Génération automatique de transactions de base de données relationnelle à partir de définitions d'attributs EB^3

Frédéric Gervais[1,2], Panawé Batanado[2], Marc Frappier[2] et Régine Laleau[3]

[1]CEDRIC, CNAM-IIE,
18 Allée Jean Rostand, 91025 Evry, France
frederic.gervais@usherbrooke.ca

[2]GRIL, Université de Sherbrooke,
2500, Boulevard de l'Université
Sherbrooke (Québec) J1K 2R1, Canada
{panawe.batanado,marc.frappier}@usherbrooke.ca

[3]LACL, Université Paris 12,
IUT Fontainebleau, Département Informatique,
Route Forestière Hurtault, 77300 Fontainebleau, France
laleau@univ-paris12.fr

Résumé EB^3 est un langage formel créé pour la spécification des systèmes d'information. Les attributs des types d'entité et des associations du système sont évalués en EB^3 grâce à des fonctions récursives définies sur les traces valides du système. Dans cet article, nous montrons comment générer automatiquement des programmes Java qui exécutent des transactions de base de données relationnelle qui correspondent aux définitions d'attributs EB^3. Dans un premier temps, nous introduisons les principes de la synthèse de transactions. Ensuite, nous présentons l'outil de génération des programmes et nous discutons des choix d'implantation qui ont été adoptés lors de son développement en Java. Un exemple est utilisé pour illustrer les différents concepts définis dans le papier.

Mots-clé Système d'information, EB^3, trace, filtrage, énoncé SQL, transaction, Java

1 Introduction

Nos intérêts portent sur la spécification formelle des systèmes d'information (SI) [Ger04]. De notre point de vue, un SI est un système logiciel qui permet à une organisation de rassembler et de manipuler des données importantes. Un SI comprend notamment des applications logicielles capables d'interroger et de mettre à jour une base de données (BD), de communiquer de manière adaptée les résultats des requêtes et de permettre aux administrateurs de contrôler et de modifier le système. L'utilisation de méthodes formelles pour développer des SI [FSD03, Mam02, Ngu98] est justifiée par la sûreté et/ou par la valeur marchande des données manipulées par des organismes comme des banques, des compagnies d'assurance, ou bien des industries de haute technologie. Actuellement, le paradigme le plus couramment utilisé pour spécifier des SI est le paradigme des transitions d'états. Ainsi, les contraintes d'intégrité du système sont décrites par des propriétés d'invariance qui doivent être préservées à chaque exécution des actions. Par exemple, les approches utilisant des transitions d'états pour spécifier des SI comprennent notamment RoZ [DLCP00], OMT-B [MS99] et UML-B [LM00].

Le langage EB^3 [FSD03] est un langage formel qui a été défini dans le but de spécifier des SI. EB^3 fournit une approche de spécification basée sur les traces d'événements, contrairement aux langages utilisés dans le paradigme des transitions d'états [FFL05]. En EB^3, les séquences possibles des événements du SI sont décrites grâce à une algèbre de processus. Une autre partie des spécifications EB^3 concerne les données du SI, décrites par l'intermédiaire des attributs des types d'entité et associations.

Le projet APIS [FFLR02] a pour objectif de générer des SI directement à partir de spécifications EB^3. L'idée est de libérer le programmeur des détails d'implantation pour qu'il se consacre aux phases d'analyse et de spécification. Plutôt que d'utiliser des techniques de raffinement comme dans les approches basées sur les transitions d'états [Edm95, Mam02], les outils d'APIS sont basés sur des techniques de synthèse automatique. Ainsi, le SI est interprété à partir des expressions de processus EB^3, grâce à un interpréteur appelé EB^{3}PAI [FF02]. Une autre composante d'APIS permet de générer une interface Web [Ter05], à travers laquelle un utilisateur peut interagir avec le SI. Pour mettre à jour ou pour interroger le système, l'utilisateur génère un événement à travers l'interface Web. Cet événement est ensuite analysé par EB^{3}PAI. S'il est valide, alors il est exécuté ; sinon un message d'erreur est envoyé à l'utilisateur.

Dans le cadre du projet APIS, nous avons développé un nouvel outil, appelé EB^{3}TG, qui permet de générer automatiquement des programmes Java qui exécutent des transactions de BD relationnelle qui correspondent à la spécification des attributs du SI en EB^3. Les transactions obtenues peuvent ainsi être utilisées en combinaison avec l'outil EB^{3}PAI afin de mettre à jour ou d'interroger la BD lorsque les événements correspondants sont considérés comme valides par l'interprétation des expressions de processus EB^3. L'outil EB^{3}TG permet également de créer et d'initialiser automatiquement une BD correspondant à une spécification EB^3. Dans cet article, nous présentons plutôt les algorithmes de génération des transactions, qui constituent la contribution originale de l'outil. Nous décrivons aussi les fonctionnalités d'EB^{3}TG.

La section 2 introduit tout d'abord le langage EB^3. En particulier, nous présentons les définitions d'attributs EB^3 qui permettent d'évaluer la valeur des attributs du SI. Afin d'illustrer les différents concepts définis dans le papier, un exemple de système de gestion de bibliothèque est présenté dans la section 2.1. Dans la section 3, les algorithmes de génération des transactions de BD relationnelle sont présentés de manière théorique, alors que la section 4 s'attarde sur la description de l'outil EB^{3}TG et sur son application. Enfin, la section 5 conclut l'article. On suppose dans la suite que les concepts de base du modèle relationnel sont des prérequis ; pour plus de détails, ce modèle est présenté, par exemple, dans [EN04].

2 EB^3

EB^3 est un langage formel inspiré de la méthode JSD [Jac83] et du concept de boîte noire de Cleanroom [PTLP99]. Une *boîte noire* est une fonction qui, à une séquence d'événements en entrée, associe une sortie. En EB^3, une algèbre de processus inspirée de CSP [Hoa85], CCS [Mil89], et LOTOS [BB87], est utilisée pour spécifier les entités du SI comme des boîtes noires. Nous utilisons les termes *type d'entité* et *entité* à la place de classe et objet, respectivement. EB^3 fournit à la fois une notation formelle et un processus de développement afin de décrire une spécification précise et complète du comportement des entrées-sorties d'un SI. Une spécification EB^3 est composée de quatre parties.

1. Un diagramme de classes décrit les types d'entité et les associations du système, ainsi que leurs actions et attributs respectifs. Ce diagramme, basé sur les concepts du modèle entité-relation (ER) [EN04], utilise les notations d'UML [Obj05].
2. Une expression de processus, appelée `main`, caractérise les traces valides d'événements à l'entrée du système. Une *trace* est une séquence d'événements.
3. Des règles d'entrée-sortie associent une sortie à chaque événement valide à l'entrée du système.

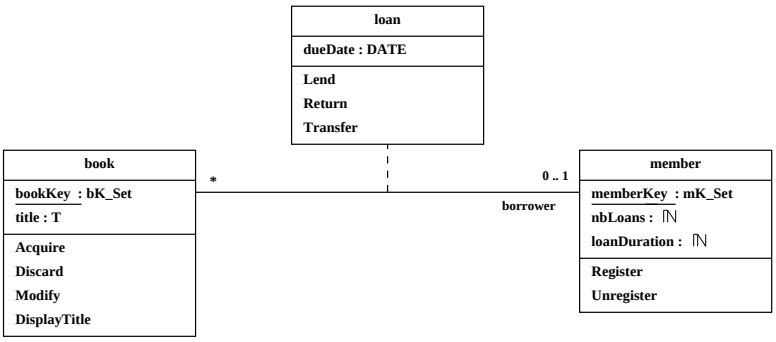

FIG. 1 – Spécification EB^3 : diagramme de classes de la bibliothèque

4. Des fonctions récursives, définies sur les traces valides de `main`, déterminent les valeurs d'attributs des types d'entité et/ou des associations.

La sémantique dénotationelle d'une spécification EB^3 est donnée par une relation R définie sur $\mathcal{T}(\texttt{main})\times O$, où $\mathcal{T}(\texttt{main})$ dénote les traces acceptées par `main` et O est l'ensemble des événements en sortie. La trace courante du système, dénotée par `trace`, est la liste finie des événements en entrée acceptés et exécutés par le système. L'expression $t :: \sigma$ représente l'insertion de l'événement σ en queue de la trace t, alors que `[]` dénote la trace vide. Le comportement de `main` est défini par :

`trace` := `[]` ;
faire toujours
  recevoir l'événement σ ;
  si `main` peut accepter `trace` :: σ **alors**
    `trace` := `trace` :: σ ;
    envoyer un événement de sortie o tel que $(\texttt{trace}, o) \in R$;
  sinon
    envoyer un message d'erreur ;

La syntaxe complète et la sémantique d'EB^3 se trouvent dans [FSD03].

2.1 Exemple

Pour illustrer les idées développées dans l'article, nous utilisons un exemple de gestion de bibliothèque. Le système doit gérer des emprunts de livres par des membres. Un livre est acheté par la bibliothèque. Il ne peut être retiré des références que s'il n'est plus emprunté. Le titre d'un livre peut être modifié ou affiché. Un membre doit s'inscrire à la bibliothèque pour pouvoir emprunter un livre. Un membre peut également transférer son prêt à un autre membre de la bibliothèque. Enfin, un membre ne peut quitter la bibliothèque que s'il a rendu tous ses livres. La figure 1 représente le diagramme de classes de cet exemple. Les signatures des actions EB^3 sont les suivantes :

```
Acquire(bId:bK_Set,bTitle:T):void
Discard(bId:bK_Set):void
Modify(bId:bK_Set,nTitle:T):void
DisplayTitle(bId:bK_Set):(title:T)
Register(mId:mK_Set,dur:NAT):void
Unregister(mId:mK_Set):void
Lend(bId:bk_Set,mId:mK_Set):void
Return(bId:bK_Set):void
Transfer(bId:bK_Set,mId:mK_Set):void
```

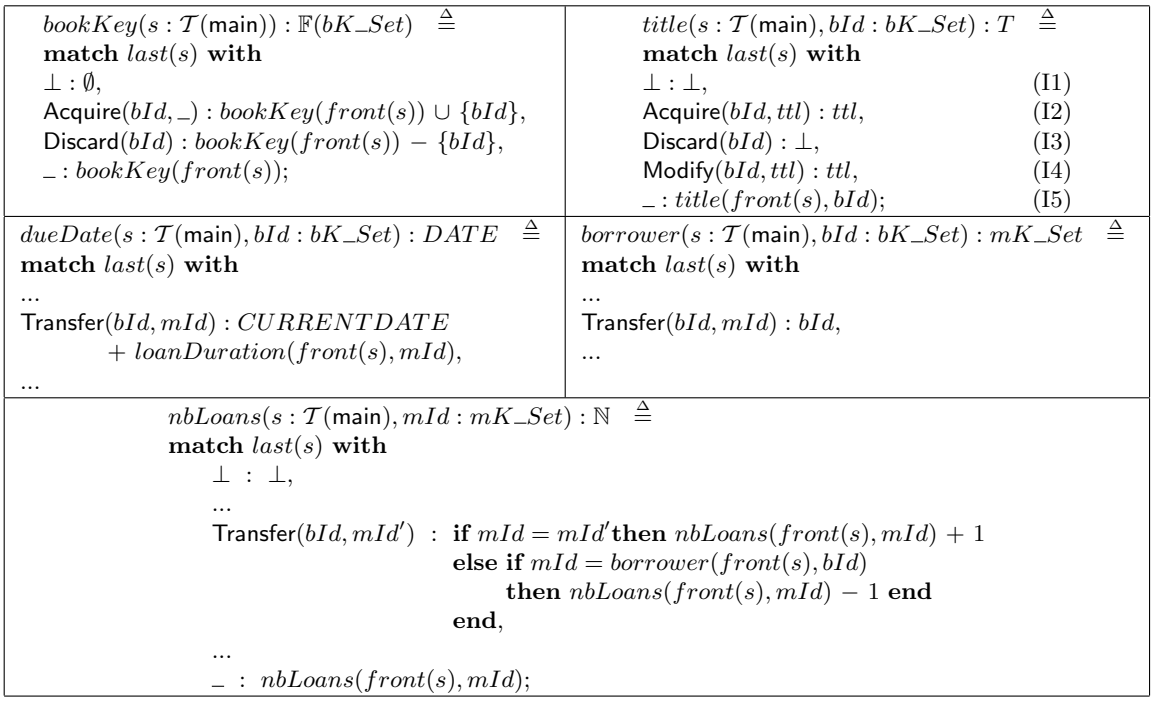

FIG. 2 – Exemples de définitions d'attributs EB^3

Le type `void` est utilisé pour dénoter une action sans entrée et/ou sortie. Toute action renvoie implicitement un paramètre de sortie pour indiquer si l'action est valide (‘`ok`’) ou pas (‘`error`’). Les expressions de processus et les règles entrée-sortie de cet exemple se trouvent dans [GFL04].

2.2 Définitions d'attributs

En EB^3, une *définition d'attribut* est une fonction récursive définie sur les traces valides du système, autrement dit, les traces d'événements acceptées par l'expression de processus `main`. La fonction est totale et est donnée dans un style fonctionnel, comme en CAML [CM98], avec un *filtrage* sur le dernier événement de la trace. La fonction retourne les valeurs d'attributs qui sont valides pour l'état dans lequel le système se trouve après avoir exécuté les événements de la trace. La syntaxe complète du langage de définition des attributs est présentée dans [GFLB05].

On différencie les attributs selon qu'ils sont attribut clé ou non. Une définition de clé renvoie l'ensemble des clés existantes, alors qu'une définition d'attribut non clé retourne la valeur d'attribut pour la clé indiquée en paramètre. Par exemple, la clé du type d'entité `book` est définie par la fonction *bookKey* dans la figure 2. *bookKey* a un unique paramètre d'entrée, $s \in \mathcal{T}(\mathsf{main})$, c'est-à-dire une trace valide du système, et elle retourne l'ensemble des clés du type d'entité `book`. La notation $\mathbb{F}(bK\_Set)$ dénote l'ensemble des sous-ensembles finis de bK_Set. L'attribut non clé *title* est défini dans la figure 2. Notons que, par souci de concision, les définitions des attributs *dueDate*, *borrower* et *nbLoans* dans la figure 2 ont été tronquées ; seuls les éléments nécessaires à l'illustration des contributions de cet article ont été conservés.

Les expressions de la forme *input* : *expr*, comme Acquire$(bId, ttl) : ttl$ dans *title*, sont appelées des *clauses d'entrée*. Quand une définition d'attribut est exécutée, alors toutes les clauses d'entrée de la définition d'attribut sont évaluées comme des filtres sur le dernier événement de la trace ; la première clause satisfaite est celle qui est exécutée. Par conséquent, l'ordre des clauses a une importance. Le filtrage s'effectue toujours sur le dernier événement de la trace $s \in \mathcal{T}(\mathsf{main})$. Si une des expressions *input* correspond avec $last(s)$, alors l'expression *expr* associée est exécutée pour évaluer la valeur d'attribut ; sinon, la fonction est appelée récursivement sur la trace s amputée du dernier élément, ce qui est dénoté par $front(s)$. Ce cas correspond

à la dernière clause avec le symbole '_'. Les définitions d'attributs EB^3 incluent toujours $\perp$, de manière à rendre les fonctions totales. En particulier, $last([\,]) = \perp$. Tout appel de fonction de clé $eKey$ ou de fonction d'attribut non clé b dans une clause d'entrée est toujours de la forme $eKey(front(s))$ ou $b(front(s), ...)$. Par exemple, les traces suivantes sont évaluées comme suit pour l'attribut *title* :

$$title([\,], b_1) \overset{(I1)}{=} \perp$$
$$title([\mathsf{Register}(m_1, d_1)], b_1) \overset{(I5)}{=} title([\,], b_1) \overset{(I1)}{=} \perp$$
$$title([\mathsf{Acquire}(b_1, t_1)], b_1) \overset{(I2)}{=} t_1$$
$$title([\mathsf{Acquire}(b_1, t_1), \mathsf{Register}(m_1, d_1), \mathsf{Modify}(b_1, t_2)], b_1) \overset{(I4)}{=} t_2$$

Dans le premier cas, la valeur est obtenue grâce à la clause (I1), puisque $last([\,]) = \perp$. Dans le second cas, nous appliquons d'abord (I5), puisqu'aucune clause d'entrée ne s'identifie avec Register, et ensuite (I1). Dans les derniers cas, les valeurs sont obtenues directement en appliquant (I2) et (I4), respectivement.

L'expression *expr* dans la clause d'entrée de la forme *input* : *expr* est un terme composé de constantes, de variables et d'appels récursifs d'attributs. Les expressions de la forme **if then else end** sont utilisées quand le filtrage des paramètres de l'action n'est pas suffisant pour déterminer la valeur de l'attribut. Une expression *expr* sans condition est appelée un *terme fonctionnel*, alors qu'une expression qui contient des **if then else end** est un *terme conditionnel*.

3 Analyse des définitions d'attributs

Cette section donne un résumé de l'algorithme de synthèse de transactions à partir des définitions d'attributs. Dans la section 3.1, nous introduisons l'algorithme principal, puis en section 3.2, nous présentons l'analyse des clauses d'entrée des définitions d'attributs. Enfin, nous montrons en section 3.3 quelle est la forme générale des transactions obtenues.

3.1 Algorithme principal

Les définitions d'attributs EB^3 décrivent la dynamique des données d'un SI. Nous avons choisi de les implanter par une BD relationnelle. Cette BD nous permet de stocker la valeur courante de chaque attribut, c'est-à-dire la valeur pour la trace courante durant l'exécution du SI. Ce choix d'implantation nous permet d'éviter de stocker toute la trace, ce qui serait difficilement réalisable, étant donnée la taille croissante de cette trace. Notre algorithme génère une transaction de BD relationnelle pour chaque action EB^3. Ainsi, à chaque fois qu'un événement est considéré comme valide, la transaction correspondante met à jour tous les attributs affectés par l'action.

Pour générer un programme qui exécute une transaction de BD relationnelle associée à l'action a, nous devons analyser toutes les clauses d'entrée des définitions d'attributs afin de déterminer :

- quels sont les attributs affectés par l'exécution de a,
- quels sont les effets de a sur ces attributs.

L'algorithme général est donc le suivant :

(0) traduire le diagramme de classes en un schéma de BD relationnelle
(1) pour chaque action a de la spécification EB^3
(2) analyser les clauses d'entrée des définitions d'attributs
(3) déterminer les attributs $Att(a)$ affectés par a
(4) déterminer les tables $T(a)$ affectées par a
(5) pour chaque table t dans $T(a)$
(6) déterminer les valeurs de clé à supprimer
(7) déterminer les valeurs de clé à ajouter et/ou à mettre à jour
(8) définir la transaction pour a

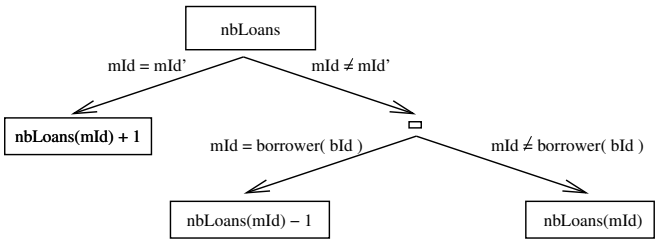

FIG. 3 – Arbre de décision de Transfer pour l'attribut *nbLoans*

3.2 Analyse des clauses d'entrée

L'analyse des clauses d'entrée est détaillée dans [GFLB05, GFL05]. Le but de cette analyse est de déterminer, pour chaque action a :

1. l'ensemble des attributs $Att(a)$ affectés par a (ligne (3) de l'algorithme),
2. l'ensemble des tables $T(a)$ affectées par a (ligne (4)),
3. les valeurs de clé concernées par l'exécution de a (lignes (6) et (7)).

Pour exécuter une définition d'attribut, un filtrage est effectué sur le dernier événement de la trace (voir Sect. 2.2). Par conséquent, un attribut b est affecté par l'action a s'il existe au moins une clause de la forme $a(\overrightarrow{p})$: *expr* dans la définition de b. On peut ainsi calculer l'ensemble $Att(a)$. Par convention, chaque attribut b est préfixé par son type d'entité ou par son association dans les définitions d'attributs EB[3]. Soit *table* la fonction qui retourne la table d'un attribut, alors $T(a)$ est défini par :

$$T(a) = \{table(b) \mid b \in Att(a)\}$$

Lors de l'analyse du filtrage, si une clause d'entrée correspond à l'événement filtré, alors chaque variable libre de la clause est liée aux paramètres actuels de l'événement. Par exemple, supposons que Acquire$(numero, titre)$ soit un nouvel événement valide du SI. Dans ce cas, pour évaluer la fonction *title* (voir Fig. 2), la clause d'entrée Acquire$(bId, _)$ s'identifie avec le nouvel événement et la variable libre bId de la définition est liée à la valeur *numero*. Par conséquent, la valeur de la clé de *title* est entièrement déterminée. Cependant, l'analyse du filtrage n'est pas toujours suffisante pour déterminer toutes les valeurs de clé affectées par une action a. Dans ce cas, il faut également analyser les conditions des termes conditionnels, autrement dit, les expressions de la forme **if then else end** des définitions d'attributs. Dans la section 3.2.1, nous montrons comment traiter ces cas. Puis, dans la section 3.2.2, nous discutons des tables temporaires que nous utilisons pour stocker les valeurs de clé concernées.

3.2.1 Arbre de décision

Pour analyser les conditions dans les prédicats **if** des termes conditionnels, nous utilisons des arbres binaires appelés *arbres de décision* ; leur construction et leur analyse sont détaillées dans [GFL04]. Par exemple, le filtrage de l'action Transfer dans l'attribut *nbLoans* ne permet pas de déterminer la valeur de la clé mId (voir Fig. 2). En effet, la variable liée par le filtrage est mId', alors que la valeur de la clé à déterminer est représentée par mId (ie, le paramètre formel de l'en-tête de l'attribut *nbLoans*). Nous construisons alors un arbre de décision pour analyser les expressions de la forme **if then else end**. Les feuilles de l'arbre représentent les termes fonctionnels des parties **then**, alors que les arcs sont étiquetés par les conditions des prédicats **if**. Ainsi, l'arbre de décision construit pour l'exemple de Transfer est donné par la figure 3. Les prédicats du terme conditionnel de la clause Transfer déterminent deux valeurs de clé pour mId : mId' et $borrower(bId)$. La première feuille correspond à la condition $mId = mId'$, et la deuxième, au prédicat $mId \neq mId' \wedge mId = borrower(bId)$. La dernière feuille est toujours un appel récursif de la définition d'attribut.

3.2.2 Tables temporaires

Quand les valeurs de clé sont déterminées à partir de prédicats impliquant des fonctions ou bien des appels récursifs d'attributs, alors un énoncé SQL de type **SELECT** est généré afin d'extraire ces valeurs de la BD. Les requêtes ainsi définies nous permettent de stocker les valeurs concernées dans des tables temporaires durant une transaction et, par conséquent, d'éviter tout problème d'incohérence au cas où la BD serait mise à jour entre-temps. Par exemple, pour retrouver l'ensemble des livres empruntés par le membre mId, une table temporaire est définie de la manière suivante :

```
CREATE TEMPORARY TABLE
    TAB (bookKey bK_Set PRIMARY KEY)
INSERT INTO TAB
    SELECT book.bookKey
    FROM book
    WHERE book.borrower = #mId;
```

Le contenu de la table temporaire est évalué une seule fois, au début de la transaction. La génération d'énoncés **SELECT** qui correspondent aux valeurs de clé satisfaisant les conditions **if** dépend essentiellement de la forme des prédicats. Nous avons identifié les principaux patterns de prédicats et leur énoncé **SELECT** correspondant [GFLB05]. Nous avons également défini dans ce rapport des mécanismes de composition entre patterns.

3.3 Génération des transactions

L'ordre des énoncés SQL dans une transaction générée par notre méthode est le suivant :
- liste des définitions de tables temporaires créées lors de l'analyse des clauses d'entrée,
- liste des énoncés SQL correspondant aux suppressions de clé,
- liste des énoncés SQL correspondant aux insertions et/ou mises à jour.

Les valeurs de clé à supprimer sont celles qui apparaissent dans une expression *expr* de la forme $eKey(front(s))-\{k\}$ dans les clauses d'une définition de clé. Un énoncé **DELETE** est alors généré au début de la transaction. Nous utilisons un pseudo-code de haut niveau pour décrire les transactions générées ; ce pseudo-code est traduit en Java dans EB^{3}TG. Par exemple, la transaction générée pour l'action Discard est la suivante :

```
TRANSACTION Discard(mId : bK_Set)
    DELETE FROM book
    WHERE book.bookKey = #bId;
    COMMIT;
```

Rappelons que cette transaction ne peut être exécutée que si l'action Discard est considérée comme valide par l'interpréteur EB^{3}PAI.

Quand une action implique des mises à jour et/ou des insertions, alors la transaction devient plus complexe. En effet, la spécification EB^3 ne permet pas toujours de distinguer les insertions des mises à jour. Par exemple, si une action apparaît dans les clauses d'entrée d'une ou plusieurs définitions d'attributs non clé, alors plusieurs cas sont possibles :
- si la valeur de clé déterminée lors du filtrage et/ou de l'analyse d'un terme conditionnel n'apparaît pas dans la définition de clé correspondante, alors il s'agit nécessairement d'une mise à jour, implantée par un énoncé de type **UPDATE** ;
- mais si l'action apparaît dans la définition de clé, et que la clause correspondante contient un terme de la forme $eKey(front(s)) \cup \{k\}$, alors il peut s'agir soit d'une insertion, soit d'une mise à jour. En effet, $eKey(front(s))$ et $\{k\}$ ne sont pas nécessairement disjoints.

Dans ce dernier cas, la solution adoptée pour rester cohérent vis-à-vis de la spécification EB^3 consiste à exécuter d'abord une mise à jour, et si elle a échoué, alors une insertion est exécutée. Par exemple, la transaction générée pour Acquire est de la forme suivante :

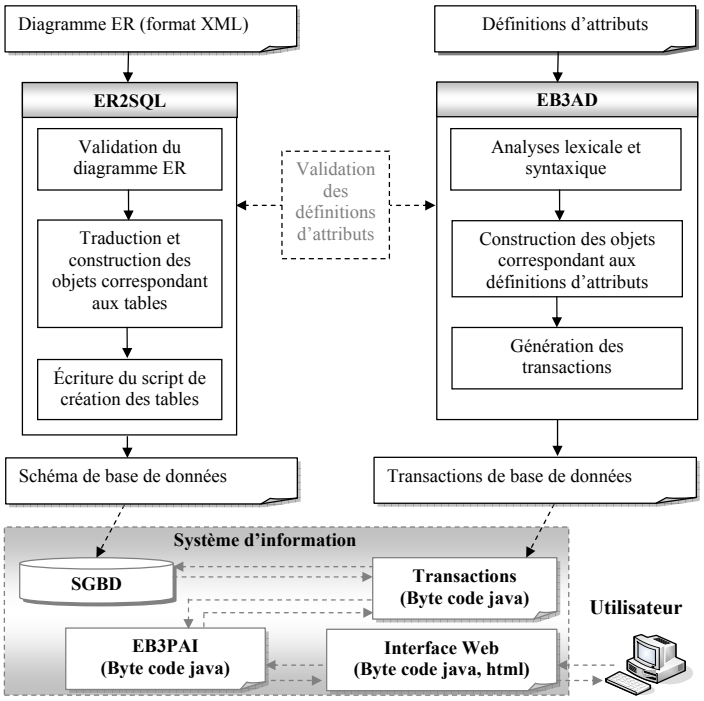

FIG. 4 – Architecture de l'outil EB^{3}TG

```
TRANSACTION Acquire(bId : bK_Set,bTitle : T)
    /* update */
    UPDATE book SET book.title = #bTitle
    WHERE book.bookKey = #bId;
    /* test */
    IF SQL%NotFound
    THEN
        /* insertion */
        INSERT INTO book(bookKey,title)
        VALUES (#bId,#bTitle);
    END;
    COMMIT;
```

La variable "SQL%NotFound" est un code renvoyé par le système de gestion de BD indiquant si la mise à jour a effectivement été réalisée.

4 Implantation : EB^{3}TG

Nous avons implanté un outil appelé EB^{3}TG (EB^3 *Transaction Generator*) qui supporte l'algorithme de génération de transactions décrit dans la section 3. Nous décrivons dans la section 4.1 les principales fonctionnalités de cet outil. Dans la section 4.2, nous présentons plus en détail le module EB3AD, qui concerne la génération des transactions. Afin d'illustrer l'application de l'outil, nous donnons dans la section 4.3 les exemples d'un schéma de BD relationnelle et d'une transaction générés par l'outil EB^{3}TG.

4.1 Architecture de EB^{3}TG

La figure 4 illustre l'architecture de l'outil EB^{3}TG. Il est composé de deux principaux modules, ER2SQL et EB3AD.

```
<entity name="member" type="strong" label="Les membres de la bibliothèque">
 <attribute name="memberKey" type="numeric" size="5" key="true"/>
 <attribute name="nbLoans" type="numeric" size="5" null="false"/>
 <attribute name="loanDuration" type="numeric" size="3" null="false"/>
</entity>
```

FIG. 5 – Exemple de balise utilisée dans la définition d'un diagramme ER

ER2SQL. À partir d'une représentation en XML du diagramme ER, ce module génère un schéma de BD et fournit une représentation objet de ce schéma. Cette représentation objet est utilisée lors de la validation des définitions d'attributs.

Les concepts du modèle ER qu'on considère sont ceux de [EN04]. La DTD (*Document Type Definition*) utilisée pour les représenter est détaillée dans [Bat05]. Par exemple, la balise qui permet de représenter le type d'entité *member* de la figure 1 est illustrée dans la figure 5. L'attribut clé de ce type d'entité est *memberKey* ; son attribut *loanDuration* par exemple n'accepte pas de valeur nulle. Une première analyse syntaxique, effectuée lors de la construction de l'arbre XML, permet de garantir la validité du document XML par rapport à la DTD du modèle ER. Une seconde analyse est effectuée afin de s'assurer de la validité des différents noms utilisés dans le document.

L'algorithme complet de traduction d'un diagramme ER en un schéma de BD relationnelle est présenté dans [GFLB05]. Les énoncés SQL de création des tables et des contraintes d'intégrité sont générés en fonction du système de gestion de BD choisi par l'utilisateur. Les principales classes de l'outil qui implantent la traduction ER vers SQL sont présentées dans la figure 6. L'analyse de l'arbre XML du diagramme ER et la traduction ER vers SQL sont effectuées dans la classe *ER2SQL*. Une instance de cette classe est passée à l'analyseur sémantique des définitions d'attributs (*TreeWalker*) qui l'utilise pour effectuer la validation de ces définitions (voir module EB3AD).

EB3AD. À partir d'une description en ASCII des définitions d'attributs EB^3, ce module assure la synthèse des transactions de BD relationnelle. Les phases classiques d'analyses lexicale et syntaxique aboutissent à la construction de l'arbre syntaxique abstrait. Le parcours de cet arbre permet d'effectuer une analyse sémantique des définitions d'attributs et de générer la représentation objet correspondante.

L'analyse sémantique garantit, entre autres, la cohérence entre les transactions et la BD générée. Nous utilisons la représentation objet de la BD générée par le module ER2SQL pour effectuer les validations des définitions d'attributs. Par exemple, tout attribut apparaissant dans les fonctions récursives doit être au préalable défini dans le diagramme ER. Une fois que les validations des définitions d'attributs sont faites, les transactions BD sont générées. La représentation objet et la génération des transactions sont détaillées dans la section 4.2.

Implantation. Notre outil a été implanté en Java. Le code source comprend 50 classes et 625 méthodes, pour un total de 20000 lignes de code. Nous avons choisi une représentation XML (*Extensible Markup Language*) pour les diagrammes ER. Ce choix permettra à l'avenir d'interfacer EB^{3}TG avec un outil graphique de saisie de diagrammes ER, comme Rational Rose [Qua98] ou ArgoUML [Arg05]. Le langage de description des transactions générées utilise la norme JDBC (*Java Database Connectivity*), qui offre une interface de programmation permettant de développer des applications en Java capables de se connecter à n'importe quelle BD. Enfin, nous avons choisi ANTLR [Par05] pour les analyses syntaxique et lexicale du langage de définition des attributs, car il est possible de générer automatiquement ces analyseurs en Java à partir de la grammaire du langage.

4.2 Module EB3AD

Dans cette section, nous détaillons le module EB3AD, dont la fonctionnalité de synthèse de transactions constitue la contribution originale de l'outil EB^{3}TG.

4.2.1 Représentation objet des définitions d'attribut

La syntaxe complète des définitions d'attributs est présentée dans [GFLB05]. Après les vérifications lexicale et syntaxique réalisées par les analyseurs générés par l'outil ANTLR, on obtient un arbre syntaxique abstrait correspondant aux définitions d'attributs. Cet arbre est ensuite analysé dans la classe *TreeWalker* afin de générer la représentation objet correspondante.

Le diagramme de classes de la figure 6 montre les principales classes représentant les différents éléments des définitions d'attributs. En général, chaque expression non terminale de la grammaire EB^3 est représentée par une classe. Une spécification d'attributs est composée de définitions de clés, d'attributs non clé et de définitions de rôles. La classe $EB3AttributeDefinitions$ représente chacune de ces définitions par une liste. Cette classe implante l'algorithme général vu à la section 3.1.

Les classes $KeyDefinition$, $NonKeyDefinition$ et *Role* permettent de représenter respectivement les trois types de définitions. Chaque définition de clé et d'attribut non clé a une clause d'entrée d'initialisation et une clause correspondant à chaque action qui l'affecte. Les clauses d'initialisation définissent les valeurs des attributs à l'initialisation du système, alors que les autres clauses d'entrée définissent l'effet de chaque action sur les attributs. Elles déterminent de ce fait les mises à jour à effectuer. Les classes $InputClauseKeyInit$, $InputClauseAttributeInit$, $InputClauseKeyUpdate$, $InputClauseAttributeUpdate$ permettent de représenter respectivement, les clauses d'initialisation et de mise à jour des clés et des attributs non clé.

Les termes qui permettent de calculer les valeurs des attributs sont soit des termes conditionnels, soit des termes fonctionnels. Ces termes sont gérés respectivement par les classes $ConditionalTerm$ et $FunctionalTerm$. Les prédicats logiques des termes conditionnels sont représentés par la classe *Predicate*. Les énoncés **SELECT**, issus de l'analyse de l'arbre de décision ou des appels récursifs d'attributs dans les termes fonctionnels (voir section 3.2), sont modélisés par la classe *SelectStatement*.

4.2.2 Génération des transactions

Pour une spécification d'attributs, EB^{3}TG génère une classe contenant les méthodes correspondant aux actions EB^3. Elle contient également une méthode permettant de se connecter à la BD choisie, une méthode permettant de créer les tables et une autre qui permet de les initialiser.

Le niveau d'isolation des transactions détermine les conflits admissibles entre elles. Nous avons choisi le plus haut niveau d'intégrité (*TRANSACTION_SERIALIZABLE*) pour l'accès en concurrence à une BD ; ce niveau permet aux transactions de s'exécuter en concurrence comme si elles s'exécutaient en série, c'est-à-dire les unes après les autres. Le mécanisme classique de gestion des transactions est utilisé pour chaque méthode correspondant à une action EB^3. Les mises à jour sont effectuées les unes après les autres ; en cas d'erreur, toute la transaction est rejetée par la méthode *rollback*(), sinon elle est acceptée avec la méthode *commit*().

L'interface de programmation d'application JDBC offre deux classes permettant d'exécuter des requêtes SQL puis d'en extraire les résultats. Il s'agit des classes *PrepareStatement* et *Statement*. La première a l'avantage d'être plus rapide en exécution parce que les requêtes sont compilées une seule fois et stockées dans un objet. Les méthodes de la forme $setXXX$ (où XXX représente un type, *String* par exemple) permettent ensuite d'affecter les paramètres de la requête, puis de l'exécuter sans avoir à la recompiler. L'atout principal de la seconde classe est d'être implantée par la plupart des gestionnaires de BD (Oracle, DB2, MySQL, etc...). Nous l'avons adoptée dans ce projet. En effet, la méthode *createStatement*() de l'exemple de la figure 8 crée un objet de type *Statement*, dont les méthodes *executeUpdate*(*uneRequete*)

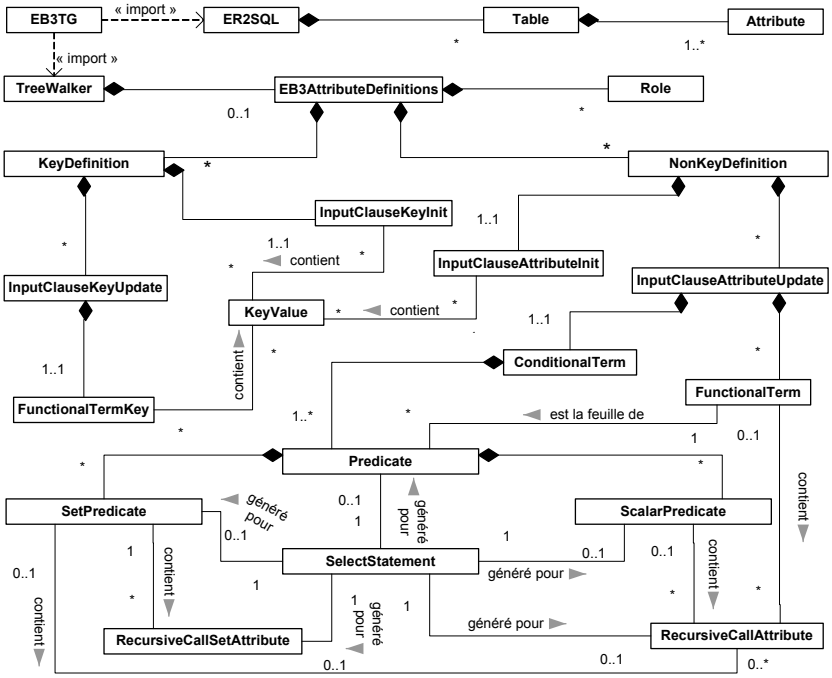

FIG. 6 – Diagramme de classes de EB^{3}TG

et *executeQuery*(*uneAutreRequete*) permettent d'exécuter respectivement des requêtes SQL de mise à jour et d'interrogation. Parmi les travaux futurs, nous prévoyons de développer une nouvelle version de l'outil utilisant la classe *PrepareStatement*. Ainsi, en regroupant les requêtes SQL dans une seule méthode qui se chargera de les compiler au démarrage de l'application, le temps d'exécution des transactions pourra être optimisé.

Nous utilisons les types de données Java équivalents aux types SQL pour stocker les variables temporaires, et la classe *ResultSet*, pour représenter la notion de table temporaire utilisée dans la section 3.2.2. Les objets de type *ResultSet* ne sont pas sensibles à des mises à jour ultérieures ; autrement dit, les mises à jour effectuées dans la BD au cours d'une transaction n'affectent pas les résultats des requêtes **SELECT** stockés dans ces objets. Un exemple de transaction utilisant des objets de type *ResultSet* est présenté à la section 4.3.2.

4.3 Exemples

Dans cette section, nous présentons les exemples d'un schéma de BD relationnelle et d'une transaction générés automatiquement par l'outil EB^{3}TG.

4.3.1 Schéma de la BD

Le premier exemple d'application de l'outil EB^{3}TG concerne la génération, par le module ER2SQL, du schéma de BD correspondant à l'exemple de la figure 1. Les tables générées pour le type d'entité *book* et pour la relation *loan* sont illustrées dans la figure 7. La syntaxe utilisée est celle d'Oracle. La table *member*, qui n'est pas indiquée dans la figure par souci de concision, est de la même forme que la table *book*. Son attribut clé est $memberKey$. L'attribut $bookKey$ est l'attribut clé des tables *book* et *loan*. Le type d'entité *book* a une participation partielle dans la relation *loan*, d'où la création d'une table pour représenter cette relation. Notons que des contraintes référentielles sont systématiquement définies par la clause **ALTER** pour éviter les problèmes de référencement mutuel entre tables. La vérification de ces contraintes est différée

```
CREATE TABLE book (
bookKey    numeric(5,2),
title      varchar(20),
CONSTRAINT PKbook PRIMARY KEY(bookKey)
);

CREATE TABLE loan (
bookKey    numeric(5,2),
borrower   numeric(5),
dueDate    date,
CONSTRAINT PKloan PRIMARY KEY(bookKey)
);

ALTER TABLE loan ADD CONSTRAINT FKMemberLoan FOREIGN KEY (borrower)
REFERENCES member (memberKey) INITIALLY DEFERRED;

ALTER TABLE loan ADD CONSTRAINT FKBookLoan FOREIGN KEY (bookKey)
REFERENCES book (bookKey) INITIALLY DEFERRED;
```

FIG. 7 – Exemple d'énoncés de création de tables

à la fin de chaque transaction, ce qui est indiqué, dans la syntaxe SQL d'Oracle, avec l'option **INITIALLY DEFERRED**.

4.3.2 Transaction

L'exemple concerne maintenant la génération de la transaction correspondant à l'action Transfer. La figure 2 contient les définitions de trois des attributs affectés par cette action : *dueDate*, *borrower* et *nbLoans*. La transaction générée par l'outil EB^{3}TG est présentée dans la figure 8.

L'action Transfer(bId, mId) a pour effet de transférer le prêt du livre bId au membre mId. La transaction correspondante doit donc mettre à jour la table des prêts, *loan*, pour y spécifier le nouvel emprunteur mId, ainsi que la nouvelle date de retour, qui est égale à la date courante augmentée de la durée normale d'un prêt pour le membre mId. Le nombre de prêts de mId doit être incrémenté de 1, alors que celui de l'emprunteur courant, dénoté par $borrower(front(s), bId)$, doit être décrémenté de 1.

Chaque condition dans les parties **if** des termes conditionnels détermine un ensemble de tuples, identifiés par leurs valeurs de clés, à mettre à jour. Chacun de ces ensembles est déterminé par un énoncé **SELECT** lors de l'analyse de l'arbre de décision. Pour chaque branche de l'arbre, une jointure permet d'obtenir en un seul énoncé **SELECT** les valeurs de clés des tuples ainsi que la nouvelle valeur de l'attribut à mettre à jour. Cette jointure s'exprime par une condition dans la clause **WHERE**. La figure 3 présente l'arbre de décision correspondant au terme conditionnel de la clause d'entrée de l'action Transfer dans la définition de l'attribut *nbLoans*. Les objets *rset0* et *rset1* (lignes 8 et 14 de la figure 8) de type *ResultSet* contiennent respectivement les valeurs temporaires pour la première et la deuxième branche de cet arbre. Il n'y a jamais d'énoncé **SELECT** généré pour la dernière branche de l'arbre, car elle représente les tuples où la valeur de l'attribut est inchangée par la réception de l'événement.

À chaque nœud de l'arbre, on exclut les clés déterminées par les prédicats des nœuds parents en utilisant l'expression **NOT IN**, comme le montre l'énoncé SQL de la ligne 18, qui correspond à la deuxième branche de l'arbre de décision. La variable *var0* à la ligne 27 stocke la valeur de l'attribut *loanDuration* durant la transaction. L'énoncé **SELECT** correspondant à l'appel récursif de cet attribut est celui de la ligne 25. Après la définition des variables temporaires, on procède aux mises à jour. Un énoncé de mise à jour est généré pour chaque branche de l'arbre de décision des termes conditionnels. Dans l'exemple de la figure 8, on distingue quatre énoncés de mise à jour, respectivement pour *borrower* (ligne 29), *dueDate* (ligne 32), et pour les deux premières branches de l'arbre de décision de *nbLoans* (lignes 35 et 41). Par souci de simplicité, nous utilisons toujours une boucle pour traiter toutes les valeurs satisfaisant une condition d'un terme conditionnel, car certaines conditions peuvent générer plusieurs valeurs de clés.

```
1     public static void Transfer(int bId,int mId){
2       try {
3         connection.createStatement().executeUpdate(
4             "CREATE TABLE eb3Tempmember ( "memberKey  numeric(5))");
5         connection.createStatement().executeUpdate(
6             "INSERT INTO  eb3Tempmember (memberKey) values("+mId+")");
7
8         ResultSet rset0 =connection.createStatement().
9           executeQuery("SELECT C.memberKey,A.nbLoans+1 "+
10          "FROM eb3Tempmember C,member A "+
11          "WHERE C.memberKey = "+mId+" "+
12          "AND A.memberKey = C.mId ");
13
14        ResultSet rset1 = connection.createStatement().
15          executeQuery("SELECT G.borrower,E.nbLoans-1 "+
16          "FROM loan G,member E "+
17          "WHERE G.bookKey = "+bId+" "+
18          "AND G.borrower NOT IN ( "+
19          "SELECT C.memberKey "+
20          "FROM eb3Tempmember C "+
21          "WHERE C.memberKey = "+mId+" ) "+
22          "AND E.memberKey = G.borrower ");
23
24        ResultSet rset2 = connection.createStatement().
25          executeQuery("SELECT D.loanDuration "+
26          "FROM member D WHERE D.memberKey = "+mId+" ");
27        String var0 = ((rset2.next())?rset2.getDouble(1)+"":"null");
28
29        connection.createStatement().executeUpdate("UPDATE loan SET "+
30          "borrower = "+mId+" WHERE bookKey = "+ bId +" ");
31
32        connection.createStatement().executeUpdate("UPDATE loan SET "+
33          "dueDate = SYSDATE+"+var0+" WHERE bookKey = "+ bId +" ");
34
35        while(rset0.next()) {
36          connection.createStatement().executeUpdate(
37          "UPDATE member SET nbLoans = "+rset0.getDouble(2)+ " "+
38          "WHERE memberKey = "+ rset0.getDouble(1));
39        }
40
41        while(rset1.next()) {
42          connection.createStatement().executeUpdate(
43          "UPDATE member SET nbLoans = "+rset1.getDouble(2)+ " "+
44          "WHERE memberKey = "+ rset1.getDouble(1));
45        }
46
47        connection.createStatement().
48          executeUpdate("DROP TABLE eb3Tempmember");
49        connection.commit();
50      } catch ( Exception e ) {
51        try{
52          connection.createStatement().
53          executeUpdate("DROP TABLE   eb3Tempmember");
54          connection.rollback();
55        } catch (SQLException s){ System.err.println(s.getMessage());}
56         System.err.println(e.getMessage());}
57    }
```

FIG. 8 – Transaction générée pour l'action Transfer

5 Conclusion

Nous avons présenté les principes de la synthèse des transactions et l'outil EB^{3}TG. Cet outil permet de générer automatiquement des programmes en Java, qui exécutent des transactions de BD relationnelle, à partir de définitions d'attributs EB^3. Cette approche est différente des paradigmes couramment utilisés pour spécifier des SI. L'objectif du projet APIS, et de l'outil EB^{3}TG en particulier, est de libérer le programmeur des détails d'implantation des transactions, pour qu'il se consacre plutôt aux phases d'analyse et de spécification.

Notre objectif est maintenant d'optimiser et d'intégrer EB^{3}TG dans les autres outils d'APIS, comme l'interpréteur EB^{3}PAI. Parmi les optimisations prévues, nous comptons mieux regrouper les mises à jour des transactions. Par exemple, les deux énoncés de mise à jour des attributs *borrower* et *dueDate* de l'exemple Transfer peuvent être regroupés puisqu'ils concernent la même valeur de clé *bId*. D'autre part, nous prévoyons d'améliorer l'analyse de l'arbre de décision en éliminant toute création de table temporaire dans le programme Java généré, comme c'est le cas actuellement. Enfin, nous souhaitons enrichir le langage de définition d'attributs, en supportant d'autres opérateurs que ceux du langage SQL et des énoncés **if then else** arbitrairement imbriqués.

Références

[Arg05] ArgoUML Project. ArgoUML User Manual. `http ://argouml.tigris.com`, 2005.

[Bat05] P. Batanado. Synthèse des transactions de base de données relationnelle à partir de définitions d'attributs EB^3. Mémoire de maîtrise, Département d'informatique, Université de Sherbrooke, Québec, 2005.

[BB87] T. Bolognesi et E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14(1), 1987.

[CM98] G. Cousineau et M. Mauny. *The functional approach to programming.* Cambridge University Press, Cambridge, 1998.

[DLCP00] S. Dupuy, Y. Ledru, et M. Chabre-Peccoud. An overview of RoZ : A tool for integrating UML and Z specifications. In *Proc. 12th Intern. Conf. CAiSE'00*, LNCS 1789, pages 417–430, Stockholm, Suède, 2000. Springer-Verlag.

[Edm95] D. Edmond. Refining database systems. In *Proc. ZUM'95*, LNCS, Limerick, Irlande, 1995. Springer-Verlag.

[EN04] R. Elmasri et S. Navathe. *Fundamentals of Database Systems.* Addison-Wesley, quatrième édition, 2004.

[FF02] B. Fraikin et M. Frappier. EB3PAI : An interpreter for the EB^3 specification language. In *15th Intern. Conf. on Software and Systems Engineering and their Applications (ICSSEA 2002)*, Paris, France, 3-5 Décembre 2002. CMSL.

[FFL05] B. Fraikin, M. Frappier, et R. Laleau. State-based versus event-based specifications for information systems : A comparison of B and EB^3. *Software and Systems Modeling*, 4(3) :236–257, Juillet 2005.

[FFLR02] M. Frappier, B. Fraikin, R. Laleau, et M. Richard. APIS - Automatic production of information systems. In *AAAI Spring Symposium*, pages 17–24, Stanford, USA, 25-27 Mars 2002. AAAI Press.

[FSD03] M. Frappier et R. St-Denis. EB^3 : An entity-based black-box specification method for information systems. *Software and Systems Modeling*, 2(2) :134–149, Juillet 2003.

[Ger04] F. Gervais. *EB^4 : Vers une méthode combinée de spécification formelle des systèmes d'information.* Examen de spécialité, Doctorat informatique, Université de Sherbrooke, Québec, Juin 2004.

[GFL04] F. Gervais, M. Frappier, et R. Laleau. *Synthesizing B substitutions for EB^3 attribute definitions*. Rapport technique n. 683, CEDRIC, Paris, France, Novembre 2004.

[GFL05] F. Gervais, M. Frappier, et R. Laleau. Generating relational database transactions from recursive functions defined on EB^3 traces. In *SEFM 2005 - 3rd IEEE International Conference on Software Engineering and Formal Methods*, Coblence, Allemagne, 7-9 Septembre 2005. IEEE Computer Society Press.

[GFLB05] F. Gervais, M. Frappier, R. Laleau, et P. Batanado. *EB^3 attribute definitions : Formal language and application*. Rapport technique n. 700, CEDRIC, Paris, France, Février 2005.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[Jac83] M. Jackson. *System Development*. Prentice-Hall, 1983.

[LM00] R. Laleau et A. Mammar. An overview of a method and its support tool for generating B specifications from UML notations. In *Proc. ASE : 15th IEEE Conference on Automated Software Engineering*, Grenoble, France, 2000. IEEE Computer Society Press.

[Mam02] A. Mammar. *Un environnement formel pour le développement d'applications base de données*. Thèse de doctorat, CNAM, Paris, France, 2002.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[MS99] E. Meyer et J. Souquières. A systematic approach to transform OMT diagrams to a B specification. In *Proc. FM'99*, volume 1708 of *LNCS*, Toulouse, France, 1999. Springer-Verlag.

[Ngu98] H.P. Nguyen. *Dérivation de spécifications formelles B à partir de spécifications semi-formelles*. Thèse de doctorat, CNAM, Paris, France, 1998.

[Obj05] Object Management Group. Unified Modeling Language. `http ://www.uml.org`, 2005.

[Par05] T. Parr. ANTLR, ANother Tool for Language Recognition. `http ://www.antlr.org`, 2005.

[PTLP99] S.J. Prowell, C.J. Trammell, R.C. Linger, et J.H. Poore. *Cleanroom Software Engineering : Technology and Process*. Addison-Wesley, 1999.

[Qua98] T. Quatrani. *Visual modeling with Rational Rose and UML*. Addison-Wesley, 1998.

[Ter05] J.-G. Terrillon. Description comportementale d'interfaces Web. Mémoire de maîtrise, Département d'informatique, Université de Sherbrooke, Québec, 2005.