Valider un système composé de modèles indépendants

Jean-Pierre Jacquot

Valider un système composé de modèles indépendants

Jean-Pierre Jacquot
LORIA, Université de Lorraine, CNRS F-54506, Vandoeuvre lès Nancy, France

Résumé

La modélisation en B-événementiel de systèmes formés de composants indépendants, homogènes ou non, n'est pas un exercice simple. Les opérations de décomposition et recomposition de modèles sont définies théoriquement mais leur usage en pratique est difficile. RODIN offre peu de support, de même que pour les CPS. Nous décrivons ici une approche pragmatique pour valider un système modélisé en plusieurs composants indépendants en utilisant l'outil JeB. Nous proposons un protocole basé sur les *websockets* qui permet l'échange de variables entre deux modèles formels. Ce travail en cours utilise l'étude de cas du Respirateur proposée par la conférence ABZ2024.

Keywords : B-Événementiel, Validation, Simulation, Animation Multi-Modèle

1 Introduction

Les méthodes formelles visent à faciliter la production de logiciels corrects. Pour cela, les recherches sont surtout orientées vers la définition de langages, de logiques et d'outils qui permettent de les modéliser sous une forme mathématique dont on puisse prouver la correction. Avec les méthodes incrémentales comme B et B-Événementiel [Abr10], il est possible de contruire les modèles par raffinements. Des outils puissants assistent dans l'écriture et la preuve. Néanmoins, dès 1993, la question de la *validation* des modèles formels a été identifiée comme un frein à la diffusion de ces techniques chez les développeurs [Rus96, Rus93]. Parmi les différentes stratégies, l'animation des modèles est un outil intéressant pour impliquer toutes les parties prenantes dans le développement [JM18]. Dans le contexte de B-Événementiel, plusieurs outils sont proposés tels que ProB [LB08], Brama ou AnimB. La limite de ces outils automatisés tient au degré de non-déterminisme du modèle qui peut entrainer une explosion combinatoire dans les calculs. [Yan13] a proposé JeB, un environnement de simulation basé sur JavaScript qui permet d'exécuter les modèles B-Événementiel quel que soit leur niveau de raffinement. L'idée est de considérer une simulation comme une collaboration entre un code automatiquement généré qui assure que la sémantique opérationnelle est respectée et du code fourni par l'utilisateur qui résoud les éléments non-déterministes.

L'utilité de JeB a été montrée sur plusieurs études de cas, par exemple [Jac16]. Ces études de cas ont mis en évidence une autre limite des outils de validation : comment valider des systèmes comportant des composants indépendants ? Dans le cas des CPS, par exemple dans l'étude de cas proposée par la conférence ABZ2014 [BW14], le système est décrit comme un composant logiciel, le contrôleur, et des sous-systèmes matériels, les capteurs et les actuateurs hydrauliques. Le modèle formel du composant logiciel peut être exécuté et validé seul jusqu'à un certain point de son raffinement, mais il devient rapidement nécessaire d'étudier l'interaction avec les modèles des composants matériels. Des projets tels que Discont [Dis] ont étudié cette question ; des solutions ont été proposée au niveau de la modélisation [VLM21, SAZ14]. Dans le cas de la décomposition de modèle, l'impératif théorique de recomposer les modèles pour les valider en limite l'utilité.

Dans cette étude, je propose une approche pragmatique, basée sur les outils existants et sur la possibilité d'étendre de manière sûre le code généré par JeB pour exécuter un système composé de plusieurs modèles indépendants. L'objectif premier est de définir un protocole, compatible avec la sémantique opérationnelle de B-Événementiel, qui permet à plusieurs modèles d'interagir. L'objectif second est de réaliser un prototype de ce protocole pour étudier la possibilité de l'inclure dans JeB et de tester son fonctionnement. Pour le prototype, j'utilise un modèle simplifié du Respirateur proposé comme étude de cas pour la conférence ABZ2024 [BG]. Celui-ci est décomposé en deux modèles, le Contrôleur et l'Interface Utilisateur (UI), tous deux modélisés en B-Événementiel. Le protocole de communication est implanté en utilisant la technologies des *Websockets*.

2 B-Événementiel et JeB

B-Événementiel [Abr10] est une extension de la méthode B qui permet de modéliser des systèmes réactifs. Elle fait évoluer graduellement une spécification abstraite vers un modèle implantable. Chaque étape est un *raffinement*. Il introduit de nouveaux éléments, tels que des structures plus concrètes pour les données, une décomposition des comportements ou un renforcement de l'invariant.

Un modèle B-Événementiel se compose de trois parties :

— un ensemble de constantes appelé *le contexte*,
— un ensemble de variables liées par un invariant appelé *l'état* et
— un ensemble *d'événements* qui font évoluer l'état.

La correction du modèle est garantie par la génération *d'obligations de preuve* qui, lorsqu'elles sont démontrées, assurent qu'il existe au moins un état valide (c'est-à-dire pour lequel l'invariant est vérifié), que les expressions sont bien typées et que l'exécution d'un événement déclenchable sur un état valide conduira à un nouvel état valide. La spécification des valeurs est écrite à l'aide d'un langage basé sur la logique du premier ordre et la théorie des ensembles. Un événement est une *transition gardée* où la garde est un prédicat sur l'état du système et les valeurs des paramètres de l'événement. Le raffinement est une relation entre deux modèles. Il

est défini par des obligations de preuves qui assurent que le modèle raffiné peut être abstrait en le modèle initial et qu'il maintient l'invariant. La plateforme RODIN fournit des éditeurs pour le langage, les générateurs pour les obligations de preuve, des assistants de preuve pour aider à montrer les obligations générées et une API pour connecter des *plug-ins*.

La sémantique opérationnelle de B-Événementiel est intuitive : dès que le système a été mis dans un état initial, déterminer quels événements ont leur garde évaluée à `vrai`, en choisir un, l'exécuter et recommencer. En pratique, les difficultés sont de nature technique : choisir la valeur d'un paramètre d'événement pour valider une garde ou calculer une valeur définie uniquement par ses propriétés par exemple. Ci-dessous, nous avons un événement extrait du modèle du *Contrôleur*.

Event PCVParamChangeAction ⟨ordinary⟩ ≙
refines SetPCVParams
  any
    RRVn
    IERn
  where
    RRValuen: $RRVn \in 4 \mathinner{..} 50$
    IEration: $IERn \in 10 \mathinner{..} 40$
    newParams: $ParamChanging = 1$
    state: $PCVcycleState = GoToInhalePCV$
  then
    RRValue: $RRValue := RRVn$
    IER: $IEratioDen := IERn$
    changeParam: $ParamChanged := 1$
  end

L'événement *PCVParamChangeAction* a deux paramètres : *RRVn* and *IERn*.

Cet événement peut être déclenché lorsque les valeurs de *RRVn* et de *IERn* possèdent des valeurs appropriées, que le respirateur est au début d'un cycle d'inhalation et que de nouveaux réglages ont été entrés.

Après exécution, les variables dans l'état du respirateur ont reçu de nouvelles valeurs.

JeB [Yan13] permet l'exécution de modèles écrits en B-Événementiel. Il se compose de trois éléments : un *plugin* RODIN qui transforme le modèle formel en un programme JavaScript et construit un ensemble de pages HTML pour contrôler et afficher la simulation, une bibliothèque qui implante les opérateurs du langage B-Événementiel, et un *runtime* qui gère l'exécution de la boucle de la sémantique opérationnelle. Lors de la génération du programme JavaSCript, JeB construit des fichiers qui contiennent les éléments que l'utilisateur peut, voire doit, programmer à la main : `jeb_user.js` qui contient le code partagé par tous les raffinements et `<machine>_user.js` pour le code spécifique à une machine. Les éléments programmables les plus importants sont les fonctions qui calculent les paramètres (`get_<param>`) des événements.

En proposant une autre implantation pour cette fonction, il est possible de piloter la simulation en suivant des scénarios prédéfinis.

3 Principes de la communication

Les paramètres des événements peuvent être vus soit comme des variables locales lorsqu'ils sont définis par une forme d'égalité sur une valeur de l'état, soit comme des variables libres lorsqu'ils sont définis par une propriété, qui peut se limiter à un typage. La communication utilise cette dernière interprétation.

L'idée est que les modèles indépendants communiquent exclusivement par des valeurs. Certains paramètres libres sont alors des variables d'état d'un autre modèle.

Ainsi lors du cycle d'exécution, les gardes sont évaluées sur les valeurs des paramètres libres fournies par les autres modèles.

Au niveau abstrait, l'architecture globale du système n'a pas à être connue des modèles qui la composent. Il faut donc que, lors de l'exécution, un modèle puisse requérir une valeur, sans savoir quel modèle la « gère ».

Enfin, le choix de l'événement à délencher est non déterministe : c'est un de ceux dont la garde est vraie. Il n'existe ni ordre, ni garantie d'éxécution. Pour conserver cette caractéristique de la sémantique, les communications sont asynchrones : ainsi, il n'y a pas de cadencement implicite des cycles d'exécution des différents modèles.

Le protocole de communication repose sur un serveur intermédiaire qui reçoit les requêtes de valeurs, les transmet aux modèles qui les gèrent et va retourner les valeurs aux requérants. Il comporte deux phases : l'initialisation et les requêtes.

L'initialisation est réalisée par deux messages :

initSimulation est un message émis par un des modèles communicants, seule la première émission est prise compte. Son rôle est d'initialiser le serveur,

register <**v**> <**m**> est un message émis pour enregistrer une variable partagée `v`. Il est émis par le modèle `m` qui gère cette variable. Le serveur conserve l'adresse du modèle pour transmettre les requêtes.

B-Événementiel ne possède pas de mécanisme de gestion des noms ou de modularisation. De fait, les techniques de décomposition / recomposition suppose que l'espace des noms est unique et qu'un seul modèle est autorisé à modifier une variable partagée. Le protocole reprend cette contrainte.

Les communications proprement dites utilisent trois messages :

request <**v**> <**m**> est un message émis lorsque l'évaluation d'une garde utilise un paramètre correspondant à une variable externe `v`. Il est émis vers le serveur qui le retransmet directement au modèle qui gère cette variable. Le modèle requérant `m` est enregistré.

value <**v**> <**val**> est un message émis en réponse à une requête vers le serveur qui le retransmet directement aux modèles requérants.

update <**v**> est un message émis lorsqu'une variable partagée est modifiée ; il est diffusé à tous les modèles connectés.

Le message **update** est une conséquence de l'asynchronisme. Tant qu'une valeur requise n'a pas été reçue, le paramètre est considéré comme *non-défini* et la garde de l'événement correspondant est évaluée à *faux*. Comme le cycle d'exécution continue, l'évaluation des gardes peut être relancée après exécution d'un autre événement. Si la valeur n'a toujours pas été reçue, il est inutile de relancer une requête. On ne relancera une requête qu'après la réception de la valeur et sa « consommation » lors de l'exécution de l'événement. Une difficulté est soulevée par les événements tels que `PCVParamChangeAction` qui font plusieurs requêtes dont les résultats arrivent de façon asynchrone. Il ne faudra donc

« consommer » les valeurs que lorsque toutes sont disponibles. Le message **update** signale que les modèles devront relancer une requête même s'ils n'ont pas « consommé » la variable.

4 Implantation du protocole

L'implantation du protocole, dans cette version expérimentale, vise à répondre à deux questions : est-ce que le protocole fonctionne ? Quelles modifications faut-il apporter à JeB pour utiliser ce protocole ?

Techniquement, les échanges sont réalisés en utilisant la technologie des *Websockets.* JeB génère les modèles exécutables en JavaScript ; l'implantation du protocole du coté des modèles utilise donc les avantages de ce langage, en particulier la possibilité d'étendre dynamiquement les *prototypes* qui remplacent la notion de classe. Le serveur est également implanté en JavaScript.

4.1 Le serveur

Le serveur est basé sur l'outil *Node.js* qui permet de programmer et d'exécuter facilement un serveur. C'est un programme d'une centaine de lignes JavaScript.

L'état du serveur est constitué de deux tables et d'une liste :

— la table `whereTo` associe chaque variable au modèle qui la gère,
— la table `requestedBy` associe aux variables dont la valeur à été requise la pile des modèles requérants et
— la liste `connectedSockets` de tous les modèles connectés.

Le code est volontairement simple ; il est nécessaire d'évaluer et de mesurer la taille et le volume des échanges avant d'envisager des optimisations. Les messages sont composés d'une structure avec trois éléments aux maximum : le type de message, la variable concernée et la valeur de cette variable. Ils sont encodés en une chaîne JSON. Les adresses des différents modèles sont directement récupérées dans les communications *Websocket.*

4.2 Les modèles

L'enjeu principal pour les modèles est de trouver une implantation qui : (1) consiste uniquement en des ajouts au code généré par JeB et (2) ne modifie pas le comportement du modèle.

Les messages d'initialisation du protocole (**initSimulation** et **register**) sont implantés par deux fonctions qui mettent en place les éléments nécessaires : ouverture de la *socket* et préparation d'une table, appelée `waitingFor`, qui mémorise les paramètres qui ont été requis. L'appel de ces fonctions est inclus dans le code d'initialisation du graphisme. Cette solution pour le prototype manque d'élégance mais elle ne perturbe pas le comportement du modèle.

La gestion des requêtes est organisée autour de la table (`waitingFor`). La requête d'une valeur externe est émise lorsque la fonction `get_<param>` associée

est exécutée. À la première requête, une entrée dans la table `waitingFor` est créée, associée à une valeur indéfinie ; les prochaines requêtes récupéreront la valeur dans la table. Lors de la réception de la valeur, celle-ci est mise dans la table et un nouveau cycle d'évaluation des toutes les gardes est lancé. Lorsque l'événement est exécuté, les valeurs externes sont ≪ consommées ≫, c'est-à-dire que l'entrée correspondante dans la table est retirée.

Lors de la réception du message `update`, l'entrée correspondante de la table `waitingFor` est effacée.

Jusqu'ici, l'implantation du protocole utilise les fonctions offertes par JeB sans les modifier. En revanche, le message **update** et la ≪ consommation ≫ des paramètres requièrent l'ajout d'une fonction nouvelle. L'émission d'un **update** ou l'effacement des paramètres sont associés à l'exécution d'un événement. L'idée est donc d'introduire dans JeB la notion de *postAction* qui est un code exécuté lorsqu'un événement a été exécuté.

Grâce au fait que JavaScript est un langage à objet basé sur des `prototypes` et non des classes, la programmation de cette nouvelle fonction peut être réalisée directement dans les fichiers utilisateurs généré par JeB sans modifier le cœur du sytème. La figure 1a montre l'extension du prototype associé aux événements ; la figure 1b montre comment adapter la boucle d'exécution en temps réel.

```
jeb.lang.Event.prototype.postAction =
  function () {}


jeb.lang.Event.prototype.doPostAction =
    function() {
      var self = this;
        if (this.postAction != null) {
          this.postAction();
   }
}
```

(a) Extension du prototype

```
jeb.scheduler.execute = function( event ) {
    jeb.scenario.save( 'parameter' );
    event.doActions();
    event.doPostAction();
    jeb.scenario.save( 'variable', event.label );
    jeb.animator.draw();
    jeb.scheduler.checkInvariants();
};
```

(b) Scheduler

FIGURE 1 – Ajout des postActions

5 Expérimentation

5.1 Modélisation du respirateur

La figure 1.1 du cahier des charges du respirateur [BG] décrit l'architecture du système comme deux composants reliés par une liaison ≪ en série ≫. J'ai suivi une approche naïve consistant à modéliser les deux composants, l'interface graphique (UI) et le contrôleur, comme deux projets RODIN indépendants. La motivation principale pour ce choix est d'étudier empiriquement une simulation composée de deux systèmes indépendants, chacun s'exécutant dans son propre environnement.

Le modèle du contrôleur est une description directe de la machine à état présentée sur la figure 4 de [BG]. Il se compose de trois raffinement (4 machines). `Cycle0`

présente la vue la plus abstraite du respirateur avec 4 modes (démarrage, PCV, PSV et erreur). `Cycle1` raffine le mode PCV en modélisant le cycle de respiration. `Cycle2` fait la même chose pour le mode PSV. `Cycle3` introduit la communication avec l'UI. Les communications sont limitées à la modification des paramètres des modes PCV et PSV, avec un petit protocole de confirmation. `Cycle3` comporte 20 événements, dont 4 concernent la communication avec l'UI.

Le cahier des charges décrit une UI complexe. Comme notre objectif est d'étudier la communication entre les exécutions de modèles, j'ai choisi de modéliser une UI minimale. Celle-ci permet de régler 5 paramètres (ITS, ETD, RRV, IEration, ApneaLag), de déclarer un nouveau patient et de confirmer la demande de modification des valeurs. On peut imaginer 2 boutons par paramètres : un pour augmenter la valeur et un pour la diminuer. Le modèle comporte 13 événements : 12 pour les $\ll$ boutons $\gg$ et un pour la réception de la confirmation venant du Contrôleur.

La communication $\ll$ en série $\gg$ suggérée à la figure 1.1 de [BG] n'est pas explicitement modélisée. Le protocole en est une modélisation implicite.

5.2 Résultats

S'il est encore trop tôt pour donner une réponse définitive aux deux questions posées dans la section 4, les premiers essais montrent que le protocole fonctionne sur l'étude de cas.

L'implantation ne nécessite aucune modification des bibliothèques et du code généré par JeB grâce aux propriétés de JavaScript. Une observation importante est que le code ne contient que des ajouts mais aucune modification interne. Ainsi, la notion de *postAction* ne modifie pas la sémantique opérationnelle intuitive des modèles, à condition, bien évidemment, que les actions ne modifient pas l'état ! Comme le protocole ne permet que d'observer les variables, il n'a pas d'impact sur la correction des comportements observés.

6 Travaux futurs et conclusion

Au delà des premiers tests, il conviendra de valider le protocole d'un point de vue théorique et pratique.

Sur le plan théorique, il faut s'assurer qu'il est utilisable pour des systèmes plus complexes que l'exemple de ce papier. Peut-on l'utiliser lorsqu'il y a plus de composants ? Peut-on l'utiliser lorsque certains modèles ont des comportements continus ? Cette dernière question est particulièrement intéressante pour l'utilisation de B-Événementiel dans le cadre de la modélisation de systèmes hybrides comme les CPS. Dans le cadre du ventilateur, on peut imaginer avoir un modèle continu de la respiration du patient. Il faudra aussi assurer que le protocole est bien compatible avec la notion de *fidelité* [MYJ17].

Sur le plan pratique, il sera nécessaire de travailler sur l'efficacité générale. Une première amélioration consistera à doter le serveur d'un mécanisme de *cache* pour

les valeurs. Il faudra également affiner les messages et leur contenu : par exemple, pourrait-on associer les valeurs avec le message **update** ?

Enfin, pour la facilité d'usage, une intégration dans JeB est souhaitable : introduction de la notion de *postAction* dans les fichiers générés, adaptation du *scheduler,* introduction d'une fonction générale d'initialisation indépendante du graphisme et extension de la bibliothèque avec les fonctions de communication. Au vu de la facilité de réalisation du prototype, ce point ne devrait pas poser de difficultés.

Références

[Abr10] Jean-Raymond Abrial. *Modeling in Event-B - System and Software Engineering*. Cambridge University Press, 2010.

[BG] S. Bonfanti and A. Gargantini. Mechanical lung ventilator. In *Proceedings of ABZ24*.

[BW14] Frédéric Boniol and Virginie Wiels. The landing gear system case study. In Frédéric Boniol, Virginie Wiels, Yamine Ait Ameur, and Klaus-Dieter Schewe, editors, *ABZ 2014 : The Landing Gear Case Study*, pages 1–18, Cham, 2014. Springer International Publishing.

[Dis] Le projet DISCONT. http://www.agence-nationale-recherche.fr/Projet-ANR-17-CE25-0005. http ://discont.loria.fr.

[Jac16] Jean-Pierre Jacquot. Premières leçons sur la spécification d'un train d'atterrissage en B Événementiel. *Revue des Sciences et Technologies de l'Information - Série TSI : Technique et Science Informatiques*, 34(5) :549–573, 2016.

[JM18] Jean-Pierre Jacquot and Atif Mashkoor. The Role of Validation in Refinement-Based Formal Software Development. In *Models : Concept, Theory, Logic, Reasoning, and Semantics*. College Publications, 2018.

[LB08] Michael Leuschel and Michael J. Butler. ProB : an automated analysis toolset for the B method. *STTT*, 10(2) :185–203, 2008.

[MYJ17] Atif Mashkoor, Faqing Yang, and Jean-Pierre Jacquot. Refinement-based validation of Event-B specifications. *Softw. Syst. Model.*, 16(3) :789–808, 2017.

[Rus93] John Rushby. Formal methods and the certification of critical systems. TR SRI-CSL-93-7, Computer Science Laboratory, SRI International, 1993. http ://www.csl.sri.com/papers/ csl-93-7/.

[Rus96] John M. Rushby. Enhancing the utility of formal methods. *ACM Comput. Surv.*, 28(4es) :123, 1996.

[SAZ14] Wen Su, Jean-Raymond Abrial, and Huibiao Zhu. Formalizing hybrid systems with Event-B and the RODIN platform. *SCP*, 94 :164–202, 2014.

[VLM21] Fabian Vu, Michael Leuschel, and Atif Mashkoor. Validation of formal models by timed probabilistic simulation. In Alexander Raschke and Dominique Méry, editors, *Proceedings ABZ 2021*, volume 12709 of *LNCS*, pages 81–96. Springer, 2021.

[Yan13] Faqing Yang. *Un environnement de simulation pour la validation de spécifications B événementiel*. PhD thesis, 2013. Université de Lorraine.