Un processus de développement Event-B pour des applications distribuées

Badr Siala, Mohamed Tahar Bhiri, Jean-Paul Bodeveix, M Filali

Un processus de développement Event-B pour des applications distribuées

Badr Siala, Mohamed Tahar Bhiri, Jean-Paul Bodeveix et Mamoun Filali
Université de Sfax
IRIT CNRS UPS Université de Toulouse

Résumé

Nous présentons une méthodologie basée sur le raffinement manuel et automatique pour le développement d'applications distribuées correctes par construction. À partir d'un modèle Event-B, le processus étudié définit des étapes successives pour décomposer et ordonnancer les calculs associés aux événements et distribuer le code sur des composants. La spécification de ces deux étapes est faite au travers de deux langages dédiés. Enfin, une implantation distribuée en BIP est générée. La correction du processus repose sur la correction des raffinements et de la traduction vers le code cible BIP.

1 Introduction

Les systèmes distribués demeurent difficiles à concevoir, construire et faire évoluer. Ceci est lié à la concurrence et au non déterminisme. Plusieurs formalismes tels que les algèbres de processus, ASM et TLA$^+$ sont utilisés pour modéliser et raisonner sur la correction des systèmes distribués. Un spécifieur utilisant ces formalismes est censé modéliser des mécanismes de bas niveau tels que canaux de communication et ordonnanceurs.

Dans cet article, nous proposons un support outillé d'aide à la conception système en utilisant une méthodologie basée sur des raffinements prouvés. Les systèmes considérés sont vus comme un ensemble d'acteurs en interaction. Les premiers raffinements fournissent une vue centralisée du système. Ils sont construits en prenant en compte progressivement les exigences du système. Ces raffinements sont exprimés à l'aide de machines abstraites décrites en Event-B [2]. Ensuite, nous proposons des schémas de raffinements destinés à prendre en compte la nature distribuée du système étudié. Ces schémas de raffinements sont guidés par l'utilsateur et les machines Event-B associées sont automatiquement générées. En conséquence, on obtient un ensemble de machines qui interagissent, dont la composition est prouvée correcte et conforme avec le niveau abstrait. Le système peut ensuite être exécuté sur une plateforme distribuée via une traduction en BIP [3]. Remarquons que notre objectif n'est pas d'automatiser complètement le processus de distribution, mais de l'assister. Tout en restant modeste, la différence est

similaire à celle entre un vérificateur de modèle où la preuve d'un jugement est automatique et un assistant de preuve où l'utilisateur doit composer des stratégies de base afin de résoudre son but. De même qu'un assistant de preuve aide à construire la preuve d'un but, notre objectif est d'aider à l'élaboration par raffinements d'un modèle distribué.

Event-B et BIP admettent une sémantique basée sur les systèmes de transitions étiquetées. Ceci favorise leur couplage. Event-B est utilisée pour la spécification et la décomposition formelles des systèmes distribués. BIP est utilisée pour leur implantation et leur déploiement sur une plateforme à mémoire répartie. Le passage d'Event-B vers BIP s'appuie sur des raffinements manuels et automatiques. Les raffinements manuels horizontaux permettent l'introduction progressive de propriétés du futur système. Les raffinements manuels verticaux permettent l'obtention de modèles Event-B traduisibles vers BIP : déterminisation d'actions et concrétisation des données. Quant aux raffinements automatiques, nous en avons élaboré deux sortes : l'une est appelée *fragmentation* (voir section 3) et l'autre *distribution* (voir section 4). Ces deux sortes de raffinement sont guidées par le spécifieur via deux langages dédiés (DSL). Cette démarche peut être comparée aux travaux portant sur la génération automatique de code source à partir de spécifications formelles. [4] propose un générateur de code efficace séquentiel en utilisant un sous-ensemble B0 impératif de B. Le raffinement automatique de machines B est également possible grâce à l'outil Bart [5]. Cependant, il ne concerne pas les modèles Event-B et est destiné aux raffinements de modèles B vers le sous-ensemble B0. Concernant Event-B, plusieurs générateurs de code source ont été proposés [6, 8]. Il est possible de générer du code Ada parallèle. Cette dernière cible est cependant plus restrictive que BIP puisqu'elle n'offre que de la synchronisation binaire.

Le processus de développement correct par construction de systèmes distribués préconisé combine les raffinements manuels et automatiques. Il se termine par la génération de code BIP. Dans la suite, après une introduction aux formalismes Event-B et BIP, nous présentons les deux transformations par raffinement (la fragmentation et la distribution), puis la génération de code BIP. Ces étapes seront illustrées sur l'exemple de l'hôtel [1].

Le passage d'Event-B vers BIP se fait en trois étapes : fragmentation, distribution et génération de code (voir Figure 1).

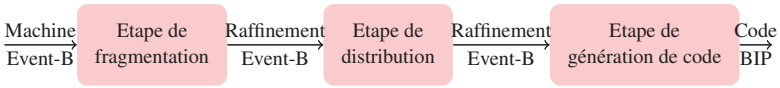

FIGURE 1 – Etapes du processus

1. Le texte complet est accessible via le lien
`https://dl.dropboxusercontent.com/u/98832434/hotelrefinements.html`.

2 Event-B et BIP

2.1 Event-B

Event-B permet de décrire le comportement d'un système par une chaine de raffinements de machines abstraites. Une machine contient des événements, éventuellement paramétrés, faisant évoluer l'état de la machine tout en préservant un invariant. Le non-déterminisme s'exprime dans le corps des événements grâce aux actions ($:\in$ et $:|$) ou via leurs paramètres introduits dans la clause `ANY` et contraints par les propriétés spécifiées dans la clause `WHERE`. Dans une optique de décomposition formelle d'une spécification centralisée en plusieurs composants Event-B, les contraintes d'un paramètre d'un événement partagé doivent être résolues localement. Nous adressons ce problème ultérieurement.

Récemment, deux opérations ont été introduites dans Event-B : la composition et la décomposition [7]. Elles ont pour but d'introduire la notion de composant dans la démarche de raffinement. Ces deux opérations sont reliées par la propriété suivante : la composition des sous-composants produits par décomposition d'un modèle doit raffiner ce modèle. Deux variantes ont été identifiées : par variables partagées et par événements partagés. La première est adaptée aux systèmes à mémoire partagée, tandis que la deuxième est plutôt adaptée aux systèmes distribués. Dans ce travail, nous nous intéressons à la composition/décomposition par événements partagés. La plateforme Rodin offre un outil interactif [7] sous forme d'un plugin permettant la composition/décomposition par événements partagés. Dans la section 4, nous situerons notre plugin de distribution vis-à-vis de ce plugin.

2.2 BIP

Le modèle de composant BIP comporte trois couches : *Behavior*, *Interaction* et *Priority*. La couche *Behavior* permet de décrire les aspects **comportementaux** des composants atomiques tandis que les couches *Interaction* et *Priority* décrivent les aspects **architecturaux** d'un système. Les contraintes de synchronisation entre les composants BIP sont exprimées par des interactions regroupées au sein de la construction `connector` de BIP tandis que les contraintes d'ordonnancement des interactions sont exprimées grâce au concept Priority de BIP. En BIP, un composant atomique englobe des données, des ports et un comportement. Les données (`data`) sont typées. Les ports (`port`) donnent accès à des données et constituent l'**interface** du composant. Le comportement est un ensemble de transitions définies par un port, une garde et une fonction de mise à jour des données.

3 La fragmentation

Cette étape, dite de fragmentation, prend en entrée un modèle Event-B quelconque (voir figure 1). Elle a pour but de réduire le non-déterminisme lié au calcul des paramètres locaux d'un événement. L'ordre de calcul des paramètres est

décrit par le spécifieur à l'aide d'un DSL (voir listing 1). En se basant sur cette description et le modèle Event-B abstrait, l'étape de fragmentation génère un modèle Event-B. Ce raffinement automatique s'appuie sur des règles simples assurant l'obtention d'un raffinement : introduction d'un nouvel événement convergent, raffinement d'un événement par plusieurs (one to many), introduction d'une nouvelle variable, renforcement de garde, renforcement d'invariant et instanciation d'un paramètre local d'un événement par une variable d'état.

Plugin de fragmentation. La transformation de fragmentation est implantée par un plugin Rodin. Elle génère un raffinement de la machine en question à partir d'une *spécification de la fragmentation*. Celle-ci comporte des déclarations donnant pour un paramètre p d'un événement ev les paramètres p_i dont il dépend et les gardes g_i le spécifiant. Une valeur initiale v est requise pour des besoins de typage.

event ev **when** p_1 ... p_n **parameter** p **init** v **with** g_1 ... g_m

Exemple d'illustration. Dans l'exemple de l'hôtel, l'événement `register` a trois paramètres : `g, r, c`. Nous spécifions que le paramètre client `g` doit être calculé en premier. Une chambre r est alors choisie et la carte d'accès c est enfin générée. La fragmentation de l'événement `register` est ainsi spécifiée :

```
splitting hotel_splitted refines hotel
events
  event register
    parameter g init g0 with tg // tg ne dépend pas de r,c
    when g parameter r init r0 with tr g1 // déclenché après le calcul de g
    when g r parameter c init c0 with tc g2 g3 // déclenché après g,r
end
```

Listing 1 – Spécification de la fragmentation

La fragmentation produit une machine dont le schéma général est le suivant :

```
machine generated refines input_machine
variables
 ev_p  ev_p_computed//témoin et statut pour param. p de l'événement ev
invariants
  @ev_gi ev_p_computed = TRUE ⇒ gi // ou p est remplacé par ev_p
variant  // compteur des paramètres restant à calculer
  {FALSE ↦ 1, TRUE ↦ 0}(ev_p_computed) + ...
events
  event INITIALISATION extends INITIALISATION
  then
    @ev_p ev_p := v
    @ev_p_comp ev_p_computed := FALSE
  end
  convergent event compute_ev_p // calcule le  paramètre p de ev
  any p where
   @gi gi // garde spécifiant  p
   @pi ev_pi_computed = TRUE //paramètres dont p dépend ont été calculés
   @p ev_p_computed = FALSE // p reste à calculer
  then
    @a ev_p := p
    @computed ev_p_computed := TRUE // décroissance du variant
```

```
  end
  event ev refines ev
  when
    @p_comp ev_p_computed = TRUE
  with
    @p p = ev_p //le paramètre p de l'événement hérité est raffiné en ev_p
  then
    // remplacer p par ev_p dans les  actions de l'événement raffiné
  end
end
```

Listing 2 – Machine générée pour le raffinement de fragmentation

En fait, cette machine implante les contraintes d'ordonnancement par l'introduction d'une variable booléenne, *computed state*, par paramètre. L'invariant de la machine est étendu par les propriétés des variables introduites. Si une variable a été calculée (`ev_p_computed = TRUE`), sa spécification, donnée par sa garde `gi`, est alors satisfaite. Lorsque tous les paramètres d'un événement ont été calculés, l'événement en question peut être activé. Enfin, la progression du calcul des paramètres requis est assurée par un variant défini comme le nombre de paramètres restant à calculer.

4 La distribution

Après l'étape de fragmentation (voir figure 1), l'étape de gestion de la distribution prend en compte les particularités de l'architecture cible. Il s'agit ici de composants BIP synchronisés par des connecteurs n-aires et supposés ne réaliser que des transferts de données (pas de traitement interne dans un connecteur). À l'instar de l'étape de fragmentation, l'étape de distribution prend en entrée un modèle abstrait et une spécification de distribution. Celle-ci est décrite par le spécifieur à l'aide d'un DSL. Elle introduit la configuration retenue : les noms des sous-composants. De même, elle répartit les variables et éventuellement les gardes sur les sous-composants. Les variables référencées par une garde doivent être localisées sur un même sous-composant. Autrement, des copies *faiblement cohérentes* de ses variables seront automatiquement ajoutées. Elles sont mises à jour par des événements convergents ordonnancés avant l'événement accédant aux copies. De même, une action est réalisée par le composant (supposé unique) sur lequel les variables modifiées sont localisées. Les variables lues par une action peuvent être distantes. Les valeurs de ces variables seront transmises lors de la synchronisation. Notons que la gestion des copies faiblement cohérentes et des variables distantes sont spécifiques à notre proposition. Ceci peut être vu comme une extension du plugin de *décomposition par événement partagé* [7]. Pour y parvenir, nous introduisons une phase de pré-traitement.

La phase de pré-traitement. Cette phase prend en entrée une machine abstraite, des identifiants de sous-composants et la répartition de ces variables sur des sous-composants. Elle produit un raffinement introduisant des copies des variables dis-

tantes lues par les gardes, les événements anticipés mettant à jour ces variables, et des paramètres recevant les valeurs des variables distantes lues par les actions. Nous supposons dans ce qui suit que les variables v_i sont localisées sur les composants C_i. Les événements convergents sont partagés par les origines (C_i) et destinations (C_j) des variables lues par les gardes. Les raffinements des événements hérités sont partagés par les origines (C_i) des copies (sur C_j) et par les composants (C_k). Ces derniers comportent les variables directement lues par les actions.

```
machine generated refines input_machine
variables
  vi // variables héritées, sur Ci
  Cj_vi // copie sur Cj de vi (utilisée par une garde sur Cj)
  vi_fresh // true si vi a été copié, sur Ci
invariants
  @Cj_vi_f vi_fresh = TRUE ⇒ Cj_vi = vi // la copie est à jour
variant
  {FALSE ↦ 1, TRUE ↦ 0}(vi_fresh) + ...
events
  convergent event share_vi // partagé par Ci et Cj
  any local_vi where
    @g vi_fresh = FALSE // sur Ci
    @l local_vi = vi // sur Ci
  then
    @to_Cj Cj_vi := local_vi // sur Cj
    @done vi_fresh := TRUE // sur Ci
  end
  event ev refines ev // partagé par Ci,Cj,Ck
  any local_vk where
    @vj_access local_vk = vk // sur Ck, accès direct par les actions
    @vi_fresh vi_fresh = TRUE // sur Ci, la copie sur Cj est à jour
    @g [vi := Cj_vi]g // sur Cj, garde héritée avec accès aux copies
  then
    @a vj := [vi := Cj_vi;vk := local_vk]e // sur Cj, synchro avec Ck
  end
end
```

Listing 3 – Pré-traitement

La phase de projection. Cette phase construit une machine pour chaque sous-composant, et reprend le plugin de décomposition par événement partagé [7]. Les sites de projection des gardes et actions sont indiqués dans le listing 3. Notons que les invariants faisant référence à des variables distantes sont naturellement écartés.

5 La génération de code BIP

L'étape de génération de code BIP prend en entrée les composants Event-B issus de l'étape de distribution. Elle génère les éléments suivants :

Types de port. Pour chaque événement de chaque sous-composant nous générons un type de port. Les variables associées à un type de port donné sont issues des variables référencées par l'événement correspondant.

Types de connecteur. Pour chaque événement faisant référence à des variables de plusieurs composants, nous générons un type de connecteur.

Squelette de sous-composants. Pour chaque sous-composant, nous générons un composant atomique BIP. Il comporte les variables du sous-composant ainsi que les variables distantes appartenant à d'autres sous-composants, les instances des types de ports associés à ce sous-composant et, pour chaque événement, une transition synchronisée sur l'instance du port correspondant.

Le composant composite. Il regroupe une instance de chaque sous-composant et connecteur. Chaque instance de connecteur admet une instance de port appartenant aux instances de sous-composants définies précédemment.

6 Conclusion

Nous avons présenté une méthode de développement de systèmes distribués corrects par construction. À partir d'une machine Event-B représentant un modèle centralisé, nous appliquons deux types de raffinements automatiques (la fragmentation et la distribution) dont les paramètres sont déclarés par deux langages dédiés. Enfin, un modèle BIP exécutable sur une architecture répartie est produit. La correction de cette méthode repose sur la correction des étapes de raffinement et de la traduction finale en code BIP.

Nous envisageons maintenant d'améliorer l'outillage de ce processus. Les transformations de modèles Event-B sont réalisées via xtext et xtend [1]. L'étape suivante sera la finalisation du générateur BIP.

Références

[1] Language engineering for everyone ! `https://eclipse.org/Xtext`. Acc : 16-01-06.

[2] J.-R. Abrial. *Modeling in Event-B : System and Software Engineering*. Cambridge University Press, New York, NY, USA, 1st edition, 2010.

[3] A. Basu, S. Bensalem, M. Bozga, J. Combaz, M. Jaber, T.-H. Nguyen, and J. Sifakis. Rigorous component-based system design using the bip framework. *IEEE Software*, 28(3) :41–48, 2011.

[4] D. Bert, S. Boulmé, M.-L. Potet, A. Requet, and L. Voisin. Adaptable translator of B specifications to embedded c programs. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME*, volume 2805 of *LNCS*, pages 94–113. Springer, 2003.

[5] Clearsy. Bart (b automatic refinement tool). `http://tools.clearsy.com/wp-content/uploads/sites/8/resources/BART_GUI_User_Manual.pdf`.

[6] A. Edmunds and M. Butler. Tasking Event-B : An extension to Event-B for generating concurrent code. Event Dates : 2nd April 2011, February 2011.

[7] R. Silva and M. Butler. Shared event composition/decomposition in Event-B. In *FMCO Formal Methods for Components and Objects*, November 2010. Event Dates : 29 November - 1 December 2010.

[8] N. K. Singh. Eb2all : An automatic code generation tool. In *Using Event-B for Critical Device Software Systems*, pages 105–141. Springer London, 2013.