Traduction de B événementiel en C pour la validation par la simulation

Faqing Yang, Jean-Pierre Jacquot, Jeanine Souquières

Traduction de B événementiel en C pour la validation par la simulation

Faqing Yang, Jean-Pierre Jacquot, and Jeanine Souquières

LORIA – équipe DEDALE – Nancy Université
Vandoeuvre-Lès-Nancy, France
{firstname.lastname}@loria.fr

Résumé Ce papier discute un traducteur de B événementiel vers C orienté vers la validation des modèles. Les outils d'animation permettent de valider les modèles simples mais échouent lorsque les événements gèrent des espaces multi-dimensionnels. Une façon de contourner cette difficulté est de générer des simulateurs, c'est-à-dire, de traduire le modèle B événementiel en C tout en offrant des points d'ancrage pour des fonctions définies manuellement. Nous décrivons comment un mécanisme d'annotations, une stratégie adaptée de traduction des gardes et actions des événements permet d'étendre les traducteur existants. Nous discutons du pilotage et des difficultés techniques qu'il induit. Nous présentons la démarche pragmatique avec laquelle nous avons abordé cette étude.

1 Introduction

L'usage des méthodes formelles telles que B [1] ou B événementiel [10,2] dépend crucialement des outils qui sont mis à disposition des développeurs. Il faut que l'environnement de travail contienne de quoi traiter toutes les tâches à réaliser. Parmi celles-ci, la validation des modèles, c'est-à-dire le contrôle que le modèle spécifié est une abstraction conforme de la réalité ou du futur système, est de première importance.

La spécification de la dynamique des modèles est notoirement difficile. L'exécution du modèle est un des moyens privilégiés de sa validation. Il existe des outils, les animateurs, qui exécutent un modèle. Ainsi, Brama [1], ProB [5] ou AnimB [2] permettent d'observer comment l'état d'un modèle B événementiel évolue en fonction du déclenchement des événements. Malheureusement, ces outils ont des limites :

- il faut modifier le texte de la spécification [8] vers une version animable,
- les constantes du modèle doivent être codées en extension pour que l'animateur puisse utiliser des stratégies d'énumération,
- certains domaines sont trop complexes pour que les valeurs puissent être énumérées efficacement.

1. http ://www.brama.fr
2. http ://wiki.event-b.org/index.php/AnimB

Dans ce dernier cas, l'animation échoue.

Une façon de contourner ces limitations est de traduire le modèle B événementiel vers un langage de programmation. La faisabilité est montrée :

- B2C [12] traduit vers C le sous-ensemble de notations B événementiel utilisées dans la spécification d'un processeur électronique.
- EB2ALL[9] [3] est issu de B2C dont il élargit l'ensemble des notations traduites. Il permet la traduction vers plusieurs langages.

La contrainte principale de ces traducteurs tient à la nature des modèles traduits. Ceux-ci doivent être déterministes, en particulier, le choix des événements et leur ordre d'exécution est fixé.

Nous pensons qu'il est possible d'associer les deux idées, animation et traduction, pour générer des simulateurs. Il faut ajouter aux traducteurs actuels :

- la traduction de machines de niveau d'abstraction quelconque,
- la prise en compte du non-déterminisme et des quantificateurs,
- la traduction séparée des *Guard* et des *Action* des événements,
- la connexion à des outils graphiques pour la visualisation et la validation.

Dans la suite, nous présentons rapidement B événementiel, notre cadre de spécification, et la notion de simulation. Ensuite, nous discutons de la stratégie de traduction et des points spécifiques à la génération de simulateurs. L'utilisation d'un simulateur est ensuite abordée. Enfin, nous décrivons la démarche que nous avons adoptée, les résultats acquis et les perspectives.

2 Validation de spécifications

2.1 Modèle B événementiel

Trois notions sont au cœur de B événementiel :

- l'état. Représenté par une fonction entre l'espace des noms et l'espace des valeurs contrainte par un invariant, l'état spécifie l'information contenue dans le modèle. Les valeurs sont construites à partir des entiers en utilisant les constructions ensemblistes usuelles. L'invariant est un prédicat du premier ordre qui explicite les relations entre les variables du modèle.
- les événements. Ils modélisent les évolutions que peut subir le système. Un événement est spécifié comme une substitution généralisée munie d'une garde. La garde est un prédicat sur l'état. Un événement dont la garde est vraie est activable, son déclenchement provoque une modification de l'état. Le choix de l'événement à déclencher est non-déterministe.
- le raffinement. Le développement d'un modèle s'effectue par étapes successives. Une étape peut réifier l'état, c'est-à-dire remplacer une variable

3. http ://eb2all.loria.fr/

par une autre plus concrète, introduire de nouvelles variables, ou introduire de nouveaux événements.

B événementiel, tout comme B, est construit autour d'une notion de « correction » très forte. Un modèle est correct s'il respecte trois conditions : (1) il est possible de construire un état initial, (2) tous les événements préservent l'invariant et (3) tous les raffinements préservent l'invariant du modèle abstrait et l'invariant du modèle concret. Ces conditions sont exprimées sous la forme « d'obligations de preuves » qu'un outil comme Rodin [11] peut construire automatiquement et aider à prouver au sens mathématique.

Avec B et B événementiel, il est alors possible de construire un système qui peut être prouvé comme étant « correct » au sens où l'implantation concrète garantit que l'invariant exprimé au niveau du modèle le plus abstrait est préservé.

Du point de vue opérationnel, un calcul modélisé en B événementiel débute par le déclenchement de l'événement *INITIALISATION* qui établit l'état initial ; il s'arrête lorsqu'aucun événement n'a sa garde vraie.

2.2 Animation et simulation

La vision opérationnelle de B événementiel est particulièrement intéressante dans la perspective de la validation des modèles. Si la notion de correction évoquée ci-dessus est nécessaire, elle n'est pas suffisante pour garantir que le modèle est une formalisation adéquate du système envisagé ou de la réalité, c'est-à-dire, qu'il est valide. Pour établir la validité, il est nécessaire d'utiliser des techniques autres que la preuve d'invariance.

L'animation est une technique, parmi d'autres, qui permet d'aborder la validation des modèles. L'évolution de l'état et le déclenchement des événements sont faciles à concevoir et à observer. Il est ainsi possible d'intégrer les futurs utilisateurs ou les spécialistes du domaine aux phases les plus en amont du cycle de développement.

Les animateurs reposent sur des stratégies de sélection et d'énumération des valeurs qui sont similaires à celles des *model-checkers,* ProB [3,6] joue les deux rôles par exemple. Le problème de l'explosion combinatoire survient donc de la même manière et pour les mêmes raisons. Notre étude de cas sur la modélisation du mouvement de véhicules est révélatrice [13]. Le modèle de déplacement mono-dimensionnel (les véhicules sont sur un rail et repérés par une seule coordonnée) est animé sans difficulté. En revanche, les animateurs échouent sur le modèle bi-dimensionnel [13] (les véhicules se déplacent sur un plan).

L'échec des animateurs ne doit pas être interprété comme une impossibilité d'exécuter le modèle. Dans notre cas, il est possible d'écrire directement, en C, un simulateur qui reprend exactement la structure et les définitions du modèle

puis d'exécuter le programme ainsi obtenu. Ceci s'explique aisément par deux observations. Il est souvent possible de remplacer une définition avec des quantificateurs par une fonction ; il est toujours possible d'impliquer l'utilisateur dans le choix des paramètres d'un événement.

Par rapport à l'animation, il est alors possible de voir la simulation comme un mode « dégradé » de validation. Nous perdons certes en rigueur, nous avons moins de garantie que l'exécution soit une image fidèle de la dynamique du modèle, mais nous gagnons en efficacité du processus de développement : les grosses anomalies ont toute chance d'être détectées tôt. Naturellement, l'usage de la simulation ne saurait suffire à valider et doit être complété par de la preuve de consistance, de l'animation ou du *model-checking* sitôt que l'état de développement de la spécification le permet.

2.3 Simuler un modèle B événementiel

Une simulation n'est intéressante que si on peut avoir une assurance raisonnable que le programme est une implantation fidèle de la dynamique du modèle spécifié. Il est donc préférable que l'essentiel du programme soit généré automatiquement à partir du texte formel. B2C ou EB2ALL montrent que la traduction d'un modèle B événementiel en C, Java, etc. est possible. Ils montrent aussi que la traduction est une transformation directe, sous certaines conditions. Ces dernières concernent deux propriétés essentielles des modèles : le déterminisme et le codage des valeurs d'états avec des structures de données simples. Il faut noter que, sous ces conditions, les animateurs n'ont pas de difficulté majeure.

En sus de ce que savent faire les traducteurs mentionnés précédemment, un générateur de simulateur doit également pouvoir :

- interpréter et représenter tous les objets mathématiques exprimables dans les constructions ensemblistes de B événementiel,
- interpréter les quantifications multiples et imbriquées pour éviter les énumérations extensives interminables et
- traiter les choix non-déterministes au cours de l'exécution.

Dans la suite de la présentation, nous détaillons les choix et techniques que nous utilisons pour répondre à ces questions.

3 Phase de traduction

3.1 Stratégie de traduction

Il est possible d'aborder le problème de la traduction de nombreuses façons. Nous avons privilégié une approche pragmatique, évolutive et intégrant

le spécifieur. Cette dernière caractéristique mérite justification. Une des raisons à l'explosion combinatoire est la sous-détermination des propriétés calculatoires des modèles : l'ordre des actions n'est pas déterminé, tout comme le choix des valeurs arguments des événements paramétrés et les données sont abstraites. Nécessaire et inhérente à la démarche par raffinements formels, cette sous-détermination doit être levée pour obtenir un programme avec un temps d'exécution raisonnable. Les progrès dans les analyses statiques permettront d'augmenter la part automatique de la traduction, mais, en attendant, le spécifieur, de part sa connaissance du modèle et son intuition sur les directions de raffinement, est le mieux à même de donner des indices pour la traduction.

Il y a un risque que les indications orientent la traduction vers des implantations trop partielles et partiales du modèle. Ce n'est pas grave lorsqu'on considère que la validation d'un modèle très abstrait est surtout la vérification que les définitions formelles et informelles ont un bon recouvrement. En outre, plus le modèle se réifie, moins le spécifieur doit intervenir. La simulation s'approche donc progressivement d'une véritable activité de validation.

Les grands choix que nous avons adoptés sont les suivants :

- possibilité de désigner certaines variables locales aux événements comme étant des paramètres. Nous voyons les événements comme des fonctions de changement d'état et les paramètres modélisent les choix extérieurs.
- possibilité de préciser l'implantation de certains types. Par exemple, une variable B événementiel de type fonction peut être représentée par un tableau ou une fonction C.
- possiblité de remplacer les quantificateurs par des fonctions, éventuellement fournies par l'utilisateur.
- calcul séparé des gardes et des actions. À chaque cycle de calcul du modèle, les gardes sont évaluées pour déterminer quels événements sont déclenchables. La fonction de choix de l'événement à déclencher peut être fournie par le spécifieur.
- l'affichage de l'état séparé de l'exécution. L'idée est de développer un serveur qui communiquera avec le simulateur pour observer les changements d'état et qui lancera les actions d'affichage, par exemple sur une web page *JSP*. Ce modèle s'inspire de l'architecture de l'animateur Brama.

La génération d'une simulation passe par quatre étapes : (1) l'annotation de la spécification, (2) la génération des fichiers C, (3) la programmation des fonctions de choix non-déterministes (paramètres, événement déclenché, etc.) et (4) la mise en place de l'interface graphique. Dans la suite, nous nous intéressons aux points (1) et (2) ; notre réflexion n'est pas encore terminée sur les deux points suivants et sera présentée dans la section suivante.

3.2 Phase d'annotation

Les annotations permettent de préciser la nature et l'implantation d'éléments atomiques de la spécification. Elles concernent principalement les constantes et les variables. Parmi les différentes techniques possibles pour implanter des annotations, nous avons choisi les commentaires qui simplifient l'implantation. Une annotation sur une constante ou une variable est simplement posée comme un commentaire sur sa déclaration. Par comparaison avec des techniques reposant sur l'addition de structures dans Rodin, notre choix coûte le développement de petits analyseurs ad-hoc et une absence de guidage de l'utilisateur lors de la phase d'annotation. En revanche, il a les gros avantages de ne pas modifier le langage ou ses outils supports, de ne pas nécessiter d'outils pour la gestion des modifications et de permettre d'expérimenter aisément.

Pour l'heure, nous avons 6 annotations qui sont résumées ci après :

Event-B Context :	@type=constant @value= @type=function
Event-B Machine :	@type=array @nature=parameter @nature=local_variable
Evaluation of guards :	@ignore

Les détails de leur usage sont donnés dans les sections suivantes. Naturellement, le spécifieur peut ajouter ou retirer une annotation à sa guise. L'absence d'annotation conduit à une traduction par défaut.

3.3 Structure des fichiers générés

Pour simplifier la compréhension et l'évaluation de la qualité de la traduction, la structure des fichiers produits est très proche de la structure de la spécification. Ainsi, pour chaque composant *Context*, nous avons deux fichiers (*.h* : déclarations des constantes et prototypes, *.c* : implantation des structures et fonctions). La machine ou le raffinement est traduit en un seul fichier. Les clauses *SEES* de machine qui réfèrent les contextes sont traduites par une inclusion (clause *#include*) du fichier *.h* correspondant.

Les définitions des structures de bases de B événementiel et des fonctions communes à toutes les simulations sont contenues dans deux fichiers : *events.c* et *event.h*.

3.4 Traduction des constantes

L'annotation des constantes définies dans les contextes permet de simplifier la traduction et d'instancier le modèle. Il est ainsi possible de contraindre le typage C, et de donner des valeurs. Par exemple, on trouvera :

CONSTANTS
VEHICLES // @type=constant @value=4
initial_xpos // @type=function
AXIOMS
axm1 : VEHICLES $\in$ 1..$\mathbb{N}$1
axm3 : initial_xpos $\in$ 1..VEHICLES $\rightarrow \mathbb{N}$

La traduction de la première forme utilise une construction *#define* de C, alors que la seconde introduit une fonction.

La table suivante donne un résumé de ces annotations et de leur traduction :

Event-B	C	Annotation	Commentaire
x $\in$ m..n	#define x	@type=constant @value=	déclaration de constante
x $\in$ Y	#define x	@type=constant @value=	déclaration de constante
x $\in$ m..n $\rightarrow$ Y	Y x()	@type=function	déclaration de fonction

3.5 Traduction des variables

L'annotation des variables a deux rôles. Comme pour les constantes, il est possible de contraindre le type. Ainsi, une variable de type fonctionnel peut être implantée comme un tableau. Un second rôle apparaît lorsqu'on analyse le statut des variables dans les événements. Clairement, les variables introduites par une clause *ANY* servent deux rôles : soit elles sont l'équivalent de variables locales, soit elles sont l'équivalent de paramètres. Dans le premier rôle, la détermination de la valeur de la variable est déterministe. Dans l'autre rôle, le choix est non-déterministe. Intuitivement, cette distinction permet d'interpréter les événements comme des fonctions au sens qu'apprécient les programmeurs. Cette interprétation est bien adaptée à la modélisation de la dynamique d'un modèle.

Dans l'exemple suivant :

VARIABLES
vehicle
xpos0 // @type=array
INVARIANTS
inv3 : vehicle $\in$ 1..VEHICLES+1
inv1 : xpos0 $\in$ 1..VEHICLES $\rightarrow \mathbb{N}$
EVENTS

move
ANY
magic_accel // @nature=parameter @type=array
nspeed // @nature=local_variable
WHERE
grd1 : magic_accel $\in$ 1..VEHICLES $\rightarrow \mathbb{N}$
grd5 : nspeed = new_speed(speed(vehicle)$\mapsto$magic_accel)
END

La variable *vehicle* sera traduite comme une variable C simple sans l'annotation, et la variable *xpos0* sera traduite comme un tableau C par l'annotation

@type=array. Les deux annotations sur l'événement *move* indiquent que *magic_accel* est un paramètre par l'annotation *@nature=parameter*, *nspeed* est une variable locale simple de *move* par l'annotation *@nature=local_variable*.

La table suivante résume les règles de traduction :

Event-B	C	Annotation	Commentaire
$x \in m..n$		-	déclaration de variable simple
$x \in m..n \rightarrow Y$	Y[]	@type=array	déclaration de tableau
		@nature=parameter	déclaration de paramètre
		@nature=local_variable	déclaration de variable locale

3.6 Traduction des événements

La traduction des événements dissocie gardes et actions. Pour chaque événement, deux fonctions sont générées. Un événement est représenté en C par la structure suivante :

```
typedef struct Event{
    char name[256];
    BOOL active;
    void (* guards)();
    void (* actions)();
}t_Event;
```

Lors du traitement d'une machine, le traducteur construit le tableau *events [TOTAL_EVENTS]* qui référence tous les événements, hors *INITIALISATION*, et leurs deux fonctions. La traduction d'une machine avec 15 événements générera le texte suivant :

```
#define TOTAL_EVENTS 15 // exclude INITIALISATION
t_Event events[TOTAL_EVENTS] = {
    {"perceive1", FALSE, perceive1_guards, perceive1_actions},
    {"perceive", FALSE, perceive_guards, perceive_actions},
    {"decide1_normal", FALSE, decide1_normal_guards, decide1_normal_actions},
    {"decide1_max", FALSE, decide1_max_guards, decide1_max_actions},
    {"decide1_min", FALSE, decide1_min_guards, decide1_min_actions},
    {"decide_normal", FALSE, decide_normal_guards, decide_normal_actions},
    {"decide_max", FALSE, decide_max_guards, decide_max_actions},
    {"decide_min", FALSE, decide_min_guards, decide_min_actions},
    {"move1_normal", FALSE, move1_normal_guards, move1_normal_actions},
    {"move1_max", FALSE, move1_max_guards, move1_max_actions},
    {"move1_reduce", FALSE, move1_reduce_guards, move1_reduce_actions},
    {"move_normal", FALSE, move_normal_guards, move_normal_actions},
    {"move_max", FALSE, move_max_guards, move_max_actions},
    {"move_reduce", FALSE, move_reduce_guards, move_reduce_actions},
    {"all_moves", FALSE, all_moves_guards, all_moves_actions}
};
```

Les noms des fonctions C sont construits en post-fixant le nom de l'événement avec *_actions* et *_guards*.

Cette traduction des événements coûte quelques indirections lors de l'exécution de la simulation. En contrepartie, elle permet une grande flexibilité dans la gestion du moteur de la simulation. En particulier, il est très simple de construire l'ensemble des événements activables sous la forme d'un tableau :
t_Events active_events[TOTAL_EVENTS].

L'événement d'initialisation bénéficie d'un traitement spécial. Il est traduit en une procédure de prototype *void INITIALISATION()*.

3.7 Traduction des gardes

Les gardes sont évaluées une par une dans leur ordre d'écriture. Pour éviter les problèmes de collision de variables liées, des blocs C sont introduits. Les gardes portant une annotation *ignore* ne sont pas traduites. L'extrait suivant

EVENTS
...
all_moves $\widehat{=}$
ANY
magic_xpos @nature=parameter @type=array
WHERE
grd1 : magic_xpos $\in$ 1..VEHICLES $\rightarrow \mathbb{N}$ // @ignore
grd2 : $\forall$v.(v $\in$ 2..VEHICLES $\Rightarrow$
((magic_xpos(v$-$1)$-$ magic_xpos(v)) $>$ CRITICAL_DISTANCE))
....

est traduit en

```
/* Event [all_moves]: evaluate guards */
BOOL all_moves_guards(NAT magic_xpos[VEHICLES+1]) {
    /* Evaluate grd1: type define for parameter magic_xpos, ignore */

    /* Evaluate grd2 */
    {
        BOOL grd2 = TRUE;
        int v;
        for (v = 2; v <= VEHICLES && grd2; v++)  {
            grd2 =( (magic_xpos[v-1] - magic_xpos[v]
                  > CRITICAL_DISTANCE) && grd2);
        }
        if (!grd2) {
            return FALSE;
        }
    }
...
```

Volontairement, nous n'avons pas cherché à produire un programme efficace, ceci ne nous paraît pas être un objectif majeur dans le cadre où nous devrons l'utiliser. Nous avons privilégié la simplicité des règles de traduction car ce sont elles, *in fine,* qui produisent un programme conforme, ou non, au modèle. Leur relative simplicité permet de vérifier leur correction plus facilement. Notons également que le code C construit reste très proche, structurellement, du texte B événementiel. Il est permis de penser que l'étude d'un comportement inattendu d'une simulation puisse alors passer par l'analyse du code C afin de remonter jusqu'à l'anomalie dans le modèle.

3.8 Traduction des actions

Les actions des événements sont des substitutions qui se traduisent naturellement par des affectations. Pour l'instant, l'opérateur "devient égal" est supporté. La difficulté tient à ce que la sémantique de B événementiel suppose que les substitutions sont concurrentes, ce qui revient à les considérer comme une affectation multiple. Les langages de programmation classiques ne disposent pas de cette construction.

Il y a deux possibilités pour surmonter le problème de l'affectation multiple. La première consiste à analyser les dépendances entre variables et affectations afin de leur trouver un ordre compatible. Ainsi, pour l'action suivante,

```
EVENTS
 ...
move1_normal ≙
 ...
THEN
     act1 : vehicle := vehicle+1
     act2 : xpos(vehicle) := nxpos
     act3 : speed(vehicle) := nspeed
 ....
```

l'ordre d'affectation doit être *act2 ;act3 ;act1* ou *act3 ;act2 ;act1*. Outre le fait que l'analyse de dépendance peut vite devenir complexe, elle ne résout pas les cas cycliques tels que act1 : a:=b ; act2 : b:=a.

Nous avons choisi la stratégie plus lourde qui consiste à introduire systématiquement des variables intermédiaires qui stockent les « anciennes » valeurs. De nouveau, nous avons privilégié la simplicité des règles et du code construit par rapport à son efficacité. Nous pouvons également penser que cette stratégie permettra de mettre en place des outils d'analyse avant/après utiles pour « déboguer » des événements complexes.

4 Phase de simulation

4.1 Choix de l'événement

Le non-déterminisme intrinsèque de B événementiel fait qu'à chaque pas de l'exécution, il peut y avoir un choix de l'événement à déclencher. Si une fonction du raffinement est de rendre déterministe une machine, il reste que l'intérêt d'un simulateur est de pouvoir observer les comportements avant que le système n'ait été rendu déterministe. Nous avons donc adopté une stratégie proche des animateurs qui passe par trois étapes :

1. détermination de l'ensemble des événements déclenchables. Toutes les gardes sont évaluées,
2. choix de l'événement à déclencher,
3. réalisation de l'action de l'événement choisi.

Les étapes 1 et 3 sont purement mécaniques ; elles sont facilitées par la structure de la traduction des événements que nous avons adoptée. Le choix de l'événement à déclencher est naturellement de la responsabilité du spécifieur. Pour cela, il peut fournir une fonction de prototype *t_events pick_event(int, t_events[])* qui implante une stratégie particulière de choix de l'événement, éventuellement limitée à l'interrogation systématique de l'utilisateur du simulateur. Une fonction de choix équiprobable est fournie par défaut.

La boucle de simulation est donc la suivante :

```
while(1){
    for (i = 0; i < TOTAL_EVENTS; i++)  {
        events[i].active = events[i].guards();
    }

    int i;
    int true_guards = 0;
    for (i = 0; i < TOTAL_EVENTS; i++)  {
        if (events[i].guard == TRUE){
            active_events[true_guards] = events[i];
            true_guards++;
        }
    }

    if (true_guards == 0){
        printf("All evaluation of events guards is false,
                deadlock occurs or system terminate normally.\n");
    }else{
        active_event = pick_event(true_guards, active_events);
        active_events[active_event].actions();
        ...
    }
}
```

4.2 Choix des valeurs de simulation

La construction *ANY* qui introduit les variables des événements est responsable de l'échec des animateurs. En effet, ceux-ci cherchent à parcourir ou évaluer l'ensemble des valeurs compatibles avec l'activation de l'événement pour en choisir une particulière. Cette stratégie est la seule qui soit formellement « correcte », malheureusement, elle atteint rapidement ses limites lorsque la complexité de l'état croît.

Notre idée est qu'on peut, dans le cas de la simulation, retourner le point de vue. Plutôt que de chercher des valeurs compatibles, il est possible de fournir des valeurs puis de vérifier leur compatibilité. La simulation est ainsi pilotée par l'utilisateur, soit à travers un dialogue ou à travers une fonction de génération des valeurs codée à la main. Il est alors possible de répondre à des questions telles que « Le comportement observé est-il conforme à celui qu'on attend pour telle valeur ? » ou « Que se passe-t-il pour telle valeur ? ». Nous sommes ainsi dans une logique identique à celle du test qui permet de découvrir des anomalies mais pas de garantir leur absence. Cette limitation, assumée, est compatible avec l'usage de la simulation des modèles abstraits.

La distinction entre variables locales et paramètres (*cf.* sec.3.5) relève de cette démarche. Les valeurs données pas l'utilisateur sont celles des paramètres. Pour l'heure, la stratégie pour donner ces valeurs n'est pas encore fixée. Deux stratégies de base sont explorées. La première retire les paramètres des gardes lors du calcul de l'ensemble des événements déclenchables et les valeurs sont fixées après le choix de l'événement. La seconde demande de fixer l'ensemble des paramètres de tous les événements avant le calcul des gardes. Dans un cas l'ensemble des événements déclenchables est sur-évalué, il est sous-évalué dans l'autre. Les expérimentations en cours sur nos études de cas nous permettrons de choisir la stratégie la plus pertinente.

5 Approche et perspectives

5.1 Démarche de travail

Les traducteurs existants, tels B2C ou EB2ALL, se situent dans la démarche de raffinement. Ils sont conçus comme la dernière étape et, dans l'idéal, garantissent la correction du code généré vis-à-vis de la spécification initiale. La contre-partie à cette garantie est que la traduction ne peut s'appliquer qu'à des modèles qui ont été raffinés jusqu'à devenir déterministes. L'objectif de notre traducteur est différent. Il doit pouvoir s'appliquer sur des modèles non-déterministes, quitte à ce que la traduction restreigne la dynamique spécifiée. Nous sommes donc dans une démarche d'ingénierie où l'expérimentation et le

compromis sont essentiels, plutôt que dans une démarche formelle où la correction sémantique est fondamentale.

Le travail s'appuie sur les études de cas qui ont été développées par l'équipe dans le domaine des transports : *platooning* simplifié mono-dimensionnel [4], *platooning* réaliste bi-dimensionnel [13] et spécification du domaine des transports [7]. Ces spécifications ont été soumises aux animateurs, avec plus ou moins de bonheur.

La première étape du travail a été de définir la stratégie générale de traduction et les règles pour les constructions élémentaires. Pour cela, nous avons programmé manuellement, en nous astreignant à suivre les règles en cours d'élaboration, des simulateurs, un par raffinement, du *platoon* mono-dimensionnel. Comme tous ces raffinements sont animables, l'objectif était de faire aussi bien. En particulier, les simulateurs devaient mettre en évidence les *deadlocks* que nous avons identifiés dans certaines versions de la spécification.

Cette étape s'étant déroulée avec succès, nous avons implanté les règles de traduction sous la forme d'un *plug-in* Rodin. Une première version expérimentale d'un traducteur est disponible. Aujourd'hui, nous obtenons par la traduction automatisée le même résultat que par la traduction manuelle pour le *platooning* mono-dimensionnel [4]. La comparaison entre l'animation des différents raffinements et la simulation indique qu'il n'y a pas de divergence dans les comportements observés.

La troisième étape, en cours, concerne essentiellement l'étude du pilotage, c'est-à-dire de l'injection des données, dans la simulation. Comme il a été montré précédemment, il faut déterminer quelle stratégie est la plus pertinente vis-à-vis de l'objectif des utilisateurs de la simulation. La spécification utilisée pour cela est le *platoon* bi-dimensionnel. Les animateurs échouent sur cette spécification, mais nous savons qu'il est possible de la simuler. L'objectif est donc de construire automatiquement des simulateurs utilisables.

5.2 Perspective

La troisième étape finie, trois questions importantes seront abordées.

Le graphisme. L'usage de la simulation comme outil de validation par les utilisateurs du futur système ou les experts du domaine impose que résultats et données de pilotage leur soient présentés sous une forme accessible. Une interface graphique est indispensable. Cette idée n'est pas originale, Brama et ProB proposent des mécanismes de connexion des animations avec des interfaces graphiques. Une critique de ces deux systèmes est que le type d'interface est trop fortement contraint par les limitations du modèle graphique ou de la communication entre l'animateur et le moteur graphique. Les limitations sont dues au

fait que les outils sont implantés dans des cadres différents qu'il est difficile de faire communiquer. Comme notre traducteur produit un code C autonome, nous pensons qu'il sera possible d'offrir des mécanismes permettant de créer des interfaces plus riches et plus conformes aux attentes des utilisateurs.

Les scénarios d'animation. Créer et gérer des scénarios d'animation sont des activités similaires à celles concernant les jeux de tests. Même si le fait est rarement mentionné, les spécifications formelles passent par une phase de mise au point comparable au « *debuging* » des programmes. Les activités de preuve y jouent certes un rôle, mais ce ne sont pas les seules. Il conviendra donc de pouvoir refaire aisément une simulation sur un cas vicieux, ou de s'assurer qu'une nouvelle version n'introduit pas une « régression » par exemple.

La correction de la simulation. La conformité du simulateur généré par rapport à la spécification repose pour l'instant sur notre intime conviction que les règles de traduction sont correctes. Nous les expliquons par la proximité des concepts de B événementiel avec le modèle de calcul impératif de C. Naturellement, il sera nécessaire de démontrer que ces règles sont sémantiquement correctes.

6 Conclusion

La généralisation de l'usage des méthodes formelles est très dépendante des outils qui sont mis à disposition des praticiens. Outre des outils mathématiques (les prouveurs) et méthodiques (les *patterns*), il faut aussi des outils d'exploration et d'analyse des modèles, en particulier de leur dynamique. Nous avons montré qu'il paraît possible de compléter l'offre actuelle des animateurs par des simulateurs qui génèrent du code exécutable. Si, par rapport aux animateurs, les simulateurs ont un lien formel plus ténu avec le modèle spécifié, ils autorisent en revanche la visualisation d'une plus large classe de modèles.

La démarche que nous avons adoptée est caractérisée par le pragmatisme. Notre ambition est comprendre quels sont les difficultés, ou les limites, de la traduction, de comprendre quels sont les problèmes pratiques et les enjeux de la mise en œuvre de la simulation et de comprendre comment raffinement formel, vérification et validation peuvent s'intégrer dans un cadre méthodique. Pour cela nous avons besoin de prototypes tels que celui que nous construisons. Nous pensons que ce n'est qu'après avoir mieux compris les trois points précédents que nous pourrons envisager la construction d'un traducteur utile qui offre des garanties formelles.

Références

1. Abrial, J.R. : The B Book. Cambridge University Press (1996)

2. Abrial, J.R. : Modeling in Event-B : System and Software Engineering. Cambridge University Press (2010)
3. Bendisposto, J., Leuschel, M., Ligot, O., Samia, M. : La validation de modèles Event-B avec le plug-in ProB pour RODIN. Technique et Science Informatiques 27(8), 1065–1084 (2008)
4. Lanoix, A. : Event-B Specification of a Situated Multi-Agent System : Study of a Platoon of Vehicles. In : 2nd IFIP/IEEE International Symposium on Theoretical Aspects of Software Engineering (TASE 2008). p. 8 pages. France (2008-06), http ://hal.archives-ouvertes.fr/hal-00260577/en/
5. Leuschel, M., Butler, M. : ProB : An Automated Analysis Toolset for the B Method. Journal Software Tools for Technology Transfer 10(2), 185–203 (2008)
6. Ligot, O., Bendisposto, J., Leuschel, M. : Debug event-b models using the prob disprover plugin. In : Approches Formelles dans l'Assistance au Développement de Logiciels(AFADL'07). Namur, Belgium (2007)
7. Mashkoor, A., Jacquot, J.P. : Utilizing Event-B for Domain Engineering : A Critical Analysis. Requirements Engineering 16(3), 191–207 (2011)
8. Mashkoor, A., Jacquot, J.P., Souquières, J. : Transformation Heuristics for Formal Requirements Validation by Animation. In : 2nd International Workshop on the Certification of Safety-Critical Software Controlled Systems (SafeCert'09). York, UK (2009)
9. Méry, D., Singh, N. : Automatic Code Generation from Event-B Models. In : Proc. 2011 Symposium on Information and Communication Technology, SoICT2011. ACM International Conference Proceeding Series, ACM, Hanoi (2011)
10. Metayer, C., Voisin, L. : The Event-B Mathematical Language (Oct 2007)
11. RODIN : Rigorous Open Development Environment for Complex Systems. website (Aug 2007), http ://rodin-b-sharp.sourceforge.net
12. Wright, S. : Automatic Generation of C from Event-B. In : Workshop on Integration of Model-based Formal Methods and Tools (2009)
13. Yang, F., Jacquot, J.P. : Scaling Up with Event-B : A Case Study. In : Bobaru, M., Havelund, K., Holzmann, G., Joshi, R. (eds.) NASA Formal Methods, Lecture Notes in Computer Science, vol. 6617, pp. 438–452. Springer Berlin / Heidelberg (2011)