Validation des propriétés d'un scénario UML/OCL à partir de sa dérivation en B

Ninh Thuan Truong, Jeanine Souquières

Validation des propriétés d'un scénario UML/OCL à partir de sa dérivation en B

Ninh-Thuan TRUONG, Jeanine SOUQUIERES

LORIA, Campus Scientifique - BP 239
54506 Vandœuvre lès Nancy cedex, France
Email: {truong, souquier}@loria.fr

Résumé: *La dérivation de spécifications UML vers B est considérée comme une approche appropriée pour utiliser conjointement UML et B dans un développement unifié, pratique et rigoureux de logiciels. On peut utiliser la spécification UML/OCL pour modéliser un système et il est possible d'utiliser des outils supports puissants de B comme AtelierB pour analyser les spécifications B dérivées afin d'identifier les défauts au sein de spécifications UML/OCL.*
Cet article aborde le couplage UML et B à l'aide d'un scénario présenté sous la forme d'un diagramme de classes, d'expressions OCL et de diagrammes de collaboration. En se basant sur l'analyse des obligations de preuves de B, la spécification UML et des contraintes OCL, nous construisons une dérivation d'un scénario UML en B en améliorant et résolvant les limites de la dérivation de diagrammes de collaboration proposée par Ledang [Led02] et vérifions également les propriétés de spécifications UML/OCL.

Mots clés: B, UML, OCL, validation, obligation de preuve.

1 Introduction

UML (Unified Modelling Language) et le langage B sont deux techniques de spécification reconnues en génie logiciel. UML propose des notations graphiques de modélisation, aisément accessibles en pratique. Les développeurs utilisent UML pour spécifier des systèmes. Cependant ces notations manquent de rigueur et de sémantique formelles, cela signifie qu'il n'est pas possible de vérifier une implémentation qui satisfasse la spécification UML ou la consistance entre des propriétés de spécifications UML. Les programmes développés à partir de ces spécifications ne sont pas sûrs et rigoureux. Une solution afin de renforcer la confiance de la spécification du logiciel est l'utilisation de techniques de validation et vérification formelles pendant le processus de développement.
B est une méthode formelle connue avec une sémantique précise utilisant les notations mathématiques. L'intégration d'UML et B est motivée par le souhait de pouvoir les utiliser conjointement dans une approche de développement de logiciels pratique, unifiée et rigoureuse. En fonction de l'objectif, il y a plusieurs approches pour coupler des notations graphiques à objets et des méthodes formelles [HB95]. Deux approches du couplage d'UML et B sont possibles:

- Couplage par adjonction: il s'agit d'une intégration partielle, dans laquelle une partie du modèle à objets UML est remplacée par une expression formelle B. Cette approche est utile lorsque l'on souhaite ajouter plus de rigueur au développement.
- Couplage par dérivation: il s'agit de la transformation systématique des diagrammes UML en B. Cela nous fournit une vue générale d'un système avec deux spécifications et avec la possibilité de détecter certaines inconsistances dans la spécification.

Concernant le couplage d'UML et B, plusieurs approches sont proposées [CB00, FLN96, Mey01, Led02]. Cependant, ces approches se concentrent sur la transformation de la spécification UML en B. Ces approches ne montrent pas clairement l'application des transformations proposées à la vérification de spécification UML. Dans un but pragmatique d'utiliser la méthode formelle B pour prouver la spécification UML, en nous basant sur la dérivation systématique de E. Meyer [Mey01] et H. Ledang [Led02], nous proposons la dérivation d'un scénario UML présenté à l'aide d'un diagramme de classes, de contraintes OCL et de diagrammes de collaboration vers B afin de vérifier la consistance des propriétés attendues d'un scénario UML/OCL.
L'article est structuré comme suit. Dans la section 2, nous présentons les caractéristiques de la méthode B. Dans la section 3, nous rappelons les notations essentielles du langage UML et OCL. Un exposé plus important est consacré au processus de preuve et de validation de spécifications dans la section 4. Nous y abordons également la dérivation de UML en B et l'analyse des machines abstraites B dérivées qui peuvent être prouvées par les obligations de preuve. Un exemple pour illustrer la preuve des propriétés de UML/OCL est présenté dans la section 5. Enfin, nous donnons les conclusions et les perspectives de ce travail dans la section 6.

2 Méthode B

La méthode B [Abr96, Sch01] conçue par J.R. Abrial est une méthode formelle qui couvre toutes les étapes de développement du logiciel, de la spécification jusqu'à la génération de code exécutable, par une succession d'étapes de raffinement [SS99]. Elle permet l'expression de propriétés sous la forme de prédicats du premier ordre et les opérateurs auxquels on peut faire appel sont ceux de l'arithmétique et de la théorie des ensembles. Les données sont encapsulées au sein de machines abstraites et ne peuvent être modifiées que par des opérations définies dans la même machine. Les caractéristiques de la méthode B sont:

Modèle B: L'utilisateur de B construit des modèles mathématiques. Ces modèles doivent exprimer les propriétés auxquelles le futur système doit obéir. Un modèle B est similaire à un module (ou encore un objet), il est représenté sous la forme d'une machine abstraite. De façon générale, il se scinde en deux parties:
- une partie statique déclare des variables qui définissent l'état du système modélisé. Ces déclarations sont complétées par des conditions (invariants) qui spécifient les propriétés que l'état du modèle doit toujours satisfaire,
- une partie dynamique définit les opérations qui décrivent l'évolution des machines abstraites (notamment son état) en utilisant les substitutions.

Raffinement: La réalisation d'un système complexe ne peut s'accomplir en une seule fois. La méthode B propose une construction par approximations successives. La notion de raffinement est fondamentale en B. D'une part, on va pouvoir l'utiliser pour introduire les détails de conception du cahier des charges qui n'étaient pas pris en compte auparavant. D'autre part, le raffinement est également utilisé pour concrétiser les modèles, ceci afin de s'orienter vers une implémentation.

Substitution: Dans un modèle B, une substitution est un moyen de représenter une transition sur l'état du modèle B. Il existe deux substitutions élémentaires, l'une notée **skip** pour conserver l'état courant et l'autre pour le modifier: la substitution simple $x := E$ a pour but d'assigner la variable x à la valeur de l'expression E. Les substitutions complexes sont construites à partir de ces deux substitutions élémentaires et des opérateurs de composition tels que le séquencement (;), le parallélisme ($\|$), la conditionnelle (if...then...end), ...

Preuve: Les preuves réalisées au cours du développement B s'assurent que le système construit possède bien les propriétés attendues. Il s'agit de preuves de conservation d'invariants et de correction du raffinement. Dans les premières preuves, on va vérifier que les propriétés décrites dans les invariants sont conservées dans l'abstraction et restent valables dans le raffinement. Il est important de noter que les preuves à satisfaire sont définies de façon rigoureuse dans la méthode, le praticien n'a pas à les écrire. Un outil (un générateur d'obligations de preuves) analyse le ou les modèles concernés et génère les différentes conditions qu'il est nécessaire de prouver pour assurer la cohérence de l'ensemble.

Outils: Un des grands avantages de B est de disposer d'ateliers logiciels performants [Ste, BC96] pour sa mise en œuvre dans des projets industriels. Les outils de preuve permettent de générer automatiquement des obligations de preuves à partir des composants du langage B. Pour qu'un composant soit correct, il faut ensuite démontrer ses obligations de preuve.

3 UML et les scénarios

UML (Unified Modeling Language) [BRJ98, OMG03] propose des notations graphiques de modélisation par objets indépendantes de tout processus de développement. Ces notations ont été conçues pour visualiser, spécifier, construire et documenter les artefacts d'un système. UML définit neuf types de diagrammes, chacun d'eux représente une vision spécifique du système. Ces diagrammes se divisent en deux catégories, quatre diagrammes présentent la structure statique, cinq présentent le comportement dynamique d'un système:

– Les aspects statiques sont décrits à l'aide des diagrammes de classe, des diagrammes d'objets, des diagrammes de composants et des diagrammes de déploiement.

– Le comportement est décrit à l'aide des diagrammes de cas d'utilisation, des diagrammes de séquences, des diagrammes d'activité, des diagrammes de collaboration et des diagrammes d'état-transition.

Les scénarios sont des processus importants dans la spécification UML, ils sont associés à des vues statiques et à des vues dynamiques.

Définition: Un scénario est une instance d'un cas d'utilisation. Il décrit un exemple d'interaction possible entre le système et les acteurs.

Un scénario est formalisé par un diagramme d'interaction (diagramme de séquence et diagramme de collaboration). Il est complété par un diagramme de classes qui spécifie sa structure statique. Les diagrammes de séquences sont une autre façon de présenter des diagrammes de collaboration, nous allons aborder les diagrammes de classes et les diagrammes de collaboration.

3.1 Diagrammes de classes

Les classes sont présentées graphiquement dans les diagrammes de classes; ces diagrammes forment les principaux constituants de la vue logique de l'architecture du système. Un diagramme de classes montre uniquement les aspects statiques du modèle et fait abstraction des aspects dynamiques ou temporels. Il déclare cependant des éléments comportementaux comme les opérations mais la dynamique de celles-ci est exprimée dans d'autres diagrammes.

3.2 Diagrammes de collaboration

Les diagrammes de collaboration montrent des interactions entre objets en insistant particulièrement sur la structure spatiale statique qui permet la mise en collaboration d'un groupe d'objets. Les diagrammes de collaboration expriment à la fois le contexte d'un groupe d'objets (au travers des objets et des liens) et l'interaction entre ces objets (par les envois de messages). Les diagrammes de collaboration peuvent être utilisés pour réaliser un cas d'utilisation ou pour implanter une opération. Dans notre travail, nous utilisons les diagrammes de collaboration pour représenter la réalisation des opérations dans un scénario. Dans la spécification UML, on utilise un diagramme de cas d'utilisation pour décrire des scénarios entre les acteurs et le système, les diagrammes d'interaction pour détailler les comportements entre les différents objets dans le système.

3.3 Vue générale d'OCL

OCL (Object Constraint Language) [WK99] est considéré comme le langage *formel* officiel au sein de modèles UML. Le point fort d'OCL est son orientation objet. Le langage fournit des variables et des opérations pour construire les expressions de contraintes. Il y a deux types de contraintes OCL:

– Invariant: les invariants sont des conditions qui portent sur toutes les instances de classificateurs. Ce sont souvent des conditions supplémentaires au sein d'une classe, d'un type ou sur l'association entre les classes que l'on ne peut pas spécifier en utilisant les notations graphiques UML.
– Pré- et post-condition: OCL fournit une syntaxe spéciale pour spécifier les pré- et post-conditions d'opérations dans le modèle UML. Pré- et post-conditions sont des contraintes qui définissent un contrat que l'implémentation d'opérations doit satisfaire. La pré-condition décrit la condition que l'on doit respecter avant l'exécution de l'opération. La post-condition décrit les effets produits par l'exécution de l'opération.

Dans une spécification UML/OCL, nous devons valider les propriétés de la manière suivante:

- invariants de classe,
- les contraintes de pré- et post-conditions des opérations séquentielles d'une implantation,
- les contraintes de pré- et post-conditions des opérations appelées par rapport aux contraintes de pré- et post-conditions d'opérations appelantes,
- les contraintes de pré- et post-conditions des opérations par rapport aux invariants dans une classe et une implantation (via les diagrammes d'interaction).

4 Processus de preuve, de dérivation et de validation

Dans ce chapitre, nous présentons les processus de preuve et la validation des propriétés dans une spécification UML/OCL. Nous présentons en détail la procédure de dérivation d'UML vers B et les propriétés de UML/OCL qui doivent être vérifiées.

4.1 Processus de preuve

Une spécification peut se révéler insatisfaisante du point de vue de la preuve de deux manières différentes. D'une part, la spécification peut être incomplète et ne pas comporter suffisamment d'informations pour mener la preuve à son terme (par exemple: "démontrer qu'une variable est toujours paire sans avoir d'information sur ses valeurs potentielles") [Mar97].

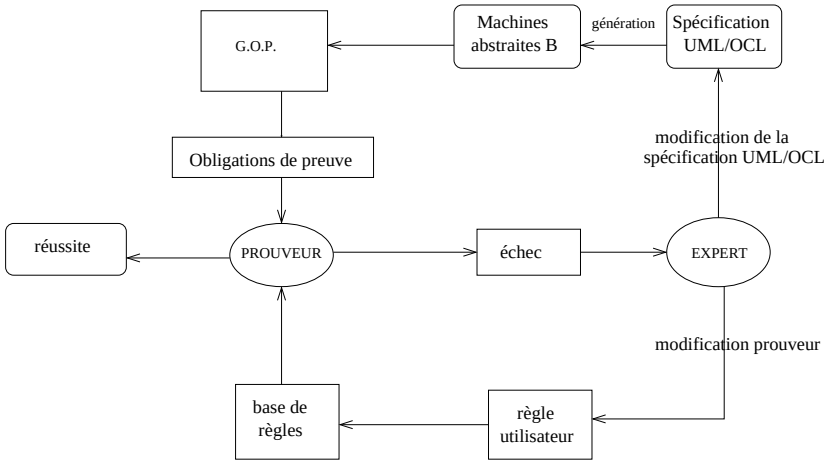

FIG. 1 – *Processus de preuve*

D'autre part, la spécification peut être incohérente et l'outil de démonstration échoue dans sa preuve (par exemple: "démontrer qu'une valeur appartient à l'ensemble vide"). Dans les deux cas, l'anomalie est révélée par l'échec d'une preuve et la correction nécessite la remise en cause des spécifications. La figure 1 exprime le processus de preuve dans ces différents cas qui associe la spécification UML/OCL avec l'outil de preuve (AtelierB) de la méthode formelle B. A partir de la spécification UML/OCL, on génère les machines abstraites B. La Génération des obligations de Preuve (G.O.P) de l'outil va utiliser les machines B pour générer des obligations de preuves. Ensuite, le prouveur utilise ces obligations de preuves pour analyser et prouver des prédicats. En cas d'échec, un expert analyse l'échec et distingue deux cas: soit il modifie la spécification, soit il ajoute de nouvelles règles à la base de règles, et puis on recommence à prouver.

4.2 Dérivation de diagrammes de collaboration d'un scénario vers B

Chaque scénario présenté par un diagramme de collaboration a un message origine. Ce message se décompose en plusieurs messages qui eux mêmes continuent à se décomposer comme le message origine. Chaque message contient un numéro et un appel à une opération. Le numéro indique l'ordre d'exécution et l'appel d'opération pour réaliser ce message. Pour la démonstration de la procédure de dérivation, nous donnons un exemple général d'un diagramme de collaboration (voir figure 2). L'opération *op*1 (contenue dans le message 1) qui appartient à Class1, va appeler les opérations *op*1.1 (Class2) et *op*1.2 (Class3). L'opération *op*1.1 va appeler les opérations *op*1.1.1 (Class4), *op*1.1.2 (Class2) et *op*1.1.3 (Class5). L'opération *op*1.1.1 va appeler l'opération *op*1.1.1.1 (Class1).

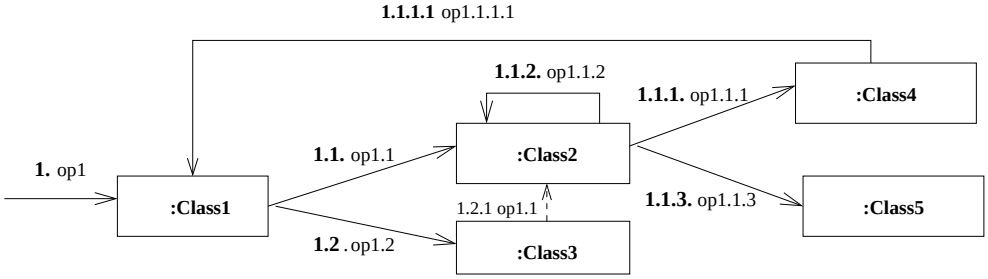

FIG. 2 – *Diagramme de collaboration*

Dans la spécification B correspondante, on considère que l'opération *op*1 se décompose en deux opérations *op*1.1 et *op*1.2 et l'opération *op*1.1 se décompose en trois opérations *op*1.1.1, *op*1.1.2 et *op*1.1.3,...
La dérivation de diagrammes de collaboration en B proposée par Ledang [Led01, Led02] utilise la notion de couches (les opérations sont organisées en couches). L'idée intuitive de la procédure de dérivation est de:

- **Créer la couche la plus haute**: toutes les opérations UML qui n'ont aucune opération appelante mais ont au moins une opération appelée forment la couche la plus haute.
- **Créer la couche la plus basse**: toutes les opérations UML qui n'ont aucune opération appelée forment la couche la plus basse.
- **Créer la(es) couches intermédiaire(s)**: à partir de la couche la plus haute, on cherche les opérations UML qui sont des opérations appelées par au moins une opération dans la couche la plus haute. Si aucune opération n'est trouvée (c'est-à-dire on a rencontré la couche la plus basse) alors on s'arrête, autrement les opérations UML trouvées forment la première couche intermédiaire. On répète cette démarche mais cette fois-ci on cherche les opérations UML qui sont des opérations appelées par les opérations dans la couche la plus haute ou bien dans les couches intermédiaires formées précédemment jusqu'à ce qu'on rencontre la couche la plus basse.

Cette dérivation a certaines limites. Les relations appelante-appelées contenant les paires récursives ou la dépendance circulaire entre les opérations ne sont pas prises en compte afin d'éviter la boucle infinie de la division. C'est le cas du message 1.1.1.1. Si l'opération *op*1.1.1.1 appelle l'opération *op*1.1 de message 1.1, il établit une dépendance circulaire et on ne peut pas distribuer les messages de diagramme du collaboration en couches. Ou bien, si le message 1.2 (Class3) appelle l'opération *op*1.1 qui appartient à la Class2. On ne sait pas l'opération *op*1.1 appartient à quelle couche , parce que avec le message 1.1, elle appartient à la couche 2 et avec le message

1.2.1 (voir Figure 2), elle appartient à la couche 3. Malheureusement, ce type de dépendance entre les messages apparaît souvent dans les diagrammes de collaboration et la programmation orienté objet. Nous proposons une amélioration de cette dérivation pour résoudre ces limites dans laquelle:

- Chaque classe du modèle UML est dérivée en B comme une machine abstraite.
- Deux machines abstraites *System* et *Basic* sont créées et les données de chaque machine sont toutes les données des machines classes dans le scénario.
- Les opérations qui appellent une ou plusieurs autres opérations dans le scénario appartiennent à la machine *System* (les opérations *op*1, *op*1.1, *op*1.1.1, *op*1.1.1.1).
- Les opérations appelées dans le scénario appartiennent à la machine *Basic* (*op*1.1, *op*1.2, *op*1.1.1, *op*1.1.2, *op*1.1.3, *op*1.1.1.1).
- L'implémentation *System.imp* raffine (REFINE) la machine abstraite *System.mch* et importe (IMPORT) la machine *Basic* (voir figure 3).
- Le contenu d'une opération de machine abstraite B est dérivé à partir de la spécification OCL de l'opération UML correspondante [Led02].
- Le contenu d'une opération d'implémentation B est dérivé à partir des messages dans le scénario.
- L'invariant de chaque machine *Basic* et *System* est associé aux invariants des classes dans le modèle UML et aux invariants OCL de ces classes.
- Toutes machines dans système voient (SEES) la machine *Types*.

Remarque: Une spécification B ne peut pas contenir plusieurs opérations ayant le même nom. C'est le cas des opérations *op*1.1,*op*1.1.1 et *op*1.1.1.1 qui apparaissent à la fois dans la machine *System* et dans la machine *Basic*. Nous proposons de renommer toutes les opérations de la machine *System* (exemple: *op*1_*sys*, *op*1.1_*sys*, *op*1.1.1_*sys* et *op*1.1.1.1_*sys*).

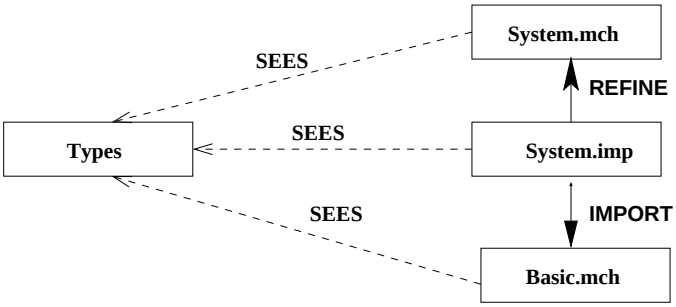

FIG. 3 – *Structure des machines dérivées*

Il est nécessaire de transposer les opérations des différentes classes dans la machine *System* et *Basic* en provenance des machines de classes parce qu'avec OCL, ces opérations utiliseront les données dans d'autres classes pour les traiter. Mais chaque classe est transformée en une machine abstraite et ces machines abstraites ne peuvent pas accéder aux propriétés des autres machines, il faut donc regrouper les données et les opérations dans une même machine. Avec cette dérivation, nous pouvons dériver tous les diagrammes de collaboration en B même lorsque les classes et les objets du diagramme d'interaction envoient des messages complexes.

4.3 Vérification et Validation

L'approche proposée est intéressante car elle permet de décomposer la vérification en s'intéressant aux différents scénarios du système de manière indépendante plutôt qu'au système vu de manière

globale. Dans cette partie nous considérerons ce que le prouveur de l'outil peut prouver et vérifier dans un scénario. Nous analysons les obligations de preuve générées par B.

Obligation de preuve dans une machine abstraite: Une machine abstraite s'exprime comme présenté figure 4. Une obligation de preuve étant de la forme : $H \Rightarrow P$, nous devons démontrer P sachant H, P et H étant des prédicats élaborés à partir du texte des spécifications. Si P est de la forme $P1 \wedge P2... \wedge Pn$, n preuves sont à faire, chacune étant de la forme $H \Rightarrow Pi$. Les obligations de preuve générées dans l'outil proviennent de machines abstraites décrites ci-dessous.

MACHINE $N(p)$
CONSTRAINTS C
SETS St
CONSTANTS k
PROPERTIES B
VARIABLES v
INVARIANT I
INITIALISATION T
OPERATIONS
$y \longleftarrow op(x) =$
pre P
then S
end;
...
END

FIG. 4 – *Structure d'une machine abstraite*

1. $\exists p.C$ obligation de preuve pour la clause CONSTRAINTS
2. $C \rightarrow (\exists St,k.B)$ obligation de preuve pour la clause PROPERTIES
3. $B \wedge C \rightarrow \exists v.I$ preuve pour la clause INVARIANT
4. $B \wedge C \rightarrow [T]I$ preuve pour la clause INITIALISATION
5. $B \wedge C \wedge I \wedge P \rightarrow [S]I$ preuve pour la clause OPERATIONS

Avec ces obligations de preuve, les invariants, l'initialisation, les pré- et post-conditions d'opérations de chaque machine seront prouvées.

Obligation de preuve dans une implémentation (ou raffinement): Chaque opération de l'implémentation est une nouvelle version (concrète) d'une opération précédemment spécifiée. Les entêtes des deux opérations sont identiques, seule la substitution généralisée définissant l'effet de l'opération est modifiée. *Une opération d'une implémentation est correcte lorsqu'elle préserve l'invariant sans contredire l'opération raffinée*. Comme pour les opérations des raffinements, le but à prouver est basé sur l'invariant de liaison de l'implémentation en conjonction avec un prédicat d'égalité entre les variables de sorties de l'opération de l'implémentation raffinement et les variables de sorties renommées de l'opération spécifiée. Le but à prouver est alors composé de l'application de l'opération de l'implémentation, de la négation de l'application de l'opération abstraite et de la négation de l'invariant [Ste]. Pour une opération de contenu T qui raffine l'opération *PRE P THEN S END*, les obligations de preuve peuvent s'écrire:

$$P \wedge I \wedge J \Rightarrow [T[u'/u]]\neg[S]\neg(J \wedge u' = u)$$

où I est l'invariant de machine raffiné, J est l'invariant de la machine d'implémentation et il se compose des invariants liés. *u* est une liste des paramètres résultats.
Outre l'application pour prouver les pré- et post-conditions des opérations préservant des invariants, les obligations de preuve de l'implémentation sont utilisées pour prouver l'intégrité entre les pré- et post-conditions des opérations séquentielles et entre les opérations décomposées et composées. Une machine implémentation qui importe (IMPORT) certaines machines abstraites va appeler les opérations de ces machines. Les opérations appelées vont s'exécuter séquentiellement (;). Nous utilisons les axiomes suivants [Abr96] pour envisager les pré- et post-conditions du séquencement entre deux opérations:

$(P \mid S); T \Leftrightarrow P \mid (S; T)$; (1)

$S; (P \mid T) \Leftrightarrow [S]P \mid (S; T)$; (2)

A partir de (1) et (2), la séquence entre deux opérations peut s'exprimer:

$$op1(); op2() = (S \mid P); (T \mid Q) \Leftrightarrow S \mid (P; (T \mid Q)) \Leftrightarrow S \mid ([P]T \mid (P; Q))$$

Cela signifie que le prouveur va prouver la consistance de toutes les pré- et post-conditions des opérations appelées pendant le temps d'exécution du système. *La valeur de variable dans la post-condition d'opération qui est exécutée précédemment va remplacer et vérifier la pré-condition de l'opération suivante ([P]T)*. De plus, les obligations de preuves qui sont générées par l'implémentation d'une opération peuvent valider la pré- et post-condition des opérations décomposées (T) qui correspondent à la pré- et post-condition d'une opération raffinée (P,Q) par:

$$P \wedge I \wedge J \Rightarrow [T[u'/u]]\neg[S]\neg(J \wedge u' = u)$$

Supposons que u est la liste des variables résultats des post-conditions de l'opération décomposée (T), u' est la liste des variables résultats des post-conditions des opérations raffinées (P,Q). E,E' sont des expressions, en provenance de l'obligation de preuve de l'implémentation:

$$A = [u' := E']\neg[u := E]\neg(J \wedge u' = u)$$
$$\Rightarrow A = [u' := E'](J \wedge u' = E)$$
$$\Rightarrow A = (J \wedge E = E')$$

Les obligations de preuve d'implémentation sont donc correctes si et seulement si $E = E'$, ou *les valeurs des variables de l'opération appelante sont identiques avec les valeurs des variables des opérations appelées*. Si une opération est appelante des autres opérations, quand une autre opération appelle cette opération, on vérifie seulement les pré- et post-conditions de celle-ci et on ne s'intéresse pas aux opérations appelées. Donc, la procédure de dérivation ci-dessus est convenable. Nous donnons dans la suite un exemple simple pour illustrer les applications de cette dérivation.

5 Exemple: un système de contrôle de trains

Un système de contrôle de trains se compose d'un contrôleur, d'un feu de signalisation et d'une barrière. A l'état normal, le feu est vert, le barrière est ouverte. Quand le train arrive, le contrôleur donne une commande pour arrêter la circulation, le feu passe à l'orange puis au rouge et la barrière se ferme. Quand le train est passé, le contrôleur donne une commande pour remettre en route la circulation. La barrière s'ouvre et le feu passe au vert.
La figure 5 présente le diagramme de classes du modèle UML de ce système. La figure 6 présente le diagramme de collaboration du scénario correspondant à la barrière fermée. L'opération *control_close* (l'opération *close* appartient à la classe *Control*) se décompose en trois opérations:

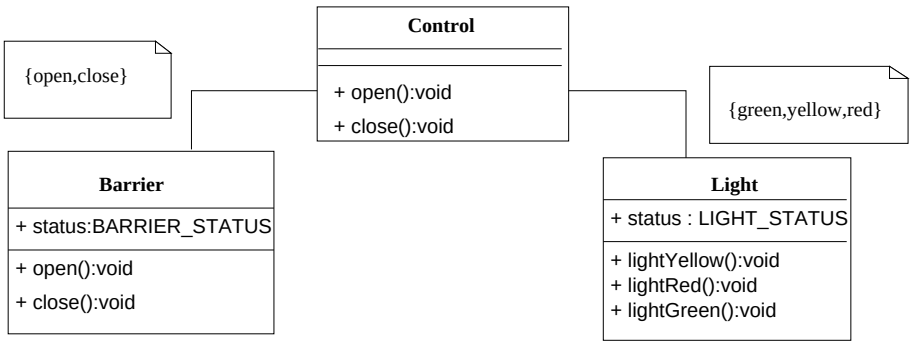

FIG. 5 – *Diagramme de classes du système de contrôle de trains*

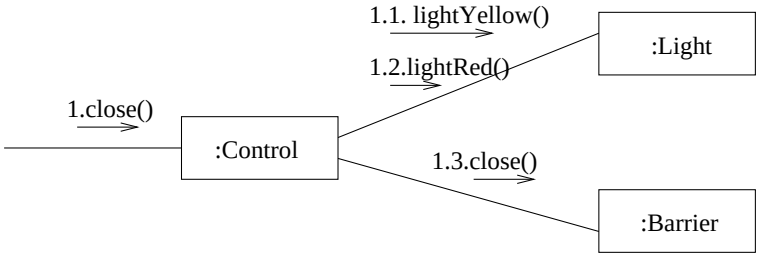

FIG. 6 – *Diagramme de collaboration du scénario barrière fermée*

light_lightYellow, *light_lightRed* et *barrier_close*. On va analyser la correction des pré- et post-conditions des opérations OCL du scénario présentées dans la figure 6:

Context Control::close():void
pre:
$light.status = green$
post:
$light.status = red$ and
$barrier.status = close$

Context Light::lightYellow():void
pre:
$self.status = green$
post:
$self.status = yellow$

Context Light::lightRed():void
pre:
$self.status = green$
post:
$self.status = red$

Context Barrier::close():void
pre:
$self.status = open$
post:
$self.status = close$

Après avoir dérivé en B la spécification du diagramme de classes UML et des opérations OCL [LS02] selon la procédure de dérivation présentée dans le paragraphe 4, lors de la dérivation UML en B, on renomme les propriétés UML (opération, attribut, ...) en les préfixant par le nom de la classe. Par exemple, l'attribute *status* de classe Light s'écrit *light_status*,... (voir les machines abstraites B dérivées dans ANNEXE). L'Atelier B donne les résultats de preuves de toutes les machines. Elles sont correctes, sauf pour la machine d'implémentation System_imp.imp, qui présente deux "unprouved". Le prouveur interactif nous donne l'obligation de preuve correspondant au premier "unprouved "(extrait de l'outil AtelierB):

```
co: control$1 &
```

```
light_status$1(controlLight$1(co)) = light_green &
"'Check preconditions of called operation, or While loop
 construction, or Assert predicates'" &
=>
(light_status$1<+{controlLight$1(co)|->light_yellow})
                        (controlLight$1(co)) = light_green
```

La raison de cette erreur est due à l'exécution séquentielle entre les opérations composées dans la spécification. L'opération *light_lightRed* est effectuée après l'opération *light_lightYellow*, les pré-conditions de l'opération *light_lightRed* (*self*.*status* = *green*) et les post-conditions de l'opération *light_lightYellow* (*self*.*status* = *yellow*) sont incohérentes. Donc, la spécification OCL de l'opération *lightRed* de la classe *Light* doit être revue de la manière suivante:

Context Light::lightRed():void
pre:
    self.*status* = **yellow**
post:
    self.*status* = *red*

L'obligation de preuve du deuxième "unprouved" est:

```
co: control$1 &
light_status$1(controlLight$1(co)) = light_green &
"'Check preconditions of called operation, or While loop
    construction, or Assert predicates'" &
 =>
barrier_status$1(controlBarrier$1(co)) = barrier_open
```

L'obligation de preuve appartient aux formules impliquées de la forme $P \Rightarrow Q$, mais on ne peut pas utiliser les hypothèses courantes pour prouver le but Q ($barrier\_status(controlBarrier(co)) = barrier\_open$ à partir des machines importées dans la spécification B, pré-condition de l'opération *barrier_close* dans la spécification OCL (*barrier_status* = *open*)). Parce que les hypothèses dérivées et le but n'ont pas de relation, on doit donc ajouter une hypothèse dans les hypothèses dérivées, c'est l'hypothèse de but prouvé. Dans la spécification OCL, on doit rajouter l'hypothèse (*barrier*.*status* = *open*) à la pré-condition de l'opération appelante *close*. Donc la spécification OCL de l'opération *close* de la classe *Control* est modifiée comme suit:

Context Control::close():void
pre:
    light.*status* = *green* and
    barrier.**status** = **open**
post:
    light.*status* = *red* and
    barrier.*status* = *close*

Cet exemple montre l'application du prouveur de la méthode B pour analyser les pré- et post-conditions des opérations décomposées et celles des opérations séquentielles. On peut également utiliser cette dérivation pour analyser le rapport entre les invariants et les pré- et post-conditions des opérations dans la spécification OCL grâce à la cohérence entre les invariants et les opérations dans la spécification B.

6 Conclusion

Dans cet article, nous avons présenté une façon de valider un scénario de spécification UML/OCL en le dérivant en B. Le scénario est exprimé par un diagramme de classes, un diagramme de collaboration et des expressions OCL. La dérivation de diagrammes de classes vers B permet de construire la structure des machines B; la dérivation des expressions OCL en B fournit le contenu des opérations (les transformations des expressions OCL en B peuvent être remplacées par les transformations des diagrammes d'état-transition en B) et la dérivation des diagrammes de collaboration vers B établit l'interaction entre les opérations dans la spécification B. Avec cette technique de transformation, nous pouvons construire un système B contenant moins de composants et résolvant les limites de la transformation de diagrammes de collaboration en B proposée par Ledang [Led02]. La rigueur dans la structure de la spécification B et les obligations de preuves générées par l'outil de preuves nous permettent de valider et de vérifier les propriétés dans la spécification UML.
Nous avons utilisé l'approche de dérivation et de validation sur un seul scénario UML. Cette approche peut être généralisée à plusieurs scénarios de diagrammes de collaboration (scénarios relationnels) en l'appliquant simultanément à chacun des scénarios. La vérification s'effectuant entre les pré- et post-condition d'opérations séquentielles, la vérification globale est donc immédiate. Le passage à l'échelle de l'utilisation de la technique proposée ne pose pas de difficultés particulières, si ce n'est dans la gestion du nombre important d'opérations. La validation des scénarios peut être appliquée dans les spécifications de systèmes critiques et de systèmes sécurisés.

Une maquette, ArgoUML+B [LSC03] a été réalisée permettant la transformation automatique d'un certains nombre de diagrammes UML en B en suivant les définitions proposées par E. Meyer [Mey01] et H. Ledang [Led02]. Ce système, implanté en Java, est basé sur une plate-forme d'édition et d'aide à la conception de diagrammes UML nommée ArgoUML [1]. Nous envisageons d'intégrer l'évolution des règles de transformation de diagrammes UML et d'expressions OCL dans cette maquette.

Références

[Abr96] Abrial (J.R.). – *The B-Book, Assigning Programs to Meanings*. – Cambridge University Press, 1996.

[BC96] B-Core(UK) (Ltd). – *B-Toolkit User's Manual*. – Oxford (UK), 1996. Release 3.2.

[BRJ98] Booch (G.), Rumbaugh (J.) et Jacopson (I.). – *The Unified Modeling Language User Guide*. – Addison-Wesley, 1998.

[CB00] Cook (C.) et Butler (M.). – Verifying Dynamic Properties of UML Models by Translation to the B Language and Toolkit. *Proceedings UML 2000 Workshop, Dynamic Behaviour in UML Models: Semantic Questions*, 2000.

[FLN96] Facon (P.), Laleau (R.) et Nguyen (H.P.). – Mapping Object Diagram into B. *Methods Integration Workshop*, Leeds, March 25-26 1996.

[HB95] Hinchey (M. G.) et Bowen (J. P.). – *Applications of Formal Methods*. – Prentice Hall, 1995.

[Led01] Ledang (H.). – Formalizing UML Behavioral Diagrams with B. *The tenth OOPSLA Workshop on Behavioral Semantics: Back to Basics*, USA, October 15 2001, pp. 162–171.

1. http://www.argouml.tigris.org

[Led02] Ledang (H.). – *Traduction Systématique de Spécifications UML en B*. – Thèse de PhD, Université Nancy 2, novembre 2002.

[LS02] Ledang (H.) et Souquières (J.). – Integration of UML and B Specification Techniques: Systematic Transformation from OCL Expressions into B. *In : Asia Pacific Software Engineering Conference APSEC*. – IEEE Computer Society, 2002.

[LSC03] Ledang (H.), Souquières (J.) et Charles (S.). – ArgoUML+B: Un outil de transformation systématique de spécification UML vers B. *Proceeding of AFADL'2003*, Rennes, France, January 15-17 2003.

[Mar97] Mariano (G.). – *Évaluation de logiciels critiques développés par la méthode B, Une approche quantitative*. – Thèse de PhD, Université de Valenciennes et du Hainaut-Cambrésis, décembre 1997.

[Mey01] Meyer (E.). – *Développements formels par objets: utilisation conjointe de B et UML*. – Thèse de PhD, Université Nancy 2, mars 2001.

[OMG03] OMG. – *Unified Modeling Language*. – OMG http://www.omg.org/docs/formal/03-03-01.pdf, Version 1.5 March 2003.

[Sch01] Schneider (S.). – *The B Method: An Introduction*. – PALGRAVE, ISBN 0-333-79284-X, 2001.

[SS99] Sekerinski (E.) et Sere (K.). – *Program Development by Refinement*. – Springer, 1999.

[Ste] Steria. – *Obligations de preuve: Manuel de référence*. – Steria - Technologies de l'information. version 3.0.

[WK99] Warmer (J.) et Kleppe (A.). – *The Object Constraint Language: Precise Modeling with UML*. – Addison-Wesley, ISBN 0-201-37940-6, 1999.

ANNEXE

MACHINE *Types*

SETS

$OBJECTS$;
$LIGHT\_STATUS = \{light\_red, light\_yellow, light\_green\}$;
$BARRIER\_STATUS = \{barrier\_open, barrier\_close\}$

CONSTANTS

$CONTROL, BARRIER, LIGHT$

PROPERTIES

$CONTROL \subseteq OBJECTS \wedge$
$BARRIER \subseteq OBJECTS \wedge$
$LIGHT \subseteq OBJECTS$

END

IMPLEMENTATION *System_imp*

REFINES *System*

SEES *Types*

IMPORTS *Basic*

OPERATIONS

control_close_sys(*co*) =
  var *li*,*ba* **in**
    li ⟵ *lightOfControl*(*co*);
    ba ⟵ *barrierOfControl*(*co*);
    light_lightYellow(*li*);
    light_lightRed(*li*);
    barrier_barrierClose(*ba*)
  end
END

MACHINE *Basic*

SEES *Types*

VARIABLES
control,*light*,*light_status*,*barrier*,*barrier_status*,
controlLight,*controlBarrier*

INVARIANT
$control \subseteq CONTROL \wedge$
$light \subseteq LIGHT \wedge$
$light\_status \in light \longrightarrow LIGHT\_STATUS \wedge$
$barrier \subseteq BARRIER \wedge$
$barrier\_status \in barrier \longrightarrow BARRIER\_STATUS \wedge$
$controlLight \in control \leftrightarrow light \wedge$
$controlBarrier \in control \leftrightarrow barrier$

INITIALISATION
$control := \varnothing \parallel light := \varnothing \parallel$
$barrier := \varnothing \parallel light\_status := \varnothing \parallel$
$barrier\_status := \varnothing \parallel controlLight := \varnothing \parallel$
$controlBarrier := \varnothing$

OPERATIONS
li ⟵ *lightOfControl*(*co*) =
pre
  $co \in control$
then
  $li := controlLight(co)$
end;

ba ⟵ *barrierOfControl*(*co*) =
pre
  $co \in control$
then
  $ba := controlbarrier(co)$
end;

co ⟵ *controlOfLight*(*li*) =
pre
  $li \in light$
then
  $co := controlLight^{-1}(li)$
end;

$co \longleftarrow controlOfBarrier(ba) =$
pre
$ba \in barrier$
then
$co := controlBarrier^{-1}(ba)$
end;

$light\_lightYellow(li) =$
pre
$li \in light \wedge$
$light\_status(li) = light\_green$
then
$light\_status(li) := light\_yellow$
end;

$light\_lightRed(li) =$
pre
$li \in light \wedge light\_status(li) = light\_yellow$
then
$light\_status(li) := light\_red$
end;

$barrier\_barrierClose(ba) =$
pre
$ba \in barrier \wedge barrier\_status(ba) = barrier\_open$
then
$barrier\_status(ba) := barrier\_close$
end;
$control\_close(co) =$
pre
$co \in control \wedge$
$light\_status(controlLight(co)) = light\_green \wedge$
then
$light\_status(controlLight(co)) := light\_red \parallel$
$barrier\_status(controlBarrier(co)) := barrier\_close$
end
END

MACHINE *System*
...
OPERATIONS
$control\_close\_sys(co) =$
pre
$co \in control \wedge$
$light\_status(controlLight(co)) = light\_green \wedge$
then
$light\_status(controlLight(co)) := light\_red \parallel$
$barrier\_status(controlBarrier(co)) := barrier\_close$
end
END