Transformation des spécifications B en des diagrammes UML

Houda Fekih, Leila Jemni, Stephan Merz

Transformation des spécifications B en des diagrammes UML

Houda Fekih[1], Leila Jemni[1] et Stephan Merz[2]

[1]Département des Sciences de l'Informatique,
Faculté des Sciences, Campus Universitaire,
1060, Le belvédère, Tunis, Tunisie

[2]INRIA Lorraine & LORIA
615, Rue du Jardin Botanique, Nancy, France

houda.fekih@fst.rnu.tn
leila.jemni@fsegt.rnu.tn
Stephan.Merz@loria.fr

Résumé

Les méthodes formelles et semi-formelles de conception de systèmes informatiques mettent en avant différents concepts principaux de développement. Plusieurs travaux se sont déjà intéressés à la formalisation des méthodes semi-formelles par des méthodes formelles telle que la formalisation de UML par B. L'objectif de ces travaux est surtout de pouvoir vérifier formellement les propriétés attendues du système. Dans cet article, nous étudions le schéma réciproque. Nous voulons représenter des spécifications B par des modèles UML afin d'obtenir une meilleure documentation et lisibilité en bénéficiant des concepts de structuration présents en UML. Nous présentons des éléments d'une transformation interactive guidant le concepteur à générer une représentation naturelle de son modèle. L'objectif global de ce travail est de pouvoir établir une démarche où les représentations en UML et en B constituent deux vues complémentaires d'un même système en offrant à l'utilisateur la possibilité de naviguer entre ces deux vues et de pouvoir choisir le langage le plus adapté.

Mots clés : UML, diagramme de classes, diagramme d'état-transitions, méthode B, raffinement.

1 Introduction

UML (Unified Modeling Language) est considéré comme le langage standard de conception orientée objet : la version 2.0 vient d'être adoptée par l'OMG (Object Management Group) [2]. Il est maintenant largement utilisé dans les communautés scientifique et industrielle. Il permet de visualiser, concevoir et documenter les composants statiques et dynamiques d'un système ; il est supporté par plusieurs outils. Cependant, UML n'est qu'un langage semi-formel. Malgré qu'il possède une syntaxe bien définie basée sur le méta-modèle de UML et complétée par des contraintes structurelles exprimées en OCL, les sémantiques de ses notations restent ambigües car elles sont décrites en langage naturel. Les propriétés attendues du système sont généralement décrites de manière informelle et elles ne sont pas vérifiées. Pour cela, UML nécessite des extensions permettant d'introduire des notions de spécifications formelles et de preuves dans le processus de modélisation. Pour remédier à cette insuffisance, plusieurs travaux se sont intéressés à la formalisation de UML en dérivant des spécifications B à partir des diagrammes UML.

Le travail réalisé par Meyer [5] permet de traduire les concepts structurels d'UML (classe, attribut, association, etc.). Ce travail a été complété par celui de Ledang [4] qui s'est intéressé à la traduction des concepts comportementaux. L'objectif de ces dérivations est d'analyser les erreurs établies dans les modèles UML au cours de la spécification des besoins, de la conception et de l'implémentation. Elle bénéficie des outils supportant la méthode B en préservant les acquis de la méthode UML.

Cependant, il s'est avéré que la traduction des modèles UML en des spécifications B engendre des machines B compliquées et difficiles à comprendre ; d'autre part, il est difficile de projeter les erreurs détectées dans la machine B dans les modèles UML correspondants. Ce qui fait que ce qu'on gagne au niveau de la formalisation d'UML on le perd au niveau de la simplicité de la spécification en B.

Dans ce contexte, nous pensons que les méthodes semi-formelles peuvent apporter des solutions à des problèmes qui n'ont pas été résolus par les méthodes formelles. En fait, une des faiblesses de la méthode B est le manque de moyens de structuration qui rend difficile la compréhension des modèles B bien qu'il simplifie la définition de la sémantique et la génération des obligations de preuve. Par contre, UML est bien connu par ses qualités de structuration grâce aux notions d'encapsulation, de composition, d'héritage ...

Il nous paraît donc utile de prévoir des méthodes de conception où les méthodes formelles et semi-formelles coexistent en offrant deux vues complémentaires d'un modèle commun sous-jacent et en permettant au concepteur de travailler sur la vue la plus adaptée. L'objectif de cette proposition consiste à aider les concepteurs à produire convenablement des diagrammes UML.

Dans cet article, nous étudions une première étape dans cette direction en présentant une initiative qui consiste à générer des diagrammes de classes et des diagrammes d'état-transitions à partir des machines B. La dérivation d'autres diagrammes UML tels que le diagramme de séquences et le diagramme de collabora-

tions est envisageable. Nous observons sur des études de cas qu'à un concept de la méthode B (les ensembles, les constantes, les variables, ...) peuvent correspondre plusieurs concepts distincts d'UML. Afin d'obtenir les modèles UML les plus «naturels», il nous paraît peu prometteur d'envisager une traduction automatique. La méthode que nous proposons est à la fois interactive et incrémentale. D'une part, l'utilisateur pourra participer dans le choix des alternatives possibles au cours de la transformation. D'autre part, une machine B est considérée comme étant un tout et la traduction de l'une de ses composants peut dépendre des autres. Nous proposerons alors des directives qui aideront l'utilisateur à choisir entre plusieurs alternatives. Les diagrammes générés seront complétés par des contraintes exprimées par le langage OCL (Object Constraint Language [8]). Ce langage permet de spécifier des invariants de classes, des contraintes ainsi que des pré-conditions et des post-conditions dans les méthodes des classes et dans les transitions d'états.

Dans la section 2, nous illustrons notre approche sur une étude de cas en considérant le modèle abstrait ainsi que les deux premiers raffinements de ce modèle. Les règles de transformation seront présentées dans la section 3. La section 4 présentera quelques éléments de comparaison avec des travaux semblables.

2 Exemple : système de contrôle d'accès aux bâtiments

Nous illustrons notre approche de transformation des spécifications B en des diagrammes UML à travers l'étude de cas du système de contrôle d'accès des personnes aux bâtiments [1]. Cet exemple montre l'application de certaines règles de transformation qui seront présentées dans la section 3. Nous nous intéressons d'abord à la transformation de la structure statique en termes d'UML (les classes, leurs attributs et les associations) avant de passer aux aspects dynamiques (les états et les transitions).

2.1 Modèle abstrait

Un modèle abstrait du système est reproduit dans la Fig. 1. (Ce modèle correspond en effet au premier raffinement du modèle décrit dans [1]). Les acteurs du système (les personnes et les bâtiments) sont représentés par des ensembles abstraits, et les autorisations *aut* d'accès des personnes aux bâtiments ainsi que les communications *com* entre les bâtiments sont représentées par des relations constantes, c'est à dire par des ensembles de couples, dont les types sont contraints dans la clause INVARIANT. La situation dynamique *sit* des personnes dans les bâtiments est représentée par une fonction totale.

Outre l'information sur les types, l'invariant affirme qu'à chaque état, les personnes ne peuvent être que dans les bâtiments où elles sont autorisées d'entrer et que *com* est irréflexive (aucun bâtiment ne communique avec lui-même). Le modèle présente un seul événement noté *pass* qui représente l'entrée d'une personne

MACHINE Batiment
SETS
$BAT; PERS$
CONSTANTS
aut, com
VARIABLES
sit
INVARIANT
$aut \in PERS \leftrightarrow BAT \wedge com \in BAT \leftrightarrow BAT \wedge$
$sit \in PERS \rightarrow BAT \wedge sit \subseteq aut \wedge com \cap \mathrm{id}(BAT) = \{\}$
OPERATIONS
$pass \mathrel{\widehat{=}}$ ANY p, b
WHERE $(p,b) \in aut \wedge (sit(p), b) \in com$
THEN $sit(p) := b$
END
END

FIG. 1 – Spécification abstraite du système de contrôle d'accès.

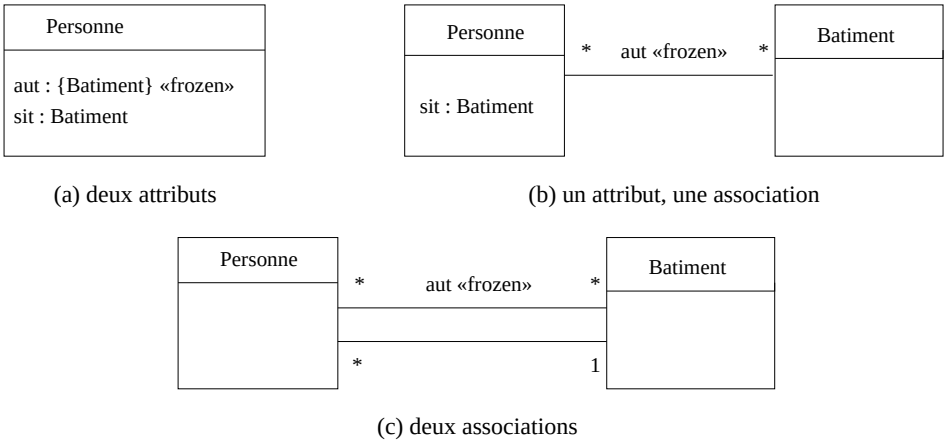
p dans un bâtiment b à condition que p soit autorisé d'entrer dans b et que b communique avec le bâtiment où se trouve p.

Nous développons maintenant un diagramme de classes UML correspondant au modèle B de la Fig. 1. Un ensemble abstrait comme *BAT* et *PERS* peut être dérivé en UML par un type de base ou par une classe. L'utilisation de tels ensembles dans les autres composants du modèle B remet en cause le choix entre ces deux possibilités. Par exemple, l'ensemble *PERS* apparaît dans le domaine de la fonction *sit* et aussi dans le domaine de la relation *aut*. En UML, une classe est plus importante qu'un type ou un attribut car elle possède des attributs et des associations avec d'autres classes. C'est sur la base de ce critère que nous devons décider si un ensemble est une classe ou plutôt un type. Un ensemble qui apparaît comme domaine de relation(s) fonctionnelle(s) telles que les fonctions, les injections, ... sera représenté par une classe. Dans le cas où il n'y a que des relations non-fonctionnelles entre les ensembles, l'utilisateur pourrait intervenir pour choisir quels sont les ensembles qui seront dérivés en des classes. Remarquons que dans le cas de la relation *com*, il s'agit d'une relation irréflexive entre les bâtiments. L'ensemble *BAT* représente à la fois le domaine et le codomaine de la relation *com*. Pour cela, cet ensemble doit être représenté par une classe au lieu d'un simple type. Ces observations nous permettent de suggérer que *PERS* ainsi que *BAT* représentent des classes. Une telle suggestion est plus naturelle du côté du concepteur en UML.

Nous passons ensuite à analyser la variable *sit* et les constantes *aut* et *com*. Rappelons que le langage B utilise les variables pour décrire les entités dont les valeurs changent pendant l'exécution du système. Les constantes décrivent les entités dont les valeurs ne sont pas modifiées. Nous avons différents choix dans la dériva-

tion des variables et des constantes en UML. Ces éléments peuvent représenter des associations, des attributs, des états, ...

La clause INVARIANT dans le modèle B offre quelques indices aidant à résoudre ce choix. La constante *aut* décrit une relation entre deux ensembles que nous avons décidé de représenter comme étant des classes en UML ; il nous paraît donc naturel de modéliser *aut* comme étant une association ayant la multiplicité (*,*) entre ces classes. Il serait également possible de représenter *aut* par un attribut multi-valué dans la classe *Personne* (pour indiquer l'ensemble des bâtiments autorisés pour une cette personne) ou dans *Batiment* (pour indiquer l'ensemble des personnes qui sont autorisées à entrer dans ce bâtiment). De façon analogue, nous représentons la relation *com* par une association irréflexive de multiplicité (*,*) de la classe *Batiment*, bien qu'elle pourrait aussi être représentée par un attribut multi-valué dans cette classe. La contrainte «frozen» prédéfinie par UML sera ajoutée aux associations *com* et *aut*. Cette contrainte affirme que la relation elle-même (*com* ou *aut*) reste inchangée (pas d'ajout ou de suppression de couples une fois que les liens sont créés).



(a) deux attributs

(b) un attribut, une association

(c) deux associations

FIG. 2 – Les representations possibles pour *aut* et *sit*.

La variable *sit* représente une fonction dont le domain *PERS* et le codomaine *BAT* sont des classes dans le modèle UML. En utilisant le même raisonnement, cette variable peut être dérivée en une association de multiplicité (*,1) ou en un attribut du type *Batiment* dans la classe *Personne*. La Fig. 2 récapitule ces choix. Pour le reste de cet article, nous choisissons le modèle décrit dans la Fig. 2(b). Nous avons maintenant identifié les classes, les associations, et les attributs correspondants aux éléments du modèle B. Les invariants qui n'ont pas été transformés en des attributs et des associations peuvent être exprimés par des contraintes OCL, comme suit :

context *Personne* **inv :**
$self.aut \rightarrow includes(self.sit)$

context *Batiment* **inv :**
$not(self.com \rightarrow includes(self))$

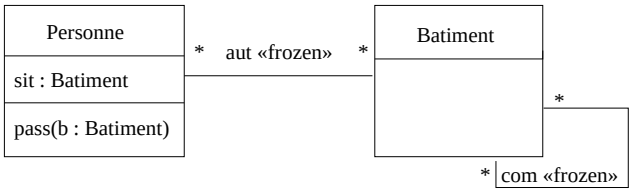


FIG. 3 – Diagramme de classes correspondant au modèle abstrait.

Nous nous intéressons maintenant à l'événement *pass* dans le modèle B de la Fig. 1. En général, les événements sont transformés en des méthodes dans des classes ou en des transitions entre les états dans des diagrammes d'état-transitions. Elles peuvent être aussi décrites par des diagrammes d'interactions, en particulier quand des instances de différentes classes sont impliquées par l'événement. Dans notre cas, nous remarquons qu'il y a deux objets (une personne p et un bâtiment b) participants dans l'événement *pass*. Les types sont implicitement donnés par la pré-condition $(p,b) \in aut$. L'effet de l'événement consiste à mettre à jour la valeur de $sit(p)$ qui est représenté en UML par un attribut dans la classe *Personne*. Cet événement peut alors être représenté par une méthode dans la classe *Personne* en prenant un paramètre b de type *Batiment*. En se basant sur la notation OCL pour les pré-conditions et les post-conditions, cette méthode peut être décrite comme suit :

context $Personne :: pass(b : Batiment)$
$pre : self.aut \rightarrow includes(b)$ *and* $self.sit.com \rightarrow includes(b)$
$post : self.sit = b$

En résumé, nous obtenons le diagramme de classe de la Fig. 3 pour représenter le modèle B montré dans la Fig. 1. Ce premier modèle B ne contient pas assez d'informations pour pouvoir dériver un diagramme d'état-transitions. Nous retardons alors la discussion sur cet aspect de transformation pour le raffinement B du même système dans la section suivante.

2.2 Premier raffinement

La Fig 4 présente un premier raffinement du système de contrôle d'accès. Il introduit les portes représentées par l'ensemble *PORTE* pour relier les bâtiments. Les fonctions *orig* et *dest* associent respectivement, à chaque porte le bâtiment origine et le bâtiment destination.

La variable *pdb* indique quelle porte, s'il en existe, est actuellement débloquée pour une personne donnée. C'est une injection partielle, exprimant qu'à tout moment, au plus une porte peut être débloquée pour une personne et, réciproquement, une porte ne peut être débloquée que pour au plus une seule personne.

Les portes sont dotées des voyants verts et rouges. Le voyant vert est allumé aussi longtemps qu'une porte est débloquée ; l'ensemble *vert* est alors défini comme étant le codomaine de la fonction *pdb*. D'autre part, le voyant rouge indique qu'on

REFINEMENT Batiment1
REFINES
Batiment
SETS
$PORTE$
CONSTANTS
$orig, dest$
VARIABLES
$pdb, rouge$
DEFINITIONS
$admis(p,q) \mathrel{\widehat{=}} orig(q) = sit(p) \wedge p \notin \mathrm{dom}(pdb) \wedge (p, dest(q)) \in aut$
$vert \mathrel{\widehat{=}} \mathrm{ran}(pdb)$
INVARIANT
$orig \in PORTE \rightarrow BAT \wedge dest \in PORTE \rightarrow BAT \wedge$
$pdb \in PERS \rightarrowtail\!\!\!\!\!\!\rightarrow PORTE \wedge rouge \subseteq PORTE \wedge vert \cap rouge = \{\} \wedge$
$com = (orig^{-1}; dest) \wedge (pdb; orig) \subseteq sit \wedge (pdb; dest) \subseteq aut$
OPERATIONS
$pass \mathrel{\widehat{=}}$ ANY q
WHERE $q \in vert$
THEN $sit(pdb^{-1}(q)) := dest(q)$
$pdb := pdb \rhd \{q\}$
END
$debloquer \mathrel{\widehat{=}}$ ANY p, q
WHERE $p \in PERS \wedge q \in PORTE \wedge$
$q \notin vert \cup rouge \wedge admis(p,q)$
THEN $pdb := pdb \cup \{p \mapsto q\}$
END
$refuser \mathrel{\widehat{=}}$ ANY p, q
WHERE $p \in PERS \wedge q \in PORTE \wedge$
$q \notin vert \cup rouge \wedge \neg admis(p,q)$
THEN $rouge := rouge \cup \{q\}$
END
$rebloquer \mathrel{\widehat{=}}$ ANY q
WHERE $q \in vert$
THEN $pdb := pdb \rhd \{q\}$
END
$liberer \mathrel{\widehat{=}}$ ANY q
WHERE $q \in rouge$
THEN $rouge := rouge - \{q\}$
END
END

FIG. 4 – Raffinement de la spécification abstraite du système de contrôle d'accès

ne permet pas à une personne de franchir une porte ; le sous-ensemble *rouge* de *PORTE* représente les portes dont le voyant rouge est allumé. L'invariant $vert \cap rouge = \{\}$ affirme que les ensembles *rouge* et *vert* doivent être disjoints. L'invariant $(pdb; orig) \subseteq sit \wedge (pdb; dest) \subseteq aut$ affirme qu'une porte ne sera débloquée que pour une personne située dans le bâtiment origine de cette porte et que la personne doit être autorisée à entrer dans le bâtiment destination. Le modèle raffiné redéfinit l'événement *pass* qui était déjà présent dans le modèle abstrait. Cet événement prend un seul paramètre formel *q* représentant une porte débloquée, puisque la personne concernée et le bâtiment destination peuvent être déduits des fonctions *pdb* et *dest*. Les événements *debloquer* et *refuser* modélisent la tentative d'une personne de débloquer une porte donnée et permettent d'allumer les voyants verts ou rouges. Les événements *rebloquer* et *liberer* modélisent la fermeture spontanée d'une porte débloquée et permettent d'éteindre les voyants verts et rouges. Normalement, ces événements devraient être activés après que les voyants aient été allumés pour une certaine période, mais ceci ne peut être exprimé formellement à ce niveau d'abstraction qui n'inclut pas le temps réel.

Nous passons maintenant à la transformation de cette spécification B en un modèle UML en nous concentrant d'abord sur la modélisation de l'aspect statique du système. L'ensemble abstrait *PORTE* apparaît comme le domaine de certaines fonctions. En suivant notre argumentation de la section 2.1 concernant la transformation, cet ensemble sera représenté par une classe *Porte*. Les fonctions *orig* et *dest* peuvent être représentées par des attributs ayant la propriété «frozen». Elles pourraient être également traduites en deux associations entre *Porte* et *Batiment*. Notons que l'utilisateur a la liberté de choisir entre ces possibilités. Nous adoptons le premier choix pour la suite de cet article. Pour ce qui concerne la traduction de *pdb* qui est une injection partielle de *PERS* vers *PORTE*, nous avons choisi de représenter cette fonction par une association de multiplicité ((0..1),(0..1)) entre ces classes. Une injection partielle peut aussi représenter un attribut optionnel. L'invariant lié à *pdb* devra être exprimé par une contrainte de OCL comme suit :

context *Personne* **inv :**
if $self.pdb \rightarrow notEmpty$
then $self.sit = self.pdb.orig$ *and* $self.aut \rightarrow includes(self.pdb.dest)$

La représentation des ensembles *vert* et *rouge* est plus intéressante. Ces deux ensembles sont des sous-ensembles de *PORTE*, que nous avons déjà identifié comme étant une classe. Les relations d'inclusion de la forme $e \subseteq C$, où *C* représente une classe peut avoir différentes interprétations. Par exemple, *e* pourrait être une sous-classe de *C* à condition que *e* représente lui-même une classe. Il pourrait également représenter l'ensemble des instances existantes de la classe *C* tel qu'il est fait dans Meyer [5] lors de la traduction d'UML vers B. En observant que les deux ensembles changent avec le temps et qu'ils doivent être disjoints, l'interprétation la plus naturelle consiste à considérer ces ensembles comme étant des états distincts d'un même objet. Dans cet exemple, nous introduisons un état *neutre* pour représenter les portes dont aucun voyant n'est allumé ($q \notin vert \cup rouge$). Ces éléments

seront traités lors de la génération du diagramme d'état-transitions d'un objet de la classe *PORTE* car ils modélisent l'aspect dynamique de cet objet.

Le prédicat *admis* affirme qu'une personne est admise à franchir une porte faisant communiquer le bâtiment où elle se trouve à un bâtiment où elle est autorisée à entrer. Nous introduisons une redondance en présentant explicitement une association *admis* ayant une multiplicité (*,*) entre les classes *Personne* et *Porte* dans le modèle UML. Un autre choix consisterait à représenter *admis* par une méthode dans la classe *Personne* ou la classe *Porte*. L'expression OCL correspondant à l'invariant de *admis* peut être exprimée comme suit :

$$\begin{array}{l}\textbf{context}\ \ Porte :: admis(p : Personne)\ \ \textbf{inv :}\\ \quad self.orig = p.sit\ \ and\ \ not(self.pdb \rightarrow includes(p))\\ \quad and\ \ p.aut \rightarrow includes(self.dest)\end{array}$$

Nous nous intéressons maintenant à la transformation des événements B. Nous convenons à dire qu'un événement B sera représenté par une méthode qui sera située dans la classe dont les attributs sont modifiés par l'événement. Si un événement modifie les attributs de plusieurs classes, la méthode correspondante peut être attribuée à une association entre les classes concernées. Par conséquent, nous choisissons encore de représenter l'événement *pass* par une méthode dans la classe *Personne* plutôt que dans la classe *Porte*. En effet, la pré-condition de *pass* et l'invariant assurent que la porte et la personne qui sont concernées par l'événement s'identifient mutuellement. La nouvelle méthode *pass* peut être décrite en utilisant la notation OCL comme suit :

$$\begin{array}{l}\textbf{context}\ \ Personne :: pass()\\ \quad pre\ \ :\ \ self.pdb \rightarrow notEmpty\ \ and\ \ self.pdb.etat = \#vert\\ \quad post\ :\ \ self.sit = (self.pdb@pre).dest\ \ and\ \ self.pdb \rightarrow isEmpty\end{array}$$

Tous les autres événements sont transformés en des méthodes dans la classe *Porte*. Nous remarquons que les événements *debloquer* et *refuser* correspondent à la tentative d'une personne de débloquer une porte avec des résultats différents selon que la personne est admise à franchir la porte ou non. En effet, les deux événements ont les mêmes paramètres et leurs gardes sont presque identiques hormis la clause indiquant si la personne est admise ou non. Nous proposons alors de regrouper ces deux événements en une seule méthode $tenter(p)$ dans la classe *Porte*. Le diagramme de classes montré dans la Fig. 5 récapitule les transformations discutées jusqu'ici.

Le raffinement B de la Fig. 4 contient assez d'informations pour produire un diagramme d'état-transitions pour un objet de la classe *Porte*. Nous avons déjà constaté qu'une porte peut être dans un des trois états *neutre*, *vert* ou *rouge*. Il ne reste alors qu'à identifier les transitions entre ces états. La présentation des événements se composant d'une garde et d'une substitution généralisée nous aide à identifier les états source et les états cibles des transition(s) associées ainsi que leurs gardes d'activation. La garde de l'événement *debloquer* exige que la porte soit dans

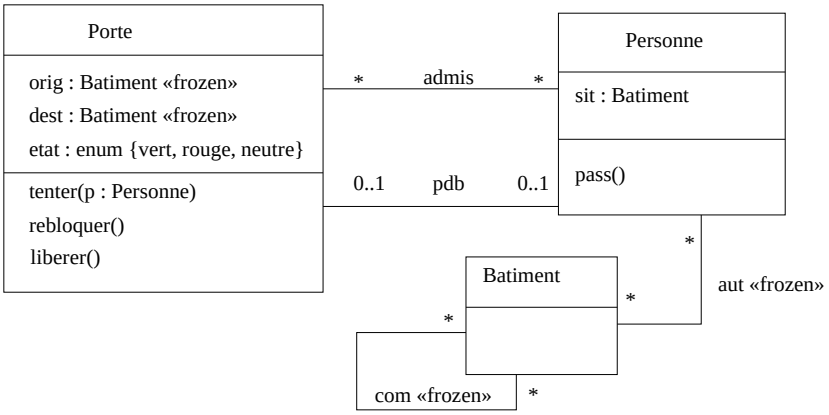


FIG. 5 – Diagramme de classes correspondant au premier raffinement.

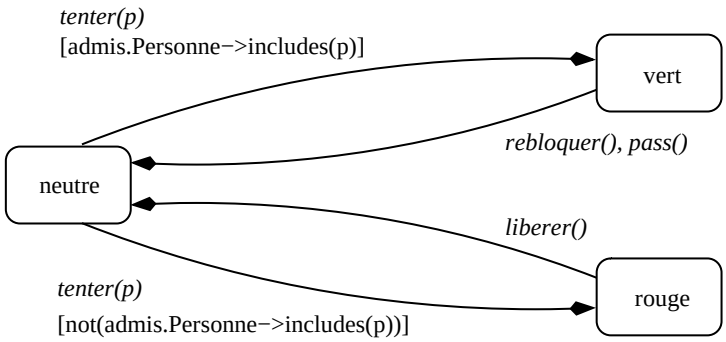


FIG. 6 – Diagramme d'état-transitions associé à la classe *Porte*.

l'état neutre ($q \notin vert \cup rouge$) et impose une condition qui indique que la personne p doit être admise pour franchir la porte. La post-condition permet de déduire que la porte sera dans l'état vert. De manière similaire, l'activation de l'événement *refuser* correspond à une transition de l'état neutre à l'état rouge. Dans le modèle UML, ces transitions correspondent à des appels à la méthode *tenter* dans des états satisfaisant des gardes différentes. En appliquant le même raisonnement aux autres méthodes de la classe *Porte*, nous obtenons le diagramme d'état-transitions de la Fig. 6, qui récapitule les comportements possibles d'une porte.

2.3 Deuxième raffinement

Le deuxième raffinement introduit notamment des lecteurs de cartes associés à chaque porte et qui permettent la capture de l'information lue sur la carte introduite et l'envoi de cette information au micro-ordinateur de contrôle. La transformation en UML suit les principes introduits précédemment et nous n'en présentons que les grandes lignes.

Un lecteur est bloqué pendant qu'une personne essaye de franchir la porte.

$$
\begin{array}{lll}
\textit{Lecture} \;\widehat{=} & \text{ANY} & p, q \\
& \text{WHERE} & p \in PERS \wedge\ q \in PORTE - LBL \\
& \text{THEN} & LBL := LBL \cup \{q\} \\
& & mLe := mLe \cup \{q \mapsto p\} \\
& \text{END} &
\end{array}
$$

$$
\begin{array}{lll}
\textit{Acquittement} \;\widehat{=} & \text{ANY} & q \\
& \text{WHERE} & q \in mLa \\
& \text{THEN} & LBL := LBL - \{q\} \\
& & mLa := mLa - \{q\} \\
& \text{END} &
\end{array}
$$

FIG. 7 – Deux événements du deuxième raffinement.

Plus précisément, il sera bloqué dès l'envoi du message identifiant la personne au contrôleur jusqu'à la réception du message d'acquittement indiquant la terminaison du protocole (avec succès ou non) et comprenant les périodes où les voyants vert ou rouge sont allumés. Le raffinement introduit les variables *LBL* (lecteurs bloqués), *mLe* (messages envoyés par les lecteurs), *mLa* (acquittements envoyés aux lecteurs) avec les invariants de typage

$$LBL \subseteq PORTE \wedge mLe \in PORTE \rightarrowtail\!\!\!\!\!\!\!+\; PERS \wedge mLa \subseteq PORTE$$

et un invariant supplémentaire qui exprime les sous-états des portes dont le lecteur est bloqué :

$$
\begin{array}{l}
\mathrm{dom}(mLe) \cup vert \cup rouge \cup mLa = LBL \wedge \\
\mathrm{dom}(mLe) \cap (vert \cup rouge \cup mLa) = \{\} \wedge \\
mLa \cap (vert \cup rouge) = \{\}
\end{array}
$$

La fonction partielle *mLe* entre les ensembles *PORTE* et *PERS*, transformés en des classes, sera représentée par une association de multiplicité ((0..1),(0..1)) entre ces classes. Le raffinement précédent a déjà permis d'établir les états *vert*, *rouge* et *neutre* d'une porte. En suivant une argumentation similaire, l'invariant supplémentaire nous suggère les trois autres états *LBL*, *mLe* et *mLa*, et nous pouvons déduire que *LBL* est un état composé des sous-états *mLe*, *vert*, *rouge* et *mLa*.

Le modèle introduit aussi deux événements *Lecture* et *Acquittement* (Fig. 7) modifiant ces variables. Ces événements seront transformés en des transitions dans le diagramme d'état-transitions associé à la classe *Porte* pour ce raffinement (Fig. 8). Notons que l'ensemble d'états *neutre*, *mLe* et *mLa* correspond au seul état *neutre* du raffinement précédent. Les autres événements correspondent à des événements du raffinement précédent en tenant compte des nouvelles variables.

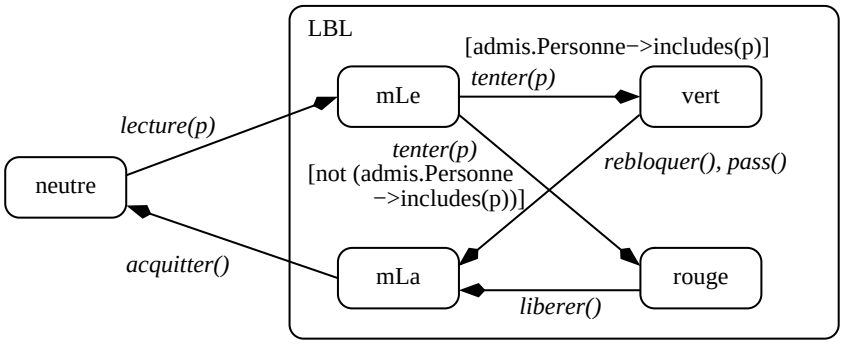


FIG. 8 – Diagramme d'état-transition du deuxième raffinement.

3 Les règles de transformation

Nous avons appliqué des transformations semblables sur les autres raffinements du modèle d'Abrial du système de contrôle d'accès [1] aussi bien que sur d'autres études de cas que nous avons trouvé dans la littérature. D'après cette expérience, il s'est avéré qu'il n'est pas raisonnable d'essayer d'établir une procédure de traduction automatisée pour transformer les spécifications B en UML parce qu'à un concept B donné peut correspondre différentes interprétations en UML et non pas une seule traduction. En plus, le choix entre des traductions dépend des autres composants de la spécification B et peut aussi dépendre des choix de l'utilisateur. Par exemple, une variable en B peut être représentée en UML par une association, un attribut ou encore par un état d'un objet. L'assistance de l'utilisateur sera nécessaire pour guider la traduction afin d'obtenir les modèles les plus convenables. Les décisions de transformation sont mutuellement dépendantes et elles doivent tenir compte de tous les composants d'une spécification B ainsi que de ses raffinements. Cependant, nous avons pu dégager certaines directives permettant d'identifier les alternatives disponibles et d'aider à choisir entre elles. Nous résumons ces directives par les règles suivantes :

Règle 1. Si un ensemble abstrait T apparaît dans le domaine de relations fonctionnelles (fonction, injection, surjection et bijection) avec d'autres ensembles dans une spécification B, alors il pourra être transformé en une classe *T*.

Exemple : l'ensemble *PORTE* dans la spécification B du système du contrôle d'accès sera transformé en une classe *Porte* dans un diagramme de classes.

Règle 2. Si un ensemble abstrait ou énuméré T apparaît dans le codomaine de certaines relations sans apparaître dans le domaine d'autres relations fonctionnelles dans une spécification B, il pourra être transformé en un type d'attribut *T* pour les classes représentant le domaine de ces relations.

Exemple : s'il n'y avait pas d'invariant spécifiant une relation entre les bâtiments, alors l'ensemble *BAT* pourrait être transformé en un type *Batiment* d'un

Type de relation	Multiplicité de A	Multiplicité de B
relation : $A \leftrightarrow B$	*	*
fonction partielle : $A \nrightarrow B$	*	0..1
fonction totale : $A \rightarrow B$	*	1
injection partielle : $A \rightarrowtail B$	0..1	0..1
injection totale : $A \rightarrowtail B$	0..1	1
surjection partielle : $A \twoheadrightarrow B$	1..*	0..1
surjection totale : $A \twoheadrightarrow B$	1..*	1
bijection partielle : $A \rightarrowtail\!\!\!\!\twoheadrightarrow B$	1	0..1
bijection totale : $A \rightarrowtail\!\!\!\!\twoheadrightarrow B$	1	1

TAB. 1 – Transformation des relations en des associations.

relation	fonction partielle (injection, surjection)	fonction totale (injection, surjection)
multi-valué	optionnel + mono-valué	obligatoire + mono-valué

TAB. 2 – Transformation des relations en des attributs.

attribut.

Règle 3. La relation d'inclusion entre deux ensembles S et E peut représenter :

- le concept de généralisation entre une sous-classe S et une super-classe E. Les invariants liés à l'ensemble E et spécifiant les attributs et les associations de la classe correspondante *E* seront hérités par la sous-classe *S* représentée par l'ensemble S.
  Exemple : un invariant $HOMME \subseteq PERS$ reliant deux ensembles *PERS* et *HOMME* peut montrer une relation de généralisation entre les deux classes *Homme* et *Personne*.
- un état possible S d'une instance de la classe E.
  Exemple : l'invariant $rouge \subseteq PORTE$ dans cette étude de cas décrit l'état *rouge* d'un objet *Porte*.
- l'ensemble des instances effectives S d'une classe E.
  Exemple : un invariant $produit \subseteq PRODUIT$ où *produit* est une variable et *PRODUIT* est un ensemble abstrait pourra être interprété comme suit : *produit* représente toutes les instances effectives de la classe *Produit* alors que *PRODUIT* représente toutes les instances possibles de cette classe.

Règle 4. Une relation $r : A \leftrightarrow B$ entre deux ensembles A et B représentant deux classes *A* et *B* peut être transformée en une association *r* entre ces classes ou en un attribut *r* de type *B* dans la classe *A*.

Dans le premier cas (tab. 1), nous distinguons deux multiplicités : l'une est associée à *A* (c-à-d le nombre d'instances de *A* associées à chaque instance de

B) et l'autre est associée à *B*. Dans le second cas (tab. 2), les attributs pourraient être obligatoires ou optionnels. Ils peuvent également être mono-valués ou multi-valués. Si *r* est une constante alors deux cas de figures sont possibles : la valeur du lien correspondant ne peut pas être modifiée s'il s'agit d'une association. Les valeurs des attributs correspondants ne pourraient pas changer une fois que l'objet est initialisé s'il s'agit d'un attribut.

Notons que dans le second cas (tab. 2), une fonction injective apportera une contrainte additionnelle affirmant qu'à chaque valeur de l'attribut *r* ne sera associée qu'une seule instance de la classe *A*. Cet attribut pourra alors jouer le rôle d'une clé primaire pour la classe *A*.

Règle 5. Les événements représentent des méthodes dans les classes et/ou des transitions dans un diagramme d'état-transitions.

Les pré-conditions indiquent des contraintes sur les classes ou les associations dans les diagrammes de classes. Elles peuvent décrire aussi les états sources et les gardes des transitions dans les diagrammes d'état-transitions.

Les post-conditions spécifient les attributs et/ou les liens ajoutés, modifiés ou supprimés dans le diagramme de classes. Ils peuvent également indiquer les états cibles des transitions.

4 Conclusion

Dans cet article, nous avons proposé une méthodologie visant à obtenir une modélisation UML à partir d'une spécification B en développant quelques règles de transformation. Cette transformation nécessite l'intervention de l'ingénieur concepteur pour assister à la prise de décisions entre les différents choix de dérivation des concepts B en des concepts UML. Le développement d'un outil supportant cette transformation est une perspective de ce travail. L'outil devra documenter les décisions prises pendant la transformation et vérifier sa cohérence. Cette expérience mènera à une clarification de notre démarche et à la dérivation d'autres règles. Puisque le raffinement est un concept central pour la méthode B, nous pensons qu'il devra être pris en considération en établissant des liens entre les modèles UML correspondants aux différents raffinements des spécifications B. Actuellement, la méthode de transformation proposée se limite au niveau des diagrammes de classes et des diagrammes d'état-transitions. La dérivation d'autres diagrammes tels que les diagrammes de séquences constitue une autre perspective de ce travail.

A notre connaissance, il n'existe pas beaucoup de travaux qui ont été établis pour générer des modèles UML à partir des spécifications B. Nous pouvons cependant citer le travail de Okalas et al. [6] qui est en cours de développement et qui vise à construire et à intégrer deux vues (UML et B) d'une même spécification. Nous citons aussi le travail de Laleau et al. [3] qui montre la nécessité de liens et de mécanismes de traçabilité entre les modèles UML et les modèles B pour pouvoir refléter les erreurs trouvées par l'atelier B dans une spécification B vers le modèle

UML correspondant. Ce travail propose des méta-structures globales permettant d'établir des relations entre les concepts UML et ceux de B mais sans donner des règles de transformation. Le travail de Tatibouët et al. [7] propose un outil de transformation de B vers UML (B2UML). Le processus de transformation de ce travail est automatique et ne permet pas l'intervention du concepteur. Ce travail impose des restrictions qui ne semblent pas être évidentes dans la pratique et qui ne lui permettent pas une pareille souplesse dans la transformation. Par exemple, d'après ce travail, les machines, les ensembles abstraits et les fonctions sont tous traduits en des classes ce qui n'est toujours convenable.

Références

[1] J.-R. Abrial. Etude de cas : Le contrôle d'accès aux bâtiments. `http://www-lsr.imag.fr/B/Documents/ClearSy-CaseStudies/PORTES/SourcesFR%/main.html`, June 2000.

[2] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modelling Language User Guide*. Addison Wesley, 1999.

[3] R. Laleau and F. Pollack. Coming and going from UML to B : a proposal to support traceability in rigorous IS development. In *Intl. Conf. Formal Specification and Development in Z and B (ZB2002)*, volume 2272 of *Lecture Notes in Computer Science*, pages 517–534, Grenoble, France, January 2002. Springer Verlag.

[4] H. Ledang. *Traduction Systematique de spécifications UML en B*. PhD thesis, Université Nancy 2 & LORIA, Nancy, France, 2002.

[5] E. Meyer. *Développement formel par objets : utilisation conjointe de B et d'UML*. PhD thesis, Université Nancy 2 & LORIA, Nancy, France, March 2001.

[6] D. Okalas Ossami, J. Souquières, and J. Jacquot. Assistance à la construction de spécifications multi-vues UML et B. In *Proceedings of MAJECSTIC*, Marseilles, France, October 29-31, 2003.

[7] B. Tatibouet and J.C. Voisinet. Generating statecharts from B specifications. In *16th Intl. Conf. Software and Systems Engineering and Their Applications (ICSSEA)*, volume 1, Paris, France, December 2003.

[8] J. Warmer and A. Kleppe. *The Object Constraint Language : Precise Modeling with UML*. Addison Wesley, 1998.