D.E.A. Informatique

Université d'Évry Val d'Essonne (EVE)
Institut d'Informatique d'Entreprise (IIE)
Institut National des Télécommunications (INT)

RAPPORT DE STAGE DE D.E.A.

Réutilisation de composants de spécification en B

Frédéric GERVAIS

Responsables de stage :

Sandrine BLAZY
et Régine LALEAU

Stage effectué au laboratoire CEDRIC de l'IIE,
du 1er Février au 2 Juillet 2002

Table des matières

Chapitre 1

Introduction

1.1 Motivation

La méthode B, qui fait partie des méthodes de spécification formelle, offre de nombreux avantages. Le langage utilisé dans cette méthode, appelé le langage B, est en effet fondé sur une sémantique précise, basée sur la notion de machine abstraite. Dans la pratique, une machine abstraite est un module de conception B qui encapsule des données et des opérations et qui représente l'état du système modélisé. Il est ainsi possible, avec la méthode B, de spécifier formellement un projet, tout en vérifiant la cohérence de ces spécifications. Les outils de preuve associés au langage permettent en effet de valider les spécifications de chaque machine abstraite. Un projet est généralement spécifié en B de manière incrémentale : une machine abstraite correctement spécifiée est “incluse” dans une machine plus grosse, et ainsi de suite ... La dernière machine, qui représente l'interface du projet, regroupe alors les spécifications de toutes les machines incluses. Enfin, il est possible de générer du code en utilisant une phase de raffinements successifs qui rendent les opérations du projet de plus en plus concrètes. Chaque étape de raffinement est ensuite vérifiée à l'aide des outils de preuve.

Mais cette forme de spécification incrémentale n'est pas toujours satisfaisante. Il arrive parfois que deux projets différents résolvent le même problème, comme une allocation de ressource, mais dans deux applications distinctes. On constate alors que les spécifications obtenues sont assez proches, à renommage près. Lorsqu'un problème est récurrent et qu'une solution est clairement exprimée, la tendance naturelle consiste à les généraliser et à les conserver en mémoire afin de pouvoir les *réutiliser* en temps voulu, l'idée étant de capitaliser un savoir-faire. Plutôt que de reprendre tout le processus d'analyse et de résolution, il suffit alors de récupérer le résultat cherché et de l'appliquer.

Mais cette forme de réutilisation n'existe pas actuellement en B : la notion de “composant” est en effet limitée à la machine abstraite ou de raffinement, et la “réutilisation” consiste à inclure des machines. Pourtant, les avantages de pouvoir utiliser des composants dans le cadre formel du langage B seraient multiples. On pourrait dans un premier temps exprimer précisément ce que font les composants et il serait possible de les prouver ou de repérer leurs incohérences. On pourrait ensuite spécifier formellement un projet à partir d'une bibliothèque de composants préalablement spécifiés et prouvés. Le but serait alors de pouvoir adapter, instancier et spécialiser ces composants avant de les combiner entre eux pour en extraire un résultat final. La figure 1.1 représente justement un processus

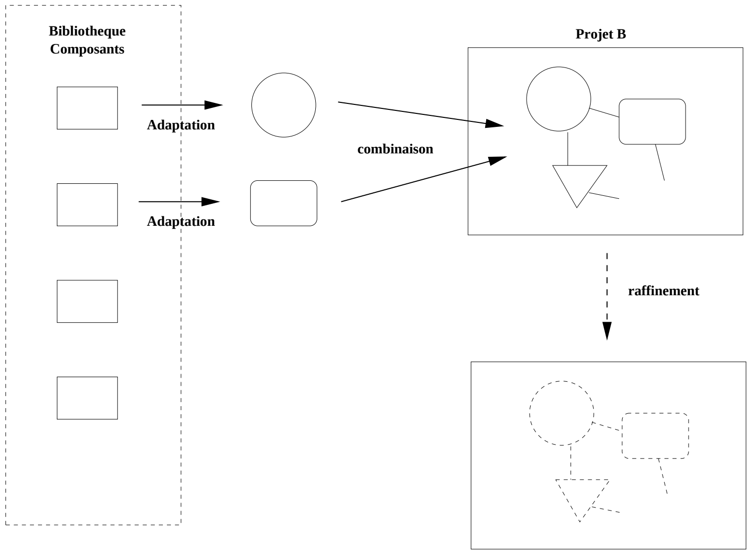

FIG. 1.1 – *Conception B utilisant des composants*

de conception possible, si le langage B était pourvu du concept de réutilisation de composants de spécification. Enfin, la spécification ainsi obtenue serait déjà partiellement prouvée, grâce à la réutilisation des preuves de ses composants.

1.2 Solutions existantes et objectifs

Il existe en informatique plusieurs types de composants réutilisables. Les composants appelés patterns, utilisés pour concevoir des applications, nous intéressent plus particulièrement. Ils correspondent en effet à la phase de conception qui pose justement problème dans le cas du langage B. Malheureusement, de tels composants sont essentiellement utilisés dans des langages semi-formels, comme par exemple UML [20]. Le défaut de ces langages est précisément leur manque de formalisation. La description des composants est souvent ambigüe. Il existe toutefois des travaux qui ont permis d'adapter et de formaliser les patterns dans des langages formels. Le langage LePUS [10] est par exemple dédié à la spécification des patterns. L'utilisation d'un langage existant, RSL, a permis aussi de formaliser certains patterns [6]. Dans Catalysis [37], des patterns appelés frameworks ont été spécifiés formellement en adaptant le projet existant. Il existe très peu de travaux concernant les composants de réutilisation avec le langage B.

Notre premier objectif est donc d'étudier les composants utilisés dans les approches semi-formelles pour concevoir des applications afin d'en sélectionner quelques-uns. Ces premiers exemples nous serviront ensuite de référence afin de spécifier formellement la notion de composant en B.

Notre deuxième objectif consiste ensuite à traduire en B un des composants sélectionnés et à le rendre réutilisable. L'idée est en effet de tout spécifier avec le langage B afin de rendre la réutilisation de composants formelle. Cette contrainte est primordiale si on souhaite vérifier ensuite la cohérence des spécifications obtenues par réutilisation.

Notre objectif final dans le cadre de ce rapport sera d'exhiber un exemple simple de réutilisation du composant spécifié en B. Ce travail préliminaire servira par la suite à la spécification et l'utilisation de nouveaux composants.

1.3 Organisation du rapport

Le rapport est séparé en deux parties principales : la première partie concerne l'état de l'art, tandis que la deuxième est un résumé et une analyse de notre contribution.

Pour commencer, nous nous intéresserons dans le chapitre 2 à la notion de composant dans la littérature et plus précisément à leur utilisation dans les langages orientés objet pour la conception d'application et à leur éventuelle spécification formelle. Pour illustrer cet état de l'art, nous présenterons quelques exemples d'utilisation et d'application.

Ensuite, dans le chapitre 3, nous essaierons de traduire la notion de composant dans le langage de spécification B. Nous utiliserons dans ce but un des exemples de composant objet étudiés dans l'état de l'art afin de le spécifier formellement en B. Nous chercherons ensuite à donner un exemple simple de réutilisation de ce composant. L'idée est de se restreindre au maximum au langage B. Cet exemple nous permettra ainsi de spécifier formellement les notions liées à la réutilisation.

Enfin, le chapitre 4 sera une conclusion de ce rapport. Nous ferons le bilan des apports de cette étude, en comparant notamment notre exemple avec les approches aperçues dans le chapitre 2. Nous terminerons, enfin, avec les perspectives et les remarques importantes de ce travail.

Chapitre 2

État de l'art sur la réutilisation de composants

2.1 Introduction

La réutilisation étant une discipline assez vaste, nous nous intéressons ici plutôt aux composants utilisés dans les approches objet pour concevoir des applications. Ce choix est justifié par le grand nombre d'exemples de composants existant dans ce domaine. L'utilisation des langages semi-formels pour les décrire rend leur spécification imprécise : nous étudierons aussi les solutions proposées pour spécifier formellement ces composants. Notre sélection de composants s'est portée sur les patterns (appelés parfois aussi des frameworks).

Le mot *pattern* signifie patron en français. Le premier exemple qui nous vient à l'esprit lorsqu'on évoque la notion de patron concerne la couture. Un patron est en effet un morceau de bois ou de toile sur lequel sont dessinées des formes de pièces de vêtements. Il indique les différentes proportions à considérer suivant les tailles désirées. Ainsi, un même patron permet de découper et de créer toutes les tailles d'une même pièce de tissu. Il suffit pour cela de choisir les “bonnes” mesures. Le patron est donc réutilisé de nombreuses fois. Dans la suite, on préfèrera le terme de “pattern” plutôt que “patron” pour conserver les mêmes conventions que dans les références.

Introduits par Christopher Alexander à la fin des années 1970 [3, 2], les patterns servirent dans le domaine de l'architecture à la construction d'une maison. L'architecte avait alors créé un *langage de patterns* dont chacun décrivait la manière de résoudre un problème particulier de la construction : la disposition des pièces, l'installation des papiers peints, du réseau de plomberie ou d'électricité, etc ... Son objectif était d'offrir à tout à chacun la possibilité de concevoir et de discuter directement avec le chef de projet des problèmes concernant sa maison grâce à ce langage. Christopher Alexander avait décrit dans ce but 253 patterns différents de manière narrative.

Si ce procédé ne s'est pas généralisé dans le domaine de la construction, les patterns ont pris une grande influence au sein de l'architecture informatique. Leur utilisation constitue en effet aujourd'hui un axe de recherche important concernant la conception et la réutilisation dans le développement logiciel. Ils apportent une solution générique pour les problèmes de conception et d'implémentation.

À travers la revue en détail de plusieurs articles, on se propose de faire la synthèse des méthodes actuelles retenues dans l'utilisation et la formalisation des patterns. Dans un premier temps, nous commencerons par définir la notion de pattern (paragraphe 2.2) qui demeure assez ambigüe. Ensuite, nous donnerons des définitions aux notions d'instanciation et de composition, qui sont liées au concept de réutilisation (paragraphe 2.3). Puis, nous poserons quelques exemples d'utilisation (paragraphe 2.4) de ces composants. Ensuite, nous exposerons et comparerons les différentes approches proposées concernant leur spécification (paragraphe 2.5) et enfin, nous conclurons par quelques pistes de recherche (paragraphe 2.6) au sujet de leur utilisation en B.

2.2 Patterns

2.2.1 Notion de pattern

Dans son article [7], Vincent Couturier définit les patterns comme "des abstractions de logiciels utilisées par des concepteurs et des programmeurs avancés dans leurs programmes". L'idée est de capitaliser un savoir-faire et d'offrir aux usagers un gain de temps et d'efficacité en proposant des solutions déjà testées et expérimentées pour des problèmes récurrents.

Les *patterns* sont en effet des descriptions de problèmes génériques régulièrement rencontrés par les programmeurs expérimentés, associées à des propositions de solutions. Ils constituent une sorte de mémoire universelle qui s'enrichit grâce à la constitution d'un catalogue et qui profite de l'expérience et de la réflexion de nombreux développeurs de logiciels. Présenté dans la langue naturelle, le pattern est un concept surtout utilisé dans les approches objet : la description en anglais du problème est ainsi complétée par des diagrammes OMT ou UML (voir [35] pour les conventions et notations utilisées) et par des extraits de code. D'un côté, cette forme de *langage* imprécis, informel et incomplet, parfois volontairement, favorise l'utilisation du pattern dans de nombreux cas, mais l'utilisation de la langue naturelle introduit facilement des équivoques. D'un autre côté, cette méthode est difficilement applicable en l'état actuel dans des langages de spécification formelle. Il est donc nécessaire d'étudier quelques exemples d'utilisation de patterns, avant de pouvoir les adapter au langage B. Ensuite, il faudra capitaliser les spécifications B, avant de se poser la question de la réutilisation.

2.2.2 Vers une classification des patterns

Types de Patterns
Analyse
Conception
Implémentation
Domaine

Il existe plusieurs types de patterns selon leur niveau d'abstraction. Certains s'appliquent plutôt à l'analyse, d'autres à la conception ou bien encore à l'implémentation du logiciel. Il existe en outre le pattern de domaine qui permet de décrire un domaine particulier et de donner une architecture adaptée. Enfin, chaque type de patterns peut lui-même être décomposé en plusieurs sous-types.

Les *patterns d'analyse et de conception* sont associés aux phases d'analyse et de conception orientées objet. La différence provient de leur moment d'utilisation dans le cycle de développement. Pendant la phase d'analyse, les patterns sont utilisés pour identifier des problèmes génériques concernant les applications et leur domaine, tandis qu'au moment de la conception, ils servent à spécifier des principes fondamentaux dans une architecture logicielle. De tels patterns sont essentiellement descriptifs et la solution y est peu développée. Les patterns d'analyse sont parfois appelés *patterns d'architecture*.

Les *patterns d'implémentation* permettent de définir les bons moyens de programmer et dépendent donc du langage de programmation. Une fois de plus, ces patterns sont descriptifs. Ils sont appelés aussi des *idiomes*.

Les *patterns de domaine* se distinguent des autres types : ils permettent de donner une architecture adaptée à un domaine de problème particulier et peuvent utiliser les autres types de patterns. La partie solution permet alors d'engendrer des solutions. Lorsque c'est le cas, un tel pattern est dit *génératif*.

Comme les composants que nous souhaitons spécifier en B concernent la phase de conception, nous nous intéresserons dans la suite de ce paragraphe aux patterns de conception.

2.2.3 Patterns de conception [20]

Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides, surnommés le "Gang of Four", ont repertorié et exposé dans leur livre [20] les principaux patterns de conception en adoptant une forme de description unique qui constitue en quelque sorte une carte d'identité de chaque pattern. Ces patterns, parfois appelés GoF Patterns, couvrent de nombreux problèmes comme le traitement d'objets composés ou bien la création de familles d'objets.

Pour rendre ce catalogue plus lisible, les patterns sont triés selon deux critères : leurs buts (création, structure ou comportement), et leurs champs d'action (classe ou objet). Le tableau 2.1 montre la classification des 23 patterns de conception décrits dans [20]. Le but décrit ce que fait le pattern. Les patterns de *création* concernent des processus de création d'objets. Les patterns de *structure* permettent de traiter des liens possibles entre les classes. Enfin, les patterns de *comportement* caractérisent les façons dont les classes et les objets interagissent entre eux. Le second critère de classification est le champ d'action des patterns. Les patterns de *classe* traitent surtout des problèmes de relations entre classes et sous-classes. Les patterns d'*objet* s'occupent des relations entre objets. Par exemple, le pattern *Composite* est dans la catégorie structure objet. Nous reviendrons sur cet exemple juste après la description des GoF patterns. Les patterns écrits en gras dans le tableau 2.1 seront utilisés dans la suite.

Description d'un pattern de conception

La description de patterns introduite par GoF [20] a un style déclaratif avec de nombreuses rubriques :

Nom et classification Nom et classe du pattern de conception.

Intention Que fait le pattern et quel est son but ?

Alias Autres noms utilisés pour désigner le pattern.

TAB. 2.1 – *Classification des GoF Patterns*

Champ d'Action	**But**		
	Création	**Structure**	**Comportement**
Classe	**Factory Method**	Adapter (classe)	Interpreter Template Method
Objet	**Abstract Factory** Builder Prototype Singleton	Adapter (objet) Bridge **Composite** Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Motivation Scénario d'utilisation du pattern de conception.

Indications d'utilisation Description des situations pour lesquelles le pattern peut être utilisé.

Structure Représentation graphique UML ou OMT des composants du pattern.

Constituants Classes et objets constituant le pattern.

Collaboration Description des collaborations entre constituants du pattern.

Conséquences Commentaires sur les effets du pattern appliqué au problème et comparaison par rapport aux objectifs.

Implémentation Conseils et astuces concernant l'implémentation (langage, difficultés, ...).

Exemple de code Fragments de code en C++ pour illustrer l'utilisation du pattern.

Utilisations Au moins deux exemples d'application du pattern dans des systèmes existants.

Patterns apparentés Liste des patterns reliés avec commentaires sur les différences et les ressemblances avec le pattern concerné.

Remarques

Cette description est un mélange de textes, de diagrammes et de code qui constitue une carte d'identité destinée à aider un concepteur à sélectionner dans le catalogue et appliquer le pattern correspondant à son problème. Ce catalogue s'adresse à n'importe quel concepteur. Le but est en effet de pouvoir réutiliser tel quel un pattern dans un cas concret. La classification apporte une première aide à la sélection du pattern recherché. Les rubriques "intention" et "motivation" permettent aux concepteurs de comprendre l'intérêt du pattern. Les "indications d'utilisation" sont utiles pour adapter le pattern au problème considéré. Enfin, la rubrique "patterns apparentés" dans la description nous montre qu'il existe non pas un unique pattern pour un problème donné, mais une *collection* de patterns reliés les uns aux autres. Les patterns sont donc des composants génériques qui peuvent s'appliquer à différents problèmes particuliers. Plutôt que de réutiliser directement du code, on réutilise ces éléments de conception qui demeurent éloignés des spécifications plus abstraites.

En revanche, ce mode mixte de description implique une lecture humaine et par conséquent, la recherche du pattern n'est pas automatisable par manque de formalisation. De plus, même lorsqu'un concepteur trouve une solution, l'application du pattern doit se faire avec la compréhension humaine, car les indications de cette description sont ambigües et parfois même très vagues. Enfin, le fait de se limiter à des éléments proches du code nous empêche de prouver des propriétés ou de repérer des erreurs.

2.2.4 Premiers exemples de GoF patterns

Terminons cet aperçu des GoF patterns par quelques exemples. Les patterns présentés dans ce paragraphe seront utilisés dans la suite.

Pattern Composite

Le pattern *Composite* sert à décomposer des objets "composites" en des structures arborescentes, en introduisant une hiérarchie entre composant et composé. Grâce à ce pattern, l'usager peut appliquer une opération sur un objet simple ou composite en toute transparence et peut donc gérer des compositions récursives d'objets. Une partie de sa description GoF est retranscrite dans l'annexe B.2. La structure du pattern *Composite* est donnée par la figure 2.1 (voir annexe A pour les notations graphiques). Les constituants de *Composite* sont :

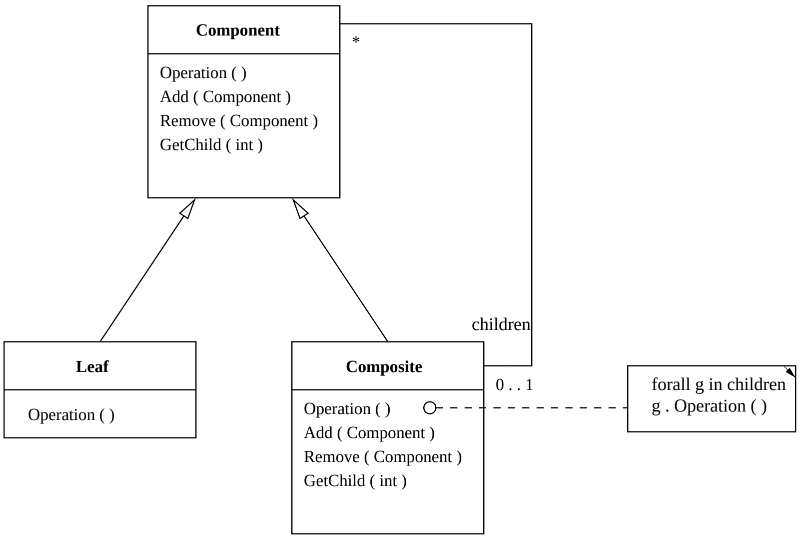

FIG. 2.1 – *Structure du pattern Composite*

- Component, qui déclare l'interface des objets dans la composition, qui implémente le comportement par défaut de l'interface commune à toutes les classes et qui définit enfin une dernière interface pour accéder aux composants fils et pouvoir les gérer.

– Leaf, qui représente les feuilles dans l'arbre de composition. Il n'a pas de fils et il déclare le comportement des objets simples.
– Composite, qui définit le comportement des composants ayant des fils et qui implémente les opérations les concernant dans l'interface Component.

Les clients utilisent alors l'interface de la classe Component pour interagir avec les différents objets de la composition.

Ce pattern est notamment utilisé dans l'article [31]. Le document fournit l'exemple (voir figure 2.2) d'un éditeur graphique utilisant la même structure que le pattern *Composite*. Il permet ainsi de traiter l'opération *Draw* dans des

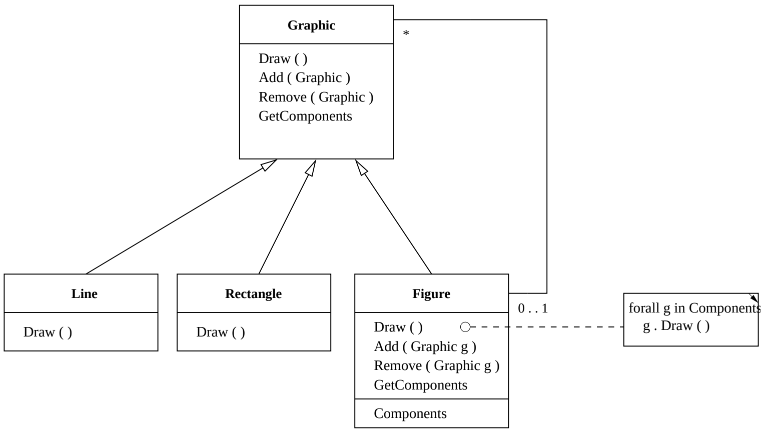

FIG. 2.2 – *Exemple : Editeur graphique*

figures composées de lignes et de rectangles.

Pattern Factory Method

Dans les approches objet, il n'est pas possible d'instancier une classe abstraite. Le pattern *Factory Method* apporte une solution à ce problème : il laisse une sous-classe créer un objet pour le compte d'une autre classe. Il permet donc à une classe de déléguer l'instanciation aux sous-classes. Une partie de sa description GoF est retranscrite dans l'annexe B.3. La structure du pattern *Factory Method* est donnée par la figure 2.3.

Afin de créer un objet de type *Product*, la classe abstraite *Creator* définit l'opération *AnOperation* qui permet d'appeler l'opération concrète *FactoryMethod* dans la sous-classe concrète ConcreteCreator. La version concrète de *FactoryMethod* permet de créer une instance de la classe concrète ConcreteProduct. Grâce au pattern, il est possible d'instancier une sous-classe concrète de la classe abstraite *Product*.

Pattern Abstract Factory

Le pattern *Abstract Factory* permet de factoriser des associations de type héritage entre des familles de classes et de sous-classes différentes. Quand des objets instanciés de classes différentes appartiennent à une famille commune, possédant le même type de propriétés par exemple, ou sont reliés entre eux et que

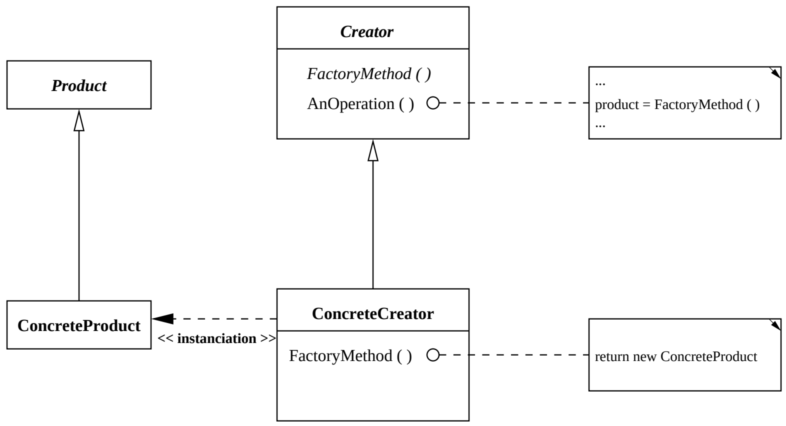

FIG. 2.3 – *Structure du pattern Factory Method*

leurs classes respectives ne sont pas des sous-classes d'une même classe abstraite, le pattern *Abstract Factory* permet de créer une classe abstraite commune dont une sous-classe concrète regroupe les différents objets de la même famille. Il crée en effet une interface pour des objets reliés ou dépendants d'une même famille sans spécifier leurs classes concrètes. Une partie de sa description GoF est retranscrite dans l'annexe B.1. La structure du pattern *Abstract Factory* est donnée par la figure 2.4. Le pattern est constitué de :

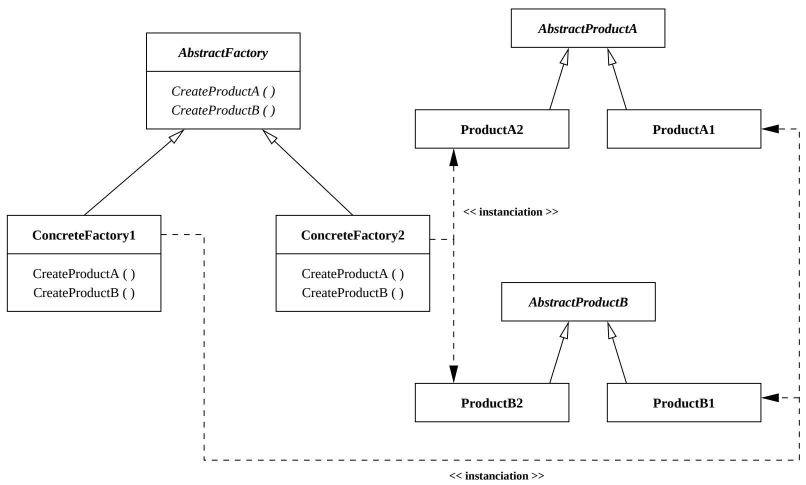

FIG. 2.4 – *Structure du pattern Abstract Factory*

- *AbstractFactory* : cette classe abstraite déclare une interface pour les opérations créant des objets de type *AbstractProductA* et *AbstractProductB*.
- ConcreteFactory1 et ConcreteFactory2 : ces deux sous-classes concrètes de *AbstractFactory* implémentent les opérations qui créent des instances de ProductA1, ProductA2, ProductB1 et ProductB2.

– *AbstractProductA* et *AbstractProductB* : ces deux classes déclarent une interface pour les deux types A et B des différentes classes Product.
– ProductA1, ProductA2, ProductB1 et ProductB2 : ces classes définissent les objets créés par les "factory methods" (les opérations *CreateProductA* et *CreateProductA*) des classes ConcreteFactory et elles implémentent les interfaces *AbstractProductA* et *AbstractProductA*.

Le pattern *Abstract Factory* est notamment utilisé dans l'article [36]. Les classes *ScrollBar* et *Window* ont pour sous-classes respectives OpenLookScrollBar et MotifScrollBar d'une part et OpenLookWindow et MotifWindow d'autre part, alors le pattern *Abstract Factory* donne comme solution la création d'une classe abstraite unique *WindowFactory* avec deux sous-classes OpenLookFactory et MotifFactory (voir la figure 2.5). La première sous-classe ne s'occupe

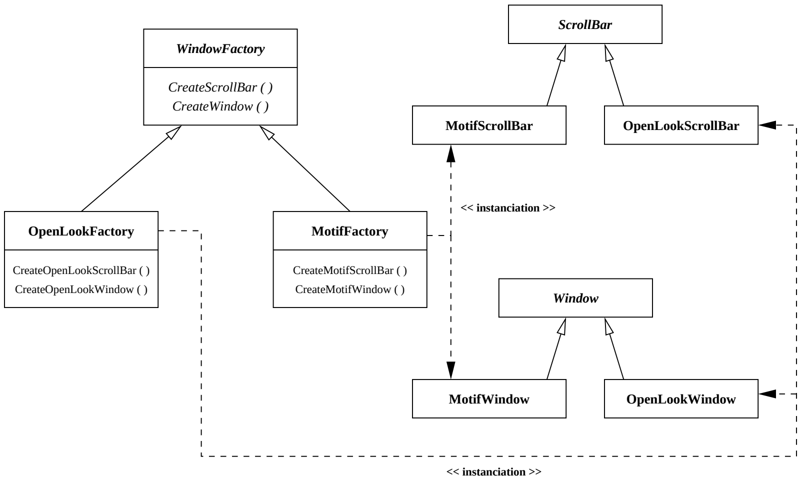

FIG. 2.5 – *Utilisation du pattern Abstract Factory*

que des objets OpenLook tandis que la seconde est destinée aux Motif.

En conclusion, les solutions présentées dans les patterns de conception permettent de résoudre des problèmes récurrents de conception, comme le traitement d'objets composites, en proposant un modèle de diagramme UML (ou OMT) qu'il est possible d'adapter et de modifier selon les cas.

2.3 Instanciation et composition de patterns

Avant de poursuivre, nous devons définir deux notions importantes qui concernent ces composants : l'instanciation et la composition.

En général, une *instance* est un exemple concret d'un élément générique. Par exemple, dans les approches objet, un objet est une instance de classe. Si on considère le composant comme un modèle générique, une instance est un cas concret de ce composant. L'*instanciation* du pattern ou du framework est un mécanisme qui consiste à concrétiser le composant. Elle permet notamment de renommer les différents éléments du composant, comme les classes, les opérations ou les associations. Mais ce n'est pas tout : il est possible d'adapter, de rajouter ou de modifier certains éléments. Par exemple, dans le paragraphe 2.2.4, nous avons présenté le pattern *Composite* (figure 2.1) et un exemple d'utilisation du pattern, l'éditeur graphique (figure 2.2). Dans les approches objet, le diagramme de l'éditeur graphique est considéré comme une instanciation du diagramme de la structure Composite. Les classes ont été renommées, l'opération générique est devenue *Draw* et la sous-classe Leaf a été dupliquée et renommée plusieurs fois. L'éditeur graphique est un cas concret du modèle *Composite*.

Lorsque les composants sont instanciés, il est intéressant de pouvoir les combiner entre eux. La *composition* est un mécanisme qui permet de créer des liens entre différents patterns. Dans ce cas, au moins une classe ou un objet est commun aux deux composants composés. Par exemple, la figure 2.6 nous donne un exemple de composition entre deux diagrammes. La classe Personne est ainsi commune aux deux diagrammes composés.

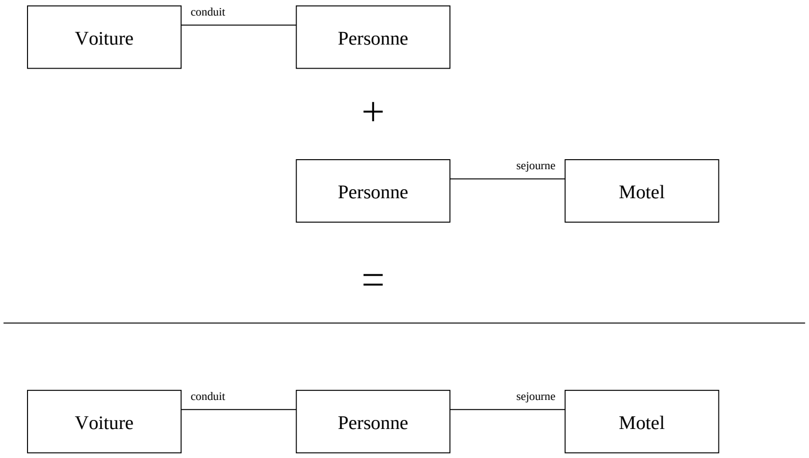

FIG. 2.6 – *Composition de deux composants*

Ces deux notions sont fondamentales dans la réutilisation, car elles permettent d'adapter et d'utiliser les différents composants. C'est pourquoi nous nous sommes intéressés aux travaux de recherche qui spécifient formellement ces deux mécanismes.

2.4 Exemples d'utilisation de patterns

Une fois cette rapide revue en détail de la notion de pattern terminée, revenons désormais à notre problème concernant la spécification de composants réutilisables. Afin d'illustrer les intérêts d'une telle recherche, considérons maintenant les deux exemples suivants.

Le premier exemple est intéressant car il est très proche de notre but. Afin de résoudre un problème de contrôle d'accès, les auteurs proposent d'adopter deux niveaux de représentations complémentaires en utilisant le langage semi-formel UML et le langage formel B. L'idée est d'utiliser ces deux points de vue pour spécifier formellement le problème.

Le second exemple concerne des patterns utilisés dans le projet Catalysis : dans ce projet, les patterns sont appelés des “frameworks”. L'exemple est intéressant car il permet d'illustrer l'instanciation d'un framework paramétré par un exemple simple, avec une approche graphique qui est un moyen efficace de ne pas manipuler du code ni même d'utiliser des spécifications abstraites. Cet exemple montre bien l'intérêt des composants réutilisables.

2.4.1 Utilisation de patterns en B [30]

L'idée de cet article est d'instancier les patterns en UML puis de les traduire en B. La démarche proposée est la suivante. La conception du logiciel s'effectue en trois étapes. Dans un premier temps, on analyse le cahier des charges afin de définir un style architectural : le résultat est l'identification des composants du système. Ensuite, on utilise des patterns de conception pour définir ces composants. Enfin, on intègre les différentes définitions générées par l'instanciation des patterns. La première étape s'effectue avec UML et la deuxième avec le langage B. Cette proposition de méthode est accompagnée par un exemple, le contrôle d'accès.

Le cahier des charges du problème du contrôle d'accès est le suivant. On souhaite gérer un système qui permet de contrôler et de vérifier les accès des différentes personnes autorisées à entrer dans les bâtiments. L'accès à un bâtiment est accordé selon l'appartenance d'une personne à un groupe, chaque groupe étant autorisé à rentrer dans certains bâtiments seulement. Le passage d'un bâtiment à un autre s'effectue à travers un point de passage en sens unique muni d'un lecteur de cartes, d'une porte et d'un timer. Le passage et l'ouverture d'une porte sont détectés par le système. Deux voyants sont installés au niveau du lecteur : un rouge en cas de refus d'accès et un vert en cas de blocage ou de déblocage de la porte. Le protocole d'accès est le suivant : la personne insère sa carte dans le lecteur et deux scénarios sont alors possibles. Soit cette personne est autorisée à entrer dans le bâtiment et dans ce cas, le voyant vert s'allume et la porte est débloquée pendant 30 secondes, soit la personne n'est pas autorisée, alors le voyant rouge s'allume pendant 2 secondes et la porte reste bloquée. Lorsque la porte se débloque, elle se rebloque et le voyant vert s'éteint soit tout de suite après le passage de la personne, soit au bout des 30 secondes accordées. En outre il est demandé au système de pouvoir rendre compte à tout instant de la présence des différentes personnes dans les différents bâtiments, ainsi que de pouvoir débloquer toutes les portes lorsque les capteurs d'incendie déclenchent une alerte.

Première étape : La lecture et l'analyse du cahier des charges permettent aux concepteurs de déterminer un style architectural. Le résultat de cette recherche est la description des différents composants et leurs liens. Dans notre cas, le style choisi est basé sur un *système distribué*. Il est divisé en trois parties :

- CONTROLEUR : ce composant s'occupe du contrôle et de la supervision des accès. Il spécifie le protocole et correspond au cœur du système à développer.
- EQUIPEMENT : il gère les capteurs, lecteurs, portes, timers, ... Ces équipements sont les périphériques du système.

– COMMUNICATION : ce dernier composant gère les communications entre les deux autres.

Une fois cette courte description terminée, la deuxième étape consiste justement à définir les composants.

Deuxième étape : L'utilisation des patterns entre en jeu à ce moment. Le but est d'associer à chaque composant un pattern de conception puis de le spécifier en B. Le problème revient à déterminer les patterns qui correspondent le mieux à nos composants.

En ce qui concerne COMMUNICATION, les patterns *Client-Serveur* et *Emetteur-Récepteur*, que nous ne détaillerons pas, se rapprochent du problème. Le premier permet de décrire un système standard de communication distribuée, avec un serveur qui fournit ses services à un ensemble de clients. Le second décrit une communication par envoi de messages, où chaque correspondant dispose de son émetteur et de son récepteur. Si on considère les messages des équipements comme des demandes de service au contrôleur, il semble intéressant de choisir le pattern *Client-Serveur*. Dans un premier temps, on instancie ce pattern en UML avec les classes **Controleur** et **Equipement** reliées entre elles par la classe-association **Communication**. Cette classe a un attribut `msg-communique` qui est un ensemble de messages (service ou réponse). **Controleur** et **Equipement** interagissent via `demander-service` et `retourner-resultat`. Ensuite, le pattern est spécifié en B à l'aide de quatre machines abstraites : **Communication**, **Controleur**, **Equipement** et **Equipement-Controleur-Interface**. Il existe en effet des règles de traduction [33] pour spécifier un diagramme UML en B. La première machine spécifie les opérations de communication entre les deux suivantes qui décrivent la structure et le comportement des objets associés, comme par exemple les opérations `executer-service` ou `traiter-resultat`. Enfin, la dernière machine inclut les trois autres et supervise les opérations. La figure 2.7 montre les deux spécifications successives du pattern *Client-Serveur* en UML et en B.

Le composant EQUIPEMENT est spécifié à l'aide du pattern *Composite*. Ce choix est justifié par le fait que les bâtiments et les points de passage sont composés de plusieurs capteurs et que les messages doivent être traités de manière transparente que ce soit pour de simples capteurs ou bien pour des portes. En se référant à la structure du pattern décrite dans le paragraphe 2.2.4, la spécification UML est composée de huit classes, dont :

– **Equipement** qui est une instanciation de Component,
– **Porte**, **Lecteur**, **Timer**, **Capteur** des instanciations de Leaf,
– **Batiment**, **Point-de-Passage**, des instanciations de Composite,
– et enfin **Equip-Compose** qui généralise les deux derniers Composites.

Par suite, il est possible de spécifier ces classes en langage B. La figure 2.8 est le graphe de dépendance des machines abstraites B du pattern *Composite*.

Le dernier composant, CONTROLEUR, n'a pas pu être spécifié par cette méthode car les patterns ne recouvrent pas tous les problèmes. De nombreux patterns ont été sélectionnés mais aucun ne s'appliquait aux fonctions du composant. La conception a alors été plus classique. La figure 2.9 est le graphe de dépendance des machines abstraites B choisies pour spécifier le composant CONTROLEUR.

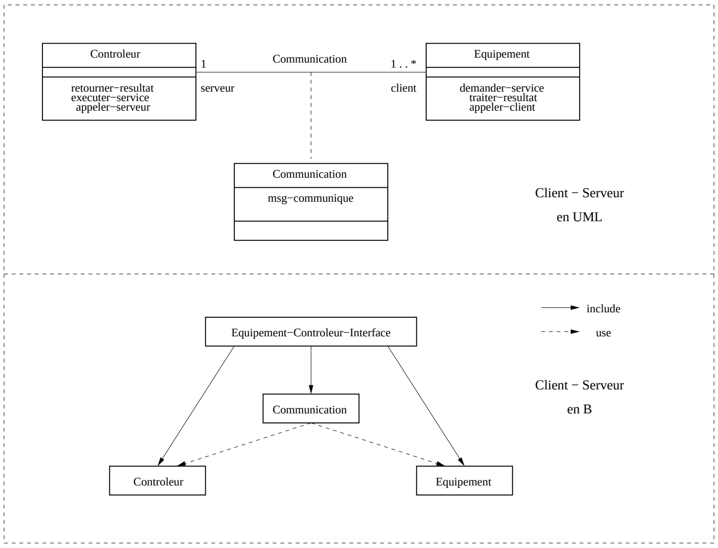

FIG. 2.7 – *Spécifications UML et B du pattern Client-Serveur*

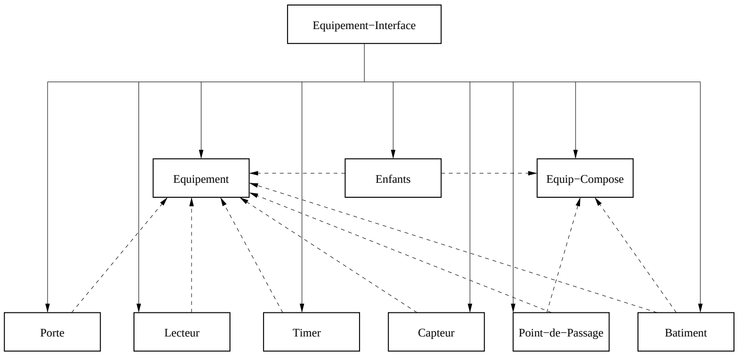

FIG. 2.8 – *Instanciation en B du pattern Composite*

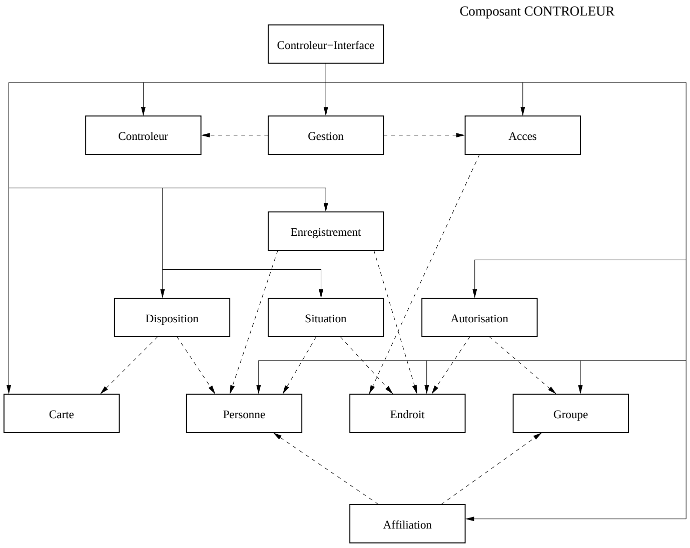

FIG. 2.9 – *Graphe des machines abstraites B du composant CONTROLEUR*

Dernière étape : Les trois composants sont alors regroupés ensemble. Par exemple, les classes **Controleur** et **Equipement** des composants respectifs CONTROLEUR et EQUIPEMENT ont été complétées par les opérations de COMMUNICATION. L'intégration finale a abouti à 25 machines abstraites en B. La spécification ainsi obtenue couvre la presque totalité du cahier des charges. Cependant, par restriction du langage B, les contraintes de temps réel ne peuvent pas être prises en compte. Enfin, les obligations de preuve générées ont toutes été prouvées, dont 91% automatiquement par le prouveur de B. Les autres preuves correspondent à la partie spécifiée sans pattern.

Remarques et conclusion : La double utilisation du langage semi-formel UML d'une part et du langage formel B d'autre part est intéressante. UML permet en effet de conserver une conception graphique facile à comprendre et d'instancier les patterns. La traduction des diagrammes UML est automatique et la spécification finale peut être validée par les outils de B. Cette méthode est donc plus formelle que la simple utilisation des patterns grâce à l'intégration du langage B. Cependant, cette double utilisation nécessite le passage d'une configuration à l'autre tout au long du projet. De plus, la combinaison entre les différents composants se fait actuellement à la main. Enfin, seuls quelques exemples d'utilisation de patterns ont été spécifiés. Le projet reste donc à développer. Leur but est ensuite de raffiner les exemples de patterns traduits en B, pour en déduire des liens entre eux. L'idée consiste en fait à utiliser des sortes de patterns de “raffinement”.

Mais ce projet est différent du nôtre, puisque les auteurs passent par UML pour instancier les patterns, avant de les traduire en B. En revanche, notre but est de tout spécifier en B : le composant générique, l'instanciation et le résultat final. L'avantage de notre projet est de pouvoir utiliser tous les outils de preuve et de raffinement disponibles en B. En outre, on espère ensuite pouvoir instancier ces composants puis les combiner entre eux à notre guise et ce, autant de fois que possible.

2.4.2 Utilisation de frameworks [37]

Cet exemple concerne les frameworks. La figure 2.10 est la représentation graphique conseillée par [37] concernant les frameworks. Cette notation est

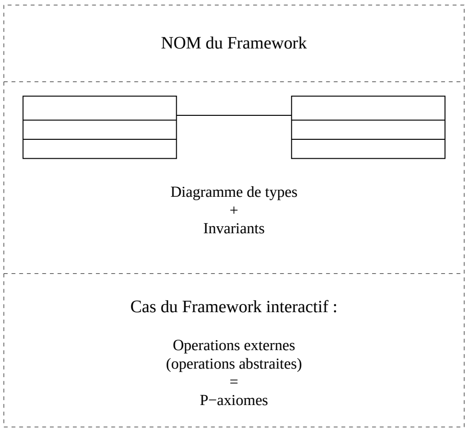

FIG. 2.10 – *Notation graphique d'un Framework*

composée de trois parties. La première partie indique le nom du framework, et par conséquent son but. La seconde boîte contient un diagramme de types : il représente les liens entre les différents types composant le framework. Le type fait référence ici aux classes des approches objet. Les instances d'une même classe sont en effet regroupées par type. Les propriétés vérifiées par les objets du framework sont exprimées sous forme d'invariant dans cette même boîte. Enfin, une dernière partie concerne les opérations abstraites externes et les axiomes correspondants (on les appelle des P-axiomes). Cette forme de présentation est suggérée par les spécifications formelles adoptées pour les frameworks, que nous détaillerons dans le paragraphe 2.5.4.

Considérons le cas d'une allocation de ressource. Ce genre de problème étant fréquent, il paraît intéressant de spécifier un tel framework de manière générique afin de pouvoir l'instancier en temps voulu. La figure 2.11 est le framework "ouvert" correspondant. L'ouverture correspond à une paramétrisation. Dans ce graphe, les rectangles à bords arrondis sont justement des paramètres du framework. Sans rentrer dans les détails, les invariants indiqués en bas du diagramme concernent les types JobOccurrence et Resource. Par exemple, le premier signifie que pour tout objet x de JobOccurrence, si la multiplicité de l'association *allocated* n'est pas nulle, alors l'ensemble des objets de la forme $x.allocated.provides$ est inclus dans l'ensemble des objets de la forme $x.job.requirements$. Le problème est le suivant : on dispose de plusieurs occurrences de tâche et de plusieurs ressources. Chaque occurrence de tâche correspond à une certaine catégorie de tâche. Chaque catégorie de tâche demande un certain nombre de moyens de ressource et chaque ressource utilise ces moyens. Le but est d'allouer les ressources aux occurrences de tâches. Le premier inva-

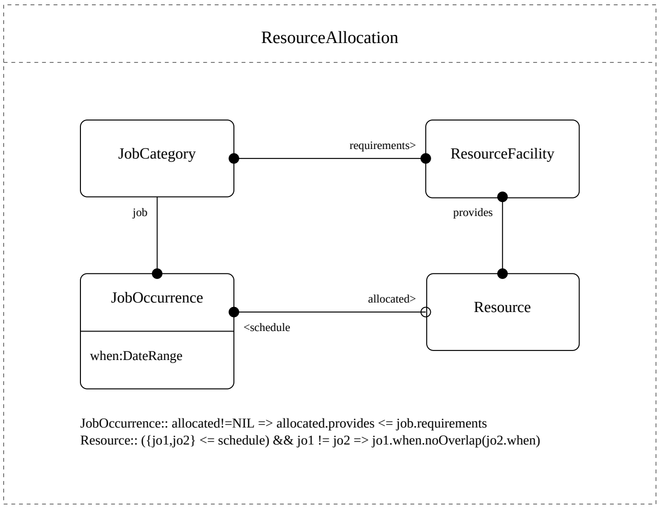

FIG. 2.11 – *Framework ResourceAllocation*

riant signifie alors que si une ressource est allouée à une occurrence de tâche, alors les moyens de ressource utilisés par cette allocation font partie des moyens demandés par la catégorie de tâche concernée. Le deuxième invariant permet d'éviter que deux occurrences ne se chevauchent.

Si on désire gérer un emploi du temps dans le domaine de l'éducation, on dispose de professeurs et de salles de cours. Chaque professeur est qualifié pour certains types de cours et certains cours exigent certaines connaissances. Chaque cour est alloué à une salle et les différentes salles de cours disposent de matériels divers suivant leur utilité. Il s'agit bien d'un problème d'allocation. Cependant, il est composé de deux problèmes combinés : l'un concernant les salles et l'autre les professeurs. On instancie alors deux fois le framework *ResourceAllocation* comme indiqué dans la figure 2.12. Et la figure 2.13 nous donne enfin le résultat de la composition de cette double instanciation. Les invariants associés sont spécifiés en fonction du renommage effectué.

En conclusion, par ce petit exemple, on montre la puissance des opérations d'instanciation et de composition. La réutilisation du framework prédéfini permet en effet de résoudre le problème d'emploi du temps. Les notations et la formalisation utilisées seront en partie détaillées et expliquées dans la suite. Ces composants ont l'avantage d'être de type boîte noire. Ils ne nécessitent pas en particulier la connaissance du code. Intéressons-nous maintenant aux spécifications de ces composants.

2.5 Spécification formelle

Les approches proposées dans le paragraphe précédent se limitaient à des exemples. Mais de nombreux chercheurs ont essayé de spécifier formellement les patterns ou des entités proches des patterns. Trois exemples sont étudiés :

– la création d'un langage de spécification : LePUS [10, 9, 14, 11, 22, 13, 12],
– la spécification des patterns avec RSL [6, 19, 34, 18, 5, 4],

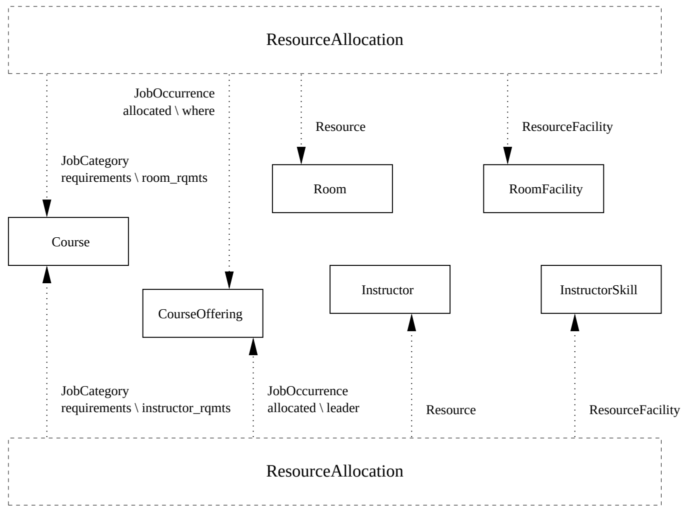

FIG. 2.12 – *Double Instanciation de ResourceAllocation*

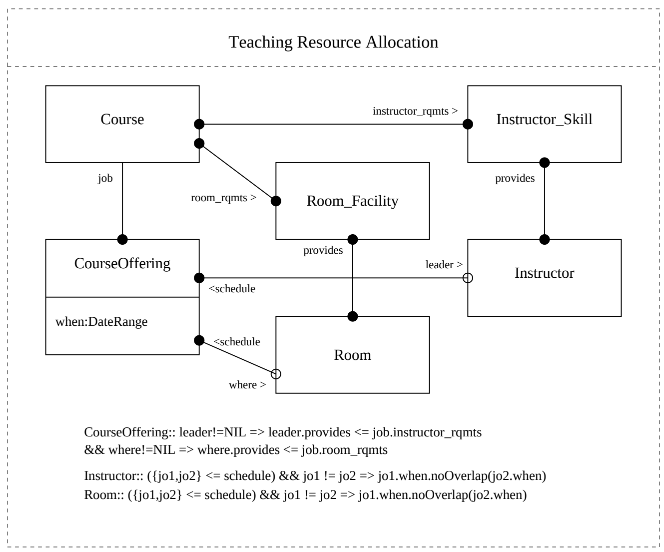

FIG. 2.13 – *Framework TeachingResourceAllocation*

– la spécification des frameworks avec des théories multi-sortes [37, 25, 24, 27, 26, 17, 16, 15, 8, 28].

Ces différentes approches seront commentées relativement à notre projet de spécification de composants réutilisables en B.

Les premières critiques concernant le manque de formalisation des patterns ou plus généralement des composants génériques se retrouvent dans les trois exemples étudiés. Elles sont à l'origine des choix et des hypothèses qui ont permis d'aboutir à ces spécifications. Avant de s'intéresser à ces propositions, exposons les principales remarques émises par les différents auteurs au sujet des patterns et de leurs défauts.

2.5.1 Remarques sur les patterns

Le principal défaut des patterns et des frameworks provient de l'utilisation des graphes semi-formels, ce qui rejoint notre point de vue. Tous les auteurs s'accordent sur le manque et le besoin de formalisation de ces composants, afin de pouvoir les utiliser, les composer et les vérifier.

En ce qui concerne la description GoF, le mode de présentation est remis en cause et un intérêt particulier est apporté à la partie solution. Amnon Eden (LePUS) introduit les notions de "lattice" et de "leitmotif" d'un pattern. Un "lattice" est la solution préconisée par le pattern, tandis qu'un "leitmotif" est un lattice associé à la raison ou la situation pour laquelle le pattern est utile et adapté. De même, Richard Moore (RSL) fait remarquer qu'un GoF pattern est découpé en quatre parties essentielles : le nom du pattern, le problème, la solution et les conséquences. L'idée est de se focaliser sur la partie solution. Les parties problème et conséquences sont en effet trop vagues pour être formalisées. Si on essayait de le faire, on se retrouverait alors à faire abstraction de l'ensemble de tous les problèmes existants et à classer le monde entier en des catégories : bien que ce soit possible sur la forme, cela n'aurait aucun sens sur le fond. Cela reviendrait à généraliser les problèmes sans en distinguer les nuances et les variantes. En revanche, si on formalise les solutions du catalogue GoF, il sera alors possible de les instancier en fonction des problèmes rencontrés.

Le manque de formalisation ne provient pas que des diagrammes : la langue naturelle est source d'erreurs et prête à confusion. Amnon Eden a analysé les différents GoF patterns et en a classé une partie suivant six catégories selon leur degré d'interprétation. Le tableau 2.2 représente l'ensemble de ces catégories avec quelques exemples. Si les quatre premiers cas ne posent pas trop de problèmes, les deux derniers sont en revanche plus difficiles voire impossibles à résoudre. Le manque d'information est trop grand pour être complété sans dénaturer la fin du pattern. Toute formalisation de ce type de composants introduit une spécialisation et retire donc une partie de ses aspects génériques. Etudions maintenant les trois approches de formalisation.

2.5.2 LePUS : un langage de spécification des patterns

Cette partie s'appuie sur les articles [10, 9, 14, 11, 22, 13, 12]. Le langage de patterns apporté par Alexander [3] n'est pas un langage dans le sens courant de la programmation mais plutôt un mélange entre structuration et langage naturel. De même, la description des GoF patterns n'est pas une formalisation

TAB. 2.2 – *Interprétation des GoF Patterns*

	Catégorie	Exemples de GoF patterns
1	Description précise	1.DECORATOR 2.VISITOR : ConcreteElement
2	Enumération d'alternatives	1.FACTORY METHOD : Creator 2.STRATEGY : Collaborations 3.DECORATOR : Collaborations
3	Généralisation précisée	1.VISITOR 2.DECORATOR
4	Termes techniques mais ouverts à plusieurs interprétations	1.PROXY : "cache" 2.PROTOTYPE : "cloning" 3.MEMENTO : "internal state"
5	Description floue, informelle ou équivoque	1.ADAPTOR : Adapter 2.COMPOSITE : Component 3.OBSERVER : Collaborations
6	Omission délibérée de détails	1.STATE

mais plutôt un mélange entre texte, diagrammes et code. L'idée [10] est la création d'un langage formel spécifique aux patterns de conception : LePUS.

Langage LePUS

Le langage LePUS (LanguagE for Patterns Uniform Specification) est basé sur des aspects de la logique monadique d'ordre supérieur (HOML : Higher Order Monadic Logic, [38]). LePUS propose de plus une notation graphique pour représenter les formules logiques. Contrairement à UML, les patterns peuvent être ainsi représentés par des diagrammes complets, sans équivoque. L'autre avantage de cette méthode est la possibilité de travailler aussi bien sur des formules que sur des diagrammes puisqu'ils sont équivalents. Afin de mieux comprendre les bases mathématiques de LePUS, introduisons les éléments de spécification de ce langage en se référant aux éléments constituant un pattern.

Représentation des classes et des fonctions Les deux sortes principales de LePUS sont les *classes* $\mathbb{C}$ et les *fonctions* $\mathbb{F}$ qui forment les *atomes* de notre univers $\mathbb{U}$ de discussion. Un programme est alors représenté par un ensemble d'entités de base (les atomes) et de relations entre elles (comme par exemple : c est une classe, f est une fonction, c_1 hérite de c_2, f crée un objet de c, c est le premier argument de f, ...). Les différentes interactions et associations entre les constituants d'un pattern sont alors formalisées à l'aide de conjonctions de *prédicats* sur les relations et les ensembles.

Relations LePUS prédéfinies Les classes et les fonctions forment la base de notre univers. Elles sont considérées ici comme des éléments irréductibles. Cependant, pour tenir compte de ce qu'en conception orientée objet on appelle des composants ou des membres d'une classe, on utilise en LePUS des *relations*. Par exemple, les arguments ou le type retourné par une fonction sont vus comme des relations sur des classes. Ces relations prédéfinies sont destinées à représenter la structure et les comportements des patterns. En particulier, $Abstract(c)$ est une relation de base qui indique si c est une classe abstraite. $Creation(f,c)$ indique que la fonction f crée un objet de la classe c, tandis que la relation $Production(f,c)$ indique que f produit un objet de c, autrement dit, f est une fonction qui crée et retourne un objet. D'autres relations du même type sont prédéfinies comme : $DefinedIn$, $Invocation$, $Inherit$, $ReturnType$. LePUS définit ensuite des *prédicats* sur les relations et les ensembles d'atomes.

Intérêt de la logique d'ordre supérieur Certaines propriétés des patterns ne pourraient pas être prises en considération dans des prédicats du *premier ordre* : il ne serait pas possible, par exemple, d'utiliser la relation $Creation$ introduite ci-dessus pour spécifier un ensemble de classes dont les objets seraient créés par un ensemble de fonctions. Or les ensembles d'ordre supérieur sont fréquents dans les conceptions orientées objet. Si on considère une hiérarchie comme un ensemble de classes et qu'une classe contient un ensemble de fonctions, un ensemble de hiérarchies n'est autre qu'un ensemble d'ensembles de classes ou bien un ensemble d'ensembles d'ensembles de fonctions. D'où la nécessité de se fonder sur une logique d'ordre supérieur comme HOML.

Ensembles d'ordre supérieur particuliers Les *hiérarchies* forment un cas particulier des héritages de classes. Une hiérarchie est composée d'une classe abstraite de base appelée racine et d'un ensemble de classes qui héritent (directement ou pas) de cette racine. Par exemple, l'ensemble des classes *ConcreteCreator* dans le pattern *Factory Method* (voir figure 2.3) est une hiérarchie dont la racine est la classe abstraite $Creator$. Par convention, nous désignerons dans la suite cette hiérarchie par $Creators$. De même, les classes $ConcreteProduct$ et la classe abstraite $Product$ forment une hiérarchie dans ce même pattern. Notons-la $Products$.

Un *clan* est un cas particulier d'ensemble de fonctions. Il s'agit d'une famille de fonctions, dont les signatures (i.e. les types) sont les mêmes, associées à des classes différentes. Par exemple, les fonctions $FactoryMethod$ forment un clan dans le pattern *Factory Method* par rapport à l'ensemble de classes $ConcreteCreator$, on note ce clan $FactoryMethods$.

Enfin, les *tribus* sont des ensembles de clans qui se réfèrent à un même ensemble de classes.

Propriétés sur les relations LePUS Les relations peuvent être vues comme des ensembles : dans ce cas, la relation $r(u)$ équivaut à $u \in r$. La relation R est dite *totale* sur les ensembles U et V si pour tout élément u de U, il existe un $v \in V$ tel que $(u,v) \in R$. Certaines relations entre ensembles peuvent être bijectives. Par exemple, dans le pattern *Factory Method*, chaque fonction $FactoryMethod$ crée un objet d'une classe dans la hiérarchie $Products$ définie auparavant. Dans ce cas, si on souhaite caractériser les liens entre $Products$ et $FactoryMethods$, la relation $Production$ définie au début de ce paragraphe est régulière sur les ensembles $FactoryMethods$ et $Products$. Ces propriétés de totalité et de régularité sont spécifiées à l'aide de prédicats. $Total$ est un prédicat ternaire : $(R,u,v) \in Total$ si et seulement si la relation $R(u,v)$ est totale. De même, on définit le prédicat $Isomorphic(R,u,v)$ dans le cas où $R(u,v)$

est régulière.

La *commutativité* permet d'indiquer que deux relations régulières ont le même rang, autrement dit, qu'elles sont égales sur les mêmes ensembles. Par exemple, en ce qui concerne le pattern *Factory Method*, deux relations régulières sont définies sur $FactoryMethods$ et $Products$. Nous avons déjà aperçu la première qui est :

$$Production^{\leftrightarrow}(f,p)$$

qui indique que les fonctions $f \in FactoryMethods$ créent et retournent un objet de la classe $p \in Products$. La double flèche $\leftrightarrow$ signifie que la relation est régulière. D'autre part, nous avons :

$$ReturnType^{\leftrightarrow}(f,p)$$

qui signifie que le type retourné par la fonction f est de la classe p. De plus, on souhaite que les deux relations coïncident sur les couples (f,p) des ensembles $FactoryMethods$ et $Products$. On dit alors que les deux relations commutent dans $FactoryMethods$ et $Products$. Dans ce cas, pour un f donné, f produit un objet de la classe p_1 et retourne un type de p_2 tels que $p_1 = p_2$.

Patterns de conception

Une dizaine de GoF patterns ont été spécifiés à l'aide de LePUS. La démarche est la suivante : dans un premier temps, on complète la description GoF avec des hypothèses supplémentaires afin de préciser le pattern, puis on spécifie la solution du pattern en LePUS avec une série de prédicats.

Prenons l'exemple du pattern *Factory Method*. La structure du pattern est représentée par la figure 2.3. Le tableau 2.3 est la traduction du pattern décrit classiquement en des propriétés qualifiées d'implicites. Ces propriétés s'expriment facilement à l'aide de prédicats LePUS. Certaines propriétés ont

TAB. 2.3 – *Propriétés du pattern Factory Method*

Constituant	Propriété implicite
1.$ConcreteCreator$	Chaque classe concrète hérite de $Creator$.
2.$ConcreteProduct$	Chaque classe concrète hérite de $Product$.
3.$FactoryMethod$	Une méthode abstraite est définie dans $Creator$ et dans chaque sous-classe.
4.$FactoryMethod$	Chaque version concrète de $FactoryMethod$ crée une instance d'un $ConcreteProduct$ correspondant.

été déjà vues dans le paragraphe précédent.

La propriété 1. se traduit par le fait que la classe abstraite $Creator$ et ses sous-classes $ConcreteCreator$ forment une hiérarchie :

$$\frac{Creators : \mathbb{P}(\mathbb{C})}{Hierarchy(Creators)}$$

où $\mathbb{P}$ indique que *Creators* est un ensemble de classes. De même pour 2. avec la hiérarchie de *Product* et de ses sous-classes *ConcreteProduct* :

$$\frac{Products : \mathbb{P}(\mathbb{C})}{Hierarchy(Products)}$$

La troisième condition indique que les fonctions *FactoryMethod* partagent la même signature et que chacune est définie dans des sous-classes différentes de la hiérarchie *Creators*. Un tel ensemble de fonctions est un clan. Cette propriété se traduit en LePUS par :

$$\frac{FactoryMethods : \mathbb{P}(\mathbb{F})}{Clan(FactoryMethods,Creators)}$$

La dernière condition implique que chaque *FactoryMethod* crée exactement une instance d'une classe de la hiérarchie *Products*. La relation entre *FactoryMethods* et *Products* est donc régulière :

$$Creation^{\leftrightarrow}(FactoryMethods,Products)$$

Si on souhaite plus de précisions, on utilise la relation de *Production*. Les liens de bijection entre les ensembles *FactoryMethods* et *Products* s'interprètent par :

1. les objets de chaque classe *Product* sont produits par une seule fonction *FactoryMethod*,
2. le type retourné par chaque *FactoryMethod* est égal à la classe *Product* de l'objet créé.

La première condition est exprimée par :

$$Production^{\leftrightarrow}(FactoryMethods,Products)$$

Pour spécifier 2., on rajoute :

$$ReturnType^{\leftrightarrow}(FactoryMethods,Products)$$

Cela ne suffit pas, car ni *Production*, ni *ReturnType* ne force que le retour du type de chaque *FactoryMethod* soit égal au type des objets créés. On utilise alors la commutativité :

$$Commute_{ReturnType,Production}(FactoryMethods,Products)$$

Les propriétés du pattern *Factory Method* sont donc spécifiées par la formule LePUS suivante :

$$\frac{\begin{array}{l} Creators : \mathbb{H} \\ Products : \mathbb{H} \\ FactoryMethods : \mathbb{P}(\mathbb{F}) \end{array}}{\begin{array}{l} Clan(FactoryMethods,Creators) \wedge \\ ReturnType^{\leftrightarrow}(FactoryMethods,Products) \wedge \\ Production^{\leftrightarrow}(FactoryMethods,Products) \wedge \\ Commute_{ReturnType,Production}(FactoryMethods,Products) \end{array}}$$

où $\mathbb{H}$ désigne les hiérarchies.

Finalement, cet exemple permet de se rendre compte comment une simple phrase écrite dans la description GoF peut être traduite. Seuls quatre prédicats ont suffi à formaliser le pattern :

1. *FactoryMethod* est un clan dans la hiérarchie *Creators*,

2. $ReturnType^{\leftrightarrow}(FactoryMethods, Products)$,
3. $Production^{\leftrightarrow}(FactoryMethods, Products)$,
4. la relation de commutativité sur $ReturnType^{\leftrightarrow}$ et $Production^{\leftrightarrow}$.

Les aspects comportementaux des patterns sont facilement représentés par des relations, même si le manque de précision du GoF catalogue, inévitable en raison de la description semi-formelle, pose parfois problème.

Relations entre patterns

Le but d'une telle démarche est de définir des relations entre patterns, comme "un pattern généralise un autre", "X utilise Y" ou "X est un composant de Y". L'avantage de LePUS est l'utilisation d'une logique d'ordre supérieur. Elle est nécessaire si on souhaite tenir compte d'ensemble d'ensembles. Une logique du premier ordre permet à des variables de se comporter comme des éléments de base. Dans une logique du second ordre, une variable peut tout aussi bien être une relation, un élément ou un ensemble de ces derniers. Une formule d'ordre k contient enfin des variables qui représentent des ensembles d'ordre k ou des relations. On se restreint à des variables sur des ensembles et non sur des relations. Une telle logique est dite *monadique.*

Sans rentrer dans les détails, LePUS est un langage basé sur le calcul des prédicats. Le tableau 2.4 contient la liste des prédicats disponibles que nous avons aperçus dans le paragraphe LePUS. Ils sont tous définis sur des ensembles d'ordre supérieur. Un pattern est représenté par une formule LePUS, i.e. une conjonction de prédicats de la forme :

TAB. 2.4 – *Liste des prédicats LePUS*

Symbole	Domaine
$Hierarchy$	$\mathbb{P}(\mathbb{C})$
$Clan$	$\mathbb{P}^n(\mathbb{F})\times\mathbb{P}^n(\mathbb{C})$
$Tribe$	$\mathbb{P}^{n+1}(\mathbb{F})\times\mathbb{P}^n(\mathbb{C})$
$Total$	$\mathbb{R}\times\mathbb{P}^m(\mathbb{U})\times\mathbb{P}^n(\mathbb{U})$
$Isomorphic$	$\mathbb{R}\times\mathbb{P}^m(\mathbb{U})\times\mathbb{P}^n(\mathbb{U})$
$Commute$	$\mathbb{R}\times\mathbb{R}\times\mathbb{P}^m(\mathbb{U})\times\mathbb{P}^n(\mathbb{U})$

$$\exists(x_1, \ldots, x_n) : \bigwedge_{j=1}^{m} P_j$$

où les P_j sont des prédicats et les x_i des variables libres dans les prédicats. Par exemple, le pattern *Factory Method* est défini en LePUS par :

$$\begin{array}{l}
\exists(Creators \in \mathbb{H}, Products \in \mathbb{H}, FactoryMethods \in \mathbb{P}(\mathbb{F})) : \\
Clan(FactoryMethods, Creators)\wedge \\
ReturnType^{\leftrightarrow}(FactoryMethods, Products)\wedge \\
Production^{\leftrightarrow}(FactoryMethods, Products)\wedge \\
Commute_{ReturnType,Production}(FactoryMethods, Products)
\end{array}$$

Une *instance* de pattern est donc un n-uplet $(x_1, \ldots, x_n)$ particulier tel que $\bigwedge_j P_j$ soit vraie.

Pour relier les patterns entre eux, on introduit deux nouvelles notions : le raffinement et la projection. Intuitivement, “un pattern en raffine un autre” signifie que le premier est un “cas particulier” de l'autre. L'idée est que tous les n-uplets vérifiant les prédicats du pattern raffiné doivent vérifier aussi les prédicats du pattern plus général. Cela implique que ce dernier est moins contraignant (il a donc moins de prédicats) ou que les ensembles intervenant dans les relations sont d'ordre supérieur. La *projection* concerne justement l'ordre des ensembles. Une projection est obtenue en remplaçant certains ensembles X dans les relations par des éléments particuliers x de ces ensembles. Intuitivement, on passe à la dimension inférieure, d'où cette notion de projection.

Prenons les exemples des patterns *Factory Method* et *Abstract Factory* (voir paragraphe 2.2.4). La formule LePUS du pattern *Abstract Factory* est de la forme :

$$\frac{\begin{array}{l} Creators : \mathbb{H} \\ Products : \mathbb{P}(\mathbb{H}) \\ FactoryMethods : \mathbb{P}(\mathbb{P}(\mathbb{F})) \end{array}}{\begin{array}{l} Tribe(FactoryMethods, Creators) \wedge \\ ReturnType^{\leftrightarrow}(FactoryMethods, Products) \wedge \\ Production^{\leftrightarrow}(FactoryMethods, Products) \wedge \\ Commute_{ReturnType, Production}(FactoryMethods, Products) \end{array}}$$

où $Creators$ est la hiérarchie composée de la classe abstraite $AbstractFactory$ et des classes concrètes $ConcreteFactory1$ et $ConcreteFactory2$, $Products$ est l'ensemble de hiérarchies $ProductsA$ (racine : $AbstractProductA$, classes concrètes : $ProductA1$ et $ProductA2$) et $ProductsB$ ($AbstractProductB$, $ProductB1$ et $ProductB2$) et $FactoryMethods$ désignent les méthodes $CreateProduct$ des classes de $Products$. Comme $FactoryMethods$ est dans ce cas un ensemble de clans qui se réfèrent tous au même ensemble de classes, $FactoryMethods$ est donc une tribu relativement à la hiérarchie $Creators$. Si on compare cette formule avec celle trouvée dans le cas du pattern *Factory Method*, on remarque qu'elles sont analogues. Le tableau suivant indique les différences :

Abstract Factory	Factory Method	Description
$Products : \mathbb{P}(\mathbb{H})$	$Products : \mathbb{H}$	Une hiérarchie contre un ensemble de hiérarchies
$FMs : \mathbb{P}(\mathbb{P}(\mathbb{F}))$	$FMs : \mathbb{P}(\mathbb{F})$	Un clan contre un ensemble de clans

où FMs désigne $FactoryMethods$. La spécification LePUS de *Factory Method* peut donc être obtenue en modifiant la dimension de deux variables dans *Abstract Factory*. On dit alors que *Factory Method* est le projeté de *Abstract Factory*.

Conclusion

LePUS est un langage de patterns basé sur la logique monadique d'ordre supérieur (HOML) qui permet de spécifier les modifications induites par application des patterns sur les programmes à l'aide de prédicats et de relations. Cette analyse est intéressante car cette formalisation se rapproche des aspects ensemblistes du langage B et les prédicats apportés par [9, 14] pourraient servir d'invariants ou de préconditions d'opérations d'une spécification de machine abstraite en B.

Toutefois, LePUS diffère de notre problème car il fonctionne avec ses propres outils et son propre langage. Il n'existe aucun outil pour produire du code. De plus, l'intégration de spécifications existantes n'est pas possible. Cette approche demande des connaissances importantes en mathématiques. D'autre part, LePUS ne résout pas encore les problèmes de composition de patterns et des aspects de réutilisation de modules dans une conception logicielle. Cependant, LePUS apporte une formalisation des notions de spécialisation (voir raffinement et projection) et une définition très abstraite de l'instanciation. Les auteurs cherchent à présent à automatiser ce langage avec l'aide de PROLOG.

2.5.3 Spécification des GoF patterns avec RSL

Cette étude est basée sur les rapports techniques de UNU/IIST [6, 19, 34, 18, 5, 4]. L'idée de ces articles est de spécifier les GoF patterns à l'aide du langage de spécification RSL (RAISE Specification Language). Les auteurs ont ensuite étendu leurs recherches aux conceptions orientées objet afin de pouvoir spécifier l'instanciation des patterns et donner des exemples de compositions.

RAISE

RAISE, un projet ESPRIT de 1985 à 1990, est inspiré à la fois de VDM (à qui il manquait la modularité et la concurrence) et des spécifications algébriques. En anglais, RAISE est en fait l'acronyme de Rigourous Approach to Industrial Software Engineering. Aujourd'hui, RSL est le langage de spécification RAISE notamment utilisé par l'université des Nations Unies UNU/IIST. Dans ce cadre, les chercheurs attachés à l'UNU/IIST se sont donnés comme objectif de spécifier des patterns avec RSL.

Pour de plus amples détails sur le langage RSL, il est conseillé de se référer à [21]. Il faut retenir que la structuration en RSL est basée sur les schémas (scheme), les classes (class), les objets (object), les types (type) et les valeurs (value).

Les *valeurs* sont associées à des types. Par exemple, on déclare les valeurs :

value

voiture : Voiture,
personne : Personne

Ce qui signifie que la valeur voiture est de type Voiture et, de même, personne est de type Personne. Un type peut être composé avec d'autres types. Par exemple,

value

voitureconduite : Voiture $\times$ Personne,
conducteur : Personne $\rightarrow$ Voiture

Les *types* se définissent de la manière suivante :

type

Voiture,
Personne,
NumeroImmatriculation,
Couleur == rouge | bleu | noir | gris,
ParcAutomobile = Voiture-**set**,

Proprietaire :: nom : Personne vehicule : Voiture plaque : NumeroImmatriculation

Dans cet exemple, Voiture et Personne sont des types simples. Couleur est un type énuméré. ParcAutomobile est un ensemble de valeurs de type Voiture. Enfin Proprietaire est un article composé de trois champs : un nom de type Personne, un vehicule de type Voiture et une plaque de type NumeroImmatriculation.

Une *expression de classe* est composée d'un ensemble de types et/ou de valeurs. Par exemple,

class

type

Voiture,

Personne,

NumeroImmatriculation,

Proprietaire :: nom : Personne vehicule : Voiture plaque : NumeroImmatriculation

CarteGrise :: nom : Personne vehicule : Voiture plaque : NumeroImmatriculation

value

voiture : Voiture,

personne : Personne,

estproprietaire : Proprietaire $\times$ Voiture $\rightarrow$ **Bool**

estproprietaire(p,v) $\equiv$ vehicule(p) = v,

estimmatricule : CarteGrise $\times$ Proprietaire $\rightarrow$ **Bool**

estimmatricule(c,p) $\equiv$ (estproprietaire(p,vehicule(c)) $\Rightarrow$ plaque(c) = plaque(p))

On remarque dans cet exemple que les valeurs utilisent les types déclarés juste avant et sont complétées par des axiomes qui les définissent. Cet exemple forme une classe. Il modélise ainsi une classe de modèles, en particulier la classe des voitures immatriculées. Un *objet* est un modèle particulier de cette classe.

Un *schéma* est constitué d'un nom suivi d'une expression de classe. Par exemple,

scheme VOITURE =

class

type

Voiture,

Personne,

NumeroImmatriculation,

etc ...

value

voiture : Voiture,

personne : Personne,

etc ...

end

en reprenant les exemples précédents. Ainsi, pour déclarer qu'un objet T provient du schéma VOITURE, on déclare :

T : VOITURE

Pour distinguer les types des différents objets d'une même classe, on utilise la notation : OBJET.type. "." est en effet un opérateur d'accès aux sous-classes. On peut alors faire appel dans une spécification d'un schéma à des types d'un objet particulier. Par exemple,

scheme USAGERS =
  class
    type
      Usager,
      Voiture :: proprietaire : Usager vehicule : T.Voiture
  end

Le schéma USAGERS utilise le type Voiture déclaré dans la classe de T pour définir son *propre* type Voiture. Après ces quelques précisions concernant RSL, étudions maintenant la spécification des patterns en RSL.

GoF patterns en RSL

Ce paragraphe est basé sur le premier article [6]. On s'intéresse, comme dans LePUS, à la partie solution du pattern. Les auteurs l'ont divisée en trois sections importantes : l'en-tête, la structure et les collaborations du pattern.

Description des sections L'en-tête constitue l'interface du pattern et il permet de le distinguer des autres. Il est composé essentiellement du nom et de la classification.

Ensuite, il est nécessaire de spécifier la structure du pattern en RSL. Le diagramme qui décrit cette structure est généralement composé de :

- classe : abstraite ou concrète, munie de méthodes,
- relation : héritage, agrégation, association ou instanciation,
- signature : l'ensemble des signatures forme une interface,
- objet : instanciation d'une classe,
- commentaires.

Cette description semi-formelle est complétée à l'aide de la description textuelle puis spécifiée dans la partie structure.

Enfin, la partie collaborations permet de rajouter des hypothèses supplémentaires pour caractériser les patterns. L'utilisation du terme "Collaborations" comme dans la description GoF montre que cette partie est fondamentale dans la spécification. Les collaborations sont définies comme des relations d'appel entre deux classes d'un pattern ou bien comme des relations d'instanciation entre deux classes. Elles se distinguent des relations : par exemple, deux classes peuvent être reliées par agrégation, mais ces classes collaborent chacune avec l'autre en tant que paire (émetteur, récepteur). Chaque collaboration représente donc une paire d'objets d'un pattern, associée à un message.

Modèle RSL Le modèle utilisé pour spécifier la solution des patterns est constitué d'un nom, d'une structure et de collaborations entre les participants du pattern. Pour des raisons de place, il n'est pas possible de retranscrire l'ensemble de la spécification RSL des GoF patterns dans ce rapport. Pour la retrouver, reportez-vous à l'appendice situé à la fin de l'article [6]. Dans la suite, nous

exposerons les principales hypothèses et les choix qui ont conduit à ce résultat.
Un GoF pattern en RSL est de la forme suivante :

GoF-Pattern =
G.Pattern-Head × PS.WF-Pattern-Structure × CO.WF-Colls

Cette déclaration définit le type GoF-Pattern composé des trois types suivants :

- Pattern-Head : l'en-tête du pattern,
- WF-Pattern-Structure : la structure du pattern,
- WF-Colls : les collaborations entre les constituants du pattern.

Dans notre cas, G désigne un objet de la classe DEFINITIONS-GENERALES, PS de la classe PATTERN-STRUCTURE et enfin CO de la classe PATTERN-COLLABORATIONS. Ces trois schémas sont des modules de spécification plus complexes qui permettent de définir de “bonnes” formulations, de “bonnes” structures et de “bonnes” collaborations. L'adjectif “bon” doit être interprété ici par “qui vérifie de bonnes propriétés”. Une fois encore, les “bonnes” propriétés sont difficiles à exprimer car elles dépendent de la manière dont on interprète (et dont on complète) la description GoF. Il faut maintenant définir ces trois types.

Par exemple, en ce qui concerne l'en-tête, un pattern est décrit par son nom et sa classification (scope et purpose). La spécification RSL est alors de la forme :

Purpose == creational | structural | behavioural
Scope == s-class | s-object
Pattern-Head ::
    pattern-name : Name
    purpose : Purpose
    scope : Scope

De façon analogue, il est possible d'exhiber une spécification RSL concernant la structure et les collaborations. On considère une structure comme un ensemble de classes bien formées (WF) munies de relations bien formées :

Pattern-Structure = C.WF-Class-**set** × R.WF-Relation-**set**

WF-Class est un type défini dans le schéma CLASSES (C) et WF-Relation provient du schéma RELATIONS (R). Les collaborations sont des paires d'objets associées à des messages :

Collaboration ::
    sender-o : G.Concrete-Object
    receiver-o : G.Concrete-Object
    signature-coll : P.Signature-Head
Coll ::
    col : Collaboration
    prereq : G.Coll-Id-**set**
Collaborations = G.Coll-Id $\vec{m}$ Coll

où $\vec{m}$ désigne une application (map). Dans cette description, Collaboration est un type composé des champs sender-o, receiver-o et signature-coll. Coll permet d'associer une collaboration avec ses pré-requis, c'est-à-dire les collaborations qui doivent être éxécutées auparavant. Par exemple, dans le pattern *Factory Method*, la méthode *AnOperation* est un pré-requis de la méthode concrète *FactoryMethod*. Collaborations est une application entre les identifiants de G (G.Coll-Id) et Coll qui garde en mémoire les pré-requis.

Exemple de spécification : la partie WF-Pattern-Structure Toutes les formalisations suivent la même démarche. Dans un premier temps, on analyse la définition informelle (apportée par la langue naturelle ou par un graphique) d'une entité pour en extraire ses composants élémentaires. Ensuite, on les spécifie en interprétant leurs contraintes implicites par des conditions explicites. Les éléments vérifiant ces propriétés sont dits bien formés.

Concernant la structure des patterns, l'article a explicité et spécifié de nombreuses contraintes, comme par exemple :

- une classe source (sourceclass) ou destination (sinkclass) dans une relation doit appartenir à l'ensemble des classes dans la structure :

  isdefinedclass : Pattern-Structure $\rightarrow$ **Bool**

  isdefinedclass(c,r) $\equiv$ ($\forall$ e : R.WF-Relation . e $\in$ r $\Rightarrow$ R.sourceclass(e) $\in$ c $\wedge$ R.sinkclass(e) $\in$ c)
- une relation est une relation bien formée :

  iscorrectrelation : Pattern-Structure $\rightarrow$ **Bool**

  iscorrectrelation(c,r) $\equiv$ ($\forall$ e1,e2 : R.WF-Relation . e1 $\in$ r $\wedge$ e2 $\in$ r $\Rightarrow$ R.isvalidrelation(e1,e2))
- s'il y a plus d'une classe dans la structure, chacune doit être reliée à une relation dans la structure :

  isdefinedrelation : Pattern-Structure $\rightarrow$ **Bool**

  isdefinedrelation(c,r) $\equiv$ **card** c = 1 $\vee$ ($\forall$ e : C.WF-Class . e $\in$ c $\Rightarrow$ ($\exists$ d : R.WF-Relation . d $\in$ r $\wedge$ (R.sourceclass(d) = e $\vee$ R.sinkclass(d) = e)))

Ces contraintes sont ensuite reliées par :

iscorrectstructure : Pattern-Structure $\rightarrow$ **Bool**

iscorrectstructure(p) $\equiv$

(isdefinedclass(p) $\wedge$ iscorrectrelation(p) $\wedge$ isdefinedrelation(p) $\wedge$ etc ...)

Dans le même esprit, il est possible d'exprimer les contraintes évitant les problèmes de circuit des liens entre classes ou objets. Par exemple, on ne souhaite pas qu'une classe puisse hériter d'elle-même ou bien qu'un objet soit l'instance de lui-même. Ensuite il faut spécifier de la même manière chacun des blocs de construction d'un diagramme, autrement dit les composants Signature, Classe, Relation. L'ensemble de ces conditions permet de définir le type WF-Pattern-Structure des structures de patterns bien formées qui rentre dans la composition du type GoF-Pattern décrit au début de ce paragraphe.

Exemple : pattern Abstract Factory Une fois ce travail réalisé, la spécification de la partie Collaborations dépend des participants des patterns. La spécification réalisée dans [6] se limite aux patterns Bridge, State, Strategy, Observer et Abstract Factory.

Par exemple, dans le cas du pattern *Abstract Factory* (voir paragraphe 2.2.4), on définit l'objet suivant qui déclare les types rôle, signature, variable et paramètre :

object

AbstractFactory :

class

type

RoleType == abstractfactory | concretefactory | abstractproduct | concreteproduct | client,

VbleType,

SigType == createProduct,

ParamType

end

On traduit alors le pattern (représenté graphiquement) en des contraintes et des collaborations exprimées en RSL, comme par exemple :

- on définit une fonction *invoke-two* pour indiquer qu'un client est lié à des classes différentes (*AbstractProduct* et *AbstractFactory*) par des relations d'association différentes,
- un client utilise les interfaces de *AbstractFactory* et *AbstractProduct* uniquement,
- une classe jouant un rôle particulier est instanciée par une unique classe jouant un autre rôle,
- on définit un prédicat *hasfactory* pour indiquer qu'un *ConcreteProduct* est instancié par une unique *ConcreteFactory*,
- etc ...

On constate alors que la spécification obtenue est limitée. L'intérêt des patterns consiste à pouvoir les instancier, ce qui n'est pas encore possible à ce stade.

Extension aux conceptions orientées objet

Un modèle des conceptions orientées objet[1] a été ensuite spécifié en RSL dans [19]. Les cinq parties suivantes constituent les modules principaux d'une spécification :

- les définitions générales (G),
- les méthodes (M),
- les classes (C),
- les relations (R),
- les structures de conception (DS).

Les lettres entre parenthèses désignent le nom des objets de ces schémas. Ce modèle des conceptions objet inclut le modèle des GoF patterns présenté auparavant, ce qui permet de faire le lien entre les deux. Il faut alors créer une application de renommage entre le modèle des conceptions objet et le modèle des patterns. La figure 2.14 est un schéma représentant le lien entre ces deux modèles.

Ce modèle des conceptions objet est plus détaillé que le modèle des GoF patterns. Il permet de mieux spécifier :

- le contenu des commentaires d'un diagramme UML,
- les méthodes d'une classe,

1. Ils utilisent le terme de modèle à la place de méta-modèle.

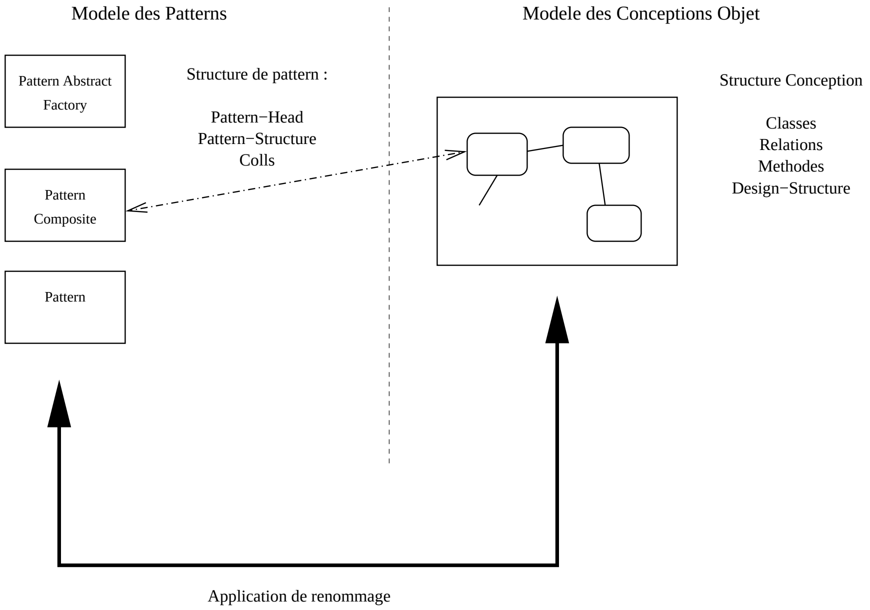

FIG. 2.14 – *Lien entre le modèle des conceptions objet et le modèle des patterns en RSL*

– les classes.

Une conception orientée objet est représentée dans le modèle par le type Design-Structure :

type

Design-Structure = Classes × WF-Relations,

WF-Design-Structure =

{| ds : Design-Structure . iswfdesignstructure(ds) |}

Ensuite, un lien entre conceptions orientées objet et patterns est défini par une application de renommage. Ainsi, les noms des différentes entités (classes, méthodes, variables d'état et paramètres) d'une conception orientée objet sont associés aux noms des entités correspondantes du pattern souhaité. Les correspondances entre les paramètres et les variables d'état sont spécifiées avec le type VariableRenaming :

type

VariableRenaming = Variable-Name $\vec{m}$ Variable-Name

Le type MethodandParametersRenaming relie les méthodes :

type

MethodandParametersRenaming = Method-Name $\vec{m}$ Method-Renaming

Method-Renaming ::

methodname : Method-Name parameterRenaming : VariableRenaming

Le renommage des classes est similaire :

type

ClassRenaming ::

classname : Class-Name methodRenaming : MethodandParametersRenaming varRenaming : VariableRenaming,

Renaming = Class-Name $\overrightarrow{m}$ ClassRenaming-**set**,

WF-Renaming =

{| r : Renaming . iswfrenaming(r) |}

Le type Design-Renaming permet d'associer une conception objet à un renommage :

type

Design-Renaming = WF-Design-Structure × WF-Renaming,

Il faut, en outre, spécifier les propriétés particulières des différents patterns concernant les classes, les méthodes, les variables d'état et les paramètres. Ce long travail est détaillé dans [34] pour les patterns de comportement, dans [18] pour les structurels et dans [5] pour les patterns de création.

Cette spécification des conceptions orientées objet et des patterns est une fois encore limitée. En effet, les types de renommage décrits jusqu'à présent rendent la multiple instanciation difficile : on ne peut pas retrouver facilement quelles instances correspondent à quels patterns. Or dans la pratique, il est fréquent d'utiliser des instances multiples d'un même pattern. Le rapport [4] apporte une solution. Il définit un nouveau type permettant d'identifier les renommages : on considère alors le triplet composé d'un rôle, des noms de patterns concernés et des instances considérées. Ce multi-renommage associe les noms de patterns avec les collections d'instances correspondantes. Plusieurs conditions implicites, pour éviter les confusions, sont également spécifiées. Chaque nom de pattern est associé à sa spécification apportée par les articles [34, 18, 5] grâce à une table de patterns.

Conclusion

En conclusion, ce travail apporte une spécification complète en RAISE des conceptions orientées objet, des GoF patterns et du lien d'instanciation entre ces derniers. Même si RSL n'est pas le langage B, il est assez proche des spécifications algébriques et de VDM. De plus, le travail d'analyse et de formulation des conditions implicites de la structure des conceptions orientées objet et des patterns peut servir de base (ou de complément) à une spécification en B (et même avec tout autre langage formel).

RSL est utilisé de manière peu abstraite : toutes les contraintes concernant les patterns sont en effet spécifiées à un niveau proche du code. Cette approche implique donc une lourde spécification des patterns. De plus, une fois la spécification terminée, la composition entre deux patterns n'est pas immédiate. Il est nécessaire d'apporter des spécifications supplémentaires pour tenir compte des liens rajoutés. À la différence de LePUS qui permettait de spécifier un pattern à l'aide de seulement quelques prédicats, la spécification RSL des patterns se traduit en revanche par un très grand nombre de valeurs exprimant de nombreuses contraintes. L'utilisation d'une application de renommage permet de résoudre le problème de la spécification de l'instanciation, malgré l'absence de plusieurs niveaux d'utilisation.

2.5.4 Spécification des frameworks

Cette étude est basée sur la série de documents [37, 25, 24, 27, 26, 17, 16, 15, 8, 28]. Notre document de référence est [26] qui est l'aboutissement des travaux préliminaires avant d'aborder les aspects dynamiques que nous ne détaillerons pas ici.

Les auteurs adoptent un formalisme à trois niveaux. Dans le plus haut niveau, on a des *frameworks de spécification* ou plus simplement des *frameworks*. Un framework de spécification est une théorie multi-sorte du premier ordre qui axiomatise un domaine de problème. La sémantique est alors basée sur les modèles isoinitiaux. Les principales définitions d'un framework de spécification sont données dans la suite. Ensuite, nous avons les *spécifications*. Elles sont toujours données dans le contexte d'un framework. Une spécification est un ensemble de formules qui définissent de nouvelles relations ou fonctions en utilisant les termes du langage de la syntaxe du framework. Enfin, au plus bas niveau, nous avons les *programmes corrects*. Pour chaque relation spécifiée, un framework doit contenir au moins un programme correct pour l'implémenter. Pour définir la correction, l'étude se base sur la notion de *robustesse* décrite dans [27]. L'intérêt est qu'il suffit de montrer la correction de modules ouverts qu'on peut ensuite instancier et composer sans problème.

Framework de spécification : définitions

Ces frameworks sont définis à l'aide des spécifications algébriques, en utilisant une logique multi-sorte du premier ordre avec l'égalité.

Dans ce contexte, une signature $\Sigma = \{S,F,R\}$ est définie par les ensembles suivants :

- S est l'ensemble des symboles de sortes,
- F est l'ensemble des déclarations de fonctions,
- R est l'ensemble des symboles de relations.

Concernant la sémantique, les modèles des spécifications algébriques sont des algèbres dépendant de la signature. Un morphisme (d'algèbres) est une application qui transporte d'une algèbre à une autre les interprétations des différents éléments de la signature.

Un "*isomorphic embedding*" est un morphisme h particulier qui conserve la propriété de négation suivante : $\forall r \in R$, égalité comprise,

$$(\alpha_1,\alpha_2,\ldots,\alpha_n) \notin r^{I^*} \Rightarrow (h(\alpha_1),\ldots,h(\alpha_n)) \notin r^I$$

où la notation r^I désigne l'interprétation de r dans le modèle I. Un modèle *initial* signifie que tous les autres modèles peuvent être obtenus à partir du modèle initial par un morphisme unique. Lorsqu'on évoque la sémantique des modèles algébriques, on y associe la notion de satisfaction de formules. Pour toute algèbre, on peut lui associer l'ensemble de toutes les formules qu'elle vérifie : cet ensemble est appelé une théorie. De la même manière qu'il est possible de définir un modèle initial par rapport aux algèbres, il est possible de définir un modèle *isoinitial* par rapport aux théories : pour toute algèbre K, un modèle I^* de la théorie de K, notée $Th(K)$, est isoinitial si pour tout autre modèle I de $Th(K)$, il existe un unique isomorphic embedding $h : I^* \to I$. L'existence d'un tel modèle n'est pas garantie.

Ces rappels nous permettent désormais de définir formellement les frameworks à l'aide des théories multi-sortes. Un *framework de spécification* $\mathcal{F}$ est une théorie qui axiomatise un type abstrait de données. Dans ce cas, le modèle cherché pour représenter ces types abstraits est le modèle isoinitial de $\mathcal{F}$. L'intérêt d'utiliser des théories isoinitiales plutôt que des théories initiales est de pouvoir mieux traiter la négation dans les propriétés à vérifier. Par exemple, un atome A est faux dans une théorie initiale s'il n'existe pas de preuve de A. Dans une théorie isoinitiale, le fait que A est faux correspond à une preuve de $\neg A$. Comme l'information est souvent contenue dans les négations, il y donc moins de perte d'information.

Un framework $\mathcal{F}$ est noté

$$\mathcal{F}(\Pi) = (\Sigma,\mathcal{T},\mathcal{TH},\Pi)$$

où Π est un ensemble de paramètres, Σ est la signature de $\mathcal{F}$, $\mathcal{T}$ est l'ensemble des axiomes et $\mathcal{TH}$ est l'ensemble des théorèmes. Π peut être vide, dans ce cas le framework n'est pas paramétré, il est dit *fermé*. Si un framework contient des paramètres, il est dit *ouvert*. Il peut alors être instancié en un framework fermé en attribuant une valeur à chacun de ses paramètres. Une telle opération est appelée une *fermeture* de framework.

Formalisation et exemples de frameworks

Un framework de spécification est déclaré par :

Framework Nom du framework (liste des paramètres)

IMPORT : Noms des frameworks utilisés

SORTS : Noms des sortes du framework

FUNCTIONS : Liste des fonctions

RELATIONS : Liste des relations

AXIOMS : Liste des axiomes

P-AXIOMS : Liste des axiomes dépendant des paramètres relations

On remarque que ce formalisme est assez proche des langages de spécifications algébriques, comme CASL. Il est en effet basé sur la logique des prédicats du premier ordre avec égalité. Mais il est aussi une adaptation du formalisme de Catalysis. Utilisons maintenant cette formalisation pour donner quelques exemples de frameworks ouverts ou fermés. Les entiers naturels peuvent être spécifiés de la manière suivante :

Framework $\mathcal{NAT}$;

SORTS : Nat;

FUNCTIONS :

$0 :\to Nat$;

$s : Nat \to Nat$;

$+,\star : (Nat,Nat) \to Nat$;

AXIOMS :

$\forall x.(x+0 = x)$;

$\forall x,y.(x+s(y) = s(x+y))$;

$\ldots$

$\mathcal{NAT}$ est un framework fermé, dont $\mathbb{N}$ peut être le modèle isoinitial, si la liste d'axiomes est bien complétée. Considérons désormais une suite finie d'éléments de type prédéfini X ; dans ce cas, la spécification est :

Framework $\mathcal{SEQ}(X)$;
IMPORT : $\mathcal{NAT}$,$\mathcal{X}$;
SORTS : Nat,X,Seq;
FUNCTIONS :
$[\,] :\to Seq$;
$cons : (X,Seq) \to Seq$;
$nocc : (X,Seq) \to Nat$;
RELATIONS : $elemi : (Seq,Nat,X)$;
AXIOMS : ...

Alors que $\mathcal{NAT}$ est fermé, $\mathcal{SEQ}(X)$ est un framework ouvert avec le type paramétré X. Si $\mathcal{NAT}$ n'a qu'un modèle, $\mathcal{SEQ}(X)$ a une classe de modèles pour le représenter. Pour chaque interprétation de la sorte X, nous avons un modèle correspondant de $\mathcal{SEQ}(X)$. Supposons par exemple que $\mathcal{INT}$ soit un framework fermé modélisant l'ensemble des entiers Int, alors $\mathcal{SEQ}(Int)$ importe automatiquement $\mathcal{INT}$ et devient un framework fermé avec un unique modèle isoinitial où Int est l'ensemble des entiers, Nat contient les entiers naturels et Seq est l'ensemble des suites finies d'entiers.

Un exemple plus intéressant de framework ouvert concerne les frameworks dits interactifs, c'est-à-dire des frameworks dont certains paramètres sont des relations et qui interagissent avec d'autres frameworks à l'aide de ces paramètres. Revenons à notre exemple $\mathcal{SEQ}(X)$. On peut l'étendre à $\mathcal{SEQ}(X,\lhd)$, où $\lhd$ est la relation d'ordre partiel. Dans ce cas, on a :

Framework $\mathcal{SEQ}(X,\lhd)$;
IMPORT : $\mathcal{NAT}$,$\mathcal{X}$;
SORTS : Nat,X,Seq;
FUNCTIONS :
$[\,] :\to Seq$;
$cons : (X,Seq) \to Seq$;
$nocc : (X,Seq) \to Nat$;
RELATIONS :
$elemi : (Seq,Nat,X)$;
$\lhd : (X,X)$;
AXIOMS : les mêmes que dans $\mathcal{SEQ}(X)$;
P-AXIOMS :
$x \lhd y \wedge y \lhd x \Leftrightarrow x = y$;
$x \lhd y \wedge y \lhd z \Rightarrow x \lhd z$

A part les axiomes usuels concernant les fonctions et les relations, le framework $\mathcal{SEQ}(X,\lhd)$ a un ensemble supplémentaire d'axiomes de paramètres, ou P-axiomes, concernant la relation $\lhd$.

Instanciation et composition de frameworks

Si $\mathcal{F}(\Pi)$ est un framework ouvert et si $\mathcal{G}$ est fermé, alors $\mathcal{F}(\Pi)$ peut être instancié par $\mathcal{G}$. L'instance correspondante, notée $\mathcal{F}(\Pi)[\mathcal{G}]$, est le résultat du renommage de la signature de $\mathcal{F}(\Pi)$ par celle de $\mathcal{G}$. Une telle opération n'est possible que si Π est l'intersection des signatures de $\mathcal{F}(\Pi)$ et de $\mathcal{G}$ et que $\mathcal{G}$ prouve les P-axiomes de $\mathcal{F}(\Pi)$.

La composition est une opération essentielle pour réutiliser les composants. Soit

$$\mathcal{F}(\Pi) = (\Sigma,\mathcal{T},\mathcal{TH},\Pi)$$

l'ensemble des P-axiomes est la restriction $\mathcal{T}|_{\Pi}$. La composition de $\mathcal{F}(\Pi)$ avec $\mathcal{F}_1(\Pi_1)$ est le résultat d'un Π-renommage ρ défini de telle sorte que : $\rho(\Sigma)$ est un Π-renommage pour Σ et Σ_1 si pour tout symbole σ utilisé dans Π, $\rho(\sigma)$ appartient à Σ_1 tandis que les symboles de sortes et les déclarations non utilisés dans Π sont associés à des noms qui n'apparaissent pas dans Σ_1. On peut alors construire un ρ-amalgame de signatures :

$$\rho(\Sigma)+\Sigma_1 = (\rho(S)\cup S_1,\rho(F)\cup F_1,\rho(R)\cup R_1)$$

où $\Sigma = \{S,F,R\}$ et $\Sigma_1 = \{S_1,F_1,R_1\}$. L'opération de composition n'est définie que si l'obligation de preuve suivante est satisfaite :

$$\rho(\mathcal{T}|_{\Pi}) \subseteq \mathcal{T}_1\cup\mathcal{TH}_1$$

ce qui prouve que les P-axiomes de $\mathcal{F}$ deviennent les théorèmes ou les axiomes de $\mathcal{F}_1$. On note $\mathcal{F}[\rho,\mathcal{F}_1]$ l'opération de composition de $\mathcal{F}$ avec $\mathcal{F}_1$ en utilisant le renommage ρ. Il est possible de montrer que cette composition est associative et non commutative.

Pour illustrer cette formalisation des frameworks, donnons un exemple simple de composition. Considérons le framework fermé suivant :

Framework $\mathcal{K}_{a,b}$;
SORTS : $Elab$;
FUNCTIONS : $a,b :\to Elab$;
AXIOMS : $\neg\ a = b$

Considérons désormais le framework ouvert qui modélise la théorie des piles :

Framework $\mathcal{STACK}(Elem)$;
SORTS : $Elem,Stacks$;
FUNCTIONS :
$empty :\to Stacks$;
$push : (Elem,Stacks) \to Stacks$;
AXIOMS :
$\neg\ empty = push(x,S)$;
$push(x_1,S_1) = push(x_2,S_2) \Rightarrow x_1 = x_2 \wedge S_1 = S_2$;
P-AXIOMS : $\exists x,y\ (\neg\ x = y)$

Dans cet exemple, le P-axiome exige que $Elem$ contienne au moins deux éléments distincts. Le framework $\mathcal{STACK}$ ainsi défini est représenté par une classe de modèles. Si on instancie cette classe avec un ensemble particulier comme $\mathcal{NAT}$ par exemple, on ferme le framework et on obtient un modèle de piles avec des entiers naturels comme éléments.

Composons le framework ouvert $\mathcal{STACK}$ avec $\mathcal{K}_{a,b}$, en utilisant comme $Elem$-renommage $Elem \mapsto Elab$ et $\rho(\sigma) = \sigma$ pour les autres symboles. La syntaxe est alors :

Framework $\mathcal{STACK}_{a,b}$;

COMPOSES : $\mathcal{STACK}(Elem)[\mathcal{K}_{a,b}]$
WITH : $Elem \mapsto Elab$;
OBLIGATION : $\exists x,y\ (\neg\ x = y)$
L'obligation de preuve découle immédiatement de l'axiome $\neg\ a = b$ dans $\mathcal{K}_{a,b}$. La composition des deux frameworks est alors définie.

Conclusion

Les formalisations présentées s'intéressent surtout à l'aspect statique des frameworks. Il est aussi possible d'étudier les aspects dynamiques. Les articles [15, 8] notamment introduisent la notion d'axiomes de transition d'état. Lors de la déclaration du framework, on rajoute une nouvelle assertion ST-axioms concernant les transitions d'état du système. Les axiomes sont de la forme pré- et post-conditions. Pour représenter l'évolution des états, on construit une structure d'événements constituée de relations cause à effet ou bien conflictuelle. Il est alors possible en cas de composition de frameworks de composer les deux structures correspondantes pour décrire les changements d'état du framework composite résultant.

En conclusion, la notion de framework a été définie formellement dans une sémantique grâce aux théories multi-sortes du premier ordre avec égalité et un langage de spécification des frameworks a été présenté. Parallèlement, une notation graphique plus détaillée a été posée. L'utilisation des théories isoinitiales comme modèles permet de mieux prendre en considération les preuves des propriétés de négation. En revanche, comme l'existence des modèles isoinitiaux n'est pas garantie, la spécification des frameworks est plus restrictive que pour une spécification algébrique plus classique.

L'avantage de cette démarche est qu'elle pose les premières pierres d'une formalisation des composants de conception comme les frameworks qu'il est éventuellement possible d'étendre à d'autres pièces génériques du langage orienté objet comme les patterns, par exemple. Mais cette démarche est assez lourde : la spécification à trois niveaux implique en effet une phase de conception plutôt longue et la question de la validation des spécifications n'est pas abordée. Les outils de Catalysis ne sont plus adaptés puisque le formalisme a été modifié. Il reste donc de nombreux points à traiter.

Notre souhait est maintenant de se rapprocher du concept de machines abstraites définies en B afin d'utiliser tous les outils de contrôle, de vérification, de preuve disponibles en B.

2.6 Conclusion générale et pistes possibles

Le premier constat est qu'il existe assez d'exemples de composant réutilisable dans les approches objet pour s'en inspirer dans les approches formelles. Il est donc inutile de chercher à créer de nouveaux composants de toute pièce en B : les patterns et les frameworks sont suffisamment nombreux.

Le pattern semble être une bonne approche pour rendre les conceptions plus modulaires. Toutefois, des problèmes persistent. Le manque de définition des patterns rend son utilisation difficile dans des langages formels, comme B. D'autre part, le catalogue ne peut en aucun cas être exhaustif, et les liens entre patterns seront de plus en plus complexes avec la découverte de nouveaux, ce qui rendra l'identification et la composition des patterns encore plus difficile. Ces remarques sont aussi valables pour les frameworks.

Dans les approches étudiées, on constate que chaque auteur apporte sa contribution dans un domaine particulier. Il semble en effet peu vraisemblable de spécifier les patterns indépendamment de leur langage de spécification. Si des formalisations assez complètes ont été apportées, le langage B n'est que peu concerné à l'heure actuelle, même si cet axe commence à devenir plus courant. Le tableau 2.5 est le bilan des trois méthodes de spécification présentées dans le paragraphe 2.5 et de l'exemple du paragraphe 2.4.1 sur UML et B. Si LePUS

TAB. 2.5 – *Bilan des approches proposées*

	LePUS	Framework	RSL	UML et B
Paragraphe	2.5.2	2.5.4	2.5.3	2.4.1
Point de départ	rien	Catalysis	VDM	UML et B
Modélisation	HOML	1er ordre	VDM	machines abstraites
Outils	graphiques	Catalysis	non	B
Instanciation	projection	paramètre	map	à la main
Composition	non	non commutative	à la main	à la main
Génération de code	non	Catalysis	non	raffinements
Pratique	$--$	$+$	$-$	$+$
Abstraction	$++$	$+$	$-$	$+$
Formalisation	formel	formel	formel	semi-formel

est la méthode de spécification la plus complète concernant les notions de composant et d'instanciation, elle est aussi la plus difficile à mettre en œuvre : peu d'outils, langage spécifique et grande abstraction. L'instanciation est plus ou moins bien traitée selon les cas : paramétrisation pour les frameworks, application de renommage en RSL, projection dans LePUS ou bien avec l'aide d'UML dans l'exemple. La composition n'est pas évoquée dans LePUS et elle n'est pas commutative pour les frameworks. Paradoxalement, l'exemple UML et B qui ressemble le plus à notre projet est finalement le plus éloigné du point de vue de la méthode. L'approche RSL est intéressante car elle donne une solution pour spécifier une instanciation avec une abstraction de bas niveau : l'utilisation d'une application de renommage. Enfin, la spécification des frameworks est formelle et assez abstraite mais trop spécifique à Catalysis pour être adaptée en B.

Le travail consiste désormais à exploiter en B certaines de ces méthodes, comme l'utilisation de relations (voir LePUS) pour spécifier le comportement des programmes par application des patterns, ou bien la formulation explicite des contraintes de l'instanciation des patterns (voir RSL) ou encore la spécification plus centrée sur les interactions entre objets que sur les classes elles-mêmes (voir frameworks). Toutes ces pistes de recherche ne sont pas nécessairement valides, mais serviront de base dans la suite. Intéressons-nous donc au problème en B.

Chapitre 3

Notion de composant en B

L'état de l'art nous a permis de sélectionner quelques composants intéressants, comme le pattern *Composite*. Notre but est maintenant de spécifier formellement ce composant en B et de réutiliser ensuite cette spécification. Pour cela, il nous faut d'abord prouver la correction de la spécification, puis traduire le mécanisme d'instanciation que nous avons défini dans le paragraphe 2.4. Notre souhait est en effet de spécifier le composant et son instanciation en B, afin que l'Atelier B prenne en charge ce qui est automatisable. L'exemple du pattern Composite introduit dans le paragraphe 2.2.4.

Mais, avant de poursuivre, commençons par quelques rappels concernant le langage B (paragraphe 3.1). Ensuite, nous étudierons les possibilités et les limites de B concernant la spécification de la notion de composant (paragraphe 3.2). Les premiers essais (paragraphe 3.3) nous permettront d'aboutir à une proposition de solution (paragraphe 3.4). Nous tenterons alors d'appliquer cette solution avec un exemple simple (paragraphe 3.5). Enfin, nous conclurons sur les limites et les conséquences de cette application (paragraphe 3.6).

3.1 Langage B [1, 23]

Le langage de spécification B est basé sur la notion de 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 :

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

Exemple de machine abstraite :

MACHINE *Exemple_Machine*

VARIABLES x

INVARIANT
$x \in 0..20$

INITIALISATION
$x := 10$

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.

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,

Exemple de raffinement de Exemple_Machine :

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.

L'Atelier B (voir `www.atelierb-societe.com`), commercialisé par la société Clearsy, est un environnement permettant de gérer des projets en langage B. L'Atelier B 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 ⇒ 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.

Le bilan de la vérification des machines est représenté par un tableau de la forme suivante :

Machines abstraites	TC	POG	Obv	nPO	nUn	%Pr
Machine1	OK	OK	4	3	0	100
Machine2	OK	OK	6	2	1	50
Machine3	-	-				

La première colonne est la liste des machines abstraites du projet en cours. TC correspond à la vérification du typage des machines. POG indique si les obligations de preuve ont été générées. Une machine doit être bien typée avant de générer les obligations de preuve. Obv indique le nombre d'obligations de preuve évidentes, nPO, le nombre d'obligations de preuves non évidentes et nUn, le nombre d'obligations restant à prouver. Enfin, %Pr est le pourcentage de preuves non évidentes terminées (automatiquement ou interactivement).

Il existe enfin en B des clauses supplémentaires qui rendent la spécification plus modulaire. Il est ainsi possible de spécifier de manière incrémentale un système qu'il aurait été difficile de spécifier en une seule étape avec une seule machine. Les différentes clauses possibles sont :

– **SEES**,
– **USES**,
– **INCLUDES**.

La visibilité et l'accessibilité des variables et des ensembles diffèrent selon les clauses utilisées.

MACHINE A

SEES B

Dans la machine A, on peut, avec **SEES**,

– accéder aux ensembles et aux constantes de la machine B,
– lire les variables de B dans les opérations.

En ce qui concerne **USES**, on déclare :

MACHINE A

USES B

Dans ce cas, dans la machine A, on peut :

– accéder aux ensembles et aux constantes de B,

– lire les variables de B dans **INVARIANT** et dans les opérations de A.

Enfin, si on utilise :

MACHINE A

INCLUDES B

On a les mêmes accessibilités qu'avec **USES**, mais on peut en outre appeler des opérations de B dans les opérations de A. La clause **PROMOTES** permet de déclarer dans A certaines opérations de la machine incluse, sans les utiliser dans d'autres opérations. Les clauses **USES** et **SEES** sont plus limitées que **INCLUDES** et sont généralement utilisées pour se référer à une machine abstraite contenant les variables et les ensembles communs à toutes les machines du projet en cours.

Intéressons-nous maintenant à l'adaptation de la notion de composant en B.

3.2 Comment spécifier un composant en B ?

La spécification d'un composant générique en B pose plusieurs problèmes.

1. Définition d'un "composant" en B. Peut-on utiliser des notions existant en B pour définir un composant ? Par exemple, la notion de machine abstraite B peut-elle être considérée comme un composant ? Enfin, dans quelle mesure ce composant est-il réutilisable ? D'où le second problème :
2. Spécification de l'instanciation et de la composition en B. Comment peut-on définir précisément les opérations d'instanciation et de composition de composants en B ? Les notions de B sont-elles suffisantes pour exprimer ces opérations ? Enfin, comment rend-on le composant réutilisable ?

Nous chercherons à apporter un début de réponses à ces questions dans les prochains paragraphes.

Cependant, on peut déjà y répondre partiellement. Un projet B est un ensemble de machines abstraites reliées par des liens de type **USES**, **SEES** et **INCLUDES**. Une **instance** de composant est donc un sous-ensemble cohérent de ces machines. Mais comment définir en B l'instanciation et surtout le composant générique ? Pour se fixer les idées, commençons par spécifier le pattern Composite en B.

3.2.1 Pattern Composite en B

Dans le cas de Composite, la première idée consiste à spécifier directement le pattern en B. Il existe en effet un outil [33] qui permet de traduire un diagramme UML en B de manière automatique. La stratégie revient à définir une machine abstraite B par classe UML, plus une interface. Il existe ensuite des règles de traduction suivant les multiplicités des associations entre les classes. Dans notre cas, le pattern Composite a été traduit à la main avec cinq machines abstraites : **Composant_Machine**, **Composite_Machine**, **Feuille_Machine**, **Pere_Machine** et **Interface_Machine**. La description de ces machines B se trouve en annexe D.1. Les différentes machines sont ensuite prouvées et on obtient :

Machines abstraites	TC	POG	Obv	nPO	nUn	%Pr
Composant_Machine	OK	OK	4	2	0	100
Composite_Machine	OK	OK	6	2	0	100
Feuille_Machine	OK	OK	6	2	0	100
Interface_Machine	OK	OK	26	20	0	100
Pere_Machine	OK	OK	5	0	0	100
Total	OK	OK	47	26	0	100

On a donc obtenu une spécification correcte du composant générique Composite en B.

3.2.2 Possibilités et limites de B [1, 23]

Une fois que nous avons spécifié le composant, on peut se demander comment adapter la spécification pour introduire la notion d'instanciation.

Solution 1 : paramétrisation Instancier un composant consiste à engendrer des composants instanciés. Il s'agit donc à partir d'une machine B spécifiant le composant à instancier de construire autant de machines que d'instances, dans le but de pouvoir utiliser (simultanément) ces différentes instances lors de la conception d'une application. D'un point de vue B, cela signifie qu'il faut modifier la machine du composant générique, afin que les instances aient des noms de variables et d'opérations distincts deux à deux. Une solution est de paramétrer la machine initiale. Fabriquer une instance reviendrait alors à donner une valeur particulière à chaque paramètre.

Mais la paramétrisation est limitée en B. En effet, les machines abstraites et leurs opérations ne peuvent être paramétrées que par des ensembles ou des scalaires. Concernant la machine abstraite, la paramétrisation est de la forme :

MACHINE *Machine*(*param_scal*, *Param_set*)

CONSTRAINTS
  $param\_scal \in S$

La clause **CONSTRAINTS** permet d'exprimer les contraintes de typage. Par exemple, *param_scal* prend ses valeurs dans S. Ces paramètres formels sont instanciés au moment d'une inclusion avec **INCLUDES** :

MACHINE *MachineSup*

INCLUDES
  Machine(n,N)

Les limites de la paramétrisation en B sont les suivantes. On ne peut pas mettre en paramètre les variables de $Machine$. De plus, il n'est pas possible de paramétrer les machines abstraites par des opérations. Enfin, les paramètres des opérations prennent nécessairement leurs valeurs parmi les valeurs possibles des variables :

MACHINE *Machine*

SETS
ENSEMBLE

VARIABLES
variable

INVARIANT
$variable \subseteq ENSEMBLE$

INITIALISATION
$variable := \emptyset$

OPERATIONS
$sorties \longleftarrow operation(entrees) =$
pre
$entrees \in variable$
then
...
end

Cette piste est donc peu intéressante en ce qui nous concerne. On ne peut pas renommer l'opération, ni changer son corps.

Solution 2 : modification de la spécification initiale La question est la suivante. Quels sont les moyens disponibles en B pour définir la spécification d'une instance en transformant la spécification initiale ? Il y en a principalement deux :

- l'inclusion de machine,
- et le raffinement.

L'idée de l'inclusion de machine est d'inclure la spécification initiale et de rajouter des variables et des opérations décrivant une instance particulière. L'inclusion de machine se fait par utilisation des clauses **INCLUDES**, **USES** et **SEES** que nous avons décrites en introduction. **INCLUDES** permet ainsi d'inclure des machines abstraites et d'utiliser ses ensembles et ses variables, ainsi que ses opérations. Supposons que la machine **Base** soit spécifiée et prouvée préalablement :

Machine de base :

MACHINE *Base*

...

Dans ce cas, on peut utiliser les variables, les ensembles et les opérations de la machine **Base** dans une autre machine, sans changer la spécification de **Base** :

Machine incluant la machine de base :

MACHINE *Incluant*

INCLUDES
Base
...

La machine **Incluant** peut alors :

- renommer les variables et les ensembles de **Base** avec la clause **DEFINITIONS**,
- déclarer certaines opérations de **Base** avec la clause **PROMOTES**,
- utiliser des opérations de **Base** dans la définition de nouvelles opérations.

Une autre idée est de raffiner progressivement la spécification initiale jusqu'à aboutir à une spécification plus détaillée et moins abstraite d'une instance. Les variables abstraites de la machine abstraite initiale sont remplacées dans les machines de raffinement par de nouvelles variables concrètes. On modifie le corps des opérations de la machine abstraite pour les rendre déterministes. Les différentes machines raffinées expriment ces transformations par l'ajout d'instructions plus déterministes que les instructions de la machine initiale en se référant aux variables concrètes. En revanche, on ne peut pas définir d'opérations supplémentaires dans le raffinement. De plus, les opérations doivent garder la même signature. Par exemple,

Machine de base :

MACHINE *Base*

SETS
Ensembles_Abstraits

VARIABLES
Variables_Abstraites

INVARIANT
INV_Base

INITIALISATION
INIT_Base

OPERATIONS
$sorties \longleftarrow operation(entrees) =$
pre
Preconditions_Base
then
Corps_Base

end

Machine raffinant la machine de base :

REFINEMENT *Raffinement*

REFINES
Base

SETS
Ensembles_Concrets

VARIABLES
Variables_Concretes

INVARIANT
INV

INITIALISATION
INIT

OPERATIONS
$sorties \longleftarrow operation(entrees) =$
pre
Preconditions
then
Corps
end

Dans ce cas, $Variables\_Concretes$ et $Ensembles\_Concrets$ sont distincts de $Variables\_Abstraites$ et $Ensembles\_Abstraits$. Les invariants et initialisations du raffinement dépendent des nouvelles variables ($Variables\_Concretes$) et des nouveaux ensembles ($Ensembles\_Concrets$) définis. L'opération raffinée a les mêmes entrées et les mêmes sorties que dans la machine **Base**, mais son corps et ses préconditions ont été modifiés. D'autre part, *Preconditions* et *Corps* utilisent les variables et les ensembles concrets définis dans le raffinement.

Nous nous intéresserons dans la suite à ces deux pistes possibles : l'inclusion et le raffinement.

3.3 Premiers essais d'instanciation

L'idée est de mélanger les deux pistes introduites dans le paragraphe précédent. On considère dans la suite que nous avons à notre disposition un composant spécifié et validé en B : par exemple, les cinq machines abstraites spécifiant le pattern Composite (voir annexe D.1). Par convention, on désignera l'ensemble de ces machines par COMPOSITE. Compte tenu des remarques précédentes, le renommage des éléments du composant peut se faire par inclusion des machines abstraites composant le pattern. Le raffinement peut ensuite servir à modifier les opérations.

3.3.1 But

En s'inspirant de l'exemple sur le contrôle d'accès [30] présenté dans le paragraphe 2.4.1, on souhaite instancier le pattern *Composite* en un point de passage dans un bâtiment, composé d'une porte et d'un lecteur. Le point de passage est alors considéré comme un objet composite composé des feuilles lecteur et porte. La structure du pattern Composite doit être adaptée et renommée et de nouvelles opérations doivent être spécifiées concernant l'acceptation ou le refus des cartes d'accès présentées au lecteur du point de passage. Notre but est donc de spécifier cet exemple avec le langage B à partir des machines abstraites de COMPOSITE décrites dans l'annexe D.1.

3.3.2 Première piste : inclusions

Une première idée consiste à créer, pour chaque machine de COMPOSITE, une nouvelle machine incluant une machine de la spécification initiale. Quelles clauses doit-on utiliser ? Les clauses **USES** et **SEES** ne sont pas transitives. Comme certaines machines de COMPOSITE utilisent déjà des **USES**, il sera impossible d'inclure à la fin toutes les machines renommées. Il nous reste uniquement la clause **INCLUDES**. Rappelons que le projet COMPOSITE est constitué de cinq machines abstraites B dont une machine “interface” incluant les quatre autres. Pour renommer les machines de COMPOSITE autres que l'interface, il suffit de créer une nouvelle machine incluant la machine à renommer. Puis on utilise la clause **DEFINITIONS** pour renommer les ensembles et les variables. Enfin, on définit les opérations avec les noms voulus mais dont le corps ne fait qu'appeler les opérations à renommer. Considérons, par exemple, la machine **Composant_Machine** :

Composant_Machine dans COMPOSITE :

MACHINE *Composant_Machine*

SETS
  COMPOSANT

VARIABLES
  Composant

INVARIANT
  $Composant \subseteq COMPOSANT$

INITIALISATION
  $Composant := \emptyset$

OPERATIONS
$Ajout\_Composant(elt) =$
pre
  $elt \in COMPOSANT - Composant$
then
  $Composant := Composant \cup \{elt\}$

end;
Retirer_Composant(*elt*) =
pre
  $elt \in Composant$
then
  $Composant := Composant - \{elt\}$
end

On déclare alors la nouvelle machine suivante :

Nouvelle machine Equipement_Machine :

MACHINE *Equipement_Machine*

INCLUDES
  Composant_Machine

DEFINITIONS
  $Equipement\ ==\ Composant;$
  $EQUIPEMENT\ ==\ COMPOSANT$

OPERATIONS
Ajouter_Equipement(*equip*) =
pre
  $equip \in EQUIPEMENT - Equipement$
then
  Ajout_Composant(*equip*)
end;
Retirer_Equipement(*equip*) =
pre
  $equip \in Equipement$
then
  Retirer_Composant(*equip*)
end

Les préconditions demeurent les mêmes, mais les éléments sont renommés. Cependant, avec ces notations, il n'est pas possible de renommer deux fois **Feuille_Machine** en Lecteur et Porte, car il y aura une confusion entre les deux renommages. Mais il est possible en B d'introduire un préfixe lors de l'inclusion d'une machine. Dans ce cas, pour renommer la machine :

Feuille_Machine dans COMPOSITE :

MACHINE *Feuille_Machine*

USES *Composant_Machine*

VARIABLES
  Feuille

INVARIANT
  Feuille $\subseteq$ *Composant*

INITIALISATION
  Feuille $:= \emptyset$

OPERATIONS
Ajout_Feuille(*elt*) = ...
Retirer_Feuille(*elt*) = ...
Operation_Feuille(*elt*) = ...

On déclare la machine incluse avec un préfixe : **lct.Feuille_Machine** et **prt.Feuille_Machine** par exemple. On peut ainsi distinguer le renommage de la variable $Feuille$ grâce aux préfixes : $lct.Feuille$ et $prt.Feuille$.

Lecteur_Machine : renommage de Feuille_Machine

MACHINE *Lecteur_Machine*

INCLUDES
  Composant_Machine, *lct.Feuille_Machine*

DEFINITIONS
  Lecteur == *lct.Feuille*;
  EQUIPEMENT == *COMPOSANT*

OPERATIONS
Ajouter_Lecteur(*lecteur*) =
pre
  lecteur $\in$ *EQUIPEMENT* $-$ *Lecteur*
then
  lct.Ajout_Feuille(*lecteur*)
end;

Retirer_Lecteur(*lecteur*) = ...
Operation_Lecteur(*lecteur*) = ...

Porte_Machine : deuxième renommage de Feuille_Machine

MACHINE *Porte_Machine*

INCLUDES
  Composant_Machine, *prt.Feuille_Machine*

DEFINITIONS
  Porte == *prt.Feuille*;
  EQUIPEMENT == *COMPOSANT*
...

En utilisant les mêmes techniques avec les autres machines (sauf l'interface), on obtient le renommage suivant :

COMPOSITE	Renommage
COMPOSANT	*EQUIPEMENT*
Feuille	*Lecteur* *Porte*
Composite	*Point*
Composant_Machine	**Equipement_Machine**
Feuille_Machine	**Lecteur_Machine** **Porte_Machine**
Composite_Machine	**Point_Machine**
Pere_Machine	**Composition_Machine**

Il est alors possible de vérifier le typage et la correction de ces machines avec l'Atelier B. Seul le renommage de la machine abstraite **Interface_Machine** de COMPOSITE pose problème :

Machine Interface du pattern COMPOSITE

MACHINE *Interface_Machine*

INCLUDES
Composant_Machine, *Composite_Machine*, *Feuille_Machine*, *Pere_Machine*

INVARIANT
$Feuille \cup Composite \ = \ Composant \wedge Feuille \cap Composite \ = \ \emptyset$

PROMOTES
Ajout_Enfants, *Retirer_Enfants*, *DonnerEnfants*

OPERATIONS
$cpt \longleftarrow \ Creer\_Composite \ =$
pre
$Composant \ \neq \ COMPOSANT$
then any xx **where** $xx \in COMPOSANT - Composant$
then
$Ajout\_Composant(xx) \ ||$
$Ajout\_Composite(xx) \ ||$
$cpt \ := \ xx$
end
end;
$feuille \longleftarrow \ Creer\_Feuille \ =$
pre
$Composant \ \neq \ COMPOSANT$
then any xx **where** $xx \in COMPOSANT - Composant$
then
$Ajout\_Composant(xx) \ ||$
$Ajout\_Feuille(xx) \ ||$

```
    feuille := xx
  end
  end;
  Supprimer_Composite(cpt) =
  pre
    cpt ∈ Composite
  then
    Retirer_Composant(cpt) ||
    Retirer_Composite(cpt) ||
    Retirer_Enfants(cpt)
  end;
  Supprimer_Feuille(feuille) =
  pre
    feuille ∈ Feuille
  then
    Retirer_Composant(feuille) ||
    Retirer_Feuille(feuille) ||
    Retirer_Enfants(feuille)
  end;
  Operation(cpt) =
  pre
    cpt ∈ Composant
  then
    select cpt ∈ Feuille then Operation_Feuille(cpt)
    when cpt ∈ Composite then Operation_Composite(cpt)
    else SKIP
    end
  end
  end
```

On ne peut pas conclure avec cette méthode : la nouvelle machine **Interface**, qui inclut à la fois les nouvelles machines (**Equipement_Machine**, **Lecteur_Machine**, **Porte_Machine**, **Point_Machine**, **Composition_Machine**) et la machine "interface" du projet COMPOSITE (**Interface_Machine**), n'est pas bien typée.

Machine Interface :

```
  MACHINE Interface

  INCLUDES
    Equipement_Machine, Lecteur_Machine, Porte_Machine
    Point_Machine, Composition_Machine, Interface_Machine

  ...
```

Pour comprendre, on doit observer le graphe de dépendance des différentes machines, représenté par la figure 3.1. Dans la nouvelle machine appelée **In-**

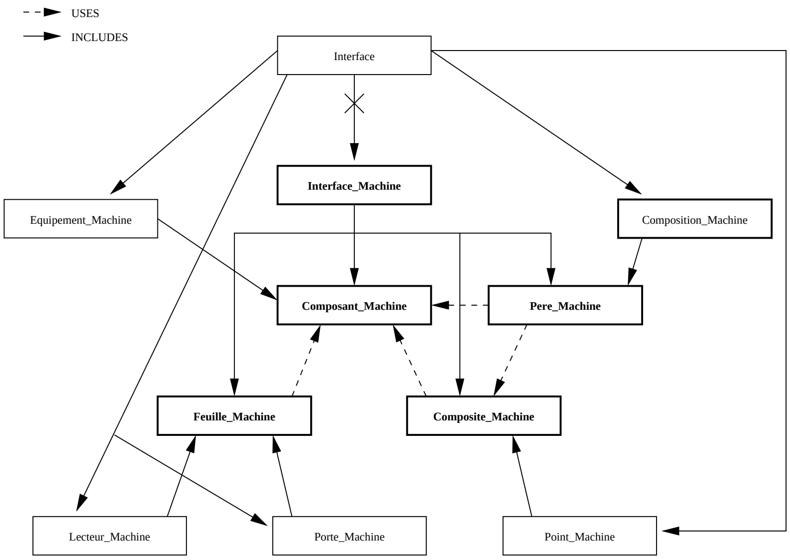

FIG. 3.1 – *Graphe de dépendance des machines B*

terface, les définitions des variables et des ensembles de la machine **Composant_Machine** sont accessibles par différents chemins (par exemple, l'ensemble $COMPOSANT$ est défini par inclusion de **Interface_Machine** mais aussi par inclusion de l'une des nouvelles machines, comme **Equipement_Machine**). C'est pourquoi le vérificateur de types rejette les machines obtenues. La machine finale tient compte des ensembles et des variables définis au début même de la chaîne, mais elle ne sait pas repérer par quel chemin. Par conséquent, la même variable donne lieu à deux définitions distinctes dans la machine finale.

Mais cette approche n'a pas que des aspects négatifs. Si on ne peut pas conclure, l'inclusion constitue tout de même un bon moyen de renommer des ensembles, des variables et des opérations. Toutes les obligations de preuve concernant les nouvelles machines sont soit évidentes, soit prouvées automatiquement :

Machines abstraites	TC	POG	Obv	nPO	nUn	%Pr
Composition_Machine	OK	OK	4	3	0	100
Equipement_Machine	OK	OK	6	0	0	100
Interface	-	-				
Lecteur_Machine	OK	OK	8	0	0	100
Point_Machine	OK	OK	8	0	0	100
Porte_Machine	OK	OK	8	0	0	100

L'instanciation du pattern COMPOSITE n'a donc pas généré de travail de preuve supplémentaire pour le concepteur (tout est pris en charge par l'Atelier B). En effet, toutes les preuves difficiles ont déjà été faites dans COMPOSITE. Par conséquent, toutes les obligations deviennent évidentes à renommage près des variables. Les machines autres que **Interface** sont décrites dans l'annexe D.2.

3.3.3 Deuxième piste : raffinement

Une autre idée consiste à raffiner les machines de COMPOSITE. On se retrouve cependant avec des problèmes de transitivité et de définitions multiples. Si on souhaite par exemple raffiner une des machines de COMPOSITE, comme **Feuille_Machine**, on doit créer une machine de raffinement de la forme :

Raffinement de Feuille_Machine :

REFINEMENT *Lecteur_Machine*

REFINES *Feuille_Machine*

VARIABLES
Lecteur

INVARIANT
$Lecteur \subseteq Composant$

INITIALISATION
$Lecteur\ := \emptyset$

OPERATIONS
$Ajouter\_lecteur(elt) =$
pre
$elt \in COMPOSANT - Feuille$
then
$Feuille := Feuille \cup \{elt\}$
end;
$Retirer\_Feuille(elt) =$
pre
$elt \in Feuille$
then
$Feuille := Feuille - \{elt\}$
end;
$Operation\_Feuille(elt) =$
pre
$elt \in Feuille$
then
...
end

Mais cette machine n'est pas valide en B, car **Feuille_Machine** utilise les ensembles et les variables de la machine **Composant_Machine** par l'intermédiaire de la clause **USES**. En cas d'utilisation de **INCLUDES**, nous aurions le même problème de définitions multiples rencontré dans le paragraphe précédent. Pour ces raisons, le raffinement ne peut concerner que la machine interface. Par conséquent, il est impossible de raffiner toutes les machines de COMPOSITE. De plus, il n'est pas possible de renommer, ni de rajouter des opérations avec le raffinement B classique.

Cependant, on peut contourner le problème du renommage en passant par une machine abstraite "vide". Par exemple, pour renommer la machine **Composant_Machine**, on déclare :

Machine intermédiaire :

MACHINE *Equipementvide*

INCLUDES
Composant_Machine

VARIABLES
Equipement

INVARIANT
$Equipement \subseteq COMPOSANT$

INITIALISATION
$Equipement := \emptyset$

Cette machine est "vide" car elle ne contient pas de nouvelles informations mais uniquement le nouveau nom, *Equipement*, de la variable *Composant*. On raffine ensuite **Equipementvide** avec :

Raffinement :

REFINEMENT *Equipement*

REFINES
Equipementvide

INCLUDES
Composant_Machine

INVARIANT
$Equipement = Composant$
...

L'invariant permet d'identifier la nouvelle variable : *Equipement* joue ainsi le rôle de l'ancienne variable *Composant*. Mais on ne peut renommer avec cette technique que les variables. Les ensembles et les opérations ne sont pas concernés. De plus, la méthode est limitée par des problèmes de transitivité des clauses **USES** et **INCLUDES**, lorsqu'on réalise la même démarche avec **Feuille_Machine**, qui utilise **Composant_Machine**.

3.3.4 Conclusion

Ces premiers essais nous ont amené à deux conclusions. Premièrement, il n'est pas possible d'utiliser uniquement des techniques B d'inclusions et de raffinements sur des composants (COMPOSITE par exemple) spécifiés à l'aide de plusieurs machines abstraites dépendantes les unes des autres. Nous avons en effet remarqué dans ces cas que des problèmes de transitivité ou de définitions multiples apparaissaient.

Deuxièmement, la création d'une machine incluant la machine B à renommer est une technique performante concernant le renommage des ensembles, des variables et des opérations, car le renommage ne rajoute pas d'obligations de preuve que le concepteur doit prouver. S'il est possible de rajouter des opérations, elle ne permet pas de modifier les opérations existantes. Concernant le raffinement, on peut modifier les opérations déjà définies, mais on ne peut pas en rajouter. De plus, il est difficile de faire du renommage avec du raffinement. En conclusion, l'inclusion et le raffinement ne permettent pas d'instancier un composant si elles sont utilisées séparemment. Cependant, ces techniques semblent complémentaires dans le cadre d'une instanciation.

3.4 Instanciation : méthode proposée

Rappelons quel est notre problème. Il existe dans les approches objet des patterns UML, peu formalisés et, de plus, les opérations d'instanciation et de composition de ces composants ne sont pas spécifiées. Notre but est d'utiliser B pour résoudre ces points. Dans ce paragraphe, nous proposons une définition de l'instanciation. La composition n'est pas traitée dans ce rapport. La mise au point d'outils automatisant la démarche d'instanciation sort du cadre de ce stage.

3.4.1 Premières solutions

L'instanciation d'un composant en B soulève des questions auxquelles nous devons répondre avant de poursuivre. Les choix sont justifiés en fonction de nos objectifs mais devront aussi être validés par la suite. Nos choix sont les suivants.

Composant B La solution choisie a été de regrouper l'ensemble des spécifications des machines incluses dans la machine interface afin d'utiliser l'inclusion de machine et le raffinement. Il existe en effet une machine abstraite unique, équivalente à toutes ces machines (voir figure 3.2). L'inconvénient de cette solution est qu'elle nécessite la création d'un outil permettant de traduire automatiquement un composant en sa machine unique équivalente ou bien la spécification dès l'origine des composants en une seule machine B. La création de l'outil en question sort du cadre du stage de DEA. Certains points concernant la traduction en une machine unique ont été traités dans [29]. Nous supposons dans la suite que le composant est spécifié dans une machine B unique et prouvée. Par exemple, nous avons spécifié "à la main" la machine unique de COMPOSITE à partir de ses différentes machines abstraites, présentées dans l'annexe D.1. Des commentaires ont été rajoutés afin de préciser les machines de provenance. Enfin, elle a été prouvée avec l'Atelier B. Cette machine est décrite dans l'annexe D.3.

Duplication des éléments du composant Dans l'exemple du point de passage, il y avait deux feuilles : la porte et le lecteur. En cas d'instanciation, cela

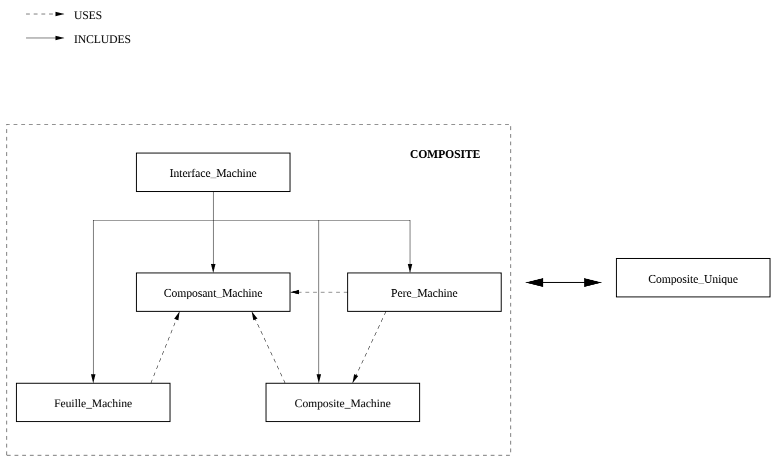

FIG. 3.2 – *Equivalence entre COMPOSITE et une machine unique*

impliquait une duplication de **Feuille_Machine** en deux machines distinctes : **Porte_Machine** et **Lecteur_Machine**. Le fait de spécifier le composant en B par une machine abstraite unique rend la duplication des machines difficile. Il est en effet difficile de distinguer les sous-machines dans la machine unique. De plus, nous considérons que cette multiplication des variables jouant le même rôle dans la machine est plus liée à la composition de composants qu'à l'instanciation. Comme nous nous intéressons maintenant à l'instanciation, nous laissons de côté pour le moment cet aspect et nous décidons de limiter l'instanciation à un renommage et à une adaptation des éléments du composant et de ne pas tenir compte des duplications possibles qui seront spécifiées à l'aide de la composition.

Une autre idée consisterait à préparer la multiplication des variables comme $Feuille$ en spécifiant dès l'origine des variables multiples $Feuille1$, $Feuille2$, etc ... dans la machine unique de COMPOSITE. Mais le problème ne serait pas résolu pour autant. Le nombre de $Feuille$ prédéfinis serait limité et il serait toujours possible de tomber sur un cas où le nombre de $Feuille$ voulu est supérieur. De plus, la preuve de cette machine avec des variables prédéfinies est difficile. Par exemple, si on prévoit deux feuilles seulement au lieu d'une, le nombre de preuves non évidentes augmente de plus de 50 %.

Nombre de feuilles prédéfinies	Obv	nPO
une feuille	39	21
deux feuilles	62	35

La multiplication des variables dans la machine unique implique en effet un nombre important d'invariants et d'initialisations supplémentaires. De manière générale, le rajout d'un ensemble dans une partition implique des contraintes supplémentaires, comme des intersections vides à vérifier entre le nouvel ensemble et chacun des éléments de la partition. Le même effet se produit ici : la duplication entraîne de nouvelles contraintes à vérifier. Cet exemple tend à confirmer l'hypothèse d'un lien avec les problèmes de composition.

Concernant les opérations du composant, il est facile de les dupliquer et de les renommer avec des liens **INCLUDES**. Il suffit de déclarer le nombre d'opérations voulu et d'appeler dans chacune d'entre elles l'opération du composant unique.

Instanciation B L'instanciation est réalisée en B par une inclusion du composant suivie d'un raffinement. Les deux mécanismes sont en effet complémentaires. L'idée consiste donc à bénéficier des avantages et des possibilités des deux techniques. Le problème revient maintenant à décider quelles spécifications donner au moment de l'inclusion et au moment du raffinement.

Par exemple, on a besoin de nouvelles variables et de nouveaux invariants pour spécifier les nouvelles opérations et pour adapter les anciennes. Si on les rajoute au moment de l'inclusion de la machine unique, ces données ne seront pas modifiables au moment du raffinement. D'autre part, il est impossible de rajouter de nouvelles opérations au moment du raffinement si on ne les a pas déclarées auparavant. Plusieurs scénarios possibles ont été testés sur des exemples et nous en avons extrait une proposition de solution.

3.4.2 Démarche proposée

L'instanciation d'un composant se réalise en deux étapes. On appelle **Composant_Unique** la machine unique B du composant à instancier.

Première étape On crée dans un premier temps une nouvelle machine abstraite **Composant_Renommage** qui inclut **Composant_Unique** :

MACHINE *Composant_Renommage*

INCLUDES *Composant_Unique*

Cette nouvelle machine sert d'étape intermédiaire. On y déclare essentiellement les renommages et les nouvelles opérations. Pour renommer les ensembles et les variables de **Composant_Machine**, on utilise la clause **DEFINITIONS**, comme suit :

DEFINITIONS
$NouveauNomEnsemble\ ==\ AncienNomEnsemble;$
$NouveauNomVariable\ ==\ AncienNomVariable$

Concernant les opérations auxquelles on ne touche pas et que l'on souhaite utiliser, on se sert de la clause **PROMOTES** pour les déclarer dans la nouvelle machine.

PROMOTES
OperationsNonModifiees

Enfin, on déclare dans **OPERATIONS** deux types d'opérations :

1. Si on souhaite renommer une opération existante, on la déclare dans la machine avec son nouveau nom. La précondition reste identique mais avec

les nouveaux noms de variables et d'ensembles définis dans **DEFINITIONS**. La nouvelle opération ainsi définie ne fait qu'appeler l'opération avec son ancien nom.

2. Si on souhaite définir des opérations supplémentaires, on doit donner leur en-tête. Le corps est spécifié par SKIP pour deux raisons :
   - cela facilite la preuve,
   - on n'a pas encore défini les variables supplémentaires, on ne peut donc pas préciser les traitements.

Les déclarations sont de la forme :

OPERATIONS

Renommage des opérations :

sorties $\longleftarrow$ *NouveauNomOperation*(*entrees*) $=$
pre
Precondition avec nouveaux noms
then
sorties $\longleftarrow$ *AncienNomOperation*(*entrees*)
end;

Pré-définition des nouvelles opérations :

sorties $\longleftarrow$ *NouvelleOperation*(*entrees*) $=$
pre
Precondition
then
SKIP
end

La preuve de cette nouvelle machine est immédiate. En effet, les initialisations et les invariants concernant les ensembles, les variables et les opérations renommées sont déjà prouvés dans la machine incluse. Quant aux nouvelles opérations, la substitution SKIP ne change rien. Par conséquent, les invariants sont préservés. L'absence de nouvelles variables, de nouveaux ensembles, de nouveaux invariants et de nouvelles initialisations n'implique que des preuves évidentes.

Deuxième étape Une fois que **Composant_Renommage** a été vérifiée et prouvée par l'Atelier B, on raffine cette machine. Cette phase nous permet alors de spécifier et de prouver les nouvelles opérations introduites dans l'étape précédente. La machine de raffinement **Composant_Instance** est déclarée par :

REFINEMENT *Composant_Instance*

REFINES *Composant_Renommage*

INCLUDES *Composant_Unique*

Lors d'un raffinement, il est possible de déclarer de nouveaux ensembles et de nouvelles variables.

SETS
$NouveauxEnsembles$

VARIABLES
$NouvellesVariables$

Il faut dans ce cas leur associer les invariants et les initialisations correspondants :

INVARIANT
$NouveauxInvariants$

INITIALISATION
$NouvellesInitialisations$

Les invariants des nouvelles variables peuvent contenir des ensembles ou des variables renommés dans la machine qu'on raffine. On rappelle dans ce cas les renommages introduits dans la machine **Composant_Renommage** :

DEFINITIONS
$NouveauNomEnsemble\ ==\ AncienNomEnsemble;$
$NouveauNomVariable\ ==\ AncienNomVariable$

On rappelle aussi les opérations non modifiées :

PROMOTES
$OperationsNonModifiees$

Enfin, concernant les opérations, on complète leur spécification :

OPERATIONS

Anciennes opérations :

$sorties\ \longleftarrow\ NouveauNomOperation(entrees)\ =$
pre
$Precondition\ avec\ nouveaux\ noms$
then
$sorties\ \longleftarrow\ AncienNomOperation(entrees)\ \|$
$NouvellesSubstitutions$
end;

Dans ce cas, il est possible de compléter une opération existante avec de nouvelles substitutions.

Nouvelles opérations :

$sorties\ \longleftarrow\ NouvelleOperation(entrees)\ =$
pre
$Precondition$
then

NouveauCorps
end

Si on veut modifier une variable de **Composant_Unique** dans les nouvelles substitutions, il y a deux cas possibles. Soit l'opération existe dans **Composant_Unique**, dans ce cas, on l'appelle. Soit l'opération n'existe pas, ce qui signifie que le composant est incomplet, et on est obligé de passer par un mécanisme particulier. Si on souhaite modifier la variable *varcomposant* définie dans **Composant_Unique**, remplacée par *nouvellevar* dans **Composant_Renommage**, on déclare :

REFINEMENT *Composant_Instance*

REFINES *Composant_Renommage*

INCLUDES *Composant_Unique*

VARIABLES
nouvellevar

DEFINITIONS
Renommage des variables

INVARIANT
$nouvellevar = varcomposant$

INITIALISATION
Initialisation de nouvellevar

OPERATIONS
$sorties \longleftarrow NouvelleOperation(entrees) =$
pre
Precondition
then
$nouvellevar := ...$
end

Mais ce dernier cas ne devrait pas arriver : si le composant est incomplet, il s'agit plutôt d'un problème de spécification de **Composant_Unique**.

Cette machine de raffinement est bien typée. Le raffinement est possible car les substitutions raffinent SKIP. Concernant la correction, seules les nouvelles substitutions spécifiées dans cette machine génèrent des obligations de preuve non triviales. Les autres obligations concernent des invariants et des initialisations, déjà prouvées dans les machines précédentes : elles sont alors évidentes.

3.4.3 Intérêt de la méthode

La figure 3.3 permet de situer notre démarche dans un schéma représentant un processus de conception B qui utiliserait des composants de spécification.

Concernant la spécification B des patterns, on utilise les patterns existant en

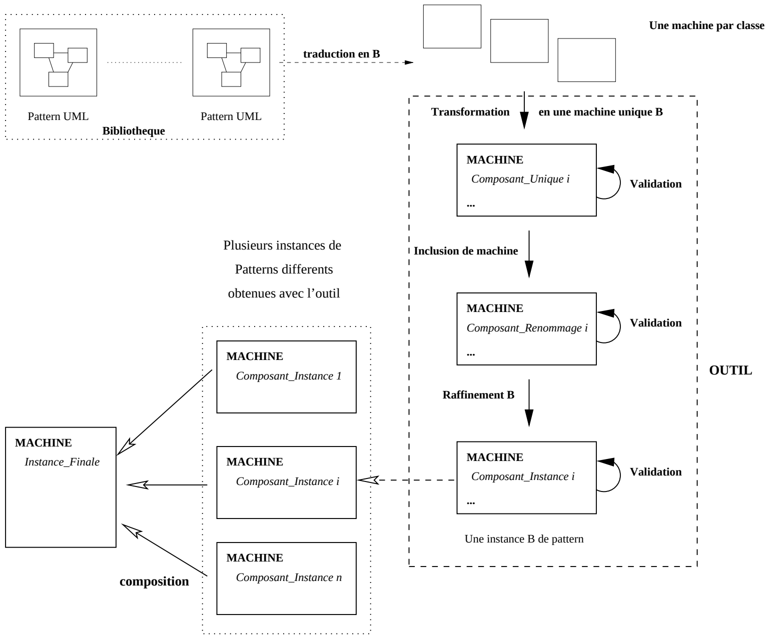

FIG. 3.3 – *Bilan de la méthode*

UML pour les traduire en B avec l'aide d'un outil (voir [33]) : des règles de traduction permettent de spécifier une machine abstraite B par classe du pattern. Ces différentes machines B sont ensuite regroupées dans une machine unique **Composant_Unique**. Il existe des règles de traduction (voir [29]) concernant cette étape.

La démarche proposée dans le paragraphe 3.4.2 permet ensuite d'instancier en B **Composant_Unique**. Elle est constituée de deux étapes : une inclusion et un raffinement. On obtient ainsi les machines B **Composant_Renommage** et **Composant_Instance** : la machine de renommage est validée automatiquement par l'Atelier B, tandis que les obligations de preuve de la machine de raffinement sont moins triviales en raison des nouvelles spécifications apportées dans cette machine. L'intérêt de cette démarche est qu'elle est automatisable. Il serait intéressant de créer un outil d'assistance permettant d'obtenir et de valider un pattern B sous la forme de **Composant_Unique**, puis de transformer ce composant par inclusion et par raffinement. L'outil pourrait générer à chaque étape les obligations de preuve non triviales afin de valider la spécification. La création d'un tel outil sort du cadre du stage de DEA.

Si on réitère ce processus sur plusieurs patterns, on obtient plusieurs instances de patterns différents spécifiées formellement en B : les machines **Composant_Machine_1**, ... , **Composant_Machine_n**. L'étape suivante, qui n'est pas abordée dans ce rapport, concerne la composition de ces machines pour obtenir une spécification finale **Instance_Finale**. Il serait alors possible de spécifier formellement en B un projet en réutilisant des composants de spécification.

3.5 Application de la démarche proposée

Afin d'illustrer cette proposition de solution, considérons l'exemple d'un éditeur comprenant des dossiers et des fichiers. Un dossier est considéré comme un élément composite pouvant comprendre des fichiers et des dossiers. On souhaite par exemple vérifier si ces éléments sont protégés ou en mode écriture. De nouvelles opérations, autres que celles spécifiées dans COMPOSITE, devront être définies afin de spécifier cet exemple.

3.5.1 Première étape : renommage

On commence par renommer la machine **Composite_Unique** en utilisant une inclusion.

MACHINE *Dossier_Renommage*

INCLUDES
Composite_Unique

L'ensemble $COMPOSANT$ et les trois variables *Composant*, *Composite* et $Feuille$ sont renommés par $ELEMENT$, *Element*, *Dossier* et *Fichier* respectivement.

DEFINITIONS
$ELEMENT \;==\; COMPOSANT$;
$Element \;==\; Composant$;
$Dossier \;==\; Composite$;
$Fichier \;==\; Feuille$

On ne souhaite pas modifier les opérations définies dans COMPOSITE qui permettent de retrouver les liens entre un composite et ses composants. Pour les déclarer dans **Dossier_Renommage**, on utilise :

PROMOTES
Ajouter_Fils, *Retirer_Fils*, *DonnerFils*

Concernant les opérations, on distingue les simples renommages et les définitions. Par exemple, l'opération *Creer_Composite* devient *Creer_Dossier*.

$dossier \longleftarrow Creer\_Dossier\ =$
pre
$Element \neq ELEMENT$
then
$dossier \longleftarrow Creer\_Composite$
end;

De même, *Creer_Feuille*, *Supprimer_Composite* et *Supprimer_Feuille* deviennent *Creer_Fichier*, *Supprimer_Dossier* et *Supprimer_Fichier* respectivement. L'opération générique *Operation* est utilisée pour définir deux opérations distinctes.

ModeEcriture(*element*) =
pre
  element ∈ *Element*
then
  Operation(*element*)
end;

Protection(*element*) =
pre
  element ∈ *Element*
then
  Operation(*element*)
end

Enfin, on a besoin de rajouter deux nouvelles opérations qui nous serviront dans le raffinement. On les déclare dans cette machine sans donner le corps.

Ajouter_Dossier(*dossier*) =
pre
  dossier ∈ *Dossier*
then
  SKIP
end;

Ajouter_Fichier(*fichier*) =
pre
  fichier ∈ *Fichier*
then
  SKIP
end;

L'ensemble des opérations est détaillé dans l'annexe D.4. La machine obtenue est vérifiée par l'Atelier B. Toutes les preuves sont évidentes. Nous reviendrons sur l'aspect preuve à la fin de ce paragraphe. Comme la machine est cohérente, nous pouvons passer à la seconde étape.

3.5.2 Deuxième étape : raffinement

On déclare la machine de raffinement **Dossier_Machine** :

REFINEMENT *Dossier_Machine*

REFINES *Dossier_Renommage*

INCLUDES *Composite_Unique*

Afin de spécifier les éléments utilisables de l'éditeur ainsi que les modes écriture et protection des dossiers et des fichiers, on déclare le nouvel ensemble $ETAT$ et les nouvelles variables $Etat$ et $Utilisables$.

SETS
$ETAT = \{ecriture, protege\}$

VARIABLES
$Etat, Utilisables$

Les initialisations et les invariants correspondants sont :

INVARIANT
$Utilisables \subseteq Element \wedge$
$Etat \in Element \nrightarrow ETAT$

INITIALISATION
$Etat, Utilisables := \emptyset, \emptyset$

On rajoute les renommages des ensembles et des variables introduits dans la machine **Dossier_Renommage**.

DEFINITIONS
$ELEMENT == COMPOSANT$;
$Element == Composant$;
$Dossier == Composite$;
$Fichier == Feuille$;

On rajoute enfin les opérations non modifiées du composant.

PROMOTES
Ajouter_Fils, *Retirer_Fils*, *DonnerFils*

Les opérations sont enfin spécifiées ou complétées en utilisant les nouvelles variables de la machine. Par exemple,

OPERATIONS

Ajouter_Dossier(*dossier*) =
pre
$dossier \in Dossier$
then
$Etat(dossier) := protege \parallel$
$Utilisables := Utilisables \cup \{dossier\}$
end;

Supprimer_Dossier(*dossier*) =
pre
$dossier \in Dossier$
then

```
  Supprimer_Composite(dossier) ||
  Utilisables := Utilisables − {dossier} ||
  Etat := {dossier} ⩤ Etat
end;

ModeEcriture(element) =
pre
  element ∈ Element
then
  select element ∈ Fichier then Etat(element) := ecriture
  when element ∈ Dossier then Etat(element) := ecriture
  else SKIP
end
end
```

Concernant *ModeEcriture*, la spécification présentée ci-dessus n'est pas très intéressante. On pourrait par exemple distinguer les variables *Etat_Dossier* et *Etat_Fichier*. Dans ce cas, *Etat* serait l'union disjointe de ces deux variables. Le raffinement des autres opérations, *Ajouter_Fichier*, *Supprimer_Fichier* et *Protection*, est détaillé dans l'annexe D.4. Enfin, la machine de raffinement est bien typée.

3.5.3 Remarques sur les preuves

La machine obtenue modélise une instance du pattern COMPOSITE. Elle représente en effet un système de fichiers comprenant des dossiers et des fichiers, protégés ou non, dont les éléments composites sont les dossiers. Cet exemple simple indique que la démarche proposée permet bien d'instancier un composant avec le langage B.

Concernant les preuves, la machine **Dossier_Renommage** est obtenue gratuitement. Les 25 obligations de preuve générées par l'Atelier B sont toutes évidentes. L'absence de nouvelles variables et de paramètres n'implique qu'une obligation concernant l'initialisation. La preuve est évidente en raison des renommages. Les autres concernent la vérification des préconditions des opérations de la machine incluse qui sont utilisées dans la nouvelle machine d'une part et la préservation de l'invariant par les substitutions des opérations d'autre part. Les preuves sont évidentes car les hypothèses contiennent à chaque fois la conclusion cherchée mais avec les variables de **Composite_Unique**, et les renommages des différentes variables.

La machine de raffinement pose plus de problèmes en raison des obligations liées au raffinement B. Par exemple, l'Atelier B génère pour **Dossier_Machine** 110 obligations de preuves évidentes et 24 preuves non triviales. Le prouveur automatique permet d'en éliminer 17. Il reste donc 7 preuves non triviales. Nous les avons toutes prouvées interactivement. Seules deux preuves sur sept sont "difficiles". Elles sont liées aux opérations de retrait d'éléments. Afin de préserver l'invariant, le prouveur doit vérifier si la variable *Etat* est bien typée en retirant une relation du type $\{element \mapsto etat\}$. La preuve n'est pas immédiate car l'utilisateur doit préciser clairement la règle utilisée pour conclure.

Les autres obligations sont faciles à prouver. Trois concernent des problèmes d'existence. Les opérations de suppression d'élément font toutes appel à l'opération $Retirer\_Enfants$ définie dans **Composite_Unique** :

$Retirer\_Enfants(pere)$ =
pre
  $pere \in Composant$
then
  any *chemin*
  where
    $chemin \in Composant \leftrightarrow Composite$
    $\wedge \forall(xx,yy).\ ((xx \mapsto yy) \in chemin \Rightarrow (yy \in Pere^{-1}[\{xx\}]$
    $\vee \exists zz.(zz \in Pere^{-1}[\{xx\}] \wedge (zz \mapsto yy) \in chemin)))$
  then
    $Pere :=$
    $(\{pere\} \cup chemin[\{pere\}]) \lhd Pere \rhd (\{pere\} \cup chemin[\{pere\}])$
  end
end;

Cette opération permet de retirer les liens entre un *Composite* et ses *Composant* fils. À cause du raffinement, les variables de **Dossier_Machine** sont distinctes des variables de la machine raffinée. Par conséquent, le prouveur cherche un élément $chemin_0$ vérifiant la condition de la clause **any**, alors qu'il existe *chemin* vérifiant la condition voulue dans les hypothèses de l'obligation de preuve. Il suffit alors de suggérer au prouveur B d'utiliser cet élément *chemin*. Enfin, les deux dernières preuves sont liées à la spécification des opérations $ModeEcriture$ et *Protection*. La préservation de l'invariant implique, en raison de la clause **select**, plusieurs cas à vérifier selon le type de l'élément considéré. Seul un cas est non trivial : il s'agit d'un cas absurde, car une des hypothèses est fausse. Mais le prouveur n'arrive pas à s'en rendre compte. Il suffit alors de lui indiquer quelle hypothèse est fausse pour terminer la preuve.

On pourrait se demander si ces cinq obligations de preuves jugées "faciles" sont liées aux spécifications rajoutées ou bien au raffinement du reste de la machine. Pour le vérifier, nous avons créé une nouvelle machine de raffinement, de la même forme que **Dossier_Machine**, qui ne rajoute ni l'ensemble $ETAT$, ni les variables *Etat* et $Utilisables$, et qui ne modifie pas les opérations. L'Atelier B génère alors 61 preuves évidentes et 5 obligations de preuve non évidentes. Ces cinq obligations ne peuvent pas être prouvées automatiquement : elles correspondent exactement aux mêmes obligations de preuve que dans le cas précédent. En conclusion, les trois preuves d'existence et les deux preuves de la clause **select** sont indépendantes de la spécification rajoutée au moment du raffinement.

Par conséquent, seules deux obligations de preuve sont réellement difficiles. À titre de comparaison, nous avons spécifié directement le même système de fichiers en B. Le détail se trouve en annexe D.5. La machine unique obtenue, appelée **Dossier_Unique**, est de la même forme que le composant **Composite_Unique**. L'Atelier B génère 89 preuves évidentes et 38 obligations non évidentes. Le prouveur B permet d'en réaliser 34. Il en reste alors 4 à prouver. Si on suppose que la machine abstraite unique de COMPOSITE est spécifiée et prouvée préalablement, la méthode combinant une inclusion et un raffinement génère 7 preuves non triviales, dont 2 sont liées aux nouvelles spécifications, tandis que la spécification directe implique 4 preuves difficiles. La comparaison n'est pas très satisfaisante, mais il s'agit d'un petit cas. De plus, les cinq preuves jugées faciles dans la machine de raffinement **Dossier_Machine** peuvent être réutilisées lors d'une instanciation ultérieure de **Composant_Unique** : une fois que ces preuves sont faites, il suffit d'utiliser les mêmes tactiques pour les reprouver.

Machines abstraites	Obv	nPO	Autom.	Reste	Diff.	Réut.
COMPOSITE						
Composite_Unique	39	21	19	2		
Méthode proposée						
Dossier_Renommage	25	0	0	0	0	0
Dossier_Machine	110	24	17	7	2	5
Total méthode	135	24	17	7	2	5
Spécification directe						
Dossier_Unique	89	38	34	4	4	0

Outre les notations Obv et nPO que nous avons déjà rencontrées, Autom. indique le nombre d'obligations prouvées automatiquement, Reste, la différence entre nPO et Autom., Diff. indique le nombre de preuves interactives jugées difficiles et Réut. est le nombre de preuves réutilisables lors d'une nouvelle instanciation. Si le nombre de preuves Reste est supérieur dans le cas de la méthode, on remarque que le nombre de preuves jugées difficiles est inférieur. D'autre part, la proportion de preuves évidentes par rapport au nombre d'obligations totales est supérieur dans le cas de la solution proposée. Ainsi, il y a moins d'obligations de preuves nPO, mais elles sont plus difficiles à prouver automatiquement. Enfin, il ne faut pas oublier que je me suis inspiré de la machine **Composite_Unique** pour spécifier **Dossier_Unique**. Par conséquent, la spécification "directe" utilise d'une certaine manière le composant COMPOSITE. Or le pattern Composite est par définition une solution de conception efficace concernant ce problème. Une spécification plus neutre (autrement dit, sans référence à COMPOSITE) de notre exemple aurait certainement généré plus d'obligations de preuve difficiles.

3.6 Conclusion

Nous avons proposé une méthode d'instanciation de composant utilisant uniquement le langage B. Cette étude nous a amené à faire des choix concernant la spécification des composants B.

On définit un *composant B* comme une machine abstraite **unique** qui décrit une manière de résoudre un problème. L'*instanciation B* est un **renommage** des ensembles, des variables et des opérations du composant B associé à une **adaptation** des opérations existantes et à un **ajout** de nouvelles opérations. Le terme renommage interdit en particulier de dupliquer les variables du composant B. Ces deux définitions répondent en partie aux problèmes que nous nous étions posés au début de ce chapitre.

Ces choix impliquent notamment la création d'un outil permettant de traduire un pattern UML en une machine unique B. L'instanciation se traduit alors en B par une inclusion suivie d'un raffinement. Cette méthode a l'avantage de pouvoir être poursuivie de manière classique par une série de raffinements. De plus, la phase d'inclusion est gratuite d'un point de vue obligation de preuve. Cependant, l'exemple d'application choisi étant simple, il faudrait tester cette méthode sur d'autres exemples, plus compliqués, afin de les étudier et d'en analyser les conséquences, notamment sur les preuves. En effet, la réutilisation implique en B une réutilisation des preuves associées. L'exemple de l'inclusion nous montre en effet que la preuve d'exactitude de la machine obtenue est une conséquence immédiate de celle du composant. La phase de raffinement reste toutefois encore à étudier. Mais un des avantages importants de cette solution est la séparation des preuves évidentes liées au renommage d'une part et des nouvelles obligations de preuve générées par les spécifications supplémentaires rajoutées au moment du raffinement, d'autre part.

Si le bilan semble plutôt encourageant, de nombreux points restent à éclaircir. Par exemple, l'aspect de la composition n'a pas été abordé dans ce rapport. De plus, la création de l'outil et la spécification de nouveaux composants sont deux points importants à développer. Enfin, une analyse plus approfondie des obligations de preuve paraît nécessaire.

Chapitre 4

Conclusions et perspectives

4.1 Apports

Nous avons, dans un premier temps, étudié les différents composants utilisés dans les approches objet, comme les patterns et les frameworks. Nous avons plus particulièrement abordé les aspects sémantiques afin d'analyser les différentes approches possibles. Nous en avons conclu qu'il existait finalement trois approches principales concernant la spécification formelle de composants :

- la création d'un langage spécifique et dédié à la conception par patterns (LePUS),
- l'adaptation d'un projet existant afin de rendre sa sémantique plus précise (Catalysis),
- et enfin, l'utilisation d'un langage de spécification existant, quitte à limiter les possibilités de réutilisation (RAISE).

En outre, nous avons constaté qu'il existait peu de travaux concernant la réutilisation de composants en B.

À partir de cette étude préliminaire, nous avons cherché à traduire les concepts de composant et d'instanciation en B, en se restreignant à une stricte utilisation des propriétés de B. Cette phase d'exploration et d'expérimentation nous a conduit à plusieurs conclusions :

1. Afin de rendre le composant B réutilisable, nous avons besoin de le spécifier et de le prouver sous forme d'une machine abstraite unique. Par conséquent, on ne peut pas se limiter au cadre strict du langage B. Nous avons besoin au moins d'un outil d'assistance afin de rendre cette machine unique "lisible". Il serait en effet difficile de conserver dans une bibliothèque chaque composant B sous la forme d'une machine unique : le programmeur ne pourrait pas comprendre les intérêts et les subtilités des différents composants.
2. D'autre part, nous n'avons pas besoin d'adapter le langage B pour instancier le composant. L'instanciation que nous avons définie est entièrement spécifiée en B :
   - une inclusion permet de renommer les variables d'état et de déclarer les nouveaux noms d'opérations,
   - un raffinement permet ensuite d'adapter et de compléter la spécification des opérations.

L'aspect composition n'a pas encore été étudié.

La méthode d'instanciation B proposée présente plusieurs avantages. Elle utilise uniquement les propriétés et les mécanismes de B. De plus, elle est compatible avec une phase successive de raffinements. Enfin, elle permet, au moment de la preuve de cohérence des machines, de distinguer :

– les preuves évidentes liées au simple renommage des ensembles et des variables du composant : elles sont réalisées dans la machine de la première étape,
– et les obligations de preuve, plus difficiles, liées aux nouvelles spécifications dans les opérations : elles sont prouvées dans la machine de la deuxième étape.

Cette proposition n'est qu'une ébauche de solution. Cette méthode ne permet pas toujours de simplifier le nombre de preuves lors d'une première instanciation. En revanche, il est possible de diminuer le nombre de preuves lors des instanciations suivantes du même composant en réutilisant certaines preuves. L'application de cette méthode est longue et difficile. Un outil d'assistance est peut-être nécessaire. Enfin, la phase de composition impliquera certainement des modifications. Le tableau 4.1 est un bilan de notre méthode comparée aux approches étudiées dans le chapitre 2.

TAB. 4.1 – *Comparaison de notre approche avec les autres méthodes*

	LePUS	Framework	RSL	UML et B	**B**
Paragraphe	2.5.2	2.5.4	2.5.3	2.4.1	3.4.2
Point de départ	rien	Catalysis	VDM	UML et B	B
Modèle	HOML	1er ordre	VDM	mach. abst.	mach. abstr.
Outils	graph.	Catalysis	non	UML, B	B
Instanciation	proj.	param.	map	à la main	inclusion + raffin.
Composition	non	non comm.	à la main	à la main	non étudié
Génération de code	non	Catalysis	non	raffin.	raffin.
Pratique	$--$	$+$	$-$	$+$	$-$
Abstrait ?	$++$	$+$	$-$	$+$	$+$
Formel ?	formel	formel	formel	semi-formel	formel

À la différence des autres méthodes, notre approche est entièrement spécifiée formellement. De plus, elle utilise uniquement le langage B, contrairement à la solution UML - B. Il manque enfin la composition pour compléter le tableau de comparaison.

4.2 Perspectives

Comme nous l'avons déjà remarqué à plusieurs reprises, l'aspect composition n'a pas été traité dans cette étude. Il existe en fait plusieurs approches possibles.

Une possibilité consiste à combiner les différentes instanciations obtenues. Dans ce cas, la phase de raffinement reste à étudier afin de trouver une propriété de B permettant de regrouper et de réutiliser les différentes machines de raffinement obtenues avec la méthode d'instanciation proposée. Les liens de type **INCLUDES** ou **IMPORT** constituent peut-être une solution. L'avantage d'une composition après instanciation est la possibilité de combiner plusieurs instances du même composant. Une autre approche consiste à combiner les composants avant l'instanciation. Dans ce cas, une solution consiste éventuellement à créer un outil cumulant les fonctions de composition et d'instanciation de composants. De toute manière, un outil sera nécessaire à la traduction automatique des composants en machines uniques.

Afin de spécifier la composition et de créer l'outil, nous aurons besoin d'exemples supplémentaires. Ils nous permettront aussi d'analyser les conséquences d'une telle démarche sur la réutilisation de preuves. L'exemple d'instanciation présenté dans ce rapport n'a qu'une portée limitée sur les preuves de cohérence. De nouveaux exemples permettront de confirmer ou d'infirmer l'hypothèse selon laquelle la méthode d'instanciation présentée peut faire diminuer le nombre de preuves difficiles, grâce à une réutilisation implicite des preuves déjà réalisées sur les composants. Une piste consiste peut être à créer dans l'outil d'assistance un complément du prouveur actuel afin de le rendre plus efficace.

En conclusion, même si de nombreux points restent à traiter, les premiers essais réalisés nous ont permis de spécifier formellement, pour la première fois en B, un petit exemple d'instanciation de composant. Si cet exemple peut se généraliser, il sera alors possible de construire une spécification formelle en B à partir de composants génériques prédéfinis et prouvés, conservés dans une bibliothèque. Nous sommes convaincus que ce type de réutilisation peut se répercuter au niveau des preuves d'exactitude des machines B.

Bibliographie

[1] J.R. Abrial. *The B-Book : Assigning programs to meanings.* Cambridge University Press, 1996.

[2] C. Alexander. *The Timeless Way of Building.* Oxford University Press, New York, 1979.

[3] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, and S. Angel. *A Pattern Language.* Oxford University Press, New York, 1977.

[4] G. Aranda and R. Moore. Formally modelling compound design patterns. Technical Report 225, UNU/IIST, P.O. Box 3058, Macau, December 2000.

[5] G. Aranda and R. Moore. GoF creational patterns : A formal specification. Technical Report 224, UNU/IIST, P.O. Box 3058, Macau, December 2000.

[6] A. Cechich and R. Moore. A formal specification of GoF design patterns. Technical Report 151, UNU/IIST, P.O. Box 3058, Macau, January 1999.

[7] Vincent Couturier. Des patterns pour la coopération de systèmes d'information : application à l'architecture coopérative ACSIS. *Journée du travail bi-thématique du GDR-PRC I3*, December 2001.

[8] I. Crnkovic, K. Ster, J. Larsson, and M. Lau. Object-oriented design frameworks: Formal specification and some implementation issues. *Proc. of 4th IEEE International Baltic Workshop in DB and IS*, 2000.

[9] A. Eden, Y. Hirshfeld, and A. Yehudai. LePUS - a declarative pattern specification language. Technical report 326/98, Department of Computer Science, Tel Aviv University, 1998.

[10] A. Eden, H. Joseph, Y. Gil, and A. Yehudai. A formal language for design patterns. *Proc. of PLoP USA*, 1996.

[11] A. H. Eden. *Precise Specification of Design Patterns and Tool Support in Their Application.* PhD thesis, Department of Computer Science, Tel Aviv University, 2000.

[12] A. H. Eden. Formal specification of object-oriented design. *International Conference on Multidisciplinary Design in Engineering CSME-MDE 2001*, November 21-22, Montreal, Canada.

[13] A. H. Eden and Y. Hirshfeld. Principles in formal specification of object-oriented design and architecture. *CASCON 2001*, November 5-8, Toronto, Canada.

[14] A. H. Eden, Y. Hirshfeld, and A. Yehudai. Towards a mathematical foundation for design patterns. Technical Report 1999-004, Department of Information Technology, Uppsala University, 1999.

[15] J. Filipe, K. Lau, M. Ornaghi, K. Taguchi, A. Wills, and H. Yatsu. Formal specification of catalysis frameworks. *Proc. 7th Asia-Pacific Software Engineering Conference*, pages 180–187. IEEE Computer Society Press, 2000.

[16] J. Kuster Filipe, K.-K. Lau, M. Ornaghi, and H. Yatsu. On dynamic aspects of OOD frameworks in component-based software development in computational logic. In A. Bossi, editor, *Proc. LOPSTR 99, Lecture Notes in Computer Science*, volume 1817, pages 43–62. Springer-Verlag, 2000.

[17] J. Kuster Filipe, K.-K. Lau, M. Ornaghi, and H. Yatsu. Intra- and inter-OOD-framework interactions in component-based software development in computational logic. In A. Brogi and P. Hill, editors, *Proc. of the Second International Workshop on Software Development in Computational Logic, Paris, France*, September 1999.

[18] A. Flores and R. Moore. GoF structural patterns : A formal specification. Technical Report 207, UNU/IIST, P.O. Box 3058, Macau, August 2000.

[19] A. Flores, L. Reynoso, and R. Moore. A formal model of object-oriented design and GoF design patterns. Technical Report 200, UNU/IIST, P.O. Box 3058, Macau, July 2000.

[20] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns : Elements of reusable object-oriented software*. Addison-Wesley, 1995.

[21] C. George, P. Haff, K. Havelund, A. Haxthausen, R. Milne, C. Nielsen, S. Prehn, and K. Wagner. *The RAISE Specification Language*. Prentice-Hall, 1992.

[22] P. Grogono and A. H. Eden. Concise and formal descriptions of architectures and patterns. Technical Report, Department of Computer Science, Concordia University, 2001.

[23] H. Habrias. *Spécification formelle avec B*. Hermes Sciences. Lavoisier, 2001.

[24] K. Lau, S. Liu, M. Ornaghi, and A. Wills. Interacting frameworks in catalysis. *Proc. 2nd IEEE Int. Conf. on Formal Engineering Methods*, Brisbane, Australia, 9-11 December 1998.

[25] K. Lau, M. Ornaghi, and A. Wills. Frameworks in catalysis: Pictorial notation and formal semantics. *Proc. 1st IEEE Int. Conf. on Formal Engineering Methods*, pages 213-220, 1997.

[26] K.-K. Lau and M. Ornaghi. OOD frameworks in component-based software development in computational logic. In P. Flener, editor, *Proc. LOPSTR 98, Lecture Notes in Computer Science 1559*, pages 101–123. Springer-Verlag, 1999.

[27] K.-K. Lau and M. Ornaghi. On specification and correctness of ood frameworks in component-based software development in computational logic. In A. Brogi and P. Hil, editors, *Proc. of the First International Workshop on Software Development in Computational Logic, Pisa, Italy*, pages 59–75, September 1998.

[28] K.-K. Lau and M. Ornaghi. A formal approach to software component specification. In G.T. Leavens D. Giannakopoulou and M. Sitaraman, editors, *Proc. of Specification and Verification of Component-based Systems Workshop at OOPSLA2001*, pages 88–96, Tampa, USA, October 2001.

[29] Amel Mammar. *Un environnement formel pour le développement d'applications à forte composante données*. PhD thesis, CNAM, 2002.

[30] R. Marcano, E. Meyer, N. Levy, and J. Souquieres. Utilisation de patterns dans la construction de spécifications en UML et B. *Journées AFADL'2000 : Approches formelles dans l'assistance au développement de logiciels*, Publication LSR, A00-R-009, January 2000.

[31] Claudia Marcos, Marcelo Campo, and Alain Pirotte. Reifying design patterns as metalevel constructs. *Proc. of the 2nd Argentine Symp. on Object Orientation, ASOO'98, Buenos Aires, Argentina*, August 1998.

[32] P.A. Muller. *Modélisation objet avec UML*. Eyrolles, 1997.

[33] H.P. Nguyen. *Dérivation de spécifications formelles B à partir de spécifications semi-formelles*. PhD thesis, CNAM, 1998.

[34] L. Reynoso and R. Moore. GoF behavioural patterns : A formal specification. Technical Report 201, UNU/IIST, P.O. Box 3058, Macau, May 2000.

[35] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorenson. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[36] L. Tokuda and D. Batory. Automated software evolution via design pattern transformations. *Proc. of the 3rd International Symposium on Applied Corporate Computing, Monterrey, Mexico*, October 1995.

[37] Alan Wills. Frameworks and component-based development. In *Object Oriented Information Systems*, 1996.

[38] Martin Wirsing. *Formal Models and Semantics*, volume B of *Handbook of Theoretical Computer Science*, chapter Algebraic Specification, pages 675–788. Jan Van Leeuwen, 1990.

Annexe A

Notations graphiques

UML, en anglais : "*Unified Modeling Language*", est un langage utilisé pour représenter les systèmes orientés objet grâce à tout un ensemble de notations graphiques et de règles de construction. À l'origine, l'objectif était en effet d'unifier les multiples méthodes objet, comme Booch, OMT, ... Cette annexe a pour but de présenter quelques notations UML utilisées notamment dans les principales références de ce rapport. Elle s'appuie sur [30, 32].

Classes d'objets : Une classe est représentée par une boîte de trois cases de la forme suivante :

Classe
attributs
opérations

où la case classe indique le nom de l'identifiant de la classe tandis que les cases attributs et opérations contiennent les listes d'attributs et d'opérations de la classe en question. Ces deux dernières cases ne sont pas nécessairement remplies.

Associations : Une fois les classes définies, il est possible de relier les différentes classes par des associations. Dans un diagramme de classes, elles sont représentées comme décrites dans la figure A.1. La multiplicité de l'association

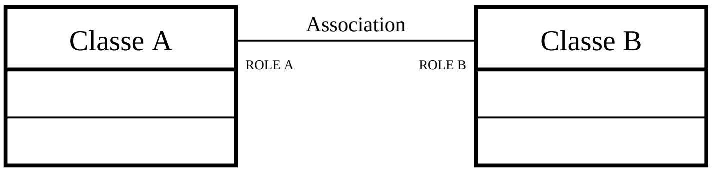

Fig. A.1 – *Association*

est précisée à l'aide des notations suivantes :

*	zéro ou plusieurs
1 .. *	1 ou plusieurs
1	un
m .. n	de m à n

La figure A.2 représente une association de type héritage. Les classes B et C spécialisent la classe A.

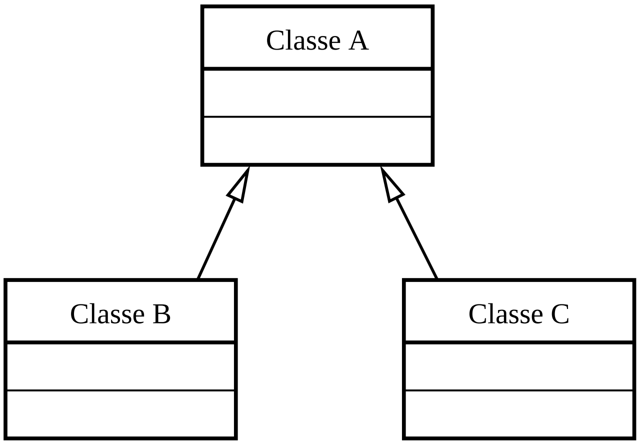

FIG. A.2 – *Héritage*

Annexe B

Description GoF des Patterns utilisés

Cette annexe contient les descriptions GoF des différents patterns de conception utilisés dans les exemples d'utilisation et dans les spécifications. Elle s'appuie sur le catalogue GoF [20].

B.1 Abstract Factory

Quand des objets instanciés de classes différentes appartiennent à une famille commune, possédant le même type de propriétés par exemple, ou sont reliés entre eux et que leurs classes respectives ne sont pas des sous-classes d'une même classe abstraite, le pattern Abstract Factory permet de créer une classe abstraite commune dont une sous-classe concrète regroupe les différents objets de la même famille. La description GoF est la suivante (pour de plus amples détails, se référer aux pages 87-96 du catalogue) :

Nom ABSTRACT FACTORY.

Classification Objet création.

Intention Il crée une interface pour des objets reliés ou dépendants d'une même famille sans spécifier leurs classes concrètes.

Alias KIT.

Motivation Considérons une interface utilisateur supportant des standards multiples "look-and-feels" comme Motif ou Presentation Manager. Ces standards définissent des apparences et des comportements différents pour une interface "widget", comme des "scroll bars", des "windows" ou des "buttons". Pour être transparente à travers tous ces standards, une application ne doit pas être spécifique à une apparence ou un comportement particulier. L'instanciation des classes de "widget" pour une application dans un certain "look-and-feel" rend difficile son adaptation dans un autre standard par la suite.

La solution consiste à définir une classe abstraite WidgetFactory qui déclare une interface pour créer chaque type de base de "widgets". Il y a aussi une classe abstraite pour chacun de ces types de base et des sous-classes concrètes qui implémentent les widgets pour des standards spécifiques.

Indications d'utilisation On utilise ABSTRACT FACTORY quand :

- un système est indépendant de la manière dont les produits sont créés, composés ou représentés,
- un système doit être configuré avec une des multiples familles de produits,
- des objets de produits d'une même famille doivent être utilisés ensemble,
- on veut fournir une librairie de classes de produits et qu'on s'intéresse uniquement à leurs interfaces et non à leur implémentation.

Structure Elle est présentée dans la figure B.1.

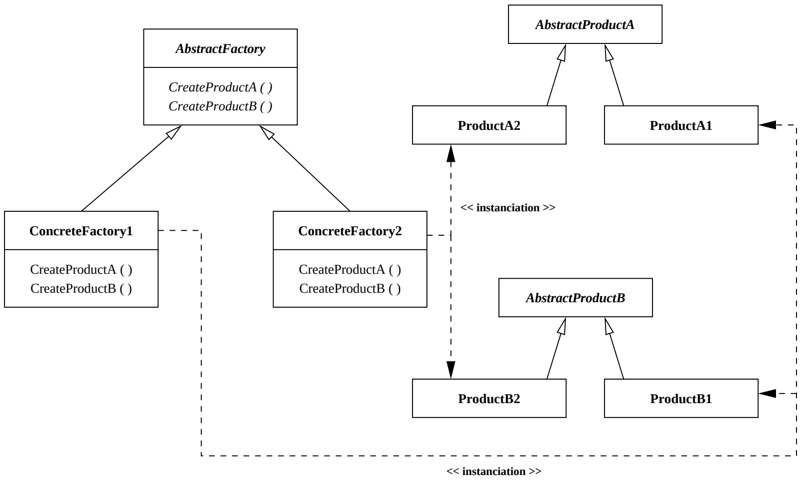

FIG. B.1 – *Structure du pattern Abstract Factory*

Constituants Le pattern est constitué de :

- AbstractFactory : il déclare une interface pour les opérations créant des objets produit abstraits.
- ConcreteFactory : il implémente les opérations qui créent des objets produit concrets.
- AbstractProduct : il déclare une interface pour un type d'objet produit.
- ConcreteProduct : il définit un objet produit créé par le ConcreteFactory correspondant et il implémente l'interface AbstractProduct.

L'usager utilise les interfaces déclarées par les classes AbstractFactory et AbstractProduct.

Collaboration Une instance simple de ConcreteFactory est créée lors de l'exécution. Elle crée des objets produits avec une implémentation particulière. Pour créer des objets produits différents, le Client doit utiliser une nouvelle ConcreteFactory. La création des produits est transférée de AbstractFactory vers ses sous-classes ConcreteFactory.

Conséquences Le pattern ABSTRACT FACTORY :

- isole les classes concrètes.

- rend les échanges de familles de produits plus faciles.
- apporte une consistence parmi les produits.
- rend l'ajout de nouvelles familles de produits plus facile.

Implémentation Voir GoF Catalogue.

Exemple de code Voir GoF Catalogue.

Utilisations Voir GoF Catalogue.

Patterns apparentés Liste des patterns concernés :
- les classes AbstractFactory sont souvent implémentées avec Factory Method ou Prototype,
- un ConcreteFactory est souvent un Singleton.

B.2 Composite

Lorsqu'un usager veut traiter de façon transparente des objets simples ou des objets composés, Composite apporte une solution en créant une classe abstraite qui représente les deux types d'objet. La description GoF est la suivante (pour de plus amples détails, se référer aux pages 163-173 du catalogue) :

Nom COMPOSITE.

Classification Objet structure.

Intention Il décompose les objets en des structures arborescentes afin de représenter les hiérarchies des différentes parties de l'ensemble. COMPOSITE permet aux clients de traiter les objets individuels et les objets composés de manière uniforme.

Motivation Les applications graphiques comme par exemple des éditeurs graphiques permettent de construire des diagrammes complexes à partir d'objets simples. L'usager peut ainsi regrouper des composants assez simples pour en former des plus complexes, et ainsi de suite. Une simple implémentation permet de définir des classes primitives d'une part et d'autres classes qui contiennent ces classes primitives, d'autre part. Par exemple, dans un éditeur graphique, les primitives peuvent être des lignes et du texte alors qu'un objet plus complexe est un rectangle avec du texte.

Mais il y a un problème avec cette approche : le code traite de manières différentes les classes primitives et les classes composées à partir de classes primitives, bien que ce soit transparent pour l'usager. Traiter de manières différentes ces classes rend l'application plus complexe. Le pattern COMPOSITE décrit comment utiliser une composition récursive en toute transparence pour l'usager. L'idée est de créer une classe abstraite qui représente à la fois les classes primitives et les compositions.

Indications d'utilisation On utilise le pattern COMPOSITE quand :
- on veut représenter la hiérarchie d'un ensemble d'objets,
- on veut que les clients ne soient pas obligés de différencier les objets simples des objets composés.

Structure La figure B.2 est le diagramme UML du pattern Composite.

Constituants Les constituants du pattern sont :
- Component : il déclare l'interface des objets dans la composition, il implémente le comportement par défaut de l'interface commune à

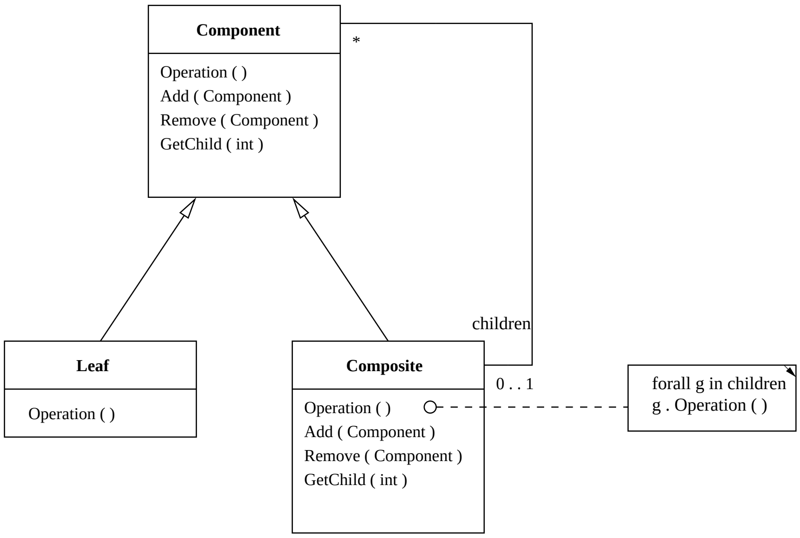

FIG. B.2 – *Structure du pattern Composite*

toutes les classes et il définit enfin une dernière interface pour accéder aux composants fils et pouvoir les gérer.

- Leaf : il représente les feuilles dans l'arbre de composition, il n'a pas de fils et il déclare le comportement des objets primitifs.
- Composite : il définit le comportement des composants ayant des fils et il implémente les opérations les concernant dans l'interface Component.
- L'usager manipule les objets dans la composition à travers l'interface Component.

Collaboration Les clients utilisent l'interface de la classe Component pour interagir avec les objets de la composition. Si l'objet en question est Leaf, alors la demande est traitée directement. Sinon, il transfère les demandes aux composants fils. Il peut éventuellement faire des opérations supplémentaires avant ou après le transfert.

Conséquences Le pattern COMPOSITE :

- définit les hiérarchies d'un ensemble d'objets simples et composés. Les objets primitifs peuvent être composés pour en former de plus complexes et ainsi de suite.
- rend Client plus simple. Les clients traitent les objets simples de la même façon que les objets composés : le code est simplifié.
- rend l'ajout de nouveaux composants plus simple. Les nouveaux Composite ou Leaf s'intègrent automatiquement avec les existants.
- peut rendre la conception trop générale. Le défaut de rendre les ajouts plus simples est de rendre les restrictions de composants plus difficiles à l'intérieur d'un même Composite.

Implémentation Voir GoF catalogue.

Exemple de code Voir GoF catalogue.

Utilisations Voir GoF catalogue.

Patterns apparentés Liste des patterns concernés :

- le lien parent-composant est souvent utilisé par Chain of Responsibility,
- Decorator est souvent combiné avec Composite,
- Flyweight permet de partager les composants mais ne peut plus se référer aux parents,
- Iterator peut être utilisé pour traverser les Composites,
- et Visitor localise les opérations qui seraient distribuées autrement par les classes Leaf et Composite.

B.3 Factory Method

On ne peut pas instancier une classe abstraite. Il est donc parfois utile de pouvoir laisser une sous-classe créer un objet pour le compte d'une autre classe. Le pattern Factory Method apporte une solution à ce problème. Le résumé de sa description dans le GoF catalogue aux pages 107-116 est :

Nom FACTORY METHOD.

Classification Classe création.

Intention Il définit une interface pour créer un objet, mais laisse les sous-classes décider quelle classe instancier. FACTORY METHOD permet à une classe de déléguer l'instanciation aux sous-classes.

Alias VIRTUAL CONSTRUCTOR.

Motivation Les frameworks utilisent des classes abstraites pour définir les relations entre objets. Considérons un framework d'application qui présente plusieurs documents à un utilisateur. Les deux classes abstraites importantes sont Application et Document. Par exemple, pour créer une application de dessin, on définit les sous-classes DrawingApplication et DrawingDocument. La classe Application est responsable de la gestion des Documents et doit parfois en créer de nouveaux. Comme la sous-classe DrawingDocument de la classe abstraite Document est spécifique à cette application de dessin, la classe Application ne peut pas prédire la sous-classe de Document à instancier (elle sait quand un document a été créé mais elle ne connaît pas son type). D'où le problème : le framework doit instancier une classe dont il connaît seulement la classe abstraite qu'il n'est pas possible d'instancier.

La solution apportée par le pattern est la suivante : on encapsule les connaissances concernant la sous-classe de Document à instancier et on l'exporte hors du framework. Les sous-classes de Application redéfinissent une opération abstraite CreateDocument sur Application qui retourne la sous-classe de Document appropriée. Une fois q'une sous-classe d'Application est instanciée, elle peut ensuite instancier les documents spécifiques sans connaître leurs classes. On appelle l'opération CreateDocument une "factory method" car elle "fabrique" un objet.

Indications d'utilisation On utilise FACTORY METHOD quand :

- une classe ne peut pas anticiper la classe des objets qu'elle doit créer,
- une classe exige que ses sous-classes spécifient la classe des objets qu'elle crée,

– les classes délèguent une responsabilité à une des sous-classes qui l'aident et qu'on veut pouvoir localiser quelle est la sous-classe en question.

Structure La figure B.3 est la structure du pattern FACTORY METHOD.

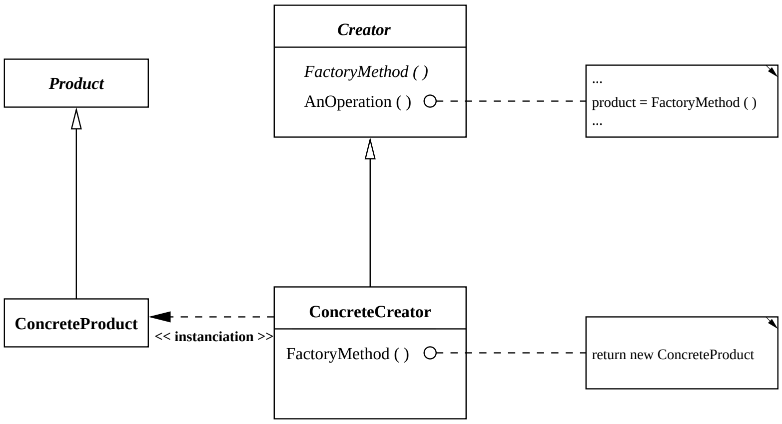

FIG. B.3 – *Structure du pattern Factory Method*

Constituants Le pattern est constitué de :

– Product : il définit l'interface des objets que la "factory method" crée.

– ConcreteProduct : il implémente l'interface Product.

– Creator : il déclare la factory method qui retourne un objet de type Product et il peut appeler cette méthode pour créer un objet.

– ConcreteCreator : il utilise la factory method pour retourner une instance de ConcreteProduct.

Collaboration La classe abstraite Creator s'appuie sur ses sous-classes pour définir la factory method de telle sorte qu'elle retourne une instance du ConcreteProduct approprié.

Conséquences Les factory methods évitent de détailler les classes spécifiques dans le code, on s'intéresse uniquement aux interfaces des Products. Le principal défaut est d'obliger le client à sous-classer Creator juste pour créer un objet particulier. Enfin, le pattern FACTORY METHOD :

– permet de raccrocher les sous-classes entre elles.

– relie les hiérarchies de classes paralèlles.

Implémentation Voir GoF Catalogue.

Exemple de code Voir GoF Catalogue.

Utilisations Voir GoF Catalogue.

Patterns apparentés Liste des patterns concernés :

– les classes AbstractFactory sont souvent implémentées avec Factory Method,

– les factory methods sont souvent appelées dans Template Method,

– Prototype n'a pas besoin de sous-classer Creator dans Factory Method. De plus, il utilise une opération Initialize dans la classe Product.

Creator en revanche se sert de Initialize pour initialiser un objet. Factory Method n'a pas besoin d'une telle opération.

Annexe C

Exemples

Nos recherches concernant l'état de l'art sur la réutilisation de composants nous ont conduit à étudier de nombreux articles sur l'utilisation de patterns orientés objet. Ces articles nous ont permis notamment de situer notre sujet en fonction de l'existant et de nous donner des exemples d'utilisation des patterns. Cette annexe regroupe donc les principaux exemples étudiés lors de ces recherches qui n'ont pas été traités dans le rapport pour des raisons de place mais dont il fait parfois référence :

1. la création d'une architecture réflexive [31],
2. la création d'un outil pour générer du code à partir d'une conception basée sur les patterns [36],

C.1 Architecture réflexive [31]

Afin de faciliter la maintenance des logiciels conçu à l'aide de patterns, il est souhaitable de retrouver dans le code les différents patterns de conception. Il existe actuellement deux axes de recherche à ce sujet : la création d'un outil qui génère des squelettes du code à partir de la conception avec patterns ou bien l'identification au niveau du code d'un ou de plusieurs patterns associés éventuels.

L'idée de ce premier exemple est d'utiliser une architecture réflexive à double niveau pour créer une interaction entre le niveau de base du développement et le niveau plus abstrait des patterns. Les auteurs définissent alors un modèle *réflexif* constitué :

- du niveau *meta* ou réflexif, qui représente les patterns à l'aide de classes de meta-objets,
- et du niveau de *base*, qui contient les informations spécifiques aux applications en cours de développement.

La conception de ce modèle s'effectue alors en trois étapes :

1. On commence par représenter les patterns de conception au niveau meta à l'aide de meta-objets. On appelle cette phase une *réification*, dans le sens où les patterns jouent désormais le rôle de constructeurs au niveau meta capables de fournir les structures de contrôle qui régissent le comportement du programme. On a donc un système réflexif : il agit sur lui-même. Les

classes de meta-objets sont implémentées indépendamment du niveau de base.

2. Ensuite, on crée des liens entre le niveau de base et le niveau meta grâce à des associations qui indiquent qu'un pattern est utilisé au niveau de l'application.
3. Enfin, on met en place le *mécanisme de réflexion* qui permet au système de réagir lorsqu'un message est envoyé à un objet de bas niveau et de le retransmettre, le cas échéant, au pattern concerné situé au niveau meta.

Le document fournit l'exemple (voir figure C.1) d'un éditeur graphique obtenu par instanciation du pattern Composite (voir annexe B.2) qui permet de

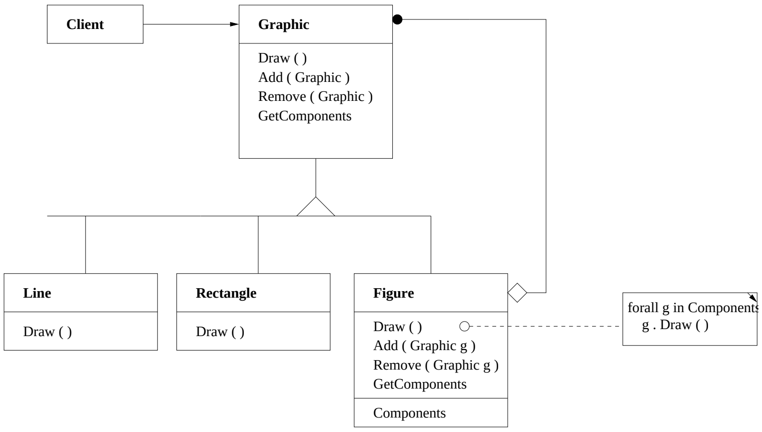

FIG. C.1 – *Exemple : Editeur graphique*

traiter des problèmes avec des objets composés d'objets élémentaires, dans notre cas des figures composées de lignes et de rectangles. Pour représenter le pattern Composite, on utilise la classe de meta-objets *MOComposite*. L'objectif de cette classe est de redistribuer une opération demandée pour un objet complexe à tous ses composants et de fournir ensuite le résultat cherché. Quatre méthodes sont définies dans la classe *MOComposite* :

- *MMGetComponents*, qui retourne les composants de l'objet composite. Elle utilise notamment l'opération *GetComponents* et la variable *Components* dans le niveau de base.
- *MMActualizeComponents*, qui peut affecter de nouvelles valeurs aux composants.
- *MMComposite*, qui retrouve les composants à l'aide de *MMGetComponents* et qui redistribue à chacun l'opération donnée en argument.
- *MMAdd* (respectivement *MMRemove*) qui ajoute (respectivement supprime) un composant dans un objet composite.

Pour associer le niveau de base et le niveau meta, on considère un meta-objet *moComposite* défini comme une instance de la classe *MOComposite*. On associe alors la classe *Figure* au niveau de base avec l'objet *moComposite* au niveau meta. On peut alors mettre en marche le mécanisme de réflexion. Par exemple, on suppose qu'un objet composite *complexfigure* est composé d'objets simples *myline* et *myrectangle*. La figure C.2 est alors le scénario du mécanisme de

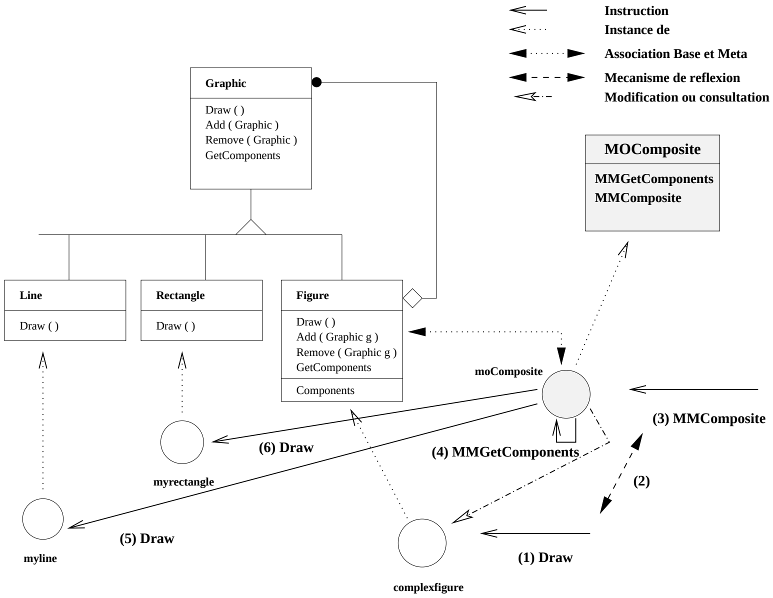

FIG. C.2 – *Mécanisme de réflexion sur l'éditeur graphique lorsque l'opération Draw est appelée*

réflexion lorsque l'opération *Draw* est appelée. Quand *complexfigure* reçoit le message *Draw* (1), le processus est redirigé vers l'objet *moComposite* (2) au niveau meta par le mécanisme de réflexion. Il appelle la méthode *MMComposite* (3) pour retrouver les composants de *complexfigure*. *MMComposite* utilise alors la méthode *MMGetComponents* (4) qui consulte la variable *Components* introduite par le pattern et qui retourne les composants : *myline* et *myrectangle*. Ensuite, *MMComposite* appelle l'opération *Draw* dans *myline* (5) et dans *myrectangle* (6). Enfin, le processus de contrôle revient à la méthode *Draw* de la figure au niveau de base, qui termine l'exécution. En cas d'appel reçu par *myline* ou par *myrectangle*, le mécanisme de réflexion ne réagit pas.

Ce modèle réflexif permet ainsi de créer une interaction entre le niveau de base et le niveau meta grâce à un mécanisme de réflexion. Par conséquent, le système conserve une trace entre la conception par pattern et les applications générées. Le but de réduire le coût de maintenance est donc atteint car il est alors possible de réutiliser la structure ou bien de rajouter de nouveaux patterns de manière dynamique en faisant attention aux nouvelles classes utilisées. Le protocole de réalisation est cependant contraignant pour le premier concepteur et son utilisation nécessite l'emploi d'un langage orienté-objet supportant les concepts de meta-objets.

C.2 Transformations de patterns [36]

L'idée de cet article est d'interpréter les patterns de conception comme des transformations. Posons un exemple. Le pattern Abstract Factory (voir Annexe B.1 ou bien [20] pages 87-96 pour plus de détails) permet de factoriser des associations de type héritage entre des familles de classes et sous-classes différentes. Dans la pratique, si les classes *ScrollBar* et *Window* ont pour sous-classes respectives OpenLookScrollBar et MotifScrollBar d'une part et OpenLookWindow et MotifWindow d'autre part, alors le pattern Abstract Factory donne comme solution la création d'une classe abstraite unique *WindowFactory* avec deux sous-classes OpenLookFactory et MotifFactory (voir la figure C.3). La première

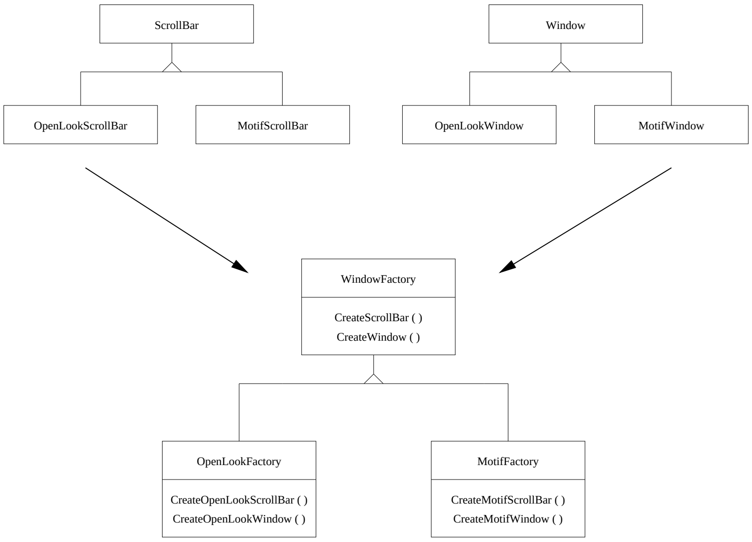

FIG. C.3 – *Utilisation du pattern Abstract Factory*

sous-classe ne s'occupe que des objets implémentés en OpenLook tandis que la seconde est destinée aux objets implémentés en Motif. Cette solution peut être vue comme l'état final d'une transformation. L'idée est de considérer des transformations dont la cible serait la solution préconisée par le pattern : de telles opérations sont appelées des *transformations de pattern*.

Il serait alors possible de réaliser sur le programme une succession de transformations de pattern simples comme par exemple :

1. rajouter des classes,
2. échanger des classes,
3. réutiliser des applications,
4. ou créer des relations de type sous-classes et sur-classes.

Le problème revient à définir de telles opérations. Un exemple de transformation est donné par **FactoryMethod**[$C1$,$C2$,$m()$,$C3$,v]. Elle rajoute une méthode $m()$ dans la classe $C1$ qui crée de nouveaux objets de la classe $C2$. Un appel de $m()$ remplace alors toutes les occurrences de ces objets dans le programme. De plus, v est une expression retournant un objet de $C1$ qui peut être donné en argument à la méthode $m()$. Enfin, $C3$ est la classe des objets retournés par m : il s'agit bien sûr de $C2$ ou bien d'une sur-classe de $C2$. Cette transformation inspirée du pattern Factory Method (voir Annexe B.2 et [20] pages 107-116) est définie à la main. Les auteurs espèrent ainsi créer de nombreuses transformations de pattern pour les analyser et les comparer.

Un exemple d'application est fourni avec le cas d'une usine de Honda Prelude. On obtient effectivement à la fin de la série de transformations une usine plus complexe. On constate en analysant le code des applications que les modifications induites par ces transformations pourraient se faire automatiquement grâce à un outil.

Quelles remarques peut-on faire concernant cet exemple ? D'une part, les changements du code semblent automatisables d'où l'intérêt d'un tel outil. D'autre part, il est plus facile de constater les effets des changements sur un diagramme que sur du code. Les auteurs proposent donc la création d'un outil qui aurait trois fonctions :

1. production d'un diagramme de classes à partir du code des applications,
2. transformation et évolution du diagramme par l'utilisateur,
3. génération automatique du code correspondant.

Cet outil pratique et facile d'utilisation permettrait des modifications du logiciel au niveau d'un diagramme et le générateur de code minimiserait les risques d'erreurs dûes aux interventions humaines. Bien que le concept soit intéressant, l'outil reste à construire. Seules quelques transformations sont définies à l'heure actuelle et il reste beaucoup de travail en perspective.

Annexe D

Composant en B

D.1 Pattern Composite traduit en B

Il est possible de traduire le diagramme (figure B.2) de la structure du pattern Composite en B [33]. Les spécifications obtenues á la main sont les suivantes.

Machine Composant

MACHINE *Composant_Machine*

SETS
$COMPOSANT$

VARIABLES
$Composant$

INVARIANT
$Composant \subseteq COMPOSANT$

INITIALISATION
$Composant := \emptyset$

OPERATIONS
$Ajout\_Composant(elt) =$
pre
$elt \in COMPOSANT - Composant$
then
$Composant := Composant \cup \{elt\}$
end;
$Retirer\_Composant(elt) =$
pre
$elt \in Composant$
then
$Composant := Composant - \{elt\}$

end

Machine Composite

MACHINE *Composite_Machine*

USES *Composant_Machine*

VARIABLES
Composite

INVARIANT
$Composite \subseteq Composant$

INITIALISATION
$Composite := \emptyset$

OPERATIONS
$Ajout\_Composite(elt) =$
pre
$elt \in COMPOSANT - Composite$
then
$Composite := Composite \cup \{elt\}$
end;
$Retirer\_Composite(elt) =$
pre
$elt \in Composite$
then
$Composite := Composite - \{elt\}$
end;
$Operation\_Composite(elt) =$
pre
$elt \in Composite$
then
SKIP
end

Machine Feuille

MACHINE *Feuille_Machine*

USES *Composant_Machine*

VARIABLES
Feuille

INVARIANT
$Feuille \subseteq Composant$

INITIALISATION
$Feuille := \emptyset$

OPERATIONS
$Ajout\_Feuille(elt) =$
pre
$elt \in COMPOSANT - Feuille$
then
$Feuille := Feuille \cup \{elt\}$
end;
$Retirer\_Feuille(elt) =$
pre
$elt \in Feuille$
then
$Feuille := Feuille - \{elt\}$
end;
$Operation\_Feuille(elt) =$
pre
$elt \in Feuille$
then
SKIP
end

Machine Pere

MACHINE $Pere\_Machine$

USES $Composant\_Machine$, $Composite\_Machine$

VARIABLES
$Pere$

INVARIANT
$Pere \in Composant \nrightarrow Composite$

INITIALISATION
$Pere := \emptyset$

OPERATIONS
$Ajout\_Enfants(parent, enfant) =$
pre
$parent \in Composite \wedge enfant \in Composant \wedge enfant \notin \mathrm{DOM}(Pere)$
then
$Pere := Pere \cup \{enfant \mapsto parent\}$
end;
$Retirer\_Enfants(pere) =$
pre
$pere \in Composant$
then

any $chemin$
where
$chemin \in Composant \leftrightarrow Composite$
$\wedge \forall(xx,yy).\ ((xx \mapsto yy) \in chemin \Rightarrow (yy \in Pere^{-1}[\{xx\}]$
$\vee \exists zz.(zz \in Pere^{-1}[\{xx\}] \wedge (zz \mapsto yy) \in chemin\ )))$
then
$Pere :=$
$(\{pere\} \cup chemin[\{pere\}]) \lhd Pere \rhd (\{pere\} \cup chemin[\{pere\}])$
end
end;
$enfants \longleftarrow DonnerEnfants(parent) =$
pre
$parent \in Composite \wedge parent \in \mathrm{RAN}(Pere)$
then
$enfants := Pere^{-1}[\{parent\}]$
end

Machine Interface du pattern Composite

MACHINE $Interface\_Machine$

INCLUDES
$Composant\_Machine$, $Composite\_Machine$, $Feuille\_Machine$,
$Pere\_Machine$

INVARIANT
$Feuille \cup Composite = Composant \wedge Feuille \cap Composite = \emptyset$

PROMOTES
$Ajout\_Enfants$, $Retirer\_Enfants$, $DonnerEnfants$

OPERATIONS
$cpt \longleftarrow Creer\_Composite =$
pre
$Composant \neq COMPOSANT$
then
any xx
where
$xx \in COMPOSANT - Composant$
then
$Ajout\_Composant(xx) \parallel$
$Ajout\_Composite(xx) \parallel$
$cpt := xx$
end
end;
$feuille \longleftarrow Creer\_Feuille =$
pre
$Composant \neq COMPOSANT$
then

```
    any xx
    where
        xx ∈ COMPOSANT−Composant
    then
        Ajout_Composant(xx) ||
        Ajout_Feuille(xx) ||
        feuille := xx
    end
  end;
  Supprimer_Composite(cpt) =
  pre
    cpt ∈ Composite
  then
    Retirer_Composant(cpt) ||
    Retirer_Composite(cpt) ||
    Retirer_Enfants(cpt)
  end;
  Supprimer_Feuille(feuille) =
  pre
    feuille ∈ Feuille
  then
    Retirer_Composant(feuille) ||
    Retirer_Feuille(feuille) ||
    Retirer_Enfants(feuille)
  end;
  Operation(cpt) =
  pre
    cpt ∈ Composant
  then
    select cpt ∈ Feuille then Operation_Feuille(cpt)
    when cpt ∈ Composite then Operation_Composite(cpt)
    else SKIP
    end
  end
```

Toutes ces machines B sont prouvées :

Machines abstraites	TC	POG	Obv	nPO	nUn	%Pr
Composant_Machine	OK	OK	4	2	0	100
Composite_Machine	OK	OK	6	2	0	100
Feuille_Machine	OK	OK	6	2	0	100
Interface_Machine	OK	OK	26	20	0	100
Pere_Machine	OK	OK	5	0	0	100
Total	OK	OK	47	26	0	100

D.2 Essai d'instanciation par inclusions

Les machines abstraites du pattern spécifiées en B dans le paragraphe précédent sont supposées données et prouvées. On crée pour chaque machine du pattern une nouvelle machine l'incluant avec la clause **INCLUDES**.

Composant_Machine est incluse dans Equipement_Machine :

MACHINE *Equipement_Machine*

INCLUDES
Composant_Machine, *equ.Composant_Machine*

DEFINITIONS
Equipement == *equ.Composant*;
EQUIPEMENT == *COMPOSANT*

OPERATIONS
Ajouter_Equipement(*equip*) =
pre
equip $\in$ *EQUIPEMENT* $-$ *Equipement*
then
equ.Ajout_Composant(*equip*)
end;
Retirer_Equipement(*equip*) =
pre
equip $\in$ *Equipement*
then
equ.Retirer_Composant(*equip*)
end

Feuille_Machine est incluse dans Lecteur_Machine :

MACHINE *Lecteur_Machine*

INCLUDES
Composant_Machine, *lct.Feuille_Machine*

DEFINITIONS
Lecteur == *lct.Feuille*;
EQUIPEMENT == *COMPOSANT*

OPERATIONS
Ajouter_Lecteur(*lecteur*) =
pre
lecteur $\in$ *EQUIPEMENT* $-$ *Lecteur*
then
lct.Ajout_Feuille(*lecteur*)
end;
Retirer_Lecteur(*lecteur*) =
pre
lecteur $\in$ *Lecteur*
then
lct.Retirer_Feuille(*lecteur*)

end;
Operation_Lecteur(*lecteur*) $=$
pre
  lecteur $\in$ *Lecteur*
then
  lct.Operation_Feuille(*lecteur*)
end

Feuille_Machine est aussi incluse dans Porte_Machine :

MACHINE *Porte_Machine*

INCLUDES
  Composant_Machine, *prt.Feuille_Machine*

DEFINITIONS
  Porte $==$ *prt.Feuille*;
  EQUIPEMENT $==$ *COMPOSANT*

OPERATIONS
Ajouter_Porte(*porte*) $=$
pre
  porte $\in$ *EQUIPEMENT* $-$ *Porte*
then
  prt.Ajout_Feuille(*porte*)
end;
Retirer_Porte(*porte*) $=$
pre
  porte $\in$ *Porte*
then
  prt.Retirer_Feuille(*porte*)
end;
Operation_Porte(*porte*) $=$
pre
  porte $\in$ *Porte*
then
  prt.Operation_Feuille(*porte*)
end

Composite_Machine est incluse dans Point_Machine :

MACHINE *Point_Machine*

INCLUDES
  Composant_Machine, *pdp.Composite_Machine*

DEFINITIONS
  Point $==$ *pdp.Composite*;
  EQUIPEMENT $==$ *COMPOSANT*

OPERATIONS
Ajouter_Point(*point*) $=$
pre
  $point \in EQUIPEMENT - Point$
then
  pdp.Ajout_Composite(*point*)
end;
Retirer_Point(*point*) $=$
pre
  $point \in Point$
then
  pdp.Retirer_Composite(*point*)
end;
Operation_Point(*point*) $=$
pre
  $point \in Point$
then
  pdp.Operation_Composite(*point*)
end

Pere_Machine est incluse dans Composition_Machine :

MACHINE *Composition_Machine*

INCLUDES *Pere_Machine*, *Composant_Machine*, *Composite_Machine*

DEFINITIONS
  Compose == *Pere*;
  Equipement == *Composant*;
  Point == *Composite*

OPERATIONS
Ajouter_Comp(*parent*,*enfant*) $=$
pre
  $parent \in Point \wedge enfant \in Equipement \wedge enfant \notin \mathrm{DOM}(Compose)$
then
  Ajout_Enfants(*parent*, *enfant*)
end;
Retirer_Comp(*pere*) $=$
pre
  $pere \in Equipement$
then
  Retirer_Enfants(*pere*)
end;
$enfants \longleftarrow DonnerComp(parent) =$
pre
  $parent \in Point \wedge parent \in \mathrm{RAN}(Compose)$
then

$enfants \longleftarrow DonnerEnfants(parent)$
end

Seule la nouvelle machine interface pose problème : on ne peut pas conclure. Le bilan de ce projet est :

Machines abstraites	TC	POG	Obv	nPO	nUn	%Pr
Composant_Machine	OK	OK	4	2	0	100
Composite_Machine	OK	OK	6	2	0	100
Composition_Machine	OK	OK	4	3	0	100
Equipement_Machine	OK	OK	6	0	0	100
Feuille_Machine	OK	OK	6	2	0	100
Interface	-	-				
Interface_Machine	OK	OK	26	20	0	100
Lecteur_Machine	OK	OK	8	0	0	100
Pere_Machine	OK	OK	5	0	0	100
Point_Machine	OK	OK	8	0	0	100
Porte_Machine	OK	OK	8	0	0	100

D.3 Machine équivalente de COMPOSITE

Il est possible d'obtenir une machine unique équivalente à un ensemble de machines B. Il suffit de remplacer dans la machine interface les **INCLUDES** par les spécifications contenues dans les machines incluses. Par exemple, le composant COMPOSITE décrit dans la section D.1 a pour machine unique équivalente :

Machine unique de COMPOSITE :

MACHINE *Composite_Unique*

SETS
COMPOSANT

VARIABLES
Composant, Machine Composant
Composite, Machine Composite
Feuille, Machine Feuille
Pere Machine Pere

INVARIANT
$Composant \subseteq COMPOSANT \wedge$ Machine Composant
$Composite \subseteq Composant \wedge$ Machine Composite
$Feuille \subseteq Composant \wedge$ Machine Feuille
$Pere \in Composant \nrightarrow Composite \wedge$ Machine Pere
$Feuille \cup Composite = Composant \wedge$ Interface 1
$Feuille \cap Composite = \emptyset$ Interface 2

DEFINITIONS

$Ajouter\_Enfants(parent, enfant)$ == $Pere$:= $Pere \cup \{enfant \mapsto parent\}$;
$Retirer\_Enfants(pere)$ ==
any $chemin$ **where** $chemin \in Composant \leftrightarrow Composite \wedge$
$\forall(xx,yy).\ ((xx \mapsto yy) \in chemin \Rightarrow (yy \in Pere^{-1}[\{xx\}] \vee$
$\exists zz.(zz \in Pere^{-1}[\{xx\}] \wedge (zz \mapsto yy) \in chemin)))$
then
$Pere := (\{pere\} \cup chemin[\{pere\}]) \lhd Pere \rhd (\{pere\} \cup chemin[\{pere\}])$
end;
$DonnerEnfants(parent)$ == $Pere^{-1}[\{parent\}]$;
$Ajouter\_Composant(cpt)$ == $Composant := Composant \cup \{cpt\}$;
$Retirer\_Composant(cpt)$ == $Composant := Composant - \{cpt\}$;
$Ajouter\_Feuille(feuille)$ == $Feuille := Feuille \cup \{feuille\}$;
$Retirer\_Feuille(feuille)$ == $Feuille := Feuille - \{feuille\}$;
$Operation\_Feuille(feuille)$ == SKIP;
$Ajouter\_Composite(cpt)$ == $Composite := Composite \cup \{cpt\}$;
$Retirer\_Composite(cpt)$ == $Composite := Composite - \{cpt\}$;
$Operation\_Composite(cpt)$ == SKIP

INITIALISATION
$Composant, Composite, Feuille, Pere := \emptyset,\emptyset,\emptyset,\emptyset$

OPERATIONS

$Ajouter\_Fils(parent,enfant)$ =
pre
$parent \in Composite \wedge enfant \in Composant \wedge enfant \notin \mathrm{DOM}(Pere)$
then
$Ajouter\_Enfants(parent, enfant)$
end;

$Retirer\_Fils(pere)$ =
pre
$pere \in Composant$
then
$Retirer\_Enfants(pere)$
end;

$enfants \longleftarrow DonnerFils(parent)$ =
pre
$parent \in Composite \wedge parent \in \mathrm{RAN}(Pere)$
then
$enfants := DonnerEnfants(parent)$
end;

Operations Interface

$cpt \longleftarrow Creer\_Composite$ =
pre
$Composant \neq COMPOSANT$
then

```
  any xx
  where
      xx ∈ COMPOSANT−Composant
  then
      Ajouter_Composant(xx) ||
      Ajouter_Composite(xx) ||
      cpt := xx
  end
end;

feuille ⟵ Creer_Feuille =
pre
  Composant ≠ COMPOSANT
then
  any xx
  where
      xx ∈ COMPOSANT−Composant
  then
      Ajouter_Composant(xx) ||
      Ajouter_Feuille(xx) ||
      feuille := xx
  end
end;

Supprimer_Composite(cpt) =
pre
  cpt ∈ Composite
then
  Retirer_Composant(cpt) ||
  Retirer_Composite(cpt) ||
  Retirer_Enfants(cpt)
end;

Supprimer_Feuille(feuille) =
pre
  feuille ∈ Feuille
then
  Retirer_Composant(feuille) ||
  Retirer_Feuille(feuille) ||
  Retirer_Enfants(feuille)
end;

Operation(cpt) =
pre
  cpt ∈ Composant
then
  select cpt ∈ Feuille then Operation_Feuille(cpt)
  when cpt ∈ Composite then Operation_Composite(cpt)
  else SKIP
  end
```

end

La machine unique a été prouvée et on obtient :

Machines abstraites	TC	POG	Obv	nPO	nUn	%Pr
Composite_Unique	OK	OK	39	21	0	100

D.4 Exemple d'instanciation

Afin d'illustrer la méthode d'instanciation proposée dans le chapitre 3, on décide de spécifier un système de fichiers à partir de COMPOSITE (voir annexe D.3).

Renommage :

MACHINE *Dossier_Renommage*

INCLUDES
Composite_Unique

DEFINITIONS
$ELEMENT\ ==\ COMPOSANT;$
$Element\ ==\ Composant;$
$Dossier\ ==\ Composite;$
$Fichier\ ==\ Feuille$

PROMOTES
Ajouter_Fils, *Retirer_Fils*, *DonnerFils*

OPERATIONS
$dossier \longleftarrow Creer\_Dossier\ =$
pre
$Element \neq ELEMENT$
then
$dossier \longleftarrow Creer\_Composite$
end;

$Ajouter\_Dossier(dossier)\ =$
pre
$dossier \in Dossier$
then
SKIP
end;

$fichier \longleftarrow Creer\_Fichier\ =$
pre
$Element \neq ELEMENT$
then
$fichier \longleftarrow Creer\_Feuille$

end;

$Ajouter\_Fichier(fichier)\ =$
pre
  $fichier\ \in\ Fichier$
then
  SKIP
end;

$Supprimer\_Dossier(dossier)\ =$
pre
  $dossier\ \in\ Dossier$
then
  $Supprimer\_Composite(dossier)$
end;

$Supprimer\_Fichier(fichier)\ =$
pre
  $fichier\ \in\ Fichier$
then
  $Supprimer\_Feuille(fichier)$
end;

$ModeEcriture(element)\ =$
pre
  $element\ \in\ Element$
then
  $Operation(element)$
end;

$Protection(element)\ =$
pre
  $element\ \in\ Element$
then
  $Operation(element)$
end

Raffinement :

REFINEMENT $Dossier\_Machine$

REFINES $Dossier\_Renommage$

INCLUDES $Composite\_Unique$

SETS
  $ETAT\ =\ \{ecriture,\ protege\}$

VARIABLES

$Etat$, $Utilisables$

INVARIANT

$Utilisables \subseteq Element \wedge$
$Etat \in Element \rightarrow\!\!\!\!\!+ \; ETAT$

INITIALISATION

$Etat$, $Utilisables$ $:=$ $\emptyset$,$\emptyset$

DEFINITIONS

$ELEMENT == COMPOSANT$;
$Element == Composant$;
$Dossier == Composite$;
$Fichier == Feuille$;
$Ecriture\_Fichier(fichier) == Etat(fichier) := ecriture$;
$Ecriture\_Dossier(dossier) == Etat(dossier) := ecriture$;
$Protection\_Fichier(fichier) == Etat(fichier) := protege$;
$Protection\_Dossier(dossier) == Etat(dossier) := protege$

PROMOTES

$Ajouter\_Fils$, $Retirer\_Fils$, $DonnerFils$

OPERATIONS

$Ajouter\_Dossier(dossier) =$
pre
$dossier \in Dossier$
then
$Etat(dossier) := protege \parallel$
$Utilisables := Utilisables \cup \{dossier\}$
end;

$Ajouter\_Fichier(fichier) =$
pre
$fichier \in Fichier$
then
$Etat(fichier) := protege \parallel$
$Utilisables := Utilisables \cup \{fichier\}$
end;

$Supprimer\_Dossier(dossier) =$
pre
$dossier \in Dossier$
then
$Supprimer\_Composite(dossier) \parallel$
$Utilisables := Utilisables - \{dossier\} \parallel$
$Etat := \{dossier\} \lhd\!\!\!- \; Etat$
end;

$Supprimer\_Fichier(fichier) =$

pre
  fichier $\in$ *Fichier*
then
  $Supprimer\_Feuille(fichier) \parallel$
  $Utilisables := Utilisables - \{fichier\} \parallel$
  $Etat := \{fichier\} \lhd Etat$
end;

$ModeEcriture(element) =$
pre
  $element \in Element$
then
  select $element \in Fichier$ **then** $Ecriture\_Fichier(element)$
  when $element \in Dossier$ **then** $Ecriture\_Dossier(element)$
  else SKIP
  end
end;

$Protection(element) =$
pre
  $element \in Element$
then
  select $element \in Fichier$ **then** $Protection\_Fichier(element)$
  when $element \in Dossier$ **then** $Protection\_Dossier(element)$
  else SKIP
  end
end

Les machines ont été vérifiées et prouvées :

Machines abstraites	TC	POG	Obv	nPO	nUn	%Pr
Dossier_Renommage	OK	OK	25	0	0	100
Dossier_Machine	OK	OK	110	24	0	100
Total	OK	OK	135	24	0	100

D.5 Spécification directe de l'exemple précédent

Afin d'analyser les conséquences sur les preuves de l'exemple d'application du paragraphe D.4, nous avons spécifié directement en B le même exemple, sans passer par des inclusions et des raffinements.

MACHINE *Dossier_Unique*

SETS
  $ELEMENT$;
  $ETAT = \{ecriture, protege\}$

VARIABLES
  Element,

$Dossier$,
$Fichier$,
$Pere$,
$Etat$,
$Utilisables$

INVARIANT

$Element \subseteq ELEMENT \wedge$
$Dossier \subseteq Element \wedge$
$Fichier \subseteq Element \wedge$
$Pere \in Element \nrightarrow Dossier \wedge$
$Fichier \cup Dossier = Element \wedge$
$Fichier \cap Dossier = \emptyset \wedge$
$Utilisables \subseteq Element \wedge$
$Etat \in Element \nrightarrow ETAT$

DEFINITIONS

$Retirer\_Enfants(pere) ==$
any $chemin$ **where** $chemin \in Element \leftrightarrow Dossier \wedge$
$\forall(xx,yy).\ ((xx \mapsto yy) \in chemin \Rightarrow (yy \in Pere^{-1}[\{xx\}] \vee$
$\exists zz.(zz \in Pere^{-1}[\{xx\}] \wedge (zz \mapsto yy) \in chemin\ )))$
then
$Pere := (\{pere\} \cup chemin[\{pere\}]) \lhd Pere \rhd (\{pere\} \cup chemin[\{pere\}])$
end;
$Ajout\_Element(cpt) == Element := Element \cup \{cpt\}$;
$Retirer\_Element(cpt) == Element := Element - \{cpt\}$;
$Ajout\_Fichier(fichier) == Fichier := Fichier \cup \{fichier\}$;
$Retirer\_Fichier(fichier) == Fichier := Fichier - \{fichier\}$;
$Ajout\_Dossier(cpt) == Dossier := Dossier \cup \{cpt\}$;
$Retirer\_Dossier(cpt) == Dossier := Dossier - \{cpt\}$;
$Ecriture\_Fichier(fichier) == Etat(fichier) := ecriture$;
$Ecriture\_Dossier(dossier) == Etat(dossier) := ecriture$;
$Protection\_Fichier(fichier) == Etat(fichier) := protege$;
$Protection\_Dossier(dossier) == Etat(dossier) := protege$

INITIALISATION

$Element, Dossier, Fichier, Pere, Etat, Utilisables := \emptyset,\emptyset,\emptyset,\emptyset,\emptyset,\emptyset$

OPERATIONS

$Ajouter\_Fils(parent,enfant) =$
pre
$parent \in Dossier \wedge enfant \in Element \wedge enfant \notin \mathrm{DOM}(Pere)$
then
$Pere := Pere \cup \{enfant \mapsto parent\}$
end;

$Retirer\_Fils(pere) =$
pre
$pere \in Element$

then
  any *chemin*
  where
  $chemin \in Element \leftrightarrow Dossier \wedge \forall(xx,yy).\ ((xx \mapsto yy) \in chemin \Rightarrow (yy \in Pere^{-1}[\{xx\}] \vee \exists zz.(zz \in Pere^{-1}[\{xx\}] \wedge (zz \mapsto yy) \in chemin)))$
  then
  $Pere := (\{pere\} \cup chemin[\{pere\}]) \lhd Pere \rhd (\{pere\} \cup chemin[\{pere\}])$
  end
end;

$enfants \longleftarrow DonnerFils(parent) =$
pre
  $parent \in Dossier \wedge parent \in \mathrm{RAN}(Pere)$
then
  $enfants := Pere^{-1}[\{parent\}]$
end;

$dossier \longleftarrow Creer\_Dossier =$
pre
  $Element \neq ELEMENT$
then
  any *xx*
  where
    $xx \in ELEMENT - Element$
  then
    $Ajout\_Element(xx) \parallel$
    $Ajout\_Dossier(xx) \parallel$
    $dossier := xx$
  end
end;

$Ajouter\_Dossier(dossier) =$
pre
  $dossier \in Dossier$
then
  $Etat(dossier) := protege \parallel$
  $Utilisables := Utilisables \cup \{dossier\}$
end;

$fichier \longleftarrow Creer\_Fichier =$
pre
  $Element \neq ELEMENT$
then
  any *xx*
  where
    $xx \in ELEMENT - Element$
  then
    $Ajout\_Element(xx) \parallel$
    $Ajout\_Fichier(xx) \parallel$
    $fichier := xx$

```
  end
end;

Ajouter_Fichier(fichier) =
pre
  fichier ∈ Fichier
then
  Etat(fichier) := protege ||
  Utilisables := Utilisables ∪ {fichier}
end;

Supprimer_Dossier(dossier) =
pre
  dossier ∈ Dossier
then
  Retirer_Element(dossier) ||
  Retirer_Dossier(dossier) ||
  Retirer_Enfants(dossier) ||
  Utilisables := Utilisables − {dossier} ||
  Etat := {dossier} ⩤ Etat
end;

Supprimer_Fichier(fichier) =
pre
  fichier ∈ Fichier
then
  Retirer_Element(fichier) ||
  Retirer_Fichier(fichier) ||
  Retirer_Enfants(fichier) ||
  Utilisables := Utilisables − {fichier} ||
  Etat := {fichier} ⩤ Etat
end;

ModeEcriture(element) =
pre
  element ∈ Element
then
  select element ∈ Fichier then Ecriture_Fichier(element)
  when element ∈ Dossier then Ecriture_Dossier(element)
  else SKIP
  end
end;

Protection(element) =
pre
  element ∈ Element
then
  select element ∈ Fichier then Protection_Fichier(element)
  when element ∈ Dossier then Protection_Dossier(element)
  else SKIP
```

end
end

Cette machine a été ensuite prouvée :

Machines abstraites	TC	POG	Obv	nPO	nUn	%Pr
Dossier_Unique	OK	OK	89	38	0	100