *The Object-Z Specification Language*, Kluwer Academic Publishers, 2000 [Smi00]
  La syntaxe et la sémantique du langage Object-Z sont présentées dans ce livre. Object-Z est une version orientée objet de Z et est utilisé dans deux exemples de combinaisons de notre état de l'art. Ce livre nous a servi de référence concernant la syntaxe d'Object-Z pendant l'élaboration de l'exemple de comparaison.

VDM :

- C.B. Jones : *Systematic Software Development using VDM*, Prentice Hall, 1990 [Jon90]
  Ce livre est le manuel de référence du langage VDM.

Statecharts :

- D. Harel : Statecharts : A Visual Formalism for Complex Systems, *Science of Computer Programming*, volume 8, 1987 [Har87]
  Cet article de journal présente le formalisme Statecharts.

Esterel :

- G. Berry, G. Gonthier : The Esterel Synchronous Programming Language : Design, Semantics, Implementation, *Science of Computer Programming*, volume 19, numero 2, 1992 [BG92]
  Cet article introduit le langage Esterel.

A.1.2 Langages basés sur les événements

EB^3 :

- M. Frappier, R. St-Denis : EB^3 : an entity-based black-box specification method for information systems, *Software and Systems Modeling*, volume 2, numéro 2, pages 134-149, Juillet 2003 [FSD03]
  Frappier et Saint-Denis introduisent ici les concepts du langage de spécification formel EB^3 dédié à la spécification des systèmes d'information. EB^3 et B sont les deux langages que nous souhaitons utiliser pour spécifier des propriétés sur des systèmes d'information. Cet article est donc notre principale référence sur le langage EB^3.
- M. Frappier, R. St-Denis : Specifying Information Systems through Structured Input-Output Traces, Département de mathématiques et d'informatique, Université de Sherbrooke, Québec, Canada, 1998 [FSD98]
  Ce document était l'article de référence sur le langage EB^3 avant la publication de l'article [FSD03].
- Guillaume Nguyen-Xuan-Dang : *Génération automatique de sites WEB pour des systèmes d'information*, Mémoire de maîtrise en génie logiciel, Université de Sherbrooke, 2003 [NXD03]
  Ce document introduit une nouvelle syntaxe dérivée du langage EB^3 appelée eb^{3}web pour spécifier des interfaces de systèmes d'information sur internet.

CSP :

- C.A.R. Hoare : *Communicating Sequential Processes*, Prentice-Hall, 1985 [Hoa85]
  Hoare présente CSP (Communicating Sequential Processes), un langage basé sur les algèbres de processus. Ce livre de référence est très cité dans les publications de la communauté CSP mais est difficilement disponible. Il constitue néanmoins notre principale référence sur CSP.
- S. Schneider : *Concurrent and Real-time Systems : The CSP Approach*, Wiley, 1999 [Sch99]
  Plus récent que [Hoa85], ce livre introduit CSP et présente des illustrations dans le domaine des systèmes concurrents à temps réel. Ce livre nous a surtout permis d'assimiler le langage CSP et de vérifier la syntaxe pour élaborer notre exemple de comparaison.
- A.W. Roscoe : *The Theory and Practice of Concurrency*, Prentice-Hall, 1997 [Ros97]
  Le livre de Roscoe introduit aussi bien le langage CSP que ses différentes sémantiques. Il nous surtout servi à comprendre les différentes notions de raffinement en CSP.

CCS :

- R. Milner : *Communication and Concurrency*, Prentice-Hall, 1989 [Mil89]
  Plusieurs langages basés sur les événements sont présentés ici, en particulier CCS. Ce livre est donc cité comme une référence pour CCS. Il nous a servi à comprendre la syntaxe des processus CCS.
- R. Milner : *A Calculus of Communicating Systems*, Springer-Verlag, 1980 [Mil80]
  Ce document est la première référence au langage CCS, introduit par Milner pour décrire les communications entre systèmes.

LOTOS :

– T. Bolognesi, E. Brinksma : Introduction to the ISO Specification Language LOTOS, *Computer Networks and ISDN Systems*, volume 14, 1987 [BB87]
LOTOS fait partie des langages qui ont inspiré la méthode EB^3, tout comme CSP et CCS. Cet article est un ouvrage de référence du langage LOTOS.

Réseaux de Petri :

– J.L. Peterson : *Petri Net Theory and the Modeling of Systems*, Prentice-Hall, 1981 [Pet81]
Les réseaux de Petri permettent de spécifier formellement et graphiquement des séquences d'événements et les états d'un système. Le livre de Peterson est notre principale référence concernant les réseaux de Petri.
– K. Jensen : *Coloured Petri nets : basic concepts, analysis methods and practical use*, Springer-Verlag, 1996 [Jen96]
Les réseaux de Petri colorés sont des réseaux de Petri dans lesquels les places sont associées à des types. Cette approche permet de généraliser les réseaux de Petri classiques. Ce livre est donc notre référence concernant les réseaux de Petri colorés.

A.1.3 Langages de spécifications algébriques

CASL :

– E. Astesiano, M. Bidoit, H. Kirchner, B. Krieg-Brückner, P. Mosses, D. Sannella, A. Tarlecki : CASL : The Common Algebraic Specification Language, *Theoretical Computer Science*, volume 286, n. 2, 2002 [ABK$^+$02]
Cet article de revue constitue le manuel de référence du langage de spécification algébrique CASL.

Larch :

– J. Guttag, J. Horning : *Larch : Languages and Tools for Formal Specification*, Springer-Verlag, 1993 [GH93]
Ce livre est la référence principale de Larch.

RAISE :

– C. George : The RAISE Specification Language : A Tutorial, *In VDM'91*, volume 551, LNCS, Springer-Verlag, 1991 [Geo91]
Il s'agit d'un tutorat sur le langage RAISE.

A.1.4 Autres approches intéressantes

Logique de Hoare :

– C.A.R. Hoare : An axiomatic basis for computer programming, *Communications of the ACM*, 12(10) :576-583, 1969 [Hoa69]
Cet article introduit la logique de Hoare. Les triplets (précondition, instruction, postcondition) de Hoare sont utilisés dans le chapitre sur le raffinement.

Jackson System Development (JSD) :

- M. Jackson : *System Development*, Prentice-Hall, 1983 [Jac83]
  Ce livre décrit une méthode de spécification utilisant notamment la notion de structure d'entité qui est reprise par le langage EB^3.

LTS :

- R. Milner : *Formal Models and Semantics*, Vol. B of *Handbook of Theoretical Computer Science*, chapter Operational and algebraic semantics of concurrent processes, MIT Press, 1990 [Mil90]
  Ce chapitre de livre introduit et compare les sémantiques des algèbres de processus de CCS. Milner y définit notamment les systèmes de transitions.

Logique temporelle :

- A. Pnueli : The temporal logic of programs, *IEEE 18th annual symposium on the Foundations of Computer Science*, pages 46-57, 1977 [Pnu77]
  La logique temporelle a été introduite par cet article. Nous nous sommes plus particulièrement intéressés à la logique linéaire temporelle propositionnelle (PLTL) qui est utilisée dans une des approches de combinaison de spécifications formelles étudiées.
- A. Pnueli : The temporal semantics of concurrent programs, *Theoretical Computer Science*, volume 13, pages 45-60, 1981 [Pnu81]
  Les principes fondamentaux de la logique temporelle sont définis dans cet article qui est un complément de [Pnu77]. En particulier, la description de la sémantique opérationnelle de PLTL était notre principal intérêt dans cet article.
- Z. Manna, A. Pnueli : *The temporal logic of reactive and concurrent systems*, Springer-Verlag, 1992 [MP92]
  Ce livre est un guide complet des définitions et propriétés des logiques temporelles. Cette référence nous a servi à comprendre la logique temporelle ainsi que les différents types de propriétés qu'on peut exprimer, comme les propriétés de sûreté ou de vivacité.

Abstract State Machines :

- Y. Gurevich : Evolving Algebras : An Attempt to Discover Semantics, *Current Trends in Theoretical Computer Science*, World Scientific, 1993 [Gur93]
  Les "evolving algebras" ou "abstract state machines" sont des algèbres particulières qui permettent de modifier la sémantique des fonctions et des relations du langage. Ce document présente sous forme d'interview les caractéristiques des evolving algebras.
- Y. Gurevich : Evolving Algebras 1993 : Lipari Guide, *Specification and Validation Methods*, Oxford University Press, 1995 [Gur95]
  Ce document présente sous une forme plus classique que [Gur93] les définitions et les propriétés des abstract state machines (ASM). Ce document nous a servi à comprendre les fondements mathématiques des ASM.

Action Systems :

- E.W. Dijkstra : *A Discipline of Programming*, Prentice-Hall, 1976 [Dij76]

Dijkstra a introduit la notion de commande gardée généralisée qui a été ensuite très répandue dans les langages formels. Elle est notamment utilisée par les Action Systems, mais elle est aussi à la base du langage de substitution généralisée du langage B.

- R.J. Back, R. Kurki-Suonio : Decentralisation of process nets with centralised control, *In 2nd ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing*, pages 131-142, 1983 [BKS83]
  Cet article introduit le formalisme des Action Systems de Back. Il est donc une des principales références concernant les Action Systems.
- J. Sinclair : Action Systems : a method combining state-based and event-based specification, *In Software Specification Methods, an overview using a case study*, Springer-Verlag, 2001 [Sin01]
  Cet article présente une étude cas résolue avec les Action Systems. Grâce à ce document, nous avons pu comprendre l'intérêt des Action Systems et nous en servir ensuite dans l'approche Circus pour construire notre exemple de comparaison.
- J. Misra, K.M. Chandy : *Parallel Program Design : A Foundation*, Addison-Wesley, 1988 [MC88]
  Ce livre introduit les Action Systems appelés UNITY.

A.1.5 Manuels des outils

- Clearsy : Manuels d'utilisation de Atelier B,
  `http ://www.atelierb-societe.com` [Cle]
  L'Atelier B est un environnement regroupant plusieurs outils permettant de spécifier et de vérifier des projets avec la méthode B. Il regroupe notamment un type-checker, un générateur d'obligations de preuve, un prouveur automatique et un prouveur interactif. Les manuels d'utilisateur introduisent àla fois la méthode et les outils, concernant toutes les étapes du processus de développement, depuis la spécification à l'implémentation, en passant par le raffinement. L'Atelier B nous a servi à vérifier la syntaxe de certaines spécifications mais aussi à prouver certains raffinements.
- B-Core (UK) Ltd. : B-Toolkit
  `http ://www.b-core.com/btoolkit.html` [B-C]
  Le B-Toolkit de B-Core constitue l'autre outil industriel développé autour de la méthode B. Il est plutôt utilisé par les anglophones, puisqu'il a été créé au Royaume-Uni, contrairement à l'Atelier B qui est une production française.
- Formal Systems (Europe) Ltd. : *Failures-Divergences Refinement : FDR2 User Manual*, `http ://www.formal.demon.co.uk`, 1997 [For97]
  FDR est un model-checker qui permet notamment de vérifier des processus CSP. Il génère et explore l'espace d'états associé au processus analysé. Nous n'avons pas utilisé l'outil FDR, mais il est cité par les articles sur CSP || B.

A.2 Systèmes d'information

A.2.1 Livres de référence sur les bases de données

Les trois livres suivants nous ont permis de comprendre les fondements et les mécanismes des bases de données :

- R. Elmasri : *Fundamentals of Database Systems*, Addison-Wesley, 2004 [Elm04]
  Ce livre en anglais est le plus complet. Les premières parties posent les fondements des bases de données, avec notamment le modèle relationnel et les BD orientées objet. Le reste du livre est consacré aux problèmes de définition et d'optimisation des requêtes, aux différentes méthodes de conception ainsi qu'aux approches les plus avancées, comme les BD actives, le *data mining* ou les BD multimédia.
- G. Gardarin : *Bases de données*, Eyrolles, 1999 [Gar99]
  Ce livre en français est le fruit de plusieurs années d'enseignement en base de données. Il présente les différents concepts des BD, de nombreux exemples et des exercices. Nous avons notamment utilisé les définitions de ce livre pour les concepts de base de données dans notre glossaire.
- J. Ullman : *Principles of Database and Knowledge-base Systems*, Computer Science Press, 1988, [Ull88]
  Le livre de Ullman est un autre exemple de référence complète concernant les définitions et les applications des bases de données.

A.2.2 Bases de données actives

- J. Widom, S. Ceri : *Active Database Systems*, Morgan Kaufmann, 1996 [WC96]
  Ce livre présente les fondements des bases de données actives. Une BD active est une base de donnée dans laquelle des règles sont définies au cas où certains événements interviennent. Dans ce cas, les règles modifient le comportement de la BD.
- A. Tanaka : *On Conceptual Design of Active Databases*, PhD Thesis, Georgia Institute of Technology, 1992 [Tan92]
  Dans sa thèse, Tanaka introduit le modèle ER^2 qui étend le modèle classique entités-associations par des règles de déclenchement capables de modifier le comportement du système.

A.2.3 Méthodes de conception des SI

Gane & Sarson, SADT :

- C. Gane, T. Sarson : *Structured Systems Analysis : Tools and Techniques*, Prentice-Hall, 1979 [GS79]
  Ce livre présente une méthode fonctionnelle de conception.
- D. Marca, C. McGowan : *SADT : Structured Analysis and Design Techniques*, McGraw-Hill, 1988 [MM88]
  Le livre de Marca et McGowan est notre principale référence concernant l'approche SADT.

Merise, Remora, Axial :

- H. Tardieu, A. Rochfeld, R. Colleti : *La méthode MERISE : Principes et Outils*, Éditions d'Organisation, 1983 [TRC83]
  Ce livre est notre référence concernant la méthode MERISE qui fait partie des méthodes systémiques de conception des systèmes d'information.
- C. Rolland, O. Foucaut, G. Benci : *Conception des systèmes d'information : la méthode REMORA*, Eyrolles, 1988 [RFB88]
  Cette référence présente la méthode de conception REMORA.
- P. Pellaumail : *La méthode AXIAL*, Éditions d'Organisation, 1986 [Pel86]
  Le livre de Pellaumail présente la méthode de conception AXIAL.

UML, OMT, OOD et OOSE :

- P.A. Muller : *Modélisation Objet avec UML*, Eyrolles, 1997 [Mul97]
  Ce livre est une introduction à UML. Les méthodes orientées objet utilisent maintenant ce langage unifié. Comme la description des principales entités des systèmes d'information est réalisée avec des diagrammes de classe en UML ou OMT, ce livre nous a servi à comprendre les diagrammes.
- J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorenson : *Object-Oriented Modeling and Design*, Prentice Hall, 1991 [RBP+91]
  Les méthodes orientées objet et en particulier OMT sont présentées et illustrées dans ce livre de référence. Ce livre nous a également servi à assimiler les diagrammes de classes qui décrivent certains systèmes d'information.
- G. Booch : *Object-Oriented Analysis and Design With Applications*, Addison-Wesley, 1994 [Boo94]
  Ce livre est notre référence pour la méthode de conception OOD.
- I. Jacobson : *Object-Oriented Software Engineering - A Use Case Driven Approach*, Addison-Wesley, 1994 [Jac94]
  Le livre de Jacobson présente l'approche orientée objet OOSE.

A.2.4 Approche UML-B-SQL

- H.P. Nguyen : *Dérivation de spécifications formelles B à partir de spécifications semi-formelles*, Thèse de Doctorat, CNAM, Paris, France, 1998 [Ngu98]
  Dans sa thèse, Nguyen définit des règles de transformation de diagrammes OMT en machines B. L'objectif est de pouvoir traduire des diagrammes graphiques et par conséquent semi-formels en des spécifications formelles en utilisant le langage B. Nous avons pu ainsi étudier une méthode possible de spécification des systèmes d'information.
- A. Mammar : *Un environnement formel pour le développement d'applications base de données*, Thèse de Doctorat, CNAM, Paris, France, 2002 [Mam02]
  Mammar présente une méthode de développement pour les applications de base de données sûres à partir de plusieurs techniques de transformations, de traductions et de raffinement qui utilisent successivement UML, B et SQL. Cette thèse nous a fourni un autre exemple de méthode de spécification des systèmes d'information.
- R. Laleau : *Conception et développement formels d'applications bases de données*, Habilitation à diriger des recherches, Université d'Évry Val d'Essonne, France, 2002 [Lal02]

Laleau présente les nombreux travaux réalisés pour développer des applications bases de données avec des méthodes formelles. Ce mémoire nous a servi à nous familiariser avec les systèmes d'information et avec les techniques utilisées pour les spécifier.

A.2.5 Autres approches

- Y. Amghar, M. Meziane, A. Flory : Using Business Rules within a Design Process of Active Databases, *In International Journal of Database Management*, volume 11, numero 3, pages 3-15, 2000 [AMF00]
  Cet article propose une méthode de spécification des systèmes d'information impliquant l'utilisation de règles actives. L'utilisation de bases de données actives constitue une méthode de spécification des systèmes d'information que nous avons étudiée à travers cet article.
- A. Flory, C. Rolland : Nouvelles perspectives des systèmes d'information, chapitre *La conception des systèmes d'information : État de l'art et nouvelles perspectives*, Eyrolles, 1990 [FR90]
  Cet état de l'art présente différentes méthodes de spécification des systèmes d'information ainsi que les perspectives attendues en 1990. Outre la revue des différentes approches, l'article nous a permis de nous rendre compte des méthodes qui ont effectivement abouti et celles qui n'ont pas évolué.
- E. Yourdon, E. Constantine : *Structured Design*, Prentice-Hall, 1979 [YC79]
  Une technique autrefois utilisée dans la spécification des systèmes d'information consistait à utiliser des flots de données. Ce livre est une référence sur les flots de données.
- G. Babin, F. Lustman : Application of Formal Methods to Scenario-based Requirements Engineering, *In International Journal of Computers and Applications*, volume 23, numero 3, pages 141-145, 2001 [BL01]
  Babin et Lustman proposent d'utiliser des scénarios pour spécifier les propriétés dynamiques d'un système d'information. Cet article nous a servi à comprendre cette approche.
- C.A. Heuser, E.M. Peres, G. Richter : Towards a complete conceptual model : Petri nets and entity-relationship diagrams, *In Information Systems*, volume 18, numero 5, pages 275-298, 1993 [HPR93]
  Une des difficultés dans le développement des systèmes d'information concerne la description des aspects dynamiques. Heuser, Peres et Richter présentent ici une approche combinant à la fois des réseaux de Petri et des diagrammes entité-relations plus classiques. Cet article nous a permis de comprendre l'approche proposée.
- G. Lausen : Modeling and Analysis of the Behavior of Information Systems, *IEEE Transactions on Software Engineering*, volume 14, numero 11, pages 1610-1620, 1988 [Lau88]
  Cet article, bien que non cité parmi les références de l'état de l'art, présente également une approche pour spécifier des systèmes d'information avec des réseaux de Petri. Il nous a servi à comprendre le grand nombre d'approches proposées pour spécifier les aspects dynamiques des systèmes d'information.

A.3 Raffinement et simulation

A.3.1 Premières références au raffinement

Les premiers travaux sur le raffinement ont été menés notamment par Back, Abadi et Lamport, Morris, Morgan et Wirth :

- R.J. Back : A calculus of refinements for program derivations, *Acta Informatica*, volume 25, 1988 [Bac88]
- R.J. Back : *On the correctness of refinement in program development*, Ph.D. thesis, University of Helsinki, 1978 [Bac78]
- M. Abadi, L. Lamport : The existence of refinement mappings, Technical report, Digital Systems Research Center, 1988 [AL88]
- J.M. Morris : A theoretical basis for stepwise refinement and the programming calculus, *Science of Computer Programming*, volume 9, 1987 [Mor87]
- C. Morgan : *Programming from Specifications*, Prentice-Hall, 1990 [Mor90]
- C. Morgan : The Specification Statement, *ACM Transactions on Programming Languages and Systems*, 11(4) :517-561, 1988 [Mor88]
- N. Wirth : Program Development by Stepwise Refinement, *Communications of ACM*, 14(4) :221-227, 1971 [Wir71]

A.3.2 Sémantique relationnelle

- A. Tarski : On the calculus of relations, *Symbolic Logic*, volume 6, 1941 [Tar41]
  Tarski présente le calcul relationnel dans cet article de revue. Plusieurs notions de raffinement, comme en Z, sont basées sur le calcul relationnel.
- J.W. de Bakker : *Mathematical Theory of Program Correctness*, Prentice-Hall, 1980 [dB80]
- A. Mili : A relational approach to the design of deterministic programs, *Acta Informatica*, volume 20, 1983 [Mil83]

A.3.3 Sémantique des jeux

La sémantique des jeux introduite par Hintikka a été utilisée par la suite pour définir le raffinement :

- J. Hintikka : *Language games and information*, Clarendon, 1972 [Hin72]
- Y.N. Moschovakis : The game quantifier,*In Proc. of the American Mathematical Society*, 1972 [Mos72]
- P. Aczel : Quantifiers, games and inductive definitions, *In Proc. 3rd Scandinavian Logic Symposium*, 1975 [Acz75]
- R.J. Back, J. von Wright : Duality in specification languages : a lattice-theoretical approach, Technical report 77, Abo Akademi, 1989 [BvW89]

A.3.4 Applications du raffinement

Relations de simulation :

- J. He, C.A.R. Hoare, J.W. Sanders : Data refinement refined, *In Proceedings ESOP 86 : European Symposium on Programming*, Springer-Verlag, Saarbrucken, Allemagne de l'Ouest, 17-19 Mars 1986 [HHS86]

Cet article introduit la notion de relations de simulation. Pour comparer deux systèmes, il est possible d'utiliser la notion de raffinement si les langages de spécification sont les mêmes. He, Hoare et Sanders définissent ici la notion de relations de simulation, basées sur des modèles relationnels, pour comparer des systèmes quelconques. Tous les articles traitant de problèmes de simulation font référence à cet article.

- M.B. Josephs : A state-based approach to communicating processes, *Distributed Computing*, volume 3, pages 9-18, 1988 [Jos88]
  Josephs définit des relations de simulation et des règles suffisantes pour montrer un lien de raffinement entre deux sysèmes. À l'instar de [HHS86], cet article est la principale référence en littérature concernant les relations de simulation.
- C. Bolton, J. Davies, J. Woodcock : On the Refinement and Simulation of Data Types and Processes, *IFM*, pages 273-292, 1999 [BDW99]
  Cet article utilise des relations de simulations inspirées de [HHS86, Jos88] pour relier des spécifications basées sur les états avec des spécifications basées sur les événements. Nous avons utilisé cet article pour comprendre les différences entre relation de simulation et raffinement.

Comparaison des relations :

- R.J. Back, J. von Wright : *Refinement Calculus : A Systematic Introduction*, Graduate Texts in Computer Science, Springer-Verlag, 1998 [BvW98]
  Ce livre présente le calcul du raffinement de Back, ainsi que ses fondements théoriques, basés sur la théorie des catégories. Il définit aussi la notion de contrat qui permet d'unifier les programmes avec leurs spécifications. Il compare également les différentes sémantiques du raffinement, comme la sémantique des jeux, du choix ou *wp*.
- W.P. de Roever, K. Engelhardt : *Data Refinement : Model-Oriented Proof Methods and their Comparison*, Cambridge University Press, 1998 [dRE98]
  Ce livre présente dans un premier temps les fondements mathématiques de la simulation. Il décrit ensuite les relations de simulation utilisées dans des langages formels comme VDM ou Z. Ce livre très théorique nous a servi à comprendre les principes des relations de simulation.
- C. Bolton : *On the refinement of state-based and event-based models*, PhD Thesis, New College, Hilary Term, 2002 [Bol02]
  La thèse contient des démonstrations et des raisonnements plus détaillés que dans [BDW99] concernant les liens entre spécifications basées sur les états et spécifications basées sur les événements. La partie qui nous a intéressé concerne les simulations et les raffinements.
- C. Bolton, J. Davies : A comparison of refinement orderings and their associated simulation rules, *In Proc. of Refine 2002*, 2002 [BD02]
  Cet article compare les principales relations de simulation en Z, Object-Z et CSP. Il fait la synthèse des principaux travaux sur les relations de simulation.
- J. Derrick, E. Boiten : Reconciling event and state-based notions of refinement, *StEve 2003*, September 2003 [DB03]
  Dans cet article, Derrick et Boiten corrigent les erreurs remarquées par Bolton concernant leurs relations de simulation en Object-Z. Ils proposent en outre une extension de leurs travaux pour rendre leurs anciennes rela-

tions de simulation compatibles avec le modèle des échecs-divergences.

Raffinement en Z et Object-Z :

– J. Derrick, E. Boiten : *Refinement in Z and Object-Z*, Springer, 2001 [DB01]
Ce livre résument les nombreux travaux sur les relations de simulation en Z et Object-Z ainsi que les travaux sur les preuves de raffinement. Il nous a permis de comprendre les nuances entre les nombreuses relations de simulation.

Raffinement des LTS :

– F. Bellegarde, J. Julliand, O. Kouchnarenko : Ready-simulation is not Ready to Express a Modular Refinement Relation, *In Proc. FASE 2000*, volume 1783, LNCS, Springer-Verlag, 2000 [BJK00]
Cet article introduit la notion de raffinement de LTS utilisée par Darlot dans son approche B événementiel - PLTL. Cette référence nous a permis d'assimiler cette définition.

A.4 Combinaisons de spécifications formelles

A.4.1 csp2B

– M. Butler : csp2B : A Practical Approach to Combining CSP and B, *In Proceedings of FM'99 World Congress on Formal Methods*, Toulouse, France, 22-24 Septembre 1999 [But99]
L'outil csp2B qui est présenté dans cet article permet de traduire des spécifications de type CSP en B. Il est également possible avec cet outil de contrôler les opérations d'une machine B avec un processus CSP. Cette approche fait partie des combinaisons de spécifications étudiées.

– M. Butler : *A CSP Approach to Action Systems*, PhD Thesis, Oxford University, 1992 [But92]
La thèse de Butler constitue une bonne introduction à l'algèbre de processus CSP. Elle contient également une présentation des "Action Systems". Elle nous a servi à comprendre les approches CSP et Action Systems.

A.4.2 CSP || B

– H. Treharne, S. Schneider : Using a Process Algebra to Control B OPERATIONS, *In Proceedings of IFM'99 : 1st International Conference on Integrated Formal Methods*, Springer-Verlag, York, Royaume-Uni, 1999 [TS99]
Treharne et Schneider introduisent dans cet article une approche pour combiner une machine B avec un processus de type CSP, appelé le contrôleur de la machine B. Les opérations B sont ainsi associées à des événements du processus. Si cet article pose les bases de l'approche, il est complété par les trois articles suivants.

– H. Treharne, S. Schneider : How to Drive a B Machine, *In Proceedings ZB2000 : Formal Specification and Development in Z and B*, LNCS, volume 1878, Springer-Verlag, York, Royaume-Uni, 2000 [TS00]

Dans cet article, les auteurs montrent des propriétés de cohérence et de non-divergence sur les paires machines B - processus CSP.

– S. Schneider, H. Treharne : Communicating B Machines, *In Proceedings ZB2002 : Formal Specification and Development in Z and B*, LNCS, volume 2272, Springer-Verlag, Grenoble, France, 2002 [ST02]
Plusieurs résultats concernant la cohérence et les propriétés de sûreté et de vivacité de paires machines - processus sont prouvés dans cet article.

– H. Treharne, S. Schneider, M. Bramble : Composing Specifications Using Communication, *In Proceedings ZB2003 : Formal Specification and Development in Z and B*, LNCS, volume 2651, Springer-Verlag, Turku, Finlande, 4-6 Juin 2003 [TSB03]
Une méthode de développement et un cas d'étude s'appuyant sur l'approche CSP || B définie dans [TS99, TS00, ST02] sont présentés dans cet article. Ces 4 articles nous ont permis de comprendre cette approche qui fait partie des exemples de combinaisons de spécifications formelles étudiés dans notre état de l'art.

A.4.3 CSP-OZ

– C. Fischer : *Combination and Implementation of Processes and Data : from CSP-OZ to Java*, PhD Thesis, Université de Oldenburg, 2000 [Fis00]
Dans sa thèse, Fischer présente une méthode de spécification utilisant notamment une intégration des langages CSP et Object-Z. La partie qui nous a intéressé concerne la définition du langage CSP-OZ. Cet exemple de combinaison fait partie de notre état de l'art.

A.4.4 CSP et Object-Z

– G. Smith, J. Derrick : Specification, Refinement and Verification of concurrent systems - An Integration of Object-Z and CSP, *In Formal Methods in Systems Design*, volume 18, pages 249-284, Kluwer Academic Publishers, 2001 [SD01]
Smith et Derrick utilisent les langages de spécification CSP et Object-Z pour spécifier et vérifier des systèmes concurrents. Une méthode de développement est également présentée et illustrée par une étude de cas. Nous avons étudié cet article pour compléter notre état de l'art et l'élaboration de notre exemple de comparaison.

– G. Smith : Extending W for Object-Z, *In Proceedings of the 9th International Conference of Z Users*, LNCS, volume 967, Springer-Verlag, 1995 [Smi95]
Cet article introduit une logique pour raisonner sur des spécifications Z qui est notamment utilisée dans [SD01]. Nous avons étudié cet article pour comprendre la logique W.

A.4.5 B événementiel et PLTL

– C. Darlot : *Reformulation et vérification de propriétés temporelles dans le cadre du raffinement de systèmes d'événements*, Thèse de Doctorat, Université de Franche-Comté, France, 2002 [Dar02]

La thèse de Darlot propose plusieurs types de combinaisons pour vérifier des propriétés temporelles : une combinaison du système d'événements B (ou B événementiel) avec la logique temporelle propositionnelle PLTL mais aussi une combinaison de preuve et de vérification par model-checking pour vérifier les propriétés logiques sur des systèmes d'événements B. Cette approche fait partie des exemples étudiés dans notre état de l'art.

A.4.6 Circus

- J.C.P. Woodcock : Unifying Theories of Parallel Programming, *In Logic and Algebra for Engineering Software*, IOS Press, 2002 [Woo02]
  Cet article introduit une théorie d'unification basée sur la théorie unifiée de programmation de [HJ98]. Elle est à la base de l'approche Circus qui sera détaillée dans les articles suivants. Cette théorie unifie les théories de Z et CSP pour introduire un calcul du raffinement intégré des deux approches.
- C.A.R. Hoare : Unified Theories of Programming, Monograph, Oxford University Computing Laboratory, 1994 [Hoa94]
  Cette monographie est une première étude sur la théorie unifiée de programmation.
- C.A.R. Hoare and H. Jifeng : *Unifying Theories of Programming*, Prentice-Hall, 1998 [HJ98]
  Ce livre propose une théorie unifiée des modèles de programmation pour simplifier l'intégration des langages de spécification. Il est une référence des travaux sur Circus, qui s'appuient sur cette approche pour intégrer Z et CSP.
- J.C.P. Woodcock, A.L.C. Cavalcanti : The Semantics of Circus, *In Proceedings ZB 2002 : Formal Specification and Development in Z and B*, LNCS, volume 2272, Springer-Verlag, Grenoble, France, 2002 [WC02]
  La sémantique du langage Circus qui est basée sur la théorie d'unification des programmes définie dans [Woo02] est spécifiée dans cet article avec le langage Z qui est utilisé comme méta-langage. Cet article nous a servi à assimiler le langage Circus.
- A.C.A. Sampaio, J.C.P. Woodcock, A.L.C. Cavalcanti : Refinement in Circus, *In Proceedings FME 2002 : Formal Methods - Getting IT Right*, LNCS, volume 2391, Springer-Verlag, 2002 [SWC02]
  Le raffinement en Circus permet d'intégrer à la fois les raffinements de données et les raffinements de processus. Cet article présente un exemple de raffinement ainsi que quelques règles de raffinement pour l'utilisateur.
- A.L.C. Cavalcanti, J.C.P. Woodcock : Refinement of Actions in Circus, *In Proceedings REFINE'2002*, Electronic Notes in Theoretical Computer Science, 2002. [CW02]
  Le raffinement Circus permet non seulement de raffiner les données et les processus mais aussi les actions qui définissent le processus principal du système. Cet article présente quelques règles de raffinement des actions.
- A.L.C. Cavalcanti, A.C.A. Sampaio, J.C.P. Woodcock : A Refinement Strategy for Circus, *Formal Aspects of Computing*, 15(2-3) :146-181, 2003 [CSW03]
  Dans cet article de revue, les auteurs de Circus proposent une stratégie de raffinement pour dériver à la fois les données et les actions d'un processus

décrit en Circus.

A.4.7 EB^3-B

- M. Frappier, R. Laleau : Proving the Refinement of Scenarios into Object-Oriented Models, Rapports techniques du CEDRIC n. 277 et du DMI n. 272, 2001 [FL01]
  Frappier et Laleau proposent de spécifier les systèmes d'information en utilisant les langages de spécifications formels B [Abr96] et EB^3 [FSD98] ainsi que des diagrammes de classes UML du modèle orienté objet. Le diagramme de classes est ainsi traduit en B et les propriétés concernant l'ordre d'exécution des opérations sont spécifiées à l'aide d'EB^3. Les propriétés sont alors vérifiées en utilisant la notion de raffinement B. Cet article fait partie de nos références concernant la combinaison des spécifications EB^3-B.
- M. Frappier, R. Laleau : Proving event ordering properties for Information Systems, *In Proceedings ZB2003 : Formal Specification and Development in Z and B*, LNCS, volume 2651, Springer-Verlag, Turku, Finlande, 4-6 Juin 2003 [FL03]
  L'utilisation du raffinement B pour prouver des propriétés EB^3 d'ordonnancement des événements sur des systèmes spécifiés avec B est ici détaillée avec la présentation de plusieurs techniques de preuve de raffinement. Cet article constitue avec [FL01] notre principale référence pour l'approche EB^3-B.
- B. Fraikin, M. Frappier : EB^{3}PAI : An Interpreter for the EB^3 Specification Language, *In Proceedings of the 15th International Conference on Software and Systems Engineering and their Applications*, CNAM, Paris, France, 3-5 Décembre 2002 [FF02]
  Fraikin et Frappier présente une approche et un outil pour interpréter des spécifications EB^3 d'un système d'information. L'objectif est à la fois d'automatiser au maximum le processus et de générer automatiquement une implémentation du système à partir de sa spécification abstraite. Cet article nous a permis d'assimiler le langage EB^3 ainsi que les méthodes de spécification des systèmes d'information avec EB^3.
- B. Fraikin, M. Frappier : Utilisation d'automates étendus pour interpréter une spécification EB^3, Département de mathématiques et d'informatique, Université de Sherbrooke, Québec, Canada, 2003 [FF03]
  Dans cet article, Fraikin et Frappier définissent un nouveau type d'automate pour améliorer les performances de l'interprétation de spécifications EB^3. Nous nous sommes plus particulièrement intéressés aux notions d'automates étendus et à l'interprétation des spécifications EB^3.
- B. Fraikin, M. Frappier, R. Laleau : A comparison of EB^3 and B for information system specification, *StEve 2003*, September 2003 [FFL03]
  Dans cet article, les approches EB^3 et B sont comparées dans le cadre des systèmes d'information. Nous avons repris l'exemple de spécification proposé pour l'adapter au cadre d'EB^4.

A.4.8 ZCCS

- A.J. Galloway, W.J. Stoddart : Integrated Formal Methods, *In Proceedings of INFORSID'97*, Toulouse, France, 11-13 Juin 1997 [GS97a]
  Le langage CCS ne prévoit pas de sémantique précise pour le passage des valeurs en paramètre des agents CCS. Cette approche propose d'utiliser le langage Z pour décrire et modéliser les passages de valeurs.
- A.J. Galloway, W.J. Stoddart : An operational semantics for ZCCS, *In Proceedings of the First IEEE International Conference on Formal Engineering Methods*, Hiroshima, Japon, 12-14 Novembre 1997 [GS97b]
  Le langage ZCCS est un langage CCS enrichi par une syntaxe Z pour décrire les passages de valeurs en paramètres des agents. Cet article définit la sémantique opérationnelle du langage ZCCS qui est basée sur les sémantiques de Z et CCS.

A.4.9 Z + Petri Nets

- F. Peschanski, D. Julien : When concurrent control meets functional requirements, or Z + Petri-Nets, *In Proceedings ZB2003 : Formal Specification and Development in Z and B*, LNCS, volume 2651, Springer-Verlag, Turku, Finlande, 4-6 Juin 2003 [PJ03]
  Cet article propose de combiner le langage Z avec les réseaux de Petri pour modéliser le comportement de systèmes décrit en Z. Pour éviter des problèmes de cohérence, l'approche propose de séparer les descriptions fonctionnelles des descriptions du comportement. Cet exemple fait partie des approches étudiées dans le chapitre des combinaisons de spécifications formelles.

A.5 Dictionnaires

On oublie souvent l'importance des dictionnaires, mais ils sont indispensables lors de la rédaction d'un rapport :

- P. Robert : *Le Petit Robert*, Dictionnaires Le Robert, 2003 [Rob03]
- V. Illingworth : *Dictionnaire d'informatique*, Hermann, 1991 [Ill91]
- *Dictionary of Computer Science : the Standardized Vocabulary*, ISO, AFNOR, 1997 [ISO97]

Annexe B

Glossaire des notions utilisées

Acteur. Personne ou entité qui prend une part active ou joue un rôle important dans un système [Rob03].

Action. Opération instantanée associée à un événement [RBP+91]. Dans un système de transitions étiqueté, un événement étiqueté ou nommé [Mil90].

Agent. Dans un système concurrent, entité agissant de manière indépendante, capable de communiquer avec d'autres agents [Mil80].

Association. Dans le modèle entités-associations, lien logique entre deux ou plusieurs entités [Gar99].

Base de données. Ensemble de données interrogeables modélisant les objets d'une partie du monde réel et qui sert de support à une application informatique [Gar99]. Ensemble de données organisées [Elm04].

Cohérence. État de ce qui est relié de manière étroite et sans contradiction [Rob03].

Contrainte d'intégrité. Dans les bases de données, règle sémantique assurant la cohérence des données lors des mises à jour de la base [Gar99].

Correction. Qualité d'un système ou d'un programme qui est conforme à sa représentation [Rob03]. Une preuve de correction est une démonstration formelle qui prouve que la sémantique d'un programme est cohérente avec sa spécification [ISO97].

Entité. Toute chose, concrète ou abstraite, qui existe, a existé ou devrait exister, incluant les associations entre ces choses [ISO97]. Objet du monde réel doué d'une unité matérielle, dont l'existence est indépendante des autres entités [Gar99].

État. Ensemble de valeurs nommées : les noms sont des variables et les valeurs sont exprimées dans des domaines issus des mathématiques (entiers naturels, réels, etc ...) [Mor90].

Événement. Action indivisible atomique pouvant être exécutée par un processus [Sch99].

Formel. Dont la précision exclut toute forme d'équivoque [Rob03]. Une notation est dite formelle si elle peut être exprimée mathématiquement [Rob03]. Une spécification est formelle si elle est écrite avec une notation formelle [ISO97]. Un langage est formel s'il comprend des règles de syntaxe et de sémantique explicites et précises [Ill91]. Une méthode est formelle si elle s'appuie sur des techniques et sur des langages formels.

Méthode. Ensemble de techniques et de notations utilisées selon certains principes ou un certain ordre pour arriver à un but. Par exemple, une méthode de conception [Rob03].

Processus. Pattern de communication [Hoa85]. Un processus est une entité indépendante qui communique avec d'autres processus [Sch99].

Programme. Ensemble d'instructions, rédigé dans un langage de programmation, permettant à un système informatique d'accomplir une tâche donnée [Rob03]. Unité syntaxique qui respecte les règles d'un langage de programmation et qui est composée de déclarations et d'instructions utilisées pour résoudre une certaine tâche ou un problème [ISO97]. Ensemble d'instructions détaillées écrit dans un langage de programmation dont la forme (la syntaxe) et le sens (la sémantique) sont définis précisément. Les programmes sont faciles à exécuter, mais difficiles à comprendre [Mor90].

Propriété. Ensemble de caractères d'un objet ou d'un système [Rob03].

Raffinement. Processus permettant de réexprimer progressivement les idées de haut niveau en des idées de plus bas niveau [Ill91]. Une instruction S' raffine S si S' satisfait toute spécification satisfaite par S [But92]. Un développement de programme implique le raffinement par étapes d'un programme abstrait en un programme exécutable par l'application d'une série de transformations préservant la correction [But92]. Concevoir un programme complexe implique en général l'application d'une méthode de raffinement qui fournit une façon de transformer graduellement un programme abstrait ou une spécification en une implémentation concrète. Le principe d'une telle méthode est que si le programme abstrait initial est correct et que les étapes de transformation préservent la correction, alors l'implémentation sera correcte par construction [dRE98].

Redondance. Caractère de ce qui apporte une information déjà donnée sous une autre forme [Rob03].

Relation. Dans les bases de données, sous-ensemble du produit cartésien entre plusieurs domaines de valeurs [Gar99].

Sémantique. Partie de la définition d'un langage qui concerne la spécification de la signification ou de l'effet d'un texte construit selon les règles de la syntaxe [Ill91].

Simulation. Relation entre deux espaces d'états qui est préservée par chaque opération. Si A est une simulation de C, alors C contient strictement plus d'informations que A. En fait, il est possible de montrer que A est une simulation de C précisément lorsque C est un raffinement de A [BDW99].

Spécification. Document qui décrit la structure et les fonctionnalités d'un système de manière détaillée, afin d'en faciliter la programmation et la maintenance [ISO97]. Énoncé précis et détaillé des résultats qu'on attend d'un système. La spécification peut être écrite dans la langue naturelle ou bien à l'aide d'un langage de spécification [Ill91]. Une spécification décrit ce qu'un système doit faire. Les spécifications sont difficiles à exécuter, mais faciles à comprendre - ou devraient l'être [Mor90]. Une spécification décrit les propriétés observables et le comportement d'un système qui n'existe pas encore dans le monde physique ; et le but est de concevoir et d'implémenter un produit qui a été prévu, en théorie, pour respecter la spécification [Hoa94].

Syntaxe. Règles définissant les séquences de symboles et/ou de caractères dans un langage [Ill91].

Système d'information. Système informatisé qui rassemble l'ensemble des informations présentes au sein d'une organisation, sa mémoire, et les activités qui permettent de les manipuler [Lal02]. Système qui comprend toutes les ressources qui sont impliquées dans le rassemblement, la gestion, l'utilisation et la dissémination des informations d'une organisation. Dans un environnement informatisé, ces ressources comprennent les données elles-mêmes, le logiciel de gestion de base de données, le matériel et les moyens de stockage du système informatique, le personnel qui utilise et qui gère les données, les logiciels d'applications qui accèdent aux données et qui les modifient, et les programmeurs qui développent ces applications [Elm04].

Transaction. Application sur la base de données qui permet d'y faire des interrogations et/ou des mises à jour et qui vérifie en général les propriétés suivantes :

- une transaction est atomique, car elle est exécutée dans son ensemble ou pas du tout,
- la base de données doit rester cohérente après l'exécution d'une transaction,
- chaque transaction est indépendante et n'interfère pas avec les autres transactions,
- et les modifications d'une transaction qui a été exécutée perdurent même en cas de panne de la base de données.

Ces propriétés sont appelées ACID (pour *Atomicity, Consistency, Isolation, Durability*) [Elm04].

Validation. Évaluation, plus ou moins subjective, de l'adaptation probable d'un système à l'environnement de travail projeté [Ill91].

Vérification. Étape du développement logiciel consistant à vérifier que le programme est conforme à sa spécification [ISO97]. Contrôle objectif de la conformité d'un programme à une spécification [Ill91].

Bibliographie

[ABK+02] E. Astesiano, M. Bidoit, H. Kirchner, B. Krieg-Brückner, P. Mosses, D. Sannella, and A. Tarlecki. CASL : The Common Algebraic Specification Language. *Theoretical Computer Science*, 286(2) :153–196, 2002.

[Abr96] J.R. Abrial. *The B-Book : Assigning programs to meanings*. Cambridge University Press, 1996.

[Abr00] J.R. Abrial. Guidelines to formal systems studies. ClearSy, November 2000.

[Acz75] P. Aczel. Quantifiers, games and inductive definitions. In *Proc. 3rd Scandinavian Logic Symposium*, North Holland, 1975.

[AL88] M. Abadi and L. Lamport. The existence of refinement mappings. Technical report, Digital Systems Research Center, Palo Alto, California, 1988.

[AM98] J.R. Abrial and L. Mussat. Introducing dynamic constraints in B. In *Second Conference on the B Method*, volume 1393 of *LNCS*, pages 83–128, 1998.

[AMF00] Y. Amghar, M. Meziane, and A. Flory. Using Business Rules within a Design Process of Active Databases. *International Journal of Database Management*, 11(3) :3–15, July 2000.

[B-C] B-Core (UK) Ltd. B-Toolkit. `http ://www.b-core.com/btoolkit.html`.

[Bac78] R.J. Back. *On the correctness of refinement in program development*. PhD thesis, University of Helsinki, 1978.

[Bac88] R.J. Back. A calculus of refinements for program derivations. *Acta Informatica*, 25 :593–624, 1988.

[BB87] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14(1), 1987.

[BD02] C. Bolton and J. Davies. A comparison of refinement orderings and their associated simulation rules. In *Proc. of Refine 2002*, volume 70 of *ENTCS*. Elsevier Science Publishers, 2002.

[BDW99] C. Bolton, J. Davies, and J. Woodcock. On the refinement and simulation of data types and processes. In *IFM*, pages 273–292, 1999.

[BG92] G. Berry and G. Gonthier. The Esterel Synchronous Programming Language : Design, Semantics, Implementation. *Science of Computer Programming*, 19(2) :87–152, 1992.

[BJK00] F. Bellegarde, J. Julliand, and O. Kouchnarenko. Ready-simulation is not ready to express a modular refinement relation. In *Proc. FASE 2000*, volume 1783 of *LNCS*. Springer-Verlag, 2000.

[BKS83] R.J. Back and R. Kurki-Suonio. Decentralisation of process nets with centralised control. In *2nd ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing*, pages 131–142, 1983.

[BL01] G. Babin and F. Lustman. Application of formal methods to scenario-based requirements engineering. *International Journal of Computers and Applications*, 23(3) :141–151, 2001.

[Bol02] C. Bolton. *On the refinement of state-based and event-based models*. PhD thesis, New College, Hilary Term, 2002.

[Boo94] G. Booch. *Object-Oriented Analysis and Design With Applications*. Addison-Wesley, 1994.

[But92] M. Butler. *A CSP approach to Action Systems*. PhD thesis, Oxford University, 1992.

[But99] M. Butler. csp2B : A practical approach to combining CSP and B. In J. Wing, J. Woodcock, and J. Davies, editors, *FM'99 World Congress on Formal Methods*, pages 223–241, Toulouse, France, 22-24th Sept 1999.

[BvW89] R.J. Back and J. von Wright. Duality in specification languages : a lattice-theoretical approach. Technical Report 77, Abo Akademi, 1989.

[BvW98] R.J. Back and J. von Wright. *Refinement Calculus : A Systematic Introduction*. Graduate Texts in Computer Science. Springer-Verlag, 1998.

[Cle] Clearsy. Atelier B. `http ://www.atelierb-societe.com`.

[CSW03] A.L.C. Cavalcanti, A.C.A. Sampaio, and J.C.P. Woodcock. A Refinement Strategy for Circus. *Formal Aspects of Computing*, 15(2-3) :146–181, 2003.

[CW02] A.L.C. Cavalcanti and J.C.P. Woodcock. Refinement of Actions in Circus. In *REFINE'2002*. Electronic Notes in Theoretical Computer Science, 2002.

[Dar02] C. Darlot. *Reformulation et vérification de propriétés temporelles dans le cadre du raffinement de systèmes d'événements*. PhD thesis, Université de Franche-Comté, 2002.

[dB80] J.W. de Bakker. *Mathematical Theory of Program Correctness*. Prentice-Hall, 1980.

[DB01] J. Derrick and E. Boiten. *Refinement in Z and Object-Z*. Springer, 2001.

[DB03] J. Derrick and E. Boiten. Reconciling event and state-based notions of refinement. ST.EVE Workshop at FM03, September 2003.

[Dij76] E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[dRE98] W.P. de Roever and K. Engelhardt. *Data Refinement : Model-Oriented Proof Methods and their Comparison*, volume 47 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1998.

[DW96] J. Davies and J.C.P. Woodcock. *Using Z : Specification, Refinement, and Proof*. Prentice-Hall, 1996.

[Elm04] R. Elmasri. *Fundamentals of Database Systems*. Addison-Wesley, 2004.

[FF02] B. Fraikin and M. Frappier. EB3PAI : an interpreter for the EB3 specification language. In *15th International Conference on Software and Systems Engineering and their Applications*, December 3-5 2002.

[FF03] B. Fraikin and M. Frappier. Utilisation d'automates étendus pour interpréter une spécification EB3. Technical report, DMI, 2003.

[FFL03] B. Fraikin, M. Frappier, and R. Laleau. A comparison of EB^3 and B for information system specification. In *State-oriented vs. Event-oriented thinking in Requirements Analysis, Formal Specification and Software Engineering (StEve)*, Pisa, Italy, September 13 2003. Satellite workshop at FM 2003.

[Fis00] C. Fischer. *Combination and Implementation of Processes and Data : from CSP-OZ to Java*. PhD thesis, University of Oldenburg, 2000.

[FL01] M. Frappier and R. Laleau. Proving the refinement of scenarios into object-oriented models. Technical report, CEDRIC n. 277 and DMI n. 272, 2001.

[FL03] M. Frappier and R. Laleau. Proving event ordering properties for Information Systems. In *ZB2003 : Formal Specification and Development in Z and B*, volume 2651 of *LNCS*, Turku, Finland, 4-6 June 2003. Springer-Verlag.

[For97] Formal Systems (Europe) Ltd. Failures-Divergences Refinement : FDR2 User Manual. `http ://www.formal.demon.co.uk`, 1997.

[FR90] A. Flory and C. Rolland. *Nouvelles perspectives des systèmes d'information*, chapter La conception des systèmes d'information : État de l'art et nouvelles perspectives. Eyrolles, 1990.

[FSD98] M. Frappier and R. St-Denis. Specifying information systems through structured input-output traces. Technical report, Département de mathématiques et d'informatique, Université de Sherbrooke, Québec, Canada, 1998.

[FSD03] M. Frappier and R. St-Denis. EB^3 : an entity-based black-box specification method for information systems. *Software and Systems Modeling*, 2(2) :134–149, July 2003.

[Gar99] G. Gardarin. *Bases de données*. Eyrolles, 1999.

[Geo91] C. George. The RAISE Specification Language : A Tutorial. In *VDM'91*, volume 551 of *LNCS*. Springer-Verlag, 1991.

[GH93] J. Guttag and J. Horning. *Larch : Languages and Tools for Formal Specification*. Springer-Verlag, 1993.

[GS79] C. Gane and T. Sarson. *Structured Systems Analysis : Tools and Techniques*. Prentice-Hall, 1979.

[GS97a] A.J. Galloway and W.J. Stoddart. Integrated formal methods. In *INFORSID'97*, Toulouse, France, 11-13 June 1997.

[GS97b] A.J. Galloway and W.J. Stoddart. An operational semantics for ZCCS. In *First IEEE International Conference on Formal Engineering Methods*, Hiroshima, Japon, 12-14 November 1997. IEEE.

[Gur93] Y. Gurevich. Evolving Algebras : An attempt to discover semantics. In *Current Trends in Theoretical Computer Science*. World Scientific, 1993.

[Gur95] Y. Gurevich. Evolving Algebras 1993 : Lipari Guide. In *Specification and Validation Methods*. Oxford University Press, 1995.

[Hab01] H. Habrias. *Spécification formelle avec B*. Hermes Sciences. Lavoisier, 2001.

[Har87] D. Harel. Statecharts : A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8 :231–274, 1987.

[HHS86] J. He, C.A.R. Hoare, and J.W. Sanders. Data refinement refined. In *ESOP 86 : European Symposium on Programming*, Saarbrucken, West Germany, 17-19 March 1986. Springer-Verlag.

[Hin72] J. Hintikka. *Language games and information*. Clarendon, 1972.

[HJ98] C.A.R. Hoare and H. Jifeng. *Unifying Theories of Programming*. Prentice-Hall, 1998.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10) :576–583, 1969.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[Hoa94] C.A.R. Hoare. Unified Theories of Programming. Monograph, Oxford University Computing Laboratory, 1994.

[HPR93] C.A. Heuser, E.M. Peres, and G. Richter. Towards a complete conceptual model : Petri nets and entity-relationship diagrams. *Information Systems*, 18(5) :275–298, 1993.

[Ill91] V. Illingworth. *Dictionnaire d'informatique*. Hermann, 1991.

[ISO97] ISO. *Dictionary of Computer Science : the Standardized Vocabulary*. ISO, AFNOR, 1997.

[Jac83] M. Jackson. *System Development*. Prentice-Hall, 1983.

[Jac94] I. Jacobson. *Object-Oriented Software Engineering - A Use Case Driven Approach*. Addison-Wesley, 1994.

[Jen96] K. Jensen. *Coloured Petri nets : basic concepts, analysis methods and practical use*. Springer-Verlag, 1996.

[Jon90] C.B. Jones. *Systematic Software Development using VDM*. Prentice Hall, 1990.

[Jos88] M.B. Josephs. A state-based approach to communicating processes. *Distributed Computing*, 3 :9–18, 1988.

[Lal02] R. Laleau. *Conception et développement formels d'applications bases de données*. Habilitation à diriger des recherches, Université d'Évry Val d'Essonne, 2002.

[Lau88] G. Lausen. Modeling and analysis of the behavior of information systems. *IEEE Transactions on Software Engineering*, 14(11) :1610–1620, November 1988.

[Mam02] A. Mammar. *Un environnement formel pour le développement d'applications base de données*. PhD thesis, CNAM, 2002.

[MC88] J. Misra and K.M. Chandy. *Parallel Program Design : A Foundation*. Addison-Wesley, 1988.

[Mil80] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag, 1980.

[Mil83] A. Mili. A relational approach to the design of deterministic programs. *Acta Informatica*, 20 :315 – 328, 1983.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[Mil90] R. Milner. *Formal Models and Semantics*, volume B of *Handbook of Theoretical Computer Science*, chapter Operational and algebraic semantics of concurrent processes. MIT Press, 1990.

[MM88] D. Marca and C. McGowan. *SADT : Structured Analysis and Design Techniques*. McGraw-Hill, 1988.

[Mor87] J.M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9 :287–306, 1987.

[Mor88] C. Morgan. The Specification Statement. *ACM Transactions on Programming Languages and Systems*, 11(4) :517–561, 1988.

[Mor90] C. Morgan. *Programming from Specifications*. Prentice-Hall, 1990.

[Mos72] Y.N. Moschovakis. The game quantifier. In *Proc. of the American Mathematical Society*, pages 245 – 250, 1972.

[MP92] Z. Manna and A. Pnueli. *The temporal logic of reactive and concurrent systems*. Springer-Verlag, 1992.

[Mul97] P.A. Muller. *Modélisation objet avec UML*. Eyrolles, 1997.

[Ngu98] H.P. Nguyen. *Dérivation de spécifications formelles B à partir de spécifications semi-formelles*. PhD thesis, CNAM, 1998.

[NXD03] G. Nguyen-Xuan-Dang. *Génération automatique de sites WEB pour des systèmes d'information*. Mémoire de maîtrise en génie logiciel, Université de Sherbrooke, 2003.

[Pel86] P. Pellaumail. *La méthode AXIAL*. Éditions d'Organisation, 1986.

[Pet81] J.L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice-Hall, 1981.

[PJ03] F. Peschanski and D. Julien. When concurrent control meets functional requirements, or Z + Petri-Nets. In *ZB2003 : Formal Specification and Development in Z and B*, volume 2651 of *LNCS*, Turku, Finland, 4-6 June 2003. Springer-Verlag.

[Pnu77] A. Pnueli. The temporal logic of programs. In *18th annual symposium on the Foundations of Computer Science*, pages 46–57. IEEE, 1977.

[Pnu81] A. Pnueli. The temporal semantics of concurrent programs. In *Theoretical Computer Science*, volume 13, pages 45–60, 1981.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorenson. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[RFB88] C. Rolland, O. Foucaut, and G. Benci. *Conception des systèmes d'information : la méthode REMORA*. Eyrolles, 1988.

[Rob03] P. Robert. *Le Petit Robert*. Dictionnaires Le Robert, 2003.

[Ros97] A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1997.

[Sch99] S. Schneider. *Concurrent and Real-time Systems : The CSP Approach*. Wiley, 1999.

[SD01] G. Smith and J. Derrick. Specification, refinement and verification of concurrent systems - an integration of Object-Z and CSP. *Formal Methods in Systems Design*, 18 :249–284, May 2001.

[Sin01] J. Sinclair. Action Systems : a method combining state-based and event-based specification. In *Software Specification Methods, an overview using a case study*. Springer-Verlag, 2001.

[Smi95] G. Smith. Extending W for Object-Z. In J. Bowen and M. Hinchey, editors, *9th International Conference of Z Users*, volume 967 of *LNCS*, pages 276–295. Springer-Verlag, 1995.

[Smi00] G. Smith. *The Object-Z Specification Language*. Kluwer Academic Publishers, 2000.

[Spi92] J.M. Spivey. *The Z Notation : a Reference Manual*. Prentice-Hall, 1992.

[ST02] S. Schneider and H. Treharne. Communicating B Machines. In D. Bert, J.P. Bowen, M.C. Henson, and K. Robinson, editors, *ZB2002 : Formal Specification and Development in Z and B*, volume 2272 of *LNCS*, pages 416–435. Springer-Verlag, 2002.

[SWC02] A.C.A. Sampaio, J.C.P. Woodcock, and A.L.C. Cavalcanti. Refinement in Circus. In *FME 2002 : Formal Methods - Getting IT Right*, volume 2391 of *LNCS*. Springer-Verlag, 2002.

[Tan92] A. Tanaka. *On Conceptual design of Active Databases*. PhD thesis, Georgia Institute of Technology, 1992.

[Tar41] A. Tarski. On the calculus of relations. *Symbolic Logic*, 6 :73 – 89, 1941.

[TRC83] H. Tardieu, A. Rochfeld, and R. Colleti. *La méthode MERISE : Principes et Outils*. Éditions d'Organisation, 1983.

[TS99] H. Treharne and S. Schneider. Using a process algebra to control B OPERATIONS. In *IFM'99 1st International Conference on Integrated Formal Methods*, pages 437–457, York, 1999. Springer-Verlag.

[TS00] H. Treharne and S. Schneider. How to drive a B machine. In J.P. Bowen, S. Dunne, A. Galloway, and S. King, editors, *ZB2000 : Formal Specification and Development in Z and B*, volume 1878 of *LNCS*, pages 188–208. Springer-Verlag, 2000.

[TSB03] H. Treharne, S. Schneider, and M. Bramble. Composing specifications using communication. In *ZB2003 : Formal Specification and Development in Z and B*, volume 2651 of *LNCS*, Turku, Finland, 4-6 June 2003. Springer-Verlag.

[Ull88] J. Ullman. *Principles of Database and Knowledge-base Systems*. Computer Science Press, 1988. 2 volumes.

[WC96] J. Widom and S. Ceri. *Active Database Systems*. Morgan Kaufmann, 1996.

[WC02] J.C.P. Woodcock and A.L.C. Cavalcanti. The Semantics of Circus. In *ZB 2002 : Formal Specification and Development in Z and B*, volume 2272 of *LNCS*, Grenoble, France, 2002. Springer-Verlag.

[Wir71] N. Wirth. Program Development by Stepwise Refinement. *Communications of ACM*, 14(4) :221–227, 1971.

[Woo02] J.C.P. Woodcock. Unifying theories of parallel programming. In *Logic and Algebra for Engineering Software*. IOS Press, 2002.

[YC79] E. Yourdon and E. Constantine. *Structured Design*. Prentice-Hall, 1979.