Un exemple de réutilisation de patterns de spécification avec la méthode B

Sandrine BLAZY, Frédéric GERVAIS et Régine LALEAU

1 Introduction

1.1 Problématique

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 représente justement un processus

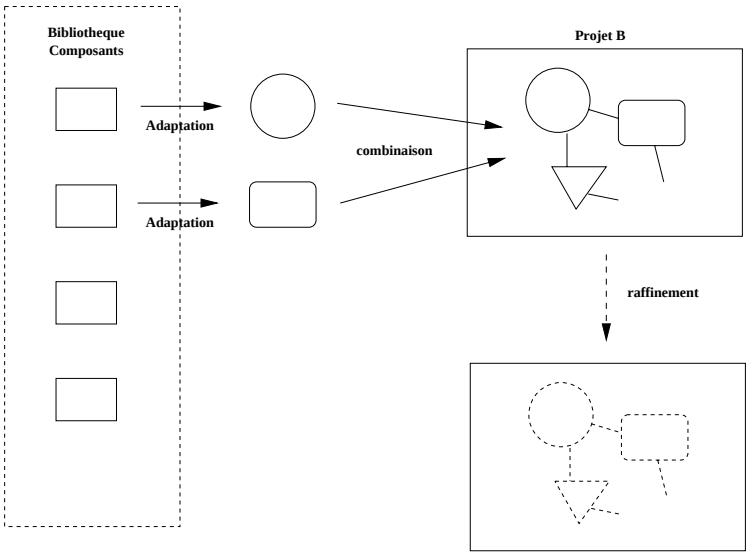

FIG. 1 – Conception B utilisant des composants

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 objectif du rapport

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 [5]. 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 (voir Chapitre 2 dans [6]). Le langage LePUS [3] est par exemple dédié à la spécification des patterns. L'utilisation d'un langage existant, RSL, a permis aussi de formaliser certains patterns [1]. Dans Catalysis [9], 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.

L'objectif de ce rapport est d'exhiber un exemple simple de réutilisation de composants de spécification avec la méthode B. Le rapport est organisé de la manière suivante. La section 2 décrit deux patterns extraits de la littérature et présente leur spécification en B. La section 3 est un exemple de réutilisation de ces patterns. Enfin, les conclusions et les perspectives sont présentées dans la section 4.

2 Patterns de spécification

2.1 Introduction sur les patterns

Les *patterns* sont des descriptions de problèmes génériques régulièrement rencontrés par les programmeurs expérimentés, associées à des propositions de solutions. Vincent Couturier [2] 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 constituent donc 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 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.

L'objectif est de sélectionner des exemples de patterns de la littérature et de les réutiliser avec la méthode B. L'idée est d'utiliser tous les mécanismes de B afin de pouvoir bénéficier des avantages de la méthode : prise en charge de la vérification syntaxique, validation des spécifications, ... Réutiliser des patterns en B signifie qu'il faut d'une part spécifier en B les patterns utilisés et, d'autre part, déterminer puis définir les mécanismes de réutilisation avec les mécanismes de la méthode B. On appelle ces composants de spécification B réutilisables des *patterns de spécification*.

Deux exemples de patterns sont traités dans cette section : le pattern **Composite** [5] et le pattern **Resource Allocation** [4].

2.2 Pattern Composite

Le pattern **Composite** [5] 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. Le diagramme de classe du pattern **Composite** est donné par la figure 2.

Les constituants de **Composite** sont :

- `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`.

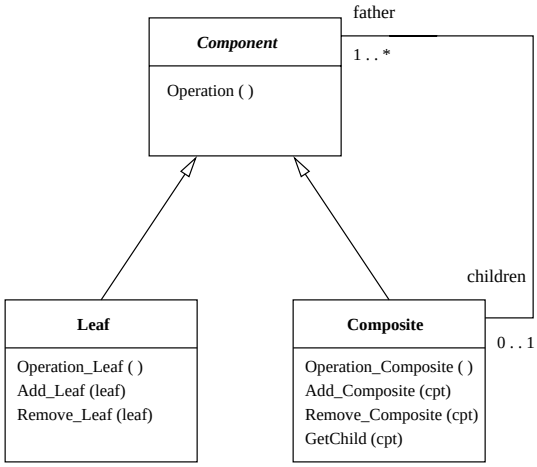

FIG. 2 – Diagramme de classe du pattern Composite

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

Pour des raisons techniques, le pattern est spécifié en B par une machine abstraite unique : cela permet notamment de raffiner un pattern de spécification. Il existe des règles [7] pour traduire automatiquement un diagramme de classe UML en une spécification B. Dans ce cas, chaque classe est spécifiée par une machine et une machine interface permet de regrouper les différentes spécifications. Le projet obtenu est alors équivalent à une machine abstraite unique. Le pattern **Composite** est spécifié en B par la machine suivante :

MACHINE $Composite\_Pattern(COMPONENT)$

VARIABLES
$Component,\ Composite,\ Leaf,\ Father$

INVARIANT
$Component \subseteq COMPONENT \wedge$
$Composite \subseteq Component \wedge$
$Leaf \subseteq Component \wedge$
$Father \in Component \nrightarrow Composite \wedge$
$Leaf \cup Composite = Component \wedge$
$Leaf \cap Composite = \emptyset$

DEFINITIONS
$Def\_Add\_Children(father,\ child) ==$
$\quad Father := Father \cup (child \times \{father\});$
$Def\_Remove\_Children\_Leaf(father) ==$
$\quad Father := \{father\} \lhd Father;$
$Def\_Remove\_Children\_Composite(father) ==$
$\quad Father := \{father\} \lhd Father \rhd \{father\};$
$Def\_Get\_Children(father) == Father^{-1}[\{father\}];$

$Def\_Add\_Component(cpt) == Component := Component \cup \{cpt\};$
$Def\_Remove\_Component(cpt) ==$
$\quad Component := Component - \{cpt\};$
$Def\_Add\_Leaf(leaf) == Leaf := Leaf \cup \{leaf\};$
$Def\_Remove\_Leaf(leaf) == Leaf := Leaf - \{leaf\};$
$Def\_Remove\_and\_Add\_Leaf(leaf1, leaf2) ==$
$\quad Leaf := (Leaf \cup \{leaf2\}) - \{leaf1\};$
$Def\_Operation\_Leaf(leaf) ==$ SKIP;
$Def\_Add\_Composite(cpt) == Composite := Composite \cup \{cpt\};$
$Def\_Remove\_Composite(cpt) == Composite := Composite - cpt;$
$Def\_Operation\_Composite(cpt) ==$ SKIP

INITIALISATION
$Component, Composite, Leaf, Father := \emptyset, \emptyset, \emptyset, \emptyset$

OPERATIONS

$children \longleftarrow GetChild(father) =$
pre
$father \in Composite \wedge father \in$ RAN$(Father)$
then
$children := Def\_Get\_Children(father)$
end;

$cpt \longleftarrow New\_Composite(comp) =$
pre
$Component \neq COMPONENT \wedge$
$comp \subseteq Component \wedge$
$comp \cap$ DOM$(Father) = \emptyset \wedge$
$comp \neq \emptyset$
then
any xx **where** $xx \in COMPONENT - Component$
then
$Def\_Add\_Component(xx) \parallel$
$Def\_Add\_Composite(xx) \parallel$
$Def\_Add\_Children(xx, comp) \parallel$
$cpt := xx$
end
end;

$Add\_Composite(cpt, comp) =$
pre
$Component \neq COMPONENT \wedge$
$comp \subseteq Component \wedge$
$comp \cap$ DOM$(Father) = \emptyset \wedge$
$comp \neq \emptyset \wedge$
$cpt \in COMPONENT - Component$
then
$Def\_Add\_Component(cpt) \parallel$
$Def\_Add\_Composite(cpt) \parallel$

Def_Add_Children(*cpt*, *comp*)
end;

leaf ⟵ *New_Leaf* =
pre
Component ≠ *COMPONENT*
then
any *xx* **where** *xx* ∈ *COMPONENT* − *Component*
then
Def_Add_Component(*xx*) ||
Def_Add_Leaf(*xx*) ||
leaf := *xx*
end
end;

Add_Leaf(*leaf*) =
pre
Component ≠ *COMPONENT* ∧
leaf ∈ *COMPONENT* − *Component*
then
Def_Add_Component(*leaf*) ||
Def_Add_Leaf(*leaf*)
end;

Remove_Composite(*cpt*) =
pre
cpt ∈ *Composite* ∧
Father(*cpt*) ≠ *cpt* ∧
cpt ∈ DOM(*Father*)
then
if $Father^{-1}[\{Father(cpt)\}] = \{cpt\}$
then
Def_Remove_Component(*cpt*) ||
Def_Remove_Composite({*cpt*, *Father*(*cpt*)}) ||
Def_Remove_Children_Composite(*cpt*) ||
Def_Add_Leaf(*Father*(*cpt*))
else
Def_Remove_Component(*cpt*) ||
Def_Remove_Composite({*cpt*}) ||
Def_Remove_Children_Composite(*cpt*)
end
end;

Remove_Leaf(*leaf*) =
pre
leaf ∈ *Leaf* ∧
leaf ∈ DOM(*Father*)
then
if $Father^{-1}[\{Father(leaf)\}] = \{leaf\}$
then

Def_Remove_Component(*leaf*) ||
Def_Remove_Composite({*Father*(*leaf*)}) ||
Def_Remove_Children_Leaf(*leaf*) ||
Def_Remove_and_Add_Leaf(*leaf*, *Father*(*leaf*))
else
Def_Remove_Component(*leaf*) ||
Def_Remove_Leaf(*leaf*) ||
Def_Remove_Children_Leaf(*leaf*)
end
end;

Operation(*cpt*) =
pre
cpt ∈ *Component*
then
select *cpt* ∈ *Leaf* **then** *Def_Operation_Leaf*(*cpt*)
when *cpt* ∈ *Composite* **then** *Def_Operation_Composite*(*cpt*)
else SKIP
end
end

2.3 Pattern Resource Allocation

Le pattern **Resource Allocation** [4] permet d'allouer des ressources. La figure 3 représente le diagramme de classe du pattern.

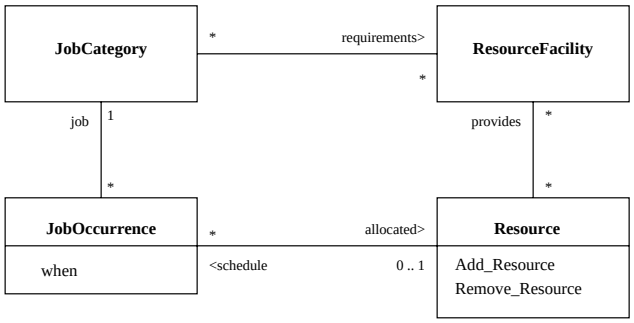

FIG. 3 – Diagramme de classe du pattern Resource Allocation

Quatre classes sont définies. La classe `Resource` représente une ressource à allouer. Une ressource utilise des moyens de ressource, représentés ici par la classe `ResourceFacility`. Une ressource est allouée à des occurrences de tâche représentées par `JobOccurrence`. Enfin, la classe `JobCategory`, qui représente les catégories de tâches, est liée aux deux dernières classes. Les besoins en moyens de ressource ne sont supportés que par certaines catégories de tâche. Une tâche est représentée par une association entre `JobOccurrence` et `JobCategory`.

Comme dans la section précédente, il est possible de spécifier en B le pattern de spécification correspondant :

MACHINE
$Resource\_Allocation(JOBS,\ CATEGORY,\ FACILITY,\ RESOURCE)$

SETS
$DATE$

VARIABLES
$JobOccurrence,\ When,\ JobCategory,\ Resource,\ ResourceFacility,$
$Job,\ Requirements,\ Provides,\ Allocated$

INVARIANT
$JobOccurrence \subseteq JOBS \wedge$
$When \in JobOccurrence \longrightarrow DATE \wedge$
$JobCategory \subseteq CATEGORY \wedge$
$Resource \subseteq RESOURCE \wedge$
$ResourceFacility \subseteq FACILITY \wedge$
$Job \in JobOccurrence \longrightarrow JobCategory \wedge$
$Requirements \in JobCategory \leftrightarrow ResourceFacility \wedge$
$Provides \in Resource \leftrightarrow ResourceFacility \wedge$
$Allocated \in JobOccurrence \nrightarrow Resource$

DEFINITIONS
$Scheduled == Allocated^{-1}$

INITIALISATION
$JobOccurrence,\ When,\ JobCategory,\ Resource,\ ResourceFacility,\ Job,$
$Requirements,\ Provides,\ Allocated := \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset$

OPERATIONS

$allocation\_Resource\_JobOccurrence(res, job) =$
pre
$res \in Resource \wedge$
$job \in JobOccurrence \wedge$
$Provides[\{res\}] \subseteq Requirements[Job[\{job\}]]$
then
$Allocated(job) := res$
end;

$Add\_Resource(res) =$
pre
$res \in RESOURCE - Resource$
then
$Resource := Resource \cup \{res\}$
end;

$Remove\_Resource(res) =$
pre
$res \in Resource$

then
  $Resource := Resource - \{res\} \parallel$
  $Provides := \{res\} \lhd Provides \parallel$
  $Allocated := Allocated \rhd \{res\}$
end;

$res \longleftarrow New\_Resource =$
pre
  $Resource \neq RESOURCE$
then
  any xx **where** $xx \in RESOURCE - Resource$
  then
    $Resource := Resource \cup \{xx\} \parallel$
    $res := xx$
  end
end

3 Réutilisation avec la méthode B

L'objectif de cette section est de donner un exemple de réutilisation de patterns de spécification en utilisant la méthode B.

3.1 Exemple de réutilisation

L'exemple considéré est le suivant : on souhaite allouer des dossiers à des secrétaires. La figure 4 est la solution suggérée. Elle réutilise les patterns décrits dans le paragraphe 2.

Comme les dossiers sont composés de dossiers et de fichiers, un dossier est considéré comme un objet composite. De plus, on souhaite réaliser une allocation de ressource. L'idée est donc d'adapter et de combiner les patterns **Composite** et **Resource Allocation**. Il existe trois types de mécanismes de réutilisation [8] : l'instanciation, la composition et l'extension. L'instanciation permet de renommer les éléments d'un pattern. La composition permet de combiner plusieurs patterns. L'extension permet enfin de rajouter, modifier ou supprimer des éléments et/ou des opérations d'un pattern.

Le but des sections suivantes est de spécifier en B ces différents mécanismes afin d'obtenir à partir des patterns de spécification décrits dans les paragraphes 2.2 et 2.3, une spécification B de l'exemple ci-dessus.

3.2 Première étape : instanciation et composition

L'instanciation et la composition sont réalisées en même temps. Les patterns de spécification `Composite_Pattern` et `Resource_Allocation` sont tous deux inclus dans une nouvelle machine : `Composition_Renaming`. Cette nouvelle machine permet à la fois d'instancier les machines des patterns mais aussi de composer leurs opérations.

Concernant l'instanciation, les given sets sont renommés par affectation des paramètres des patterns. Les variables des machines incluses sont renommées

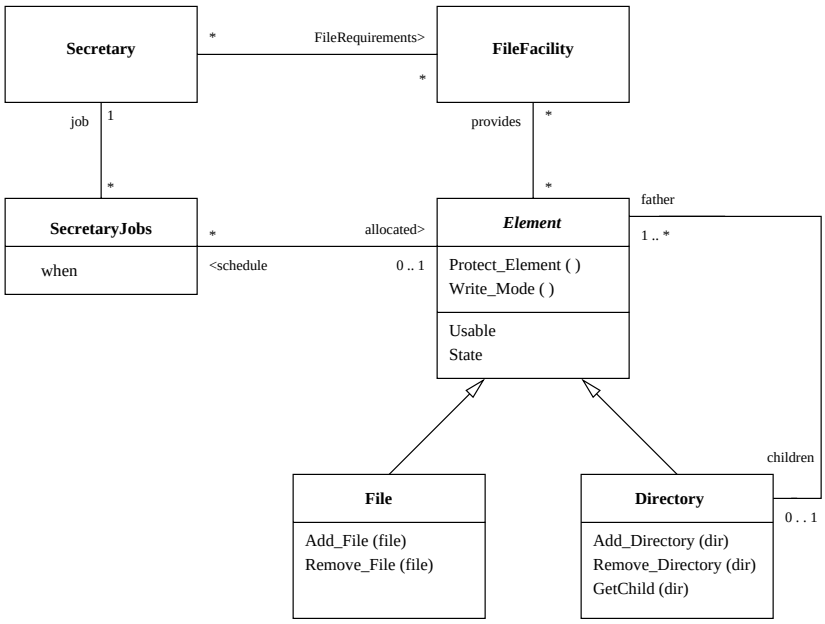

FIG. 4 – Un exemple de réutilisation

dans la clause **DEFINITION**. Par exemple, la variable `Composite` est renommée par `Directory`. Le renommage des opérations sera discuté en même temps que le mécanisme de composition.

La composition se traduit dans l'invariant par l'unification de deux variables : `Resource` et `Component`. Ce nouvel invariant ne peut être respecté que si les opérations des patterns de spécification initiaux ont été correctement composées entre elles. En effet, si une opération qui a pour effet de modifier une des variables unifiées n'est pas complétée par une opération ayant le même effet sur l'autre variable, alors l'invariant n'est pas préservé par cette opération. Par exemple, l'opération `Remove_Composite` doit être composée avec l'opération `Remove_Resource` afin de préserver l'invariant. La composition de patterns se traduit donc en B par une inclusion des machines concernées et par la composition des opérations réutilisées. Cette composition n'est possible que si tous les patterns de spécification comportent un minimum d'opérations élémentaires. Cette composition des opérations s'accompagne d'un renommage des préconditions et d'un nommage du résultat de la composition d'opérations.

Certaines opérations supplémentaires sont définies dans cette machine, comme `New_File`. Ces définitions permettent d'étendre la spécification obtenue. Les raisons de ce choix sont expliquées dans le paragraphe suivant.

MACHINE *Composition_Renaming*

SETS
ELEMENT; *JOBS*; *DATE*; *CATEGORY*; *FACILITY*

INCLUDES

Resource_Allocation(*JOBS*, *CATEGORY*, *FACILITY*, *ELEMENT*),
Composite_Pattern(*ELEMENT*)

DEFINITIONS
Directory == *Composite*;
File == *Leaf*;
SecretaryJobs == *JobOccurrence*;
Secretary == *JobCategory*;
FileFacility == *ResourceFacility*;
FileRequirements == *Requirements*;
Element == *Resource*

INVARIANT
Resource = *Component*

PROMOTES
GetChild

OPERATIONS

dir ⟵ *Add_Directory*(*files*) =
pre
Component ≠ *ELEMENT* ∧
files ⊆ *Component* ∧
files ∩ DOM(*Father*) = ∅ ∧
files ≠ ∅
then
any *xx* **where** *xx* ∈ *ELEMENT* − *Component*
then
Add_Composite(*xx*, *files*) ||
Add_Resource(*xx*) ||
dir := *xx*
end
end;

New_Directory(*dir*) =
pre
dir ∈ *Directory*
then
SKIP
end;

file ⟵ *Add_File* =
pre
Component ≠ *ELEMENT*
then
any *xx* **where** *xx* ∈ *ELEMENT* − *Component*
then
Add_Leaf(*xx*) ||
Add_Resource(*xx*) ||

$file := xx$
end
end;

$New\_file(file) =$
pre
$file \in File$
then
SKIP
end;

$Remove\_Directory(dir) =$
pre
$dir \in Directory \wedge$
$Father(dir) \neq dir \wedge$
$dir \in \text{DOM}(Father)$
then
$Remove\_Composite(dir) \parallel$
$Remove\_Resource(dir)$
end;

$Remove\_File(file) =$
pre
$file \in File \wedge$
$file \in \text{DOM}(Father)$
then
$Remove\_Leaf(file) \parallel$
$Remove\_Resource(file)$
end;

$Protect\_Element(element) =$
pre
$element \in Component$
then
$Operation(element)$
end;

$Write\_Mode(element) =$
pre
$element \in Component$
then
$Operation(element)$
end;

$allocation\_File\_Secretary(file, job) =$
pre
$file \in Resource \wedge$
$job \in SecretaryJobs \wedge$
$Provides[\{file\}] \subseteq FileRequirements[Job[\{job\}]]$
then

allocation_Resource_JobOccurrence(*file*, *job*)
end

3.3 Deuxième étape : extension

Une fois que les deux patterns ont été composés, il est possible d'étendre la spécification du résultat obtenu. Pour rajouter de nouvelles spécifications, l'utilisateur a besoin de nouveaux ensembles et de nouvelles variables concrètes. Les opérations spécifiées dans `Composition_Renaming` sont complétées par des substitutions utilisant les nouveaux ensembles et variables introduits dans le raffinement `Composition_Machine`. Par exemple, les variables `State` et `Usable` permettent de représenter l'état d'un fichier et d'indiquer si il est en mode écriture ou lecture. Dans ce cas, l'opération `Remove_File` est complétée par de nouvelles substitutions concernant ces deux variables concrètes.

Pour rajouter de nouvelles opérations, leur définition est spécifiée dans la machine `Composition_Renaming` avec le corps "skip". Cela permet d'une part de définir une nouvelle opération, ce qui n'est pas possible dans un raffinement, et de simplifier les preuves dans la machine de composition d'autre part. Ensuite, le corps de l'opération est raffiné par de nouvelles substitutions ou par des appels d'opérations dans le raffinement. Comme toute substitution raffine "skip", le raffinement est possible.

Enfin, l'opération générique `Operation` du pattern de spécification `Composite_Pattern` a été réutilisée deux fois dans la machine `Composition_Renaming`. Cette opération générique permet d'indiquer à l'utilisateur comment spécifier une opération qui agit à la fois sur des objets composites et simples. Cette opération est étendue afin de spécifier les opérations `Protect_Element` et `Write_Mode`. Le raffinement permet de spécifier effectivement le contenu de ces opérations. Un tel mécanisme aurait été impossible dans le cadre d'une inclusion de machine.

REFINEMENT *Composition_Machine*

REFINES *Composition_Renaming*

INCLUDES
Resource_Allocation(*JOBS*, *CATEGORY*, *FACILITY*, *ELEMENT*),
Composite_Pattern(*ELEMENT*)

SETS
$STATE = \{write, protected\}$

VARIABLES
State,
Usable

INVARIANT
$Usable \subseteq Element \wedge$
$State \in Usable \nrightarrow STATE$

DEFINITIONS
$Directory == Composite$;
$File == Leaf$;
$SecretaryJobs == JobOccurrence$;
$Secretary == JobCategory$;
$FileFacility == ResourceFacility$;
$FileRequirements == Requirements$;
$Element == Resource$;
$Write(element) == State(element) := write$;
$Protect(element) == State(element) := protected$

INITIALISATION
$Usable,\ State := \emptyset, \emptyset$

PROMOTES
$GetChild$

OPERATIONS

$New\_Directory(dir) =$
pre
$dir \in Directory$
then
$State(dir) := protected \parallel$
$Usable := Usable \cup \{dir\}$
end;

$New\_file(file) =$
pre
$file \in File$
then
$State(file) := protected \parallel$
$Usable := Usable \cup \{file\}$
end;

$Remove\_Directory(dir) =$
pre
$dir \in Directory \wedge$
$Father(dir) \neq dir \wedge$
$dir \in \text{DOM}(Father)$
then
$Remove\_Composite(dir) \parallel$
$Remove\_Resource(dir) \parallel$
$Usable := Usable - \{dir\} \parallel$
$State := \{dir\} \lhd State$
end;

$Remove\_File(file) =$
pre
$file \in File \wedge$

```
  file ∈ DOM(Father)
then
  Remove_Leaf(file) ||
  Remove_Resource(file) ||
  Usable := Usable − {file} ||
  State := {file} ⩤ State
end;

Protect_Element(element) =
pre
  element ∈ Element
then
  if element ∈ Usable
  then
      Protect(element)
  else
      SKIP
  end
end;

Write_Mode(element) =
pre
  element ∈ Element
then
  if element ∈ Usable
  then
      Write(element)
  else
      SKIP
  end
end
```

3.4 Analyse des preuves

Ces machines ont été validées par l'Atelier B (voir `www.atelierb-societe.com`). Concernant `Composition_Renaming`, toutes les obligations de preuves (PO) sont soit évidentes, soit automatiquement prouvées. Comme aucun invariant, à part celui lié à la composition, n'est spécifié dans cette machine, les PO liées au renommage sont évidentes car elles réutilisent les preuves des patterns de spécification réutilisés. Si la composition des opérations est correctement réalisée, elle permet de préserver l'invariant de la composition et par conséquent, toutes les PO générées par ce nouvel invariant sont automatiquement prouvées par le prouveur.

Concernant l'extension, les PO générées se partagent en deux groupes : les PO concernant les nouvelles substitutions et les PO liées au raffinement des opérations. Les autres obligations sont évidentes. Six PO concernent les nouvelles substitutions et ont toutes été prouvées interactivement. Les quatre autres PO sont liées au raffinement et ne dépendent pas de l'extension des spécifications. Ces PO prouvées interactivement sont donc "réutilisables" : si la machine

`Composition_Renaming` est de nouveau étendue, les quatre PO seront à nouveau générées et pourront être prouvées de la même manière.

Par conséquent, les preuves liées au renommage et à la composition sont automatiques et seule l'extension génère des preuves non évidentes. Cette propriété est intéressante car elle permet de séparer les preuves difficiles des preuves faciles. De plus, elle introduit une notion de "réutilisation de preuves" : réutilisation au moment de l'instanciation et de la composition, car le travail de validation est pris en charge par l'Atelier B et réutilisation au moment de l'extension, car la stratégie de preuve de certaines PO est réutilisable.

Machines	Obv.	PO.	Aut.	Int.	Un.
Composite_Pattern	32	59	47	12	0
Resource_Allocation	42	16	15	1	0
Composition_Renaming	43	10	10	0	0
Composition_Machine	284	33	23	10	0

3.5 Spécification directe

À titre de comparaison, l'exemple de la figure 4 a été spécifié directement en B par la machine suivante :

MACHINE *Exemple_Direct*

SET
$JOBS$; $DATE$; $CATEGORY$; $FACILITY$; $ELEMENT$;
$STATE = \{write, protected\}$

VARIABLES
State, *Usable*, *SecretaryJobs*, *When*, *Secretary*, *Element*, *FileFacility*, *Job*, *FileRequirements*, *Provides*, *Allocated*, *Directory*, *File*, *Father*

INVARIANT
$SecretaryJobs \subseteq JOBS \wedge$
$When \in SecretaryJobs \longrightarrow DATE \wedge$
$Secretary \subseteq CATEGORY \wedge$
$Element \subseteq ELEMENT \wedge$
$FileFacility \subseteq FACILITY \wedge$
$Job \in SecretaryJobs \longrightarrow Secretary \wedge$
$FileRequirements \in Secretary \leftrightarrow FileFacility \wedge$
$Provides \in Element \leftrightarrow FileFacility \wedge$
$Allocated \in SecretaryJobs \nrightarrow Element \wedge$
$Directory \subseteq Element \wedge$
$File \subseteq Element \wedge$
$Father \in Element \twoheadrightarrow Directory \wedge$
$File \cup Directory = Element \wedge$
$File \cap Directory = \emptyset \wedge$
$Usable \subseteq Element \wedge$
$State \in Usable \nrightarrow STATE$

INITIALISATION

$State$, $Usable$, $SecretaryJobs$, $When$, $Secretary$, $Element$, $FileFacility$, Job, $FileRequirements$, $Provides$, $Allocated$, $Directory$, $File$, $Father := \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset$

OPERATIONS

$children \longleftarrow GetChild(parent) =$
pre
$parent \in Directory \wedge parent \in \text{RAN}(Father)$
then
$children := Father^{-1}[\{parent\}]$
end;

$dir \longleftarrow Add\_Directory(files) =$
pre
$Element \neq ELEMENT \wedge$
$files \subseteq Element \wedge$
$files \cap \text{DOM}(Father) = \emptyset \wedge$
$files \neq \emptyset$
then
any xx **where** $xx \in ELEMENT - Element$
then
$Directory := Directory \cup \{xx\} \parallel$
$Element := Element \cup \{xx\} \parallel$
$Father := Father \cup (files \times \{xx\}) \parallel$
$dir := xx$
end
end;

$file \longleftarrow Add\_File =$
pre
$Element \neq ELEMENT$
then
any xx **where** $xx \in ELEMENT - Element$
then
$Element := Element \cup \{xx\} \parallel$
$File := File \cup \{xx\} \parallel$
$file := xx$
end
end;

$New\_Directory(dir) =$
pre
$dir \in Directory$
then
$State(dir) := protected \parallel$
$Usable := Usable \cup \{dir\}$
end;

$New\_file(file) =$

pre
  $file \in File$
then
  $State(file) := protected \parallel$
  $Usable := Usable \cup \{file\}$
end;

$Remove\_Directory(dir) =$
pre
  $dir \in Directory \wedge$
  $Father(dir) \neq dir \wedge$
  $dir \in \text{DOM}(Father)$
then
  if $Father^{-1}[\{Father(dir)\}] = \{dir\}$
  then
      $Element := Element - \{dir\} \parallel$
      $Directory := Directory - \{dir,\ Father(dir)\} \parallel$
      $Father := \{dir\} \lhd Father \rhd \{dir\} \parallel$
      $File := File \cup \{Father(dir)\}$
  else
      $Element := Element - \{dir\} \parallel$
      $Directory := Directory - \{dir\} \parallel$
      $Father := \{dir\} \lhd Father \rhd \{dir\}$
  end $\parallel$
  $Provides := \{dir\} \lhd Provides \parallel$
  $Allocated := Allocated \rhd \{dir\} \parallel$
  $Usable := Usable - \{dir\} \parallel$
  $State := \{dir\} \lhd State$
end;

$Remove\_File(file) =$
pre
  $file \in File \wedge$
  $file \in \text{DOM}(Father)$
then
  if $Father^{-1}[\{Father(file)\}] = \{file\}$
  then
      $Element := Element - \{file\} \parallel$
      $Directory := Directory - \{Father(file)\} \parallel$
      $Father := \{file\} \lhd Father \parallel$
      $File := (File \cup \{Father(file)\}) - \{file\}$
  else
      $Element := Element - \{file\} \parallel$
      $File := File - \{file\} \parallel$
      $Father := \{file\} \lhd Father$
  end $\parallel$
  $Provides := \{file\} \lhd Provides \parallel$
  $Allocated := Allocated \rhd \{file\} \parallel$
  $Usable := Usable - \{file\} \parallel$
  $State := \{file\} \lhd State$

```
end;

Protection(element) =
pre
  element ∈ Element
then
  if element ∈ Usable
  then
      State(element) := protected
  else
      SKIP
  end
end;

WriteMode(element) =
pre
  element ∈ Element
then
  if element ∈ Usable
  then
      State(element) := write
  else
      SKIP
  end
end

end
```

Les obligations de preuve générées par la machine `Exemple_Direct` sont résumées dans le tableau suivant :

Machines	Obv.	PO.	Aut.	Int.	Un.
Exemple_Direct	167	87	67	20	0

Dans cet exemple, il reste vingt obligations de preuve à démontrer après utilisation du prouveur automatique de l'Atelier B. Si on suppose que les patterns **Composite** et **Resource Allocation** sont préalablement spécifiés et prouvés (ils peuvent par exemple être stockés dans une bibliothèque de patterns de spécification), notre approche permet de spécifier le même exemple à partir de ces patterns en prouvant seulement dix PO de manière interactive. De plus, comme quatre de ces PO sont "réutilisables" (voir section 3.4), seules six PO difficiles restent à prouver.

Même si l'exemple présenté est assez simple, il permet de montrer que la méthode proposée ouvre de nouvelles perspectives, en réduisant le travail de preuve. Il montre également le besoin d'un outil pour assister le programmeur.

4 Conclusion

Nous avons présenté un exemple simple de réutilisation de patterns de spécification avec la méthode B. Cette étude nous a amené à faire des choix concernant

la spécification de la réutilisation.

On définit un pattern de spécification en B comme une machine abstraite unique qui décrit une manière de résoudre un problème de conception. Cette limite provient des restrictions de B concernant les mécanismes d'inclusion et de raffinement. D'un point de vue pratique, l'utilisation d'une machine unique est la principale limite de notre approche.

La réutilisation des patterns de spécification a été spécifiée en deux étapes consécutives. L'inclusion des machines de patterns permet dans un premier temps d'instancier et de composer les patterns de spécifications. Le mécanisme de raffinement B permet d'étendre la spécification obtenue. Ces deux étapes sont validées grâce aux outils de la méthode.

Le principal travail consiste maintenant à généraliser cet exemple, afin de spécifier formellement et d'automatiser la réutilisation de patterns de spécification en B. Dans cette perspective, l'utilisation des mécanismes de la méthode B pour spécifier ces mécanismes de réutilisation nous permettra de valider toutes les étapes du processus. En revanche, les limites concernant la mise en pratique de cette approche ne pourront être comblées que par la création d'un outil d'assistance.

Références

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

[2] 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.

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

[4] Martine Fowler. *Analysis Patterns : Reusable Object Models*. Addison-Wesley, 1997.

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

[6] Frédéric Gervais. Réutilisation de composants de spécification en B. Technical Report 394, CEDRIC, 2002.

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

[8] Ruben Prieto-Diaz and Peter Freeman. Classifying software for reusability. *IEEE Software*, 4(1), 1987.

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