Décomposition formelle des spécifications centralisées Event-B : application aux systèmes distribués BIP

Badr Siala

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE ET DE L'UNIVERSITÉ DE SFAX

Délivré par : l'Université Toulouse III Paul Sabatier **et**
Faculté des Sciences Économiques et de Gestion de Sfax

Discipline ou spécialité :
Informatique

Présentée et soutenue le *15 Décembre 2017* **par :**

Badr SIALA

Décomposition formelle des spécifications centralisées Event-B : application aux systèmes distribués BIP

JURY

Nadia Bouassida	Maître de Conférences, HU	ISIMS, Sfax
J. Christian Attiogbé	Professeur des Universités	Université de Nantes
Hassan Mountassir	Professeur des Universités	Université Franche-Comté
Mohamed Tahar Bhiri	Maître de Conférences, HDR	FSS, Sfax
Jean-Paul Bodeveix	Professeur des Universités	IRIT/UPS, Toulouse
Mamoun Filali	Chargé de recherche	IRIT/CNRS, Toulouse

École doctorale et spécialité :
MITT : Domaine STIC : Sûreté du logiciel et calcul haute performance
Unité de Recherche :
Institut de Recherche en Informatique de Toulouse
Directeurs de Thèse :
Jean-Paul Bodeveix, Mohamed Tahar Bhiri et *Mamoun Filali*
Rapporteurs :
Jérémie Christian Attiogbé et Hassan Mountassir

Remerciements

Au terme de ce travail de recherche, je suis convaincu que la thèse est loin d'être un travail solitaire. En effet, l'aboutissement de ce travail n'aurait été possible sans l'intervention, le soutien et la générosité de certaines personnes. Qu'elles trouvent ici l'expression de mes plus sincères remerciements.

En premier lieu, j'exprime ma reconnaissance et ma gratitude à mes directeurs de thèse : M. Jean-Paul Bodeveix, Professeur à l'université Paul Sabatier, M. Mohamed Tahar Bhiri, Maître de conférences à l'université de Sfax et M. Mamoun Filali, Chargé de recherche CNRS. Messieurs, je tiens à vous remercier pour la confiance que vous m'avez accordée en acceptant de diriger cette thèse, pour les conseils que vous m'avez prodigués et le temps que vous m'avez consacré tout au long de ces années de recherches. J'espère que vous trouverez ici l'expression de mon respect pour vos qualités humaines et scientifiques. Je vous en saurais infiniment gré.

Un grand merci à M. Jean-Paul Bodeveix pour avoir accepté de partager son bureau avec moi durant mes séjours à l'IRIT et d'avoir eu la patience de répondre à mes nombreuses questions.

J'adresse mes vifs remerciements à M. Jérémie Christian Attiogbé, Professeur à l'université de Nantes, et M. Hassan Mountassir, Professeur à l'université Franche-Comté d'avoir accepté de rapporter ma thèse. J'aimerais leur dire à quel point j'ai apprécié leur grande disponibilité et leur respect sans faille des délais serrés. Je suis prodigieusement reconnaissant de l'intérêt que vous avez manifesté pour mes travaux.

Mes remerciements vont également à Mme Nadia Bouassida, Maître de conférences à l'université de Sfax pour avoir accepté d'être examinatrice de ma thèse.

Ces remerciements seraient incomplets si je n'en adressais pas à mes amis et membres de mon équipe ACADIE à l'IRIT pour l'accueil, la bonne ambiance et l'encouragement. Des remerciements plus particuliers à Jean-Baptiste Raclet, Yamine Aït-Ameur, Meriem Ouederni, Silvano Dal Zilio et Erik Martin-Dorel.

Également, je n'oublie pas de remercier les administratifs de l'école doctorale à l'université Paul Sabatier et particulièrement Mme Agnès Requis et Mme Martine Labruyère pour leur efficacité et disponibilité et pour leur bonne humeur.

Je ne voudrais pas terminer sans remercier tous mes amis pour tout, en particulier, Boulbaba et Habib.

Je remercie ma petite famille, ma femme Sirine, mon fils Mohamed Mahdi, ma mère Essia, mon frère Riadh, ma soeur Mouna, mes neveux Mokhlis, Mohamed et Marwen et ma nièce Meriem pour tout le soutient, l'amour et l'aide qu'ils m'ont toujours apportés.

*Aux plus chers à mon coeur : ma femme Sirine, mon fils Mohamed Mahdi **"babitou"**, ma mère Essia, mon frère Riadh et ma soeur Mouna.*

♡

Table des matières

Table des figures

Listings

Chapitre 1

Introduction générale

La sûreté des systèmes informatiques est aujourd'hui une obligation implicite. En effet, la plupart de nos activités s'articulent autour d'un ou plusieurs de ces systèmes. Ces activités nous concernent aussi bien à un titre personnel, e.g., édition d'un document à l'aide d'un traitement de texte, qu'à un titre collectif : utilisation d'un système bancaire. Ces systèmes sont dans certains cas critiques : ils mettent en jeu des intérêts importants, e.g. transactions bancaires, voire vitaux (pace maker, système de transport, ...). Historiquement, la sûreté des systèmes a concerné des systèmes de taille modeste. Les systèmes embarqués tels que commande de vol d'un avion, logiciel de contrôle d'un train sont parmi les premiers systèmes pour lesquels le concept de sûreté a fait l'objet d'une définition précise, de critères qualifiant la sûreté de développement des logiciels [1]. Par la suite, cette notion de sûreté de logiciel, a été généralisée et a abouti à la définition de normes appelées *critères communs*. Ces normes sont internationalement reconnues. Elles associent un degré de confiance à l'évaluation d'un produit. A titre d'exemple, les EAL (Evaluation Assurance Level) [1] introduisent une hiérarchie d'évaluation à 7 niveaux. L'EAL 1 est le niveau le plus faible et repose principalement sur une justification informelle de la conformité du produit par rapport à sa description dans la documentation. L'EAL 7 est le niveau le plus haut. Il introduit l'utilisation de spécifications formelles, ainsi qu'un processus de développement pour justifier formellement la satisfaction des propriétés requises. Notre travail se situe à ce niveau dans la mesure où nous nous appuierons principalement sur la méthode formelle de développement de logiciel B et de sa version "contemporaine" Event-B [8].

Dans le cadre de cette méthode, nous nous intéressons au développement de systèmes réactifs et plus particulièrement de systèmes répartis. Pour nous, ces systèmes se caractérisent par leur nature répartie qu'il est important de prendre en compte lors du processus de développement. A ce titre, elle est à la charge du développeur. Cette distribution peut être appréhendée à deux niveaux : au niveau contrôle où il s'agit d'exprimer le passage d'un modèle initialement centralisé à un modèle réparti : nous utiliserons le terme de *fragmentation*, et au niveau spatial où il s'agit de placer les fragments précédemment distingués sur les unités de localisation de l'architecture répartie sous-jacente : nous utiliserons le terme de *distribution*. Cette vision de la distribution repose sur l'existence d'un exécutif distribué. Pour cela, nous avons utilisé l'approche BIP [16] et son environnement.

Ainsi, *Notre première contribution a consisté à définir une façon de prendre en compte la distribution au sein de la méthode Event-B*. Il s'agit là d'une étape intéressante pour la production de code réparti correct par construction par raffinement.

Toujours dans le cadre de la méthode Event-B, nous nous sommes aussi intéressés à

1. Nous nous intéressons au développement de logiciels qu'il convient de distinguer de son fonctionnement. Plus précisément, lors du développement, nous faisons des hypothèses sur le fonctionnement futur que nous supposerons vérifiées.

l'environnement de développement. Dans la mesure du possible, nous avons intégré notre approche de la distribution à l'environnement standard Event-B : Rodin [10] basé sur la plateforme de développement logiciel Eclipse. Pour cela, nous avons utilisé les outils "avancés" de développement logiciels disponibles au sein de cette plateforme. *Notre seconde contribution a consisté à réaliser un ensemble de plug-ins destinés à une mise en oeuvre intégrée de la répartition.* Il est à noter que ces transformations sont certes automatisées mais sont guidées par l'utilisateur. Enfin, on notera que l'annotation de ces transformations a permis que les machines automatiquement générées peuvent être relues par l'utilisateur et éventuellement raffinées.

Ces contributions sont exposées dans les chapitres suivants :

État de l'art. Ce chapitre présente le contexte dans lequel se situe notre travail. Ainsi, nous présenterons le langage BIP et plus généralement l'approche BIP, qui nous servira d'environnement de développement de systèmes répartis. Nous nous intéresserons aussi bien à son modèle de composant ainsi qu'à la sémantique de ses différents concepts. Nos travaux s'appuient sur cette sémantique. Ensuite, nous présentons le langage Event-B qui nous servira de cadre de développement formel par raffinement. Nous portons une attention plus particulière à la composition et à la décomposition en Event-B. En effet, d'une part, ces deux notions ont donné lieu à des propositions récentes [71, 72, 74, 75]. Et, d'autre part, nos travaux s'inscrivent dans la continuité de ces propositions.

Démarche de développement de systèmes distribués. Dans ce chapitre, nous élaborons la démarche de développement de systèmes distribués préconisée. Pour cela, nous présentons tout d'abord les hypothèses d'utilisation de l'environnement BIP ainsi que les rudiments de distribution de machines Event-B. Enfin, nous présentons les trois étapes qui constituent notre démarche de développement de systèmes distribués : la fragmentation, la distribution et la génération de code distribué BIP. La fragmentation introduit des événements permettant d'évaluer les paramètres des événements de synchronisation traités. La distribution introduit des composants et le placement des variables et gardes sur ces composants. Enfin, la génération de code prend en compte le modèle d'exécution BIP et élabore un code pour cette cible.

Étude de cas : Système de clés électroniques pour les hôtels. Dans ce chapitre, nous reprenons une étude de cas précédemment développée dans la communauté des méthodes formelles [45, 55]. Nous l'étudions plus particulièrement sous l'angle de la répartition. Cette étude a été pour nous l'occasion de préciser un processus d'élicitation des exigences dédié pour la répartition. Par suite, l'énoncé adapté de ces exigences a permis de faciliter l'expression de la fragmentation et de la distribution et enfin d'aboutir à une implantation distribuée exprimée en BIP.

Chapitre 2

État de l'art

Sommaire

2.1 Introduction

Le paradigme distribué est de plus en plus utilisé en informatique. Mais le développement des systèmes distribués demeure un challenge pour la plupart des informaticiens. Dans un premier temps, nous allons cerner les aspects principaux des systèmes distribués et présenter l'application d'Hôtel [8, 45] utilisée comme support pour cette thèse. Dans un deuxième temps, nous étudierons d'une façon approfondie le langage BIP [15, 70] en tant que langage de modélisation et programmation des systèmes à base de composants. Également, nous apporterons une évaluation empirique du langage BIP. Dans un troisième temps, nous aborderons la méthode formelle Event-B [8] notamment celle orientée composant [26]. Un soin particulier lié aux outils de vérification et validation formelle autour de la plateforme Rodin [9, 10] sera accordé. L'application de l'Hôtel sera modélisée aussi bien en BIP qu'en Event-B orientée composant.

2.2 Systèmes distribués

2.2.1 Généralités

En informatique, trois paradigmes d'exécution sont bien établis : séquentiel, concurrent et distribué. Les systèmes distribués [78] sont au coeur d'enjeux industriels : systèmes embarqués temps-réel, orchestration des services Web, protocoles de communication, algorithme d'élection d'un leader, vote électronique, etc. Un système distribué comporte un ensemble d'agents. Ces derniers **coopèrent** afin d'atteindre un objectif global bien défini. Un agent, pris individuellement, est censé exécuter un programme séquentiel sur son ordinateur.

La coopération entre les agents formant le système distribué est primordiale. Elle traduit les **interactions** définies inter-agents. De telles interactions expriment des choix architecturaux ou contraintes liés à l'application distribuée.

Le développement des systèmes distribués est une tâche ardue. En effet, la plupart des informaticiens éprouvent des difficultés importantes pour spécifier, concevoir, implémenter, déployer et faire évoluer des systèmes distribués. En outre, l'obtention des systèmes distribués **fiables** demeure un défi. Les erreurs fréquentes et difficilement détectables relatives aux programmes distribués sont : **interblocage**, **famine** et **panne** notamment celle de type byzantin.

Au milieu des années 90, la communauté d'architecture logicielle a développé plusieurs langages appelés ADL (Architecture Description Language) [38]. Ces derniers offrent des constructions permettant de décrire les **aspects architecturaux** du futur système. Les

ADL s'appuient sur divers modèles de composants. Cependant, ils introduisent des concepts architecturaux communs comme composant (atomique et composite), interface, connecteur (atomique et composite), configuration et style. En outre, ces ADL sont souvent dotés des langages permettant de décrire les **aspects comportementaux** : PSM, algèbre de processus, automate et réseau de Petri.

Les ADL peuvent être utilisés avec profit afin de **modéliser** les systèmes distribués en se servant judicieusement des aspects architecturaux et comportementaux offerts par ces ADL. De plus, certains ADL comme Wright [12] permettent par model-checking [64] de vérifier la cohérence d'un composant, absence de blocage d'un connecteur et compatibilité entre composant et connecteur.

2.2.2 Étude de cas support

Nous décrivons ici l'étude de cas qui servira de support à la présentation des langages sur lesquels reposent les travaux décrits dans ce document. Il s'agit d'un système de clé électronique pour hôtels [8, 45] dont les contraintes sont les suivantes :

— Chaque porte d'une chambre d'hôtel est équipée d'une serrure électronique indépendante contenant une clé électronique. La serrure est dotée d'un lecteur permettant d'insérer une carte électronique.

— Une nouvelle réservation (check-in) donne naissance à l'édition d'une carte magnétique. Celle-ci comporte deux clés électroniques : `k1` et `k2` (voir la figure 2.1). La clé `k1` devrait être identique à la clé mémorisée par la serrure de la chambre concernée. La clé `k2` est utilisée pour mettre à jour la serrure lors de la première entrée dans la chambre. Les deux clés `k1` et `k2` doivent être différentes.

— Les clients de l'hôtel ne sont pas censés rendre les cartes électroniques lors de la sortie (check-out).

— A l'entrée dans la chambre, le client insère sa carte dans le lecteur de la serrure. Celui-ci lit la carte insérée et ouvre la porte à condition que sa propre clé électronique soit l'une des deux clés de la carte insérée. Une mise à jour de la clé portée par la serrure (elle devient `k2`) est effectuée lors de la première entrée dans la chambre.

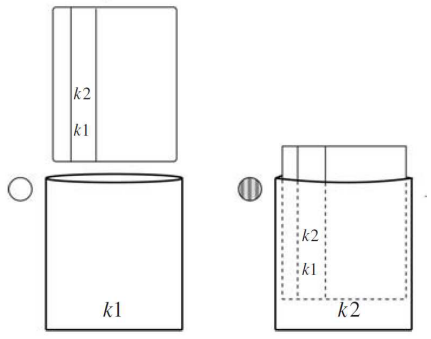

FIGURE 2.1 – Une carte insérée dans une serrure de porte [8]

2.3 BIP

Dans cette section, nous allons présenter et évaluer les aptitudes du langage BIP considéré comme ADL pour la modélisation des systèmes distribués.

2.3.1 Le modèle de composants BIP

Le langage BIP (Behavior, Iteraction, Priority) [15, 70] est un langage orienté composant permettant la modélisation et la programmation des systèmes complexes. Il introduit un modèle de composants permettant la manipulation des composants et connecteurs aussi bien au niveau design-time (types de composants et connecteurs) qu'au niveau runtime (instances de composants et connecteurs). Également, le modèle de composants supporte les composants composites et connecteurs hiérarchiques. Le langage BIP induit une approche **ascendante** de développement des systèmes à base de composants. Un système BIP peut être décrit par une arborescence dont les noeuds terminaux correspondent aux composants atomiques (**atom** en BIP), les noeuds non terminaux correspondent aux composants composites (**compound** en BIP) et la racine correspond à l'application. Ainsi, une application décrite en BIP est représentée (voir la figure 2.2) par :

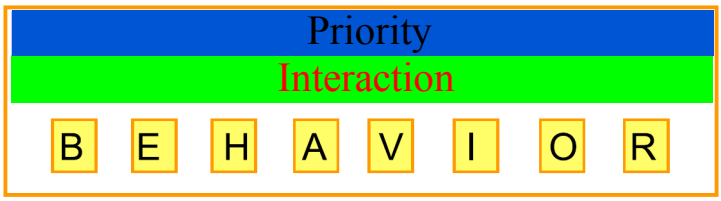

FIGURE 2.2 – Architecture de BIP

— Le comportement (**Behavior**) spécifié par un ensemble de composants atomiques et composites. Le comportement d'un composant atomique est exprimé par un automate ou un réseau de Petri. Et, à terme, le comportement des composants composites est obtenu par la composition des composants atomiques.
— Les interactions (**Interaction**) sont spécifiées à l'aide de connecteurs (**connector** en BIP). Elles définissent les synchronisations et communications possibles entre les composants.
— Les priorités (**Priority**) permettent la description des politiques d'ordonnancement des interactions.

Dans la suite, nous nous limitons aux deux couches Behavior et Interaction.

2.3.1.1 Les composants atomiques

Un composant atomique BIP est décrit à l'aide de la construction **atom type**. Celle-ci comporte des variables typées (**data**) en utilisant les possibilités de typage fournies par le langage hôte C++. Ces variables sont utilisées pour stocker les données locales du composant atomique. De plus, un composant atomique comporte un ensemble de portes (**port**). Une porte BIP possède un nom et peut être bâtie sur plusieurs variables. Elle sert pour la synchronisation et la communication avec d'autres composants en passant par des connecteurs. Les variables et les portes forment l'interface d'un composant atomique BIP. Le comportement d'un composant atomique BIP est décrit par un automate comportant un ensemble d'états de contrôle (**place** en BIP) et un ensemble de transitions. Une transition modélise le changement d'un composant d'un état de contrôle vers un autre. Une garde (simple ou élaborée) et action (simple ou séquence) sont associées à une transition. La garde est une condition booléenne exprimée en utilisant les opérateurs fournis par C++. L'action d'une transition modélise le calcul exécuté lors du franchissement de ladite transition. Elle est exprimée par une séquence d'instructions issues du langage C++ comportant : l'affectation, l'appel d'un sous-programme, la séquentialité et la structure conditionnelle if/then/else.

BIP distingue trois types de transitions :

— Une transition appelée **initial** permettant d'initialiser l'état de contrôle et les variables locales. Une telle transition ne possède ni garde ni région de contrôle. Elle ne peut pas être observée par les autres composants.

— Une transition appelée **internal**. Elle est invisible par les autres composants. Elle est prioritaire sur les transitions étiquetées par les portes (**port**).

— Une transition étiquetée par une porte (**port**). Elle est visible par les autres composants via des connecteurs.

Notons au passage, qu'un composant atomique BIP peut avoir des paramètres qui sont des variables simples au sens du langage C.

A titre d'exemple, le listing 2.1 est un composant atomique BIP qui modélise un processus producteur (modèle producteur/consommateur). Il exporte deux portes `prod`, `work` et définit deux états de contrôle `PRODUCE` et `WORK`. L'état initial du composant est l'état `PRODUCE`. A partir de l'état initial, le composant change d'état via les interactions sur ces deux portes (`prod` et `work`). La figure 2.3 décrit la représentation graphique du composant `Producer`.

```
port type ePort()          // Porte sans données
port type intPort(int i)   // Porte associée à une donnée entière

atom type Producer()
  data int x
  export port intPort prod(x)
  export port ePort work()

  place PRODUCE, WORK

   initial to PRODUCE do {x=10;}
  on prod from PRODUCE to WORK do {printf("Sending %d\n", x);}
  on work from WORK to PRODUCE do {x = x + 1;}
end
```

Listing 2.1 – Description du composant atomique Producer

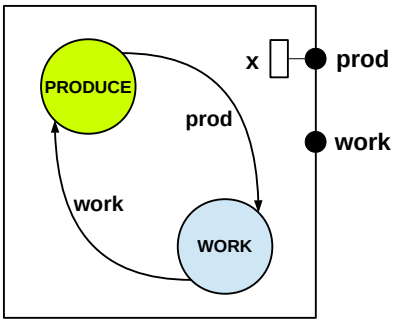

FIGURE 2.3 – Représentation graphique du composant `Producer` en BIP

2.3.1.2 Les connecteurs atomiques

Un connecteur atomique BIP est une entité sans état (stateless, contrairement à un composant atomique) permettant l'**interaction** des composants via des portes. A l'instar des composants, un connecteur BIP peut être défini aussi bien au niveau type (**connector type**) qu'au niveau instance (**connector**). Les constituants d'un connecteur sont :

— les portes issues des composants à faire communiquer ;

— les variables temporaires (**data**) ayant une durée de vie égale à celle de l'interaction ;

— une interface comportant des portes bâties sur les variables temporaires ;

— la nature des interactions autorisées : diffusion et rendez-vous (**define**). Sachant que les portes primées sont des portes déclencheurs et les portes non primées sont des portes de synchronisation ;

— les interactions possibles. Sachant qu'une interaction comporte les portes concernées (**on**), une garde (**provided**) et les fonctions de transfert (**up** et **down**) permettant l'échange de données entre les composants qui participent à ladite interaction.

Le listing 2.2 décrit deux connecteurs `SendRec` et `Singleton` qui autorisent une interaction rendez-vous. Le connecteur `Singleton`, est paramétré par une porte de type `ePort` et le connecteur `SendRec` est paramétré par deux portes de type `intPort`. La figure 2.4 est la représentation graphique d'un tel connecteur.

```
connector type Singleton(ePort p) // Singleton
 define p
end

connector type SendRec(intPort s, intPort r) // Rendez-vous entre s, r
  define s r
  on s r
    down {r.i = s.i;}
end
```

Listing 2.2 – Description d'un connecteur atomique `SendRec`

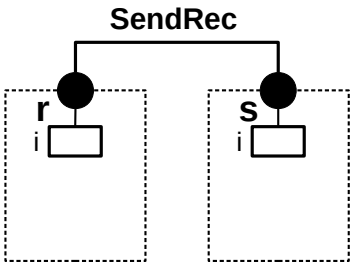

FIGURE 2.4 – Représentation graphique du connecteur `SendRec` en BIP

2.3.1.3 Les composants composites

Le langage BIP fournit une construction appelée **compound** permettant de composer aussi bien des composants atomiques que des composants composites. Les composants composites BIP peuvent être définis au niveau type (**compound type**) qu'au niveau instance (**compound**). Les constituants d'un composant composite BIP sont :

— un ensemble de composants atomiques ou composites déclarés avec le mot-clef **component** ;

— un ensemble de connecteurs annoncés avec le mot-clef **connector**. De tels connecteurs permettent de faire communiquer les composants formant le composant composite concerné ;

— l'interface du composant composite déclarée avec le mot-clef **export**. Une telle interface regroupe les portes exportées.

Le listing 2.3 montre un exemple de composant composite nommé `ProdCons`. Il comporte une instance du composant atomique `Producer`, une instance du composant atomique `Consumer` et trois connecteurs de type rendez-vous. Deux connecteurs de type `Singleton` paramétrés par une porte de type `ePort` et un connecteur paramétré par deux portes de

type `SendRec` nommé `Communicate`. La représentation graphique du composant composite `ProdCons` est donnée dans la figure 2.5.

```
compound type ProdCons()
  component Producer P()
  component Consumer C()

  connector SendRec Communicate(P.prod, C.cons)
  connector Singleton  P_work(P.work)
  connector Singleton  C_work(C.work)
end
```

Listing 2.3 – Description d'un composant composite `ProdCons`

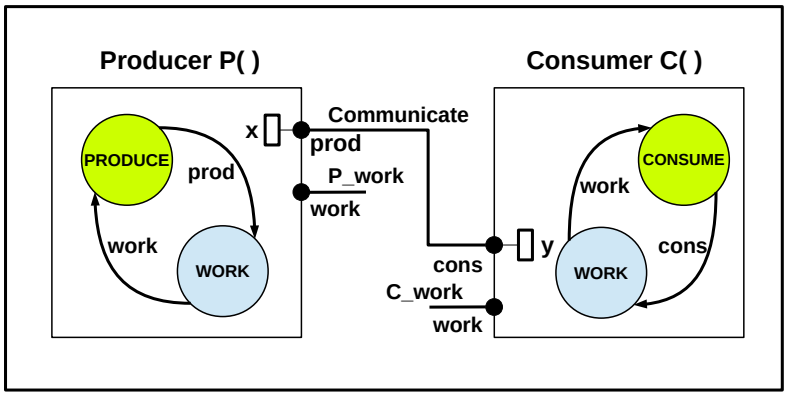

FIGURE 2.5 – Représentation graphique du composant composite `ProdCons` en BIP

Notons au passage, qu'un composant composite BIP peut être aplati en un composant atomique via une transformation source à source. Une telle transformation remplace la hiérarchie des composants par un ensemble de connecteurs hiérarchiquement structurés pour interconnecter les composants atomiques [23]. L'outil BIP2BIP [23] compose le comportement des différents composants formant le composant composite BIP pour générer un composant atomique équivalent.

2.3.1.4 Les connecteurs hiérarchiques

Un connecteur hiérarchique BIP exporte une porte sur laquelle d'autres connecteurs peuvent être attachés. Cette porte peut donner accès à des données locales au connecteur. Celles-ci sont alors synthétisées par l'action **up** à partir des données provenant des sous-composants. L'action **down** du connecteur parent peut alors mettre à jour les données du connecteur courant qui activera alors son action **down**. Il est à noter que les gardes des composants ou des connecteurs sont évaluées lors de la phase ascendante, c'est-à-dire avant l'écriture de données via la porte. Ce principe est illustré par le listing 2.4 où chaque instance de composant effectue une transition vers son état final si la valeur `v` qu'il possède est minimale parmi les valeurs des composants n'ayant pas encore terminé. Il la compare pour cela avec la valeur minimale `m` calculée par la hiérarchie de connecteurs lors de la synchronisation sur la porte `getMin`. La figure 2.6 donne la structure de la composition et illustre les transferts de données réalisés par les diverses actions **up** (flèches montantes) et **down** (flèches descendantes ou racine) lors de la synchronisation.

```
@cpp(include="stdio.h", include="stdlib.h")

package min
  port type intPort(int v, int m)
```

```
  extern function printf(string, int)
  extern function int min(int, int)
  extern function int rand()

atom type Composant()
    data int v
    data int m
    export port intPort getMin(v,m)
    place Start, Test, End
     initial to Start do { v = rand(); m = 0; }
    on getMin from Start to Test
     internal from Test to End provided (v==m) do { printf("%d\n", v); }
     internal from Test to Start provided (v>m) do { m = 0; }
end

connector type cMin(intPort p1, intPort p2)
    data int v
    data int m
    export port intPort getMin(v,m)
    define p1' p2'
    on p1 p2
      up { v = min(p1.v, p2.v); }
      down { p1.m = m; p2.m = m; }
    on p1 up { v = p1.v; } down { p1.m = m; }
    on p2 up { v = p2.v; } down { p2.m = m; }
end


connector type root (intPort p)
    define p
    on p down { p.m = p.v; }
end

compound type Sort()
    component Composant c1()
    component Composant c2()
    component Composant c3()
    component Composant c4()
    connector cMin cm12(c1.getMin,c2.getMin)
    connector cMin cm34(c3.getMin,c4.getMin)
    connector cMin cm(cm12.getMin,cm34.getMin)
    connector root R1(cm.getMin)
  end

end

component Sort main()
```

Listing 2.4 – Connecteur hiérarchique

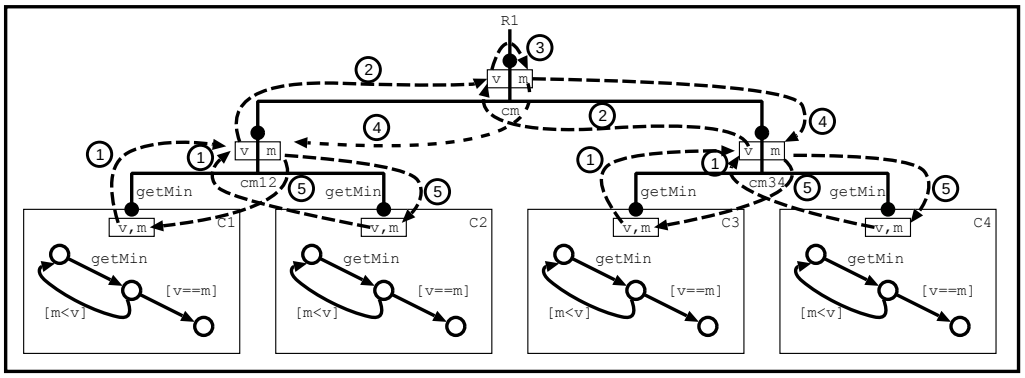

FIGURE 2.6 – Représentation graphique de connecteurs hiérarchiques en BIP

L'outil BIP2BIP [23] permet d'aplatir un connecteur BIP hiérarchique en un connecteur atomique N-aire. En effet, le connecteur hiérarchique `cMin` (voir le listing 2.4) peut être aplati

en un connecteur atomique quaternaire.

2.3.2 Sémantique

Nous décrivons ici la sémantique d'un modèle BIP, tout d'abord de manière informelle, puis de façon plus formelle via l'introduction d'une syntaxe abstraite pour les deux constituants principaux d'un modèle BIP que sont les composants et les connecteurs.

2.3.2.1 Sémantique informelle

L'exécution d'une application BIP comporte les étapes suivantes :

Étape 1 : Initialisation. Exécution des composants atomiques comme des processus

Étape 2 : Situation stable. Attente des composants atomiques pour qu'ils atteignent une situation stable afin de les coordonner : leurs exécutions sont suspendues lorsque les transitions font référence à des portes.

Étape 3 : Interactions exécutables. Cette étape a pour objectif le calcul des interactions exécutables dites encore légales : les composants associés aux portes de l'interaction sont prêts à exécuter une transition sur cette porte et la garde associée à l'interaction est vérifiée.

Étape 4 : Ordonnancement des interactions exécutables. L'ordonnancement des interactions exécutables se fait en tenant compte des contraintes de priorité : une interaction est conservée si aucune interaction de plus grande priorité n'est légale.

Étape 5 : Exécution de l'interaction choisie. L'exécution d'une interaction se traduit par l'exécution de deux fonctions de transfert **up** (transfert de données des portes des composants participant à l'interaction vers la porte de sortie) et **down** (transfert de données vers les composants participant à l'interaction).

Étape 6 : Évolution des composants atomiques. Les transitions des composants atomiques concernés par l'interaction sont exécutées.

Étape 7 : Aller en **Étape 2**.

2.3.2.2 Expression formelle de la sémantique

Dans cette section, nous donnons une sémantique opérationnelle de la composition d'un ensemble de composants BIP $(C_i)_{i\in 1..n}$ connectés par un ensemble de connecteurs γ. Tout d'abord, nous résumons la syntaxe des composants et des connecteurs comme suit :

— $C_i = \langle \Sigma_i, P_i, X_i, \rightarrow_i \rangle$ avec Σ_i sont les états de C_i, P_i l'ensemble de ces portes, X_i l'ensemble de ces variables, $\mathcal{G}(X_i)$ est un ensemble de prédicats sur les X_i, $\mathcal{A}(X_i)$ est un ensemble prédicats sur X_i et $\rightarrow_i \subseteq \Sigma_i \times P_i \times \mathcal{G}(X_i) \times \mathcal{A}(X_i) \times \Sigma_i$ ces transitions étiquetées par une garde et une action. Nous écrirons $\sigma_i \xrightarrow{p_i/g_i/a_i}_i \sigma_i'$ pour un élément de $\rightarrow_i$.

— $\gamma \subseteq \{\langle I \subseteq 1..n, (p_i(x_i))_{i\in I} \in \Pi_{i\in I} P_i(X_i), p, G, (D_i)_{i\in I}, U\rangle\}$ est un ensemble de connecteurs, I est l'ensemble des index des composants en interaction, $(p_i(x_i))_{i\in I}$ l'ensemble sélectionné de portes (une dans chaque composant) avec leur vue x_i sur les variables d'un composant, p la porte sortante, G la garde du connecteur, D_i l'ensemble des fonctions de transfert de type *down* spécifiant la mise à jour des états des sous-composants et U les fonctions de transfert de type *up* spécifiant les données de la porte sortante.

La sémantique opérationnelle de la composition (voir la règle d'inférence (2.1)) est définie par les transitions suivantes sur les états et les valuations des variables v_i du composant. Elle peut être lue comme suit :

— Sélectionner un connecteur de l'ensemble γ,

— Vérifier que chaque composant sélectionné a une transition sur la porte associée au connecteur,

— Vérifier que les gardes des transitions et du connecteur sont satisfaites,

— Exécuter les actions des transitions après l'exécution des actions du connecteur définis dans dans la fonction *down* ,

— Ne pas mettre à jour l'état des composants qui ne participent pas à l'interaction sélectionnée,

— Émettre un événement marqué par la porte du connecteur et le résultat de la fonction `up`.

Un connecteur est sélectionné sur les portes activées. Les actions de la fonction *down* D_i du connecteur sont exécutées avant l'exécution des actions locales a_i de chaque composant.

$$\frac{\begin{array}{c}\langle I, (p_i)_{i\in I}, p, G, (D_i)_{i\in I}, U\rangle \in \gamma \\ \bigwedge_{i\in I} \sigma_i \xrightarrow{p_i/g_i/a_i}_i \sigma'_i \ \wedge \ \bigwedge_{i\notin I} \sigma'_i = \sigma_i \\ (\bigwedge_{i\in I} g_i(v_i)) \ \wedge \ G(\langle x_i \triangleleft v_i \mid i \in I\rangle) \\ \bigwedge_{i\in I} v'_i = a_i(v_i \ \lhd\!\!+ \ D_i(\langle x_j \triangleleft v_j \mid j \in I\rangle)) \ \wedge \ \bigwedge_{i\notin I} v'_i = v_i\end{array}}{\langle(\sigma_1, v_1), \ldots, (\sigma_n, v_n)\rangle \xrightarrow{p(U(\langle x_i \triangleleft v_i \mid i\in I\rangle))} \langle(\sigma'_1, v'_1), \ldots, (\sigma'_n, v'_n)\rangle} \tag{2.1}$$

Pour des raisons de lisibilité, les priorités ne sont pas prises en compte. Nous devrions ajouter que l'interaction tirée n'est pas cachée par des interactions prêtes ayant une priorité inférieure.

2.3.3 L'environnement BIP

2.3.3.1 Présentation

L'environnement de développement BIP (voir la figure 2.7) comporte un langage de modélisation et de programmation des systèmes à base de composants et des outils d'édition et d'analyse lexico-syntaxique et sémantique de programmes BIP. En outre, il fournit une plateforme dédiée à la validation des modèles BIP. Une telle plateforme englobe un moteur d'exécution (voir la section 2.3.3.2) d'un programme BIP [22]. De plus, elle permet de générer l'espace d'états lié au programme BIP afin de l'analyser par des outils de validation formelle. Ainsi, un concepteur de modèles BIP peut utiliser des model-checkers externes comme Evaluator [50, 51] afin de vérifier des propriétés temporelles sur son modèle. De même, il peut utiliser des model-checkers internes comme D-Finder [17] et D-Finder2 [18] spécialisés dans la détection des erreurs de blocage. Enfin, il peut se servir du model-checker VCS [40] acceptant en entrée un modèle BIP et des propriétés temporelles décrites en CTL (Computation Tree Logic). Le model-checker VCS permet de spécifier les interactions avec transfert des données inter-composants BIP.

Cependant, pour des systèmes complexes, l'utilisation de ces model-chechers est limitée. En effet, l'exploration exhaustive de l'espace d'états du modèle conduit à un problème d'explosion d'états qui reste non résolu. C'est ainsi que l'environnement de développement BIP est équipé d'une méthode de vérification à l'exécution (Runtime verification) [35] qui permet l'analyse dynamique de systèmes. En effet, pour vérifier les spécifications à l'exécution, les systèmes BIP peuvent être augmentés par des moniteurs. Ces derniers permettent la validation de ces systèmes en utilisant l'outil de vérification RV-BIP [34].

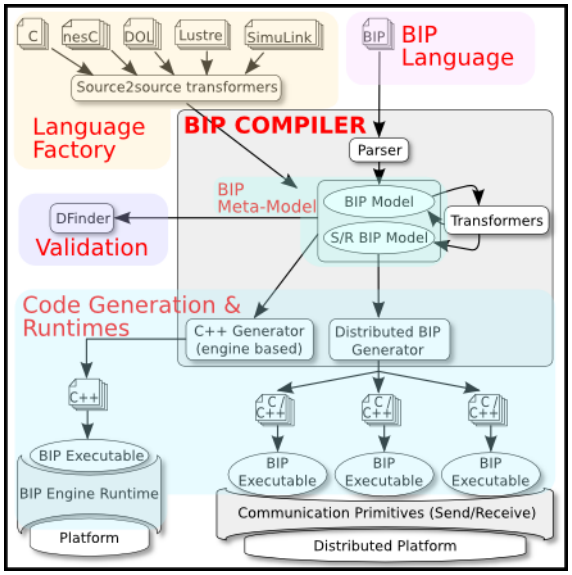

FIGURE 2.7 – Chaîne d'outils BIP [4]

2.3.3.2 Moteur d'exécution

Le moteur d'exécution de BIP [22] démarre avec le calcul des interactions exécutables (couche Interaction d'un modèle BIP). Ensuite, il ordonnance ces interactions en tenant compte des contraintes de priorité (couche Priority d'un modèle BIP) afin de choisir l'interaction à exécuter. Enfin, les transitions des composants atomiques concernés par l'interaction sont exécutées (couche Behavior d'un modèle BIP).

Il existe trois implantations différentes pour le moteur d'exécution de BIP [44] : moteur énumératif centralisé, moteur symbolique centralisé et moteur d'exécution distribuée.

Les générateurs de code exécutable (C/C++) sont de deux types [44] :

— single-threaded : il supporte deux types d'exécution : effectuer l'exploration exhaustive ou lancer une exécution.

— multi-threaded : il permet de stocker les états des composants et de surveiller l'espace d'états global nécessaire pour la simulation exhaustive.

2.3.4 Étude de cas en BIP

Dans cette section, nous allons modéliser en BIP le système de clés électroniques pour les hôtels introduit dans la section 2.2.2.

2.3.4.1 Programme BIP proposé

Le programme BIP (voir le listing 2.5) permettant de modéliser et programmer l'application de l'Hôtel comporte :

— deux types de composants atomiques : `Chambre` et `Client` ;

— deux types de connecteurs atomiques : `client_chambre` et `synchronisation` ;

— un type de composant composite : `hotel`.

Le type de composant `Chambre` (voir les lignes de 5 à 26 du listing 2.5) factorise les éléments communs aux chambres. Il encapsule des données locales (**data**) permettant de mémoriser la clé portée par la serrure (`cle_serrure`) et la carte électronique introduite dans le

lecteur (`premier` et `deuxieme`). Les transitions `passer_carte`, `attribuer`, `repasser_carte` et `quitter` modélisent respectivement les événements l'introduction d'une carte dans le lecteur d'une chambre, le premier accès à une chambre, les accès suivants à la chambre et le fait de quitter l'hôtel. L'effet de ces transitions est exprimé notamment par le changement de la région de contrôle (**place**) du composant de type `Chambre`. La transition `passer_carte` est censée **récupérer des données** venant de son environnement. Les deux transitions `quitter` et `repasser_carte` sont en **compétition**. En effet, la condition de déclenchement de ces deux transitions est la même : **from** `chambre_attribuee`. Enfin, le type de composant `Chambre` est paramétré par la clé portée par la serrure lors de l'instanciation des entités de type `Chambre` (voir le composant composite `hotel`, ligne 58 du listing 2.5).

Le type de composant `Client` (voir les lignes de 28 à 46 du listing 2.5) possède une structure similaire à celle de `Chambre`. Il est paramétré sur la carte électronique contenant deux clés : `cle1` et `cle2`. Ces cartes sont attribuées par le composant composite `hotel` lors de l'instanciation des entités de type `Client`. La transition `prendre_carte` est censée **transmettre des données** à son environnement. Le type de connecteur `client_chambre` (voir les lignes de 48 à 51 du listing 2.5) permet de connecter deux composants exportant deux portes de type `carte`. Il propose une interaction de type rendez-vous avec échange de données. Une telle interaction est gardée et effectue une copie d'une porte émettrice vers une porte réceptrice.

Le type de connecteur `synchronisation` (voir les lignes de 53 à 55 du listing 2.5) permet de connecter deux composants exportant deux portes de type `sans_variable`. Il propose une interaction de type **rendez-vous pur** (sans échange de données).

Le type de composant composite `hotel` (voir les lignes de 57 à 76 du listing 2.5) permet d'instancier des composants de type `Chambre` et `Client`. Également, il instancie des connecteurs de type `client_chambre` et `synchronisation`. La configuration introduite par `hotel` est fournie par la figure 2.8.

```
1  package Hotel
2    port type carte(int k1, int k2)
3    port type sans_variable()

5    atom type Chambre(int cle)
6      // Interface
7      // variables
8      data int cle_serrure  //cle de la serrure
9      data int premier   //premiere cle de la carte
10     data int deuxieme  //deuxieme cle de la carte

12     export port carte passer_carte(premier, deuxieme)
13     export port sans_variable attribuer()
14     export port sans_variable quitter()
15     export port sans_variable repasser_carte()

17     //places
18     place dispo, carte_valide, chambre_attribuee
19      initial to dispo do {cle_serrure=cle;}
20     on passer_carte from dispo to carte_valide
21     on attribuer from carte_valide to chambre_attribuee
22           provided((cle_serrure==premier)||(cle_serrure==deuxieme))
23            do {if(cle_serrure==premier) then cle_serrure=deuxieme; fi}
24     on repasser_carte from chambre_attribuee to carte_valide
25     on quitter from chambre_attribuee to dispo
26   end

28   atom type Client(int cle1, int cle2)
29     // Interface
30     // variables
31     data int premier  //premiere cle de la carte
32     data int deuxieme //deuxieme cle de la carte

34     export port carte prendre_carte(premier, deuxieme)
```

```
35      export port sans_variable attribuer ()
36      export port sans_variable quitter ()
37      export port sans_variable repasser_carte ()

39      //places
40      place pas_de_carte, carte_initialisee , carte_a_valider
41       initial to pas_de_carte do {premier=cle1; deuxieme=cle2;}
42      on attribuer from carte_initialisee to carte_a_valider
43      on quitter from carte_a_valider to pas_de_carte
44      on repasser_carte from carte_a_valider to carte_initialisee
45      on prendre_carte from pas_de_carte to carte_initialisee provided (premier !=
            deuxieme)
46    end

48    connector type client_chambre(carte c, carte r)
49      define c r
50      on c r provided (c.k1 != c.k2) down {r.k1=c.k1; r.k2=c.k2;}
51    end

53    connector type synchronisation (sans_variable p1, sans_variable p2)
54          define p1 p2
55    end

57    compound type hotel()
58      component Chambre r1(10), r2(20), r3(30)
59      component Client c1(10,60), c2(20,80), c3(30,90)

61      connector client_chambre c1_r1(c1.prendre_carte, r1.passer_carte)
62      connector client_chambre c2_r2(c2.prendre_carte, r2.passer_carte)
63      connector client_chambre c3_r3(c3.prendre_carte, r3.passer_carte)

65      connector synchronisation attribuer_c1_r1(c1.attribuer ,r1.attribuer )
66      connector synchronisation quitter_c1_r1(c1.quitter ,r1.quitter )
67      connector synchronisation repasser_carte_c1_r1(c1.repasser_carte ,r1.
            repasser_carte )

69      connector synchronisation attribuer_c2_r2(c2.attribuer ,r2.attribuer )
70      connector synchronisation quitter_c2_r2(c2.quitter ,r2.quitter )
71      connector synchronisation repasser_carte_c2_r2(c2.repasser_carte ,r2.
            repasser_carte )

73      connector synchronisation attribuer_c3_r3(c3.attribuer ,r3.attribuer )
74      connector synchronisation quitter_c3_r3(c3.quitter ,r3.quitter )
75      connector synchronisation repasser_carte_c3_r3(c3.repasser_carte ,r3.
            repasser_carte )
76    end

78  end
```

Listing 2.5 – Étude de cas hôtel en BIP

2.3.4.2 Expérimentation

Le moteur d'exécution de BIP génère un espace d'états associé au programme BIP exécuté. Chaque état possède un numéro (**state #numéro**) et comporte les interactions franchissables numérotées. Par défaut, le moteur d'exécution tire au hasard l'interaction à franchir. En utilisant le moteur d'exécution de BIP, nous avons expérimenté notre programme `Hotel` introduit précédemment. Pour y parvenir, nous avons établi les trois scénarios suivants :

Scénario 1 : "Avant de quitter l'hôtel, un client est censé accéder à la chambre allouée au moins une fois".

Afin de tracer la séquence d'exécution d'un tel scénario et par mesure de simplification, nous avons opté pour la configuration illustrée par le listing 2.6. Celle-ci définit un composant composite `hotel` qui instancie un composant **`c1`** de type `Client` qui interagit avec une instance **`r1`** du composant de type `Chambre`. Ces deux instances sont synchronisées via

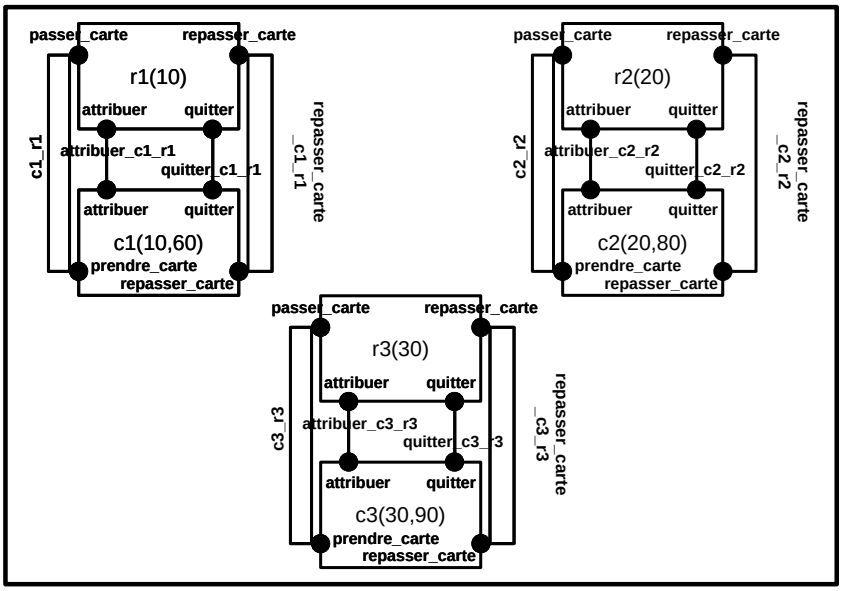

FIGURE 2.8 – Architecture générale du composant composite hotel

les quatre instances de connecteurs `c1_r1`, `attribuer_c1_r1`, `repasser_carte_c1_r1` et `quitter_c1_r1`.

```
compound type hotel()
  component Chambre r1(10)
  component Client c1(10,60)

  connector client_chambre c1_r1(c1.card, r1.passer_carte)
  connector synchronisation attribuer_c1_r1(c1.attribuer, r1.attribuer)
  connector synchronisation quitter_c1_r1(c1.quitter, r1.quitter)
  connector synchronisation repasser_carte_c1_r1(c1.repasser_carte, r1.repasser_carte)
end
```

Listing 2.6 – Composant composite `hotel` (scénario 1)

Après la compilation du code BIP, nous avons adopté un mode d'exécution **interactif** [6] pour générer la séquence d'exécution donnée par le listing 2.7. Cette séquence montre que le programme BIP admet une seule interaction franchissable pour les états de numéro 0 et 1 (voir les lignes 5 et 10 du listing 2.7) et deux interactions franchissables pour l'état numéro 2 (voir la ligne 13 du listing 2.7). Ainsi, le client ne peut sortir de la chambre (interaction `quitter_c1_r1`) qu'après avoir accédé au moins une fois (interaction `attribuer_c1_r1`) à la chambre réservée. L'accès à la chambre pour la première fois nécessite le franchissement de l'interaction `c1_r1`. Notons que les deux interactions `repasser_carte_c1_r1` et `quitter_c1_r1` franchissables de l'état numéro 3 (voir la ligne 13 du listing 2.7) sont en compétition. Ceci explique le fait qu'un client, après son premier accès à sa chambre, aura le choix entre quitter ou accéder à nouveau à la chambre.

```
1  [BIP ENGINE]: BIP Engine (version 2015.04-RC7 )
2  [BIP ENGINE]:
3  [BIP ENGINE]: initialize components...
4  [BIP ENGINE]: random scheduling based on seed=1491492841
5  [BIP ENGINE]: state #0: 1 interaction:
6  [BIP ENGINE]:  [0]  ROOT.c1_r1: c1.prendre_carte(k1=10;k2=50;) r1.passer_carte(k1=0;k2=0;)
7  [BIP ENGINE]: →enter interaction / internal port number (0..0), or 'r' for a random choice: 0
8  [BIP ENGINE]: →choose [0] ROOT.c1_r1: c1.prendre_carte(k1=10;k2=50;) r1.passer_carte(k1=0;k2=0;)
9  [BIP ENGINE]: state #1: 1 interaction:
10 [BIP ENGINE]:  [0]  ROOT.attribuer_c1_r1: c1.attribuer() r1.attribuer()
```

```
11 [BIP ENGINE]: →enter interaction / internal port number (0..0), or 'r' for a random choice: 0
12 [BIP ENGINE]: →choose [0] ROOT.attribuer_c1_r1: c1.attribuer() r1.attribuer()
13 [BIP ENGINE]: state #2: 2 interactions:
14 [BIP ENGINE]: [0] ROOT.repasser_carte_c1_r1: c1.repasser_carte() r1.repasser_carte()
15 [BIP ENGINE]: [1] ROOT.quitter_c1_r1: c1.quitter() r1.quitter()
16 [BIP ENGINE]: →enter interaction / internal port number (0..1), or 'r' for a random choice:
```

Listing 2.7 – Trace d'exécution du composant composite `hotel` (scénario 1)

Scénario 2 : "Une mauvaise instanciation de type de composant `Client` peut entraîner une situation de blocage".

La configuration du composant composite `hotel` retenue pour ce scénario est illustrée par le listing 2.8. Cette configuration instancie trois composants `c1`, `c2` et `c3` de type `Client` qui interagissent respectivement avec les composants `r1`, `r2` et `r3` de type `Chambre`. Les composants de type `Client` sont correctement instanciés, chaque client se faisant attribuer une carte électronique dont la première clé ne correspond pas à la clé de la serrure de la chambre réservée. Par exemple, le client `c1` possède la carte dont les clés sont (100,60) ne peut pas accéder à la chambre réservée `r1` avec la clé 10 portée par sa serrure.

```
compound type hotel()
  component Chambre r1(10), r2(20), r3(30)
  component Client c1(100,60) c2(200,80) c3(400,90)

  connector client_chambre c1_r1(c1.prendre_carte, r1.passer_carte)
  connector client_chambre c2_r2(c2.prendre_carte, r2.passer_carte)
  connector client_chambre c3_r3(c3.prendre_carte, r3.passer_carte)

  connector synchronisation attribuer_c1_r1(c1.attribuer,r1.attribuer)
  connector synchronisation quitter_c1_r1(c1.quitter,r1.quitter)
  connector synchronisation repasser_carte_c1_r1(c1.repasser_carte,r1.repasser_carte)

  connector synchronisation attribuer_c2_r2(c2.attribuer,r2.attribuer)
  connector synchronisation quitter_c2_r2(c2.quitter,r2.quitter)
  connector synchronisation repasser_carte_c2_r2(c2.repasser_carte,r2.repasser_carte)

  connector synchronisation attribuer_c3_r3(c3.attribuer,r3.attribuer)
  connector synchronisation quitter_c3_r3(c3.quitter,r3.quitter)
  connector synchronisation repasser_carte_c3_r3(c3.repasser_carte,r3.repasser_carte)
end
```

Listing 2.8 – Composant composite `hotel` (scénario 2)

Après la compilation du code BIP du listing 2.8, nous avons exécuté une seule séquence d'interaction (mode d'exécution par défaut du moteur d'exécution [6]) pour générer la séquence d'exécution donnée par le listing 2.9. Ceci montre que le programme BIP est en situation de blocage (voir la ligne 17 du listing 2.9) après le franchissement des trois interactions possibles du programme BIP (voir les lignes de 5 à 16 du listing 2.9).

```
1  [BIP ENGINE]: BIP Engine (version 2015.04-RC7 )
2  [BIP ENGINE]:
3  [BIP ENGINE]: initialize components...
4  [BIP ENGINE]: random scheduling based on seed=1491230880
5  [BIP ENGINE]: state #0: 3 interactions:
6  [BIP ENGINE]: [0] ROOT.c1_r1: c1.prendre_carte(k1=100;k2=60;) r1.passer_carte(k1=0;k2=0;)
7  [BIP ENGINE]: [1] ROOT.c2_r2: c2.prendre_carte(k1=200;k2=80;) r2.passer_carte(k1=0;k2=0;)
8  [BIP ENGINE]: [2] ROOT.c3_r3: c3.prendre_carte(k1=400;k2=90;) r3.passer_carte(k1=0;k2=0;)
9  [BIP ENGINE]: →choose [1] ROOT.c2_r2: c2.prendre_carte(k1=200;k2=80;) r2.passer_carte(k1=0;k2=0;)
10 [BIP ENGINE]: state #1: 2 interactions:
11 [BIP ENGINE]: [0] ROOT.c1_r1: c1.prendre_carte(k1=100;k2=60;) r1.passer_carte(k1=0;k2=0;)
12 [BIP ENGINE]: [1] ROOT.c3_r3: c3.prendre_carte(k1=400;k2=90;) r3.passer_carte(k1=0;k2=0;)
13 [BIP ENGINE]: →choose [1] ROOT.c3_r3: c3.prendre_carte(k1=400;k2=90;) r3.passer_carte(k1=0;k2=0;)
14 [BIP ENGINE]: state #2: 1 interaction:
15 [BIP ENGINE]: [0] ROOT.c1_r1: c1.prendre_carte(k1=100;k2=60;) r1.passer_carte(k1=0;k2=0;)
16 [BIP ENGINE]: →choose [0] ROOT.c1_r1: c1.prendre_carte(k1=100;k2=60;) r1.passer_carte(k1=0;k2=0;)
```

```
17 [BIP ENGINE]: state #3: deadlock!
```

Listing 2.9 – Trace d'exécution du composant composite `hotel` (scénario 2)

Scénario 3 : "Calculer toutes les séquences d'interactions possibles d'un composant composite".

Pour ce scénario nous explorons toutes les séquences d'interactions possibles du composant composite `hotel` spécifié par le listing 2.8. L'exécution **exhaustive** [6] de toutes ces séquences d'interactions est fournie par le listing 2.10.

```
1 [BIP ENGINE]: BIP Engine (version 2015.04−RC7 )
2 [BIP ENGINE]:
3 [BIP ENGINE]: initialize components...
4 [BIP ENGINE]: computing reachable states: . . . . . .
5 . . . . . .
6 . . . . . .
7 . . . . . found 125 reachable states, 0 deadlock, and 0 error in 0 state
```

Listing 2.10 – Trace d'exécution du composant composite `hotel` (scénario 3)

2.3.4.3 Discussion

Les moyens offerts par BIP comme **atom type**, **connector type**, **compound type**, **component** et **connector** facilitent le développement des applications dans ce langage. De même, le paramétrage des composants atomiques apporte une solution élégante à l'initialisation des données (**data**) d'un composant par la transition **initial** (voir les composants `Client` et `Chambre` dans le listing 2.5).

Les données introduites dans les composants atomiques `Chambre` et `Client` possèdent uniquement des types définis en utilisant les possibilités de typage offertes par le langage hôte C++. Ainsi, les propriétés intra et inter-données ne peuvent pas être exprimées par BIP. Par exemple, dans le composant `Chambre`, la propriété qui stipule que les deux clés (`premier`, `deuxieme`) d'une carte électronique sont différentes **n'est pas explicite**. De même, la propriété qui consiste à dire que `cle_serrure` $\in$ {`premier`, `deuxieme`} dès que la chambre est attribuée à un client **n'est pas explicite**. En outre, les deux paramètres `cle1` et `cle2` du composant atomique `Client` sont uniquement typés comme int. En principe, ils doivent être différents. Le paramètre `cle1` doit être identique à `cle_serrure` de la chambre attribuée au client.

Les transitions étiquetées par des portes bâties sur des variables (comme `passer_carte` pour `Chambre` et `prendre_carte` pour `Client`) ne peuvent pas être totalement comprises au sein du composant où elles étaient introduites. Par exemple, la définition de la transition `passer_carte` est répartie sur `Chambre`, `Client`, `client_chambre` et `hotel`. Une telle répartition peut être difficile à maîtriser lors de la modélisation d'une application BIP.

Un connecteur BIP peut **modifier** les données d'un composant. Par exemple, les connecteurs de type `client_chambre` agissent **d'une façon implicite** sur les données portées par les composants de type `Client`. Ceci peut entraîner des erreurs difficiles à détecter.

2.3.5 Évaluation

BIP est un langage de modélisation et programmation des systèmes à base de composants. En tant que langage de modélisation, il peut être considéré comme ADL (Architecture Description Language). A l'instar, des ADL comme Wright [12] et Acme [54], BIP supporte le concept de connecteur pour exprimer la coordination entre les composants. Cependant, un connecteur BIP est sans état (stateless). En BIP, le concept d'architecture est un concept

de première classe. Il englobe les connecteurs et priorités : les deux couches Interaction et Priority (voir la figure 2.2). Une architecture BIP possède une sémantique bien définie et peut être analysée et transformée par un concepteur de systèmes en se servant de divers outils fournis par la plateforme BIP [4, 6]. De plus, une architecture BIP se distingue nettement du comportement exprimé par les composants. Cette séparation entre architecture et comportement est un atout en BIP. En tant que langage de programmation, BIP exécute les composants atomiques d'une façon concurrente et les coordonne moyennant des mécanismes de haut niveau tels que protocoles et politiques d'ordonnancement. Ceci distingue BIP des modèles de composants orientés implantation comme Java Beans, J2EE et CORBA. En effet, ces derniers utilisent des mécanismes de bas niveau tels que processus léger (thread), interaction point à point et appel distant pour faire coopérer les composants. Enfin, en tant que langage de modélisation et programmation, BIP offre un cadre homogène permettant d'assurer la continuité entre ces deux phases et le développement rigoureux (comportement formel de composants, architecture bien définie) des systèmes à base de composants.

Le langage BIP induit une méthode ascendante de développement des applications ou systèmes à base de composants. Ainsi, la correction (vérification et validation) d'une application BIP s'appuie sur la correction des composants atomiques. En effet, le comportement d'un composant composite est obtenu par composition des composants atomiques utilisés. Mais le langage BIP ne permet pas l'analyse a priori des composants atomiques. En effet, les outils de vérification associés à BIP comme D-Finder [17] et D-Finder 2 [18] spécialisés dans la détection des erreurs de blocage interviennent a posteriori, une fois l'application construite : composants atomiques, composants composites, connecteurs simples, connecteurs hiérarchiques et politiques d'ordonnancement. De plus, à notre connaissance, il n'existe pas des travaux liés à la vérification formelle des composants BIP pris individuellement. Une telle entreprise n'est pas facile car le langage BIP autorise l'appel des sous-programmes C++ (langage hôte de BIP) entre autres avec effet de bord sur l'état du composant (passage par adresse des données). A notre avis, la vérification de la correction des composants atomiques favoriserait l'obtention des connecteurs corrects et par conséquent des architectures cohérentes.

2.4 Event-B

La méthode Event-B [8] permet le développement de logiciels et systèmes corrects par construction. Elle est mieux adaptée que son ancêtre la méthode B [7] pour la modélisation de systèmes réactifs et distribués. En effet, elle supporte une notion de raffinement plus riche que celle de la méthode B (possibilité d'introduire de nouveaux événements). Il est à noter qu'elles sont basées sur une logique des prédicats du premier ordre et la théorie des ensembles. Event-B est utilisée pour décrire formellement les systèmes et raisonner mathématiquement sur leurs propriétés. Le concept fondamental de la modélisation en Event-B est celui de la machine abstraite décrivant les comportements possibles du système. Elle permet de décrire des événements et les modifications qu'ils introduisent sur l'état interne de la machine. Il s'agit d'un modèle à états qui peut être interprété formellement par un système de transitions.

Cependant Event-B ne supporte pas d'une façon native des mécanismes de réutilisation très utiles pour le développement formel des systèmes complexes. Les travaux de Bulter [25, 24, 36, 74], Silva [73, 75] et Abrial [11] ont permis d'augmenter Event-B par des techniques de réutilisation basées sur la composition des sous-spécifications dites aussi sous-composants et la décomposition des spécifications centralisées. Deux méthodes de composition/décomposition ont été identifiées pour Event-B : par variable partagée et par événement partagé. La première est adaptée aux systèmes parallèles à mémoire partagée et la deuxième est plutôt adaptée aux systèmes distribués.

2.4.1 Modèles Event-B

Les concepts fondamentaux de la modélisation en Event-B sont les machines et les contextes. Un modèle Event-B est formé par un ensemble de machines et/ou de contextes. Plus précisément, il peut contenir uniquement des contextes, ou uniquement des machines, ou les deux. Dans le premier cas, le modèle représente une structure purement mathématique des ensembles, constantes, axiomes et théorèmes. Dans le troisième cas, le modèle est paramétré par les contextes. Enfin, le second cas représente un modèle qui n'est pas paramétré.

Les machines en Event-B, prennent en charge uniquement la définition de la partie dynamique du système. Quant à la partie statique, elle est à la charge des contextes. Une machine Event-B peut réutiliser les éléments de modélisation définis dans des contextes comme les ensembles et les constantes via la relation **sees**. En outre, Event-B supporte une relation clef pour le développement formel basé sur le raffinement prouvé. Celle-ci est appelée **refines** liant deux machines : machine abstraite et machine raffinée ou concrète. Un contexte étend (relation **extends**) au moins un autre contexte en rajoutant de nouveaux éléments de modélisation (ensemble, constante, axiome ou théorème). On note que les relations de raffinement et d'extension sont des relations transitives et elles ne doivent pas entraîner de cycles. La figure 2.9 montre les relations entre machines et contextes au sein d'un modèle Event-B.

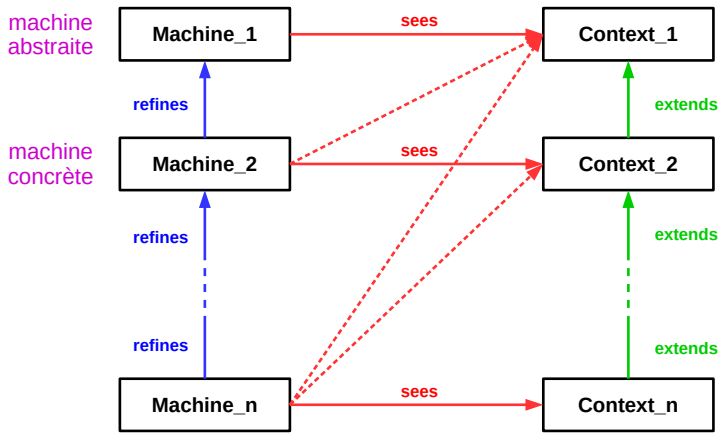

FIGURE 2.9 – Relations potentielles entre les constructions machine et contexte

La méthode Event-B se base sur un langage d'actions simple pour décrire les traitements et sur un langage logico-ensembliste pour décrire les données. Le langage d'actions comportant cinq types d'actions : parallèle, affectation déterministe, affectation non déterministe ensembliste ($:\in$), affectation non déterministe régie par un prédicat ($:|$) et skip (action qui ne fait rien). Une machine modélise un système au moyen de variables d'état et d'un ensemble d'événements faisant évoluer ces variables. Un événement comporte une garde et une action. Un événement est dit franchissable, si sa garde est satisfaite. En effet, la garde d'un événement représente une condition nécessaire mais pas suffisante. Dans le cas où existe plusieurs événements franchissables au sein d'une même machine alors, un et un seul événement sera déclenché et le choix de celui-ci sera arbitraire. Ceci est connu sous le nom du non-déterminisme externe permettant de régir l'exécution des événements franchissables. Quant au non-déterminisme interne, il concerne l'exécution d'un événement comportant des actions non déterministes telles que l'affectation ensembliste non déterministe ($:\in$). Dans la suite, nous présentons les structures des deux composants contexte et machine d'Event-B.

2.4.1.1 Contexte

Le contexte définit les données statiques, dites encore des paramètres, liées à un système. Ces paramètres englobent les ensembles abstraits et les constantes. Les propriétés sur ces constantes (typage et autres contraintes) et ces ensembles sont décrites par des prédicats logiques sous forme d'axiomes et éventuellement de théorèmes. Toutefois un contexte Event-B peut étendre plusieurs autres contextes. La structuration de ces éléments est englobée dans des clauses comme le montre le Listing 2.11.

```
context
  < context_identifier >
extends
  < context_identifier >∗
  . . .
sets
  < set_identifier >∗
  . . .
constants
  < constant_identifier >∗
  . . .
axioms
  {<label> <predicate>}∗
  . . .
theorems
  {<label> <predicate>}∗
  . . .
end
```

Listing 2.11 – Structure d'un contexte

La clause `axioms` permet de typer les constantes et éventuellement les ensembles introduits dans les deux clauses `sets` et `constants`. Également, elle décrit les contraintes mettant en relation les constantes et les ensembles. Au sens du langage Event-B les axiomes sont des prédicats supposés vrais a priori. Tandis que, les théorèmes introduits dans la clause `theorems` doivent être déduits par preuve mathématique à partir des axiomes et des théorèmes introduits textuellement avant. Notons que tout contexte Event-B est identifié par un nom unique déclaré dans la clause `context`.

```
context Fibonacci
constants n fibo
axioms
  @n_typ n∈ℕ
  @n_val n=25
  @fib_typ fibo∈0. .n→ℕ
  @fib0 fibo(0)=0
  @fib1 fibo(1)=1
  @fibn ∀i·(i∈2. .n⇒fibo(i)=fibo(i−2)+fibo(i−1))
  theorem @fibo_trie ∀i·(i∈0. .n−1⇒fibo(i)≤fibo(i+1))
end
```

Listing 2.12 – Contexte Fibonacci

Le listing 2.12 illustre un exemple de contexte nommé `Fibonacci`. Il définit en Event-B la fameuse suite de Fibonacci. Ce contexte comporte deux constantes `n` et `fibo`. Le contexte définit un ensemble d'axiomes qui sont des prédicats supposés vrais a priori au sens du langage Event-B et un théorème d'étiquette `@fibo_trie`.

2.4.1.2 Machine

Les machines modélisent le comportement dynamique du système. Une machine est composée d'un ensemble d'éléments de modélisation définissant principalement l'état et les évé-

nements.

La cohérence d'une machine est décrite par des invariants (clause `invariants`) portant sur ses variables d'état (clause `variables`). Ces invariants doivent être établis par l'initialisation et préservés par chaque événement. En effet, un événement est perçu comme une transition permettant de passer d'un état cohérent vers un autre. Ils sont définis par des prédicats décrits à l'aide du langage logico-ensembliste d'Event-B. A l'instar des théorèmes introduits dans un contexte, les théorèmes d'une machine Event-B sont des prédicats qui doivent être prouvés statiquement. Notons qu'une machine peut en raffiner une autre et voir un ou plusieurs contextes (voir la figure 2.9). La clause `refines` indique l'identifiant de la machine abstraite que la machine raffine et la clause `sees` fournit la liste de contextes vus explicitement par la machine.

```
machine
  <machine_identifier>
refines
  <machine_identifier>*
sees
  < context_identifier >*
  . . .
variables
  < variable_identifier >*
  . . .
invariants
  {<label> <predicate>}*
  . . .
theorems
  {<label> <predicate>}*
  . . .
variant
  <variant>
events
  <event>*
  . . .
end
```

Listing 2.13 – Structure d'une machine

Les événements, qui agissent sur l'état du système, doivent respecter les propriétés invariantes en permanence et ceci pour garantir la cohérence de la machine au sein de laquelle ils sont définis. La description de l'ensemble d'événements se fait dans la clause `events`. En outre, cet ensemble comporte obligatoirement un événement particulier, noté **initialisation**, qui prend en charge l'initialisation de l'état de la machine.

Les événements d'une machine Event-B sont des transitions. Un événement est composé de deux parties : une garde qui définit la condition selon laquelle l'événement peut ou non se déclencher et une action qui définit l'évolution des variables d'état. La structure générale d'un événement est donnée par le listing 2.14.

```
< event_identifier > ≙
  {ordinary, convergent, anticipated}
refines
  < abstract_event_identifier >*
  . . .
any
  <parameter_identifier>*
  . . .
where
  {<label> <predicate>}*
  . . .
when
  {<label> <predicate>}*
  . . .
with
```

```
  {<label> <witness>}*
  . . .
then
  {<label> <action>}*
  . . .
end
```

Listing 2.14 – Structure d'un événement

En Event-B, nous pouvons distinguer trois formes d'événements où les mots-clés utilisés pour définir un événement changent. En effet, un événement paramétré admet des paramètres (clause `any`), des gardes (clause `where`) et des actions (clause `then`) et il ne se déclenche que s'il existe des valeurs de paramètres qui satisfont les gardes.

Lorsqu'un événement raffine un événement abstrait qui admet un paramètre et le fait disparaître, il est indispensable de fournir un témoin sur l'existence de ce paramètre dans la version concrète (clause `with`). La seconde forme dite gardée dans laquelle l'événement n'admet pas de paramètres mais définit des gardes (clause `when`) et des actions (clause `then`). Autrement dit, puisque la clause `any` est omise le mot clé `where` est remplacé par `when`. C'est ainsi que les gardes et actions ne dépendent que des variables d'état du système. La dernière forme définit des événements simples où la garde est toujours vraie alors que les actions (clause `then`) agissent sur les variables d'état du modèle.

Event-B supporte trois catégories d'événements : **ordinary**, **convergent** et **anticipated**. Par défaut, un événement est **ordinary**. En général, les événements qui forment le modèle initial abstrait sont de genre **ordinary**. Et ils demeurent ordinary tout au long de la chaîne de raffinements successifs. Un événement **convergent** ne doit pas s'exécuter indéfiniment. Un événement **anticipated** devrait finir par être raffiné par un événement **convergent**. Pour y parvenir, le spécifieur doit introduire dans la clause `variant` une expression entière qui doit décroitre à chaque occurrence d'un événement **convergent** et ne doit pas croitre pour chaque occurrence d'un événement **anticipated**.

2.4.1.3 Sémantique d'une machine Event-B

Le comportement dynamique d'une machine Event-B est exprimé par l'algortithme non déterministe suivant :

```
Exécuter l'événement INITIALISATION
Calculer dans Evt_f l'ensemble des événements franchissables
Tant que Evt_f ≠  ∅ Faire
  Choisir d'une façon arbitraire dans evt un élément de l'ensemble Evt_f
  Exécuter evt
  Calculer dans Evt_f l'ensemble des événements franchissables
Fin tant que
```

Une machine Event-B est un triplet $\langle V, I, E\rangle$ où V est un ensemble de variables, I un prédicat sur V dit l'invariant supposé préservé par les événements, E est un ensemble d'événements de la forme $(e = \textbf{any}\ X\ \textbf{where}\ G(X)\ \textbf{then}\ S(V, X))$.

La sémantique dynamique est directement donnée par un système de transitions dont les transitions sont les suivantes :

$$\frac{(e = \textbf{any}\ X\ \textbf{where}\ G(V,X)\ \textbf{then}\ S(V,X)) \in M \quad G(v,p),\ v' = S(v,p),}{v \xrightarrow{e(p)} v'}$$

2.4.1.4 Exemple : modélisation d'un algorithme de calcul des nombres premiers

La méthode massivement parallèle appelée Crible de Darwin [19] permet de résoudre le problème des calculs des nombres premiers en se basant sur le principe d'élimination progressive des multiples. Cette méthode est représentée par la figure 2.10. Chaque nombre est écrit sur un poisson nageant dans la mer, et tout poisson peut et doit manger ses multiples. Seuls survivent les poissons premiers. Ce modèle est appelé **machine chimique** [19] car la nage des poissons simule le mouvement brownien des molécules, avec comme règle de réaction chimique $p, k \times p \rightarrow p$. Techniquement, la machine chimique s'exprime par la réécriture de multi-ensembles, où chaque élément peut avoir plusieurs occurrences. Pour le problème de calcul des nombres premiers, on peut mettre plusieurs poissons pour chaque nombre. Ceci permet d'accélérer le mouvement brownien des molécules. Un tel mouvement est indéterministe, parallèle et désordonné.

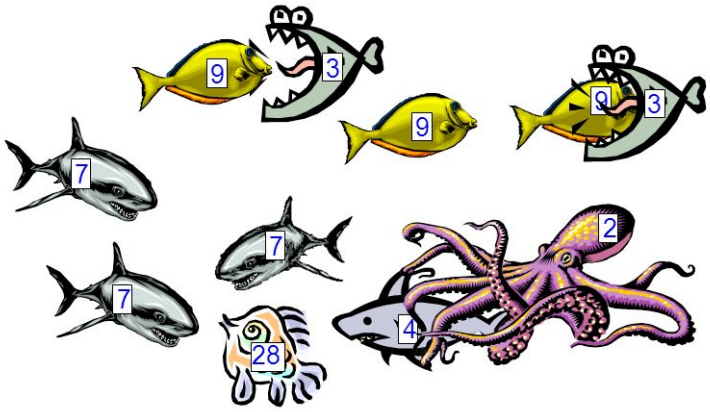

FIGURE 2.10 – Crible de Darwin [19]

Nous avons modélisé en Event-B le Crible de Darwin permettant le calcul des nombres premiers (voir le listing 2.15). La soupe chimique (variable `soupe_chimique`) est modélisée par un multi-ensemble exprimé en Event-B par une fonction totale dont l'ensemble de départ représente les molécules (ici les nombres > 1) et l'ensemble d'arrivée représente l'occurrence de chaque molécule. La réaction chimique $p, k \times p \rightarrow p$ est modélisée par l'événement `darwin` permettant de mettre à jour la variable `soupe_chimique`. Notre machine `Chimique` en Event-B simule bien l'indéterminisme du mouvement des molécules mais pas son caractère parallèle.

```
machine Chimique
variables soupe_chimique //nombres qlqs, multi−ensemble
invariants
  @var_typ soupe_chimique∈ℕ1\{1}→ℕ

events
  event INITIALISATION
    then
      @act1 soupe_chimique:∈ℕ1\{1}→ℕ1
  end

  event darwin // Probleme de terminaison
    any n1 n2 p
    where
      @grd1 n1∈dom(soupe_chimique)
      @grd2 n2∈dom(soupe_chimique)
      @grd3 p∈ℕ1∧p>1
      @grd4 n2=p ∗ n1 // n2 est un multiple de n1
      @grd5 soupe_chimique(n1)>0
      @grd6 soupe_chimique(n2)>0
```

```
    then
      @act1 soupe_chimique(n2):=soupe_chimique(n2)−1
  end
end
```

Listing 2.15 – Crible de Darwin en Event-B

2.4.2 Raffinement en Event-B

Le raffinement est une technique de développement incrémental qui consiste à modéliser des systèmes complexes étape par étape en partant d'une spécification abstraite. Il s'agit d'un concept fondamental de la méthode Event-B. En effet, un développement en Event-B commence par la rédaction d'une spécification, dont on prouve la cohérence. On construit ensuite des raffinements, pour lesquels on établit la cohérence interne et la correction vis-à-vis de la spécification abstraite. Ainsi, la modélisation d'un système en Event-B consiste en une suite de modèles construits de manière incrémentale où chacun est censé être un raffinement du modèle précédent. A chaque raffinement, de nouveaux détails sont rajoutés progressivement tout en conservant les propriétés des modèles abstraits.

Le raffinement de modèles en Event-B consiste à étendre les contextes et à raffiner les machines. Un contexte peut être étendu en ajoutant de nouveaux éléments de modélisation (ensembles, constantes et axiomes). Une machine Event-B peut être raffinée en modifiant son espace d'états (nouvelles variables et nouveaux invariants), en ajoutant de nouveaux événements et/ou en raffinant les événements abstraits (renforcement des gardes et raffinement des actions).

Event-B est supportée par la plateforme Rodin [9, 10] permettant le développement par raffinements successifs. Rodin permet le raffinement des événements (**one-by-one**), l'éclatement des événements (**one-by-many**) et l'introduction des nouveaux événements. En utilisant judicieusement les différents genres d'événement supportés par Rodin à savoir **ordinary**, **convergent** et **anticipated** (voir la section 2.4.1.2), deux statégies de développement en Event-B sont bien établies [62] : sans extension d'interface (no interface extension) et avec extension d'interface (interface extension), sachant que l'interface d'un modèle Event-B comporte les événements de statut **ordinary** introduits par le modèle initial. Autrement dit la deuxième stratégie (interface extension) permet l'introduction des nouveaux événements de statut **ordinary**.

2.4.2.1 Raffinement horizontal vs vertical

Il existe deux types de raffinement en Event-B : raffinement horizontal et raffinement vertical. Le raffinement horizontal [13], dit aussi superposition, consiste à compléter une spécification abstraite d'un système en ajoutant de nouvelles fonctionnalités. Un tel processus de raffinement permet d'ajouter étape par étape des exigences issues du cahier des charges du système à modéliser. Le modèle final obtenu doit incorporer toutes les fonctionnalités à formaliser sans contredire la spécification abstraite. De plus, ce type de raffinement peut être utilisé avec profit pour le développement des systèmes parallèles et distribués en intégrant des mécanismes de distribution dans un modèle initialement centralisé. Notons que la technique de raffinement horizontal ne doit pas lever l'indéterminisme des modèles abstraits à raffiner.

Le raffinement vertical [60], dit aussi raffinement de données, permet de raffiner un modèle issu d'un processus de raffinement horizontal. En effet, il permet d'apporter une implantation pas-à-pas à la spécification issue d'un processus de raffinement horizontal. Ce type de raffinement ne permet pas d'ajouter des fonctionnalités au modèle, comme le raffinement horizontal, mais il raffine le modèle afin de se rapprocher d'un modèle exécutable. Autrement dit, ce raffinement consiste à raffiner un modèle abstrait en ajoutant de manière incrémentale

des décisions d'implantation. De telles décisions concernent le remplacement des structures de données purement mathématiques par des structures concrètes implantables et le changement de variables abstraites par des variables concrètes en ajoutant des invariants de collage. La décomposition de modèles complexes en sous-modèles à développer séparément est vue comme un raffinement vertical. Notons qu'un tel type de raffinement peut être réalisé par des outils semi-automatiques [8].

En Event-B via la plateforme Rodin, l'activité de preuve mathématique (obligations de preuve à décharger) et de model-checking (via la boîte à outils ProB, voir la section 2.4.3) assure la cohérence des modèles produits et la correction de chaque étape de raffinement aussi bien pour le raffinement horizontal que vertical. Mais Event-B ne permet pas d'exprimer directement la préservation de propriétés de vivacité par les étapes de raffinement. Le travail décrit dans [63] apporte des réponses à la préservation des propriétés temporelles dans un processus par raffinements successifs en Event-B.

2.4.2.2 Aides méthodologiques

La plupart des spécifieurs rencontrent des **difficultés réelles** pour mener à bien et à terme un processus formel de développement basé sur des raffinements successifs. Ces difficultés touchent à l'élaboration d'une stratégie de raffinement "optimale", l'ordonnancement des étapes de raffinement et la gestion des retours arrière mettant en cause des choix antérieurs de spécification, conception ou implémentation. Pour faire face **en partie** à ces difficultés, la communauté formelle autour de B et d'Event-B a exploré deux pistes de recherche. La première piste est très ambitieuse. Elle vise l'**automatisation** de l'activité de raffinement. En effet, l'outil BART [47] faisant partie intégrante de l'Atelier B implémente une approche d'automatisation des raffinements successifs des machines B. Un tel outil s'appuie sur des règles de raffinement **génériques** décrites à l'aide d'un langage adéquat [27, 59]. Utilisé en phases aval -notamment dans la phase d'implémentation-, l'outil BART améliore sensiblement la productivité des modélisateurs B. A l'instar des patrons de conception de GoF célèbres dans le monde orienté objet, la deuxième piste **réutilise** ce paradigme dans un monde formel celui d'Event-B. Le travail décrit dans [48] introduit le concept de **patron de conception prouvé**. Également, il propose des patrons de conception tels que : synchronisation faible et synchronisation forte utilisables notamment dans les systèmes réactifs à événements discrets. En outre, le travail décrit dans [42] propose un patron de conception appelé synchronous multiple message communication formalisé en Event-B. De plus, il propose une démarche à quatre étapes permettant de **réutiliser** des patrons de conception décrits en Event-B. Dans la suite de ce travail, nous allons apporter deux types de raffinements automatiques guidés par des DSL en tant qu'aides pour le développement sûr des systèmes distribués (voir le chapitre 3).

2.4.3 Correction des modèles Event-B

Dans cette section, nous allons examiner les techniques permettant de vérifier la correction de modèles Event-B.

2.4.3.1 Preuve

L'un des points d'orgue d'Event-B est la preuve de correction des modèles. En effet, un modèle Event-B est dit correct si les obligations de preuve (OP) associées sont prouvées (ou déchargées). Plus précisément, une OP est un séquent dont on doit fournir une démonstration pour vérifier un critère de correction sur le modèle. Ces obligations de preuve concernent différents aspects du modèle tels que la vérification des propriétés d'invariance d'une machine Event-B ou la preuve de la correction d'un raffinement ou encore pour prouver que

les propriétés de la clause `theorems` sont correctes. Elles sont confiées aux prouveurs internes et externes de la plateforme Rodin dédiée à Event-B. Une fois prouvée, une étiquette est attribuée à chaque obligation de preuve pour des raisons de traçabilité. Cette étiquette renseigne sur la provenance de l'obligation de preuve (théorème, invariant, renforcement de garde, simulation d'action, ...).

2.4.3.2 Vérification comportementale à l'aide de ProB

ProB [39, 49] est un outil incontournable pour un développement formel. Il permet l'animation, le model-checking des propriétés de sûreté et d'absence de blocage, le model-checking des propriétés de vivacité, la vérification de la correction de la relation de raffinement, la résolution de contraintes et la contre-preuve. L'outil ProB supporte plusieurs formalismes tels que B, Event-B, Z, CSP et TLA$^+$. ProB complète l'activité de preuve supportée par Event-B. En effet, les propriétés de sûreté (établissement de l'invariant et préservation de l'invariant) et la vérification formelle de la correction de la relation de raffinement sont couvertes par des obligations de preuve standards définies par la théorie Event-B [8] et implémentées dans la plateforme Rodin [9, 10]. Mais l'intégration des propriétés de vivacité dans Event-B sous forme d'obligations demeure un challenge. En effet, le travail décrit dans [41] propose des obligations de preuve couvrant plusieurs types de propriétés de vivacité tels que : Eventually, Until, Progress et Persistence. Cependant, l'expression directe de ces propriétés de vivacité n'est pas actuellement supportée par la plateforme Rodin. Le model-checker temporel ProB apporte une solution générique pour la description et la vérification des propriétés de vivacité. Pour y parvenir, il propose un langage puissant pour la description des propriétés temporelles LTL^e [58]. Ce dernier englobe :

- trois types des propositions atomiques : orientés état au sens LTL, orientés transition comme **e(**`evt`**)** qui teste si l'événement `evt` est couramment franchissable et orientés occurrence d'événement comme [`evt`] qui teste si `evt` a été exécuté dans l'instant suivant.
- des opérateurs logiques : **not**, **&**, **or** et ⇒.
- des opérateurs temporels tels que **G**(**Globally**), **F**(**Finally**), **X**(**neXt**), **U**(**Until**) et **W**(**Weak until**).
- des opérateurs d'équité tels **WF** et **SF** [46].

Notons au passage que le langage LTL^e de ProB fournit plusieurs propositions atomiques liées au blocage d'un ensemble d'événements (deadlock(evt1,evt2,...,evtk)), au déterminisme externe (deterministic(evt1,evt2,...,evtk)) et à la controlabilité (controller(evt1,evt2,...,evtk)). De plus, ProB est doté d'un solveur de contraintes permettant entre autres la vérification de la cohérence des axiomes associés aux constantes. Par exemple, pour le contexte `Fibonacci` (voir le listing 2.12), qui définit en Event-B la suite de Fibonacci, ProB permet de vérifier la cohérence de cette formalisation et génère les n+1 premiers nombres de Fibonacci. De plus, le théorème `@fibo_trie` est déchargé par le contre-prouveur ProB [39, 49].

2.4.3.3 Animation et simulation

La validation d'un modèle Event-B est relative aux attentes de l'utilisateur supposées décrites dans le cahier des charges. Un modèle Event-B correct vis-à-vis de la théorie Event-B n'est pas forcément valide. En effet, les critères de correction introduits par Event-B ne couvrent pas toutes les classes d'erreurs : oublis des fonctionnalités, mauvaise compréhension des besoins, propriétés dynamiques. Pour y parvenir, on peut utiliser avec profit des outils de validation associés à Event-B. Contrairement aux outils de preuve, ces outils permettent

une certaine exécution des modèles Event-B. Ceci, avec l'aide de l'utilisateur final, permet de détecter des erreurs non couvertes par l'analyse statique des prouveurs.

En Event-B, l'activité de validation permet de vérifier la correction des modèles vis-à-vis du cahier des charges. D'une façon générale, un outil de validation permet l'exécution des modèles Event-B à la recherche de contre-exemples. Un contre-exemple traduit forcément la présence des erreurs dans les modèles Event-B traités. Ces erreurs peuvent provenir des défauts de modélisation (incohérence des axiomes, oublis, cas non traités ou mal compris dans le cahier des charges), ou des défauts de conception introduits lors des étapes de raffinement.

La plateforme Rodin est dotée d'outils de validation à base d'animation tels que AnimB [53] et ProB [39, 49] permettant l'exécution de modèles formels Event-B en explorant d'une manière exhaustive l'espace d'états des modèles construits. En effet, de point de vue pratique, le fonctionnement de tels outils repose sur trois phases à savoir : évaluation des gardes, calcul de l'ensemble d'événements permis (ou accessibles ou franchissables) et finalement exécution proprement dite de l'événement choisi par l'utilisateur.

D'autres outils de validation à base de simulation permettent de transformer des modèles Event-B vers des programmes écrits en langage Javascript supportés par des environnements Web. Pour l'exécution des modèles après leur traduction, les simulateurs reposent sur les trois phases assurées par les outils d'animation comme décrits ci-dessus. Le simulateur JeB [82, 83] génère et exécute des simulations de modèles Event-B.

2.4.4 Composition et décomposition en Event-B

La méthode Event-B ne supporte pas d'une façon native des mécanismes de réutilisation très utiles pour le développement formel des systèmes complexes. Récemment, Event-B a été augmentée par des techniques de réutilisation : instanciation, composition et décomposition. La technique d'instanciation est au coeur de l'utilisation des Types de Données Abstraits (TDA) en Event-B [37]. Ainsi, elle apporte une séparation nette entre la spécification (contexte abstrait) et l'implémentation (contexte concret) d'un TDA. Les deux techniques complémentaires de composition et décomposition permettent la combinaison formelle des spécifications via la technique de raffinement. La décomposition peut être perçue comme le processus inverse de la composition. Un modèle Event-B contenant plusieurs variables et événements obtenus suite à des raffinements peut être décomposée en plusieurs sous-composants simples. Ceci peut être motivé par des raisons liées à la maîtrise de la complexité (décomposer pour régner) ou à l'introduction d'aspects architecturaux. Deux méthodes de composition/décomposition ont été identifiées pour Event-B : variable partagée [75] et événement partagé [72, 74]. La composition/décomposition par variable partagée est adaptée pour les systèmes parallèles à mémoire partagée, tandis que la composition/décomposition par événement partagé est plutôt adaptée pour le développement des systèmes distribués.

La méthode Event-B admet un développement formel descendant (top-down style) [8, 74] de spécifications qui débute avec un modèle initial abstrait du futur système. Par raffinements successifs, le modèle initial devient de moins en moins abstrait et de plus en plus concret jusqu'à ce que le modèle ultime soit prêt à être implémenté. Grâce au raffinement formel, le modèle ultime est correct par construction. Mais ce processus de raffinement entraîne des modèles complexes et difficiles à gérer. Pour faire face à ces problèmes, la décomposition formelle permet de diviser un modèle en plusieurs sous-composants d'une façon systématique. Ceci permet d'obtenir des sous-composants de tailles raisonnables et de travailler à plusieurs d'une façon parallèle.

En Event-B, deux méthodes de décomposition sont proposées [74] : décomposition par événement partagé (Shared Event Decomposition) et décomposition par variables partagés (Shared Variables Decomposition).

2.4.4.1 Composition

Dans le cadre d'Event-B, la composition des machines permet de réutiliser des spécifications existantes déjà prouvées pour construire des systèmes complexes. Bulter et Silva [71, 72, 74] augmentent Event-B par une nouvelle construction **composed machine**. Elle permet la composition par événement partagé de machines Event-B élémentaires en regroupant les éléments de modélisation venant de ces machines : variables et propriétés invariantes. En outre, une machine de type **composed machine** introduit des événements appelés des événements de synchronisation permettant de composer plusieurs événements issus des machines composées.

Le listing 2.16 décrit la structure générale d'une machine composée Event-B. Elle possède un identifiant et comporte plusieurs clauses. En effet, une machine de type **composed machine** peut en raffiner une autre machine du même projet Event-B (clause `refines`). Les machines formant la machine composée sont mentionnées dans la clause `includes`. Il est à noter que les contextes vus par les machines incluses sont implicitement visibles par la machine qui les compose. De plus, des propriétés de composition entre les différents espaces d'états (impérativement disjoints) des machines incluses sont décrits par des invariants explicites (clause `invariants`). Enfin, la clause `composes events` comporte une description des événements composés. Ils sont définis par la synchronisation d'événements appartenant aux machines incluses.

```
composed machine
  <machine_identifier>
refines
  <machine_identifier>*
sees
  < context_identifier >*
  . . .
includes
  {<machine_identifier> (no|include)<invariant>}*
  . . .
invariants
  {<label> <predicate>}*
  . . .
composes events
  <event>*
  . . .
end
```

Listing 2.16 – Structure d'une machine composée

La machine composée de type **composed machine** est censée respecter les obligations de preuve standard d'Event-B telles que : établissement de l'invariant, préservation de l'invariant et correction de raffinement. Lors de la composition de plusieurs sous-composants, on peut rajouter **un invariant de composition** permettant de mettre en relation plusieurs variables issues des machines composées. Un tel invariant devrait être déchargé pour vérifier la machine composée. A l'instar de la composition parallèle des processus CSP [43], la composition par événement partagé en Event-B respecte la propriété de monotonie [74]. En effet, les composants formant la machine composite peuvent être raffinés indépendamment sans toucher à terme à la correction de la machine initiale composée.

Pour des besoins de réutilisation, une machine composée peut être expansée (ou aplati) en générant une machine Event-B élémentaire qui comporte toutes les propriétés définies dans la machine composée et dans ses composants.

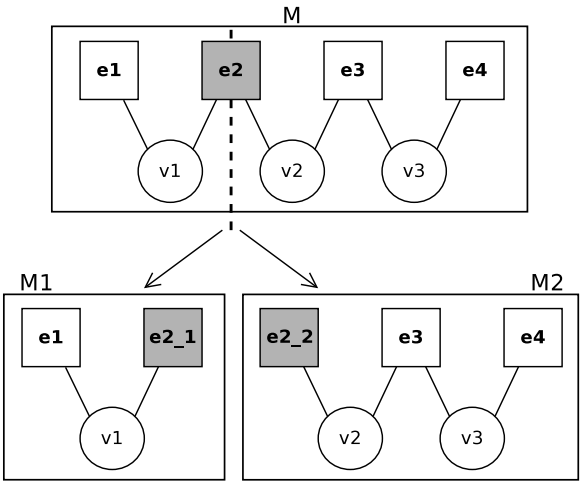

FIGURE 2.11 – Décomposition par événement partagé

2.4.4.2 Décomposition par événement partagé

Proposée par Bulter [74], la décomposition par événement partagé en Event-B permet la distribution des événements et variables sur plusieurs sous-composants. Elle n'autorise pas les variables partagées et les événements peuvent être scindés sur plusieurs sous-composants. Les sous-composants obtenus peuvent être raffinés par plusieurs acteurs qui travaillent en parallèle.

La figure 2.11 issue de [72] montre la décomposition par événement partagé de la machine M en deux machines M1 et M2. Les variables (v1, v2 et v3) de M sont réparties sur les deux machines M1 (v1) et M2 (v2 et v3). Cette répartition engendre une distribution des événements de la machine M. L'événement e2 est un événement partagé par les deux machines M1 et M2. Un tel événement, dit de synchronisation, doit être scindé en deux sous événements e2_1 et e2_2 alloués respectivement aux deux sous-machines M1 et M2.

La décomposition en Event-B possède deux aspects : syntaxique et sémantique. L'aspect syntaxique est caractérisé par la répartition des entités syntaxiques (variables, invariants et événements) sur plusieurs sous-composants. Quant à l'aspect sémantique, il concerne le comportement des sous-composants recomposés. Un tel comportement est le même que le composant initial non décomposé. Ceci peut être vérifié formellement en Event-B par une relation de raffinement entre le composant initial non décomposé et les sous-composants recomposés.

Concrètement, la décomposition par événements partagés est obtenue par la répartition des variables sur les sous-composants retenus. Or la répartition des variables se heurte à des problèmes : prédicats complexes (invariants et gardes) ou actions complexes faisant intervenir des variables réparties sur des sous-composants différents. Ceci nécessite la **séparation** de ces variables par des raffinements avec preuves mathématiques. Notons au passage, que la composition des sous-composants devrait raffiner le modèle centralisé initial.

Dans le cadre de cette thèse, nous avons utilisé avec profit la technique de décomposition par événement partagé. Celle-ci est inspirée de CSP de Hoare [43] et favorise le développement de systèmes distribués. En CSP, les processus se synchronisent sur le même événement et peuvent échanger des messages de type entrée ou sortie. En Event-B, les événements des sous-composants (ou sous-spécifications) synchronisés communiquent via **des paramètres partagés**.

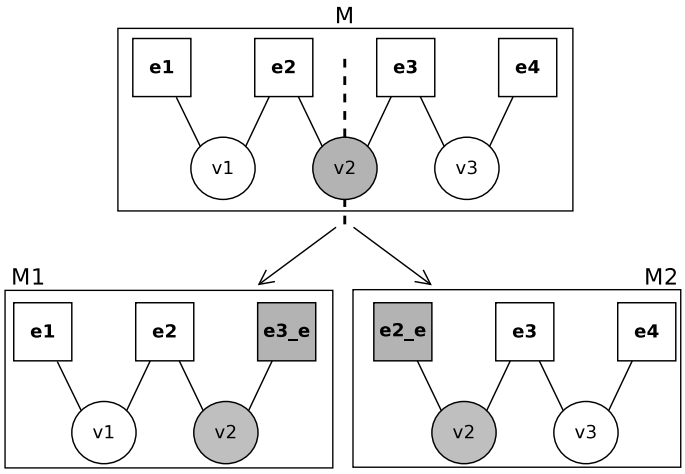

FIGURE 2.12 – Décomposition par variable partagée

2.4.4.3 Décomposition par variable partagée

La technique de décomposition par variable partagée a été proposé par Abrial [11]. Elle favorise la conception des algorithmes parallèles [24]. Techniquement cette approche de décomposition consiste à répartir les événements d'un modèle sur les sous-composants retenus. Elle permet l'introduction de variables partagées et d'événements externes. Ceux-ci garantissent la préservation du comportement des variables partagées dans tous les sous-composants. A l'instar de la décomposition par événement partagé, la recomposition des sous-composants raffinés donne naissance à un composant qui devrait raffiner le composant abstrait initial.

La figure 2.12 issue de [72] montre la décomposition par variable partagée de la machine M en deux machines M1 et M2.

Les événements (e1, e2, e3 et e4) de M sont répartis sur les deux machines M1 (e1 et e2) et M2 (e3 et e4). Les deux variables v1 et v3 sont allouées respectivement aux deux sous-composants M1 et M2. Quant à la variable v2, elle est partagée par les deux événements e2 et e3 et appartient aux deux sous-composants M1 et M2 issus de la décomposition de M. Les deux événements externes (e3_e et e2_e) sont introduits afin de simuler le traitement de la variable partagée aussi bien dans M1 (e3_e) que dans M2 (e2_e). Les deux sous-composants M1 et M2 peuvent être raffinés indépendamment. Mais les variables partagées et les événements externes doivent figurer à chaque étape et ne peuvent pas être affinées.

2.4.4.4 Outil de composition/décomposition

Les mécanismes de composition et décomposition décrits dans les deux sections 2.4.4.2 et 2.4.4.3 sont implémentés dans la plateforme Rodin sous forme d'un plug-in [75]. Il est intitulé **Shared Event Decomposition Plug-in**. Dans la suite et pour des raisons de simplification, le nom de l'outil sera abréger par **SEDT** (pour **S**hared **E**vent **D**ecomposition **T**ool).

L'entrée de l'outil SEDT de décomposition est une machine d'un projet Rodin donné sélectionné par l'utilisateur final. Après la sélection de la configuration et du style de décomposition, l'outil génère automatiquement les sous-composants si et seulement si la spécification centralisée à décomposer ne renferme pas d'actions ou de gardes complexes. Une garde ou une action est dite complexe si elle fait intervenir plusieurs variables réparties sur des sous-composants différents. Pour résumer voici les étapes à suivre afin de décomposer une machine Event-B :

1. Sélectionner une machine à décomposer ;
2. Définir les sous-composants qui doivent être générés ;
3. Sélectionner le style de décomposition à utiliser :
   - Shared Variable : le spécifieur sélectionne les événements à répartir sur les sous-composants. L'outil décompose automatiquement le reste du modèle conformément aux événements répartis : variables privées/variables partagées et événements externes.
   - Shared Event : le spécifieur répartit les variables sur les sous-composants. Le reste est assuré automatiquement par l'outil : événements privés, événements scindés.
4. L'utilisateur peut choisir de décomposer les contextes observés dans les sous-composants de manière similaire à la décomposition de la machine ;
5. Les sous-composants sont réalisés selon la configuration à décomposer ;
6. La configuration décomposée est stockée comme une machine composée (**composed machine**) ;
7. Les sous-composants générés peuvent être ultérieurement raffinés.

2.4.4.5 Exemple

Dans cette section, nous recherchons à montrer le fonctionnement de l'outil SEDT. Pour y parvenir, nous partons d'une spécification centralisée appelée `Essai` à décomposer en trois sous-composants **MA**, **MB** et **MC**. Une telle spécification centralise trois fonctionnalités à savoir le transfert de valeur, la copie de valeur et l'échange de valeur. Nous nous sommes inspiré d'une spécification centralisée `communication` issue de la thèse de Renato Silva [72]. Le modèle `Essai` comporte un contexte `Essai_C0` et une machine `Essai_M0` (voir l'annexe A.2). En utilisant l'outil SEDT, nous avons introduit, d'une façon interactive, l'architecture de décomposition de la machine `Essai_M0` est décrite dans la table 2.1.

Sous-composants	Variables d'état
MA	a
MB	b
MM	m,ctrl

TABLE 2.1 – Configuration de décomposition d'`Essai_M0`

L'outil échoue à décomposer la machine `Essai_M0` en renvoyant un message d'erreur (voir la figure 2.13). En effet, l'événement `Copy` manipule deux variables (b, m) appartenant à deux sous-composants différents (**MB**, **MM**). Ainsi l'action b := m est considérée comme action complexe par l'outil SEDT. Le problème rencontré par l'outil SEDT peut être résolu en raffinant l'événement `Copy`. La version raffinée de l'événement `Copy` (voir le listing 2.18) découple les deux variables b et m en introduisant un paramètre p (clause **any**). Afin de préserver la sémantique de l'action (@act1 b := m), le paramètre p doit être égal à la variable m (@grd3 p = m).

```
event Copy // evenement de copie de valeur·
  where
    @grd1 ctrl = TRUE // possibilite de copier
  then
    @act1 b := m
end
```

Listing 2.17 – Version abstraite de l'évévement `Copy`

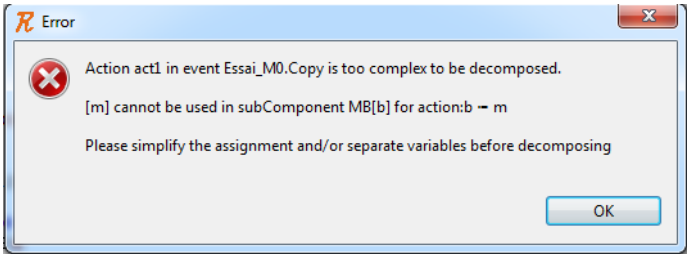

FIGURE 2.13 – Message d'erreur de la décomposition d'`Essai_M0` par SEDT

```
event Copy // evenement raffine
 refines Copy
  any p // parametre de raffinement·
  where
    @grd1 ctrl = TRUE
    @grd2 p∈ℕ
    @grd3 p=m
  then
    @act1 b ≔ p // b prends une valeur p dont le contenu est equivalent a celle de m
 end
```

Listing 2.18 – Version raffinée de l'évévement `Copy`

Nous rencontrons le même problème pour l'événement `Transfer_mid`. Cet événement comporte une action complexe qui manipule deux variables (a, m) appartenant à deux sous-composants différents. Ceci nécessite le raffinement de l'événement `Transfer_mid`. Les composants `MA`, `MB` et `MM` issus de la décomposition de la machine raffiné `Essai_M1` ainsi que la machine composée sont décrits dans l'annexe A.2.

2.4.5 Génération de code

Contrairement à la méthode B qui est dotée d'un outil de génération de code à partir d'une machine de type **`implementation`**, Event-B n'est pas équipée d'un outil standard de génération de code. Ceci peut être expliqué par le fait qu'un modèle Event-B pourra toucher à plusieurs domaines couvrant divers paradigmes séquentiel, concurrent, parallèle et distribué.

En effet, un modèle Event-B peut formaliser des futurs programmes (séquentiels, concurrents ou distribués) et systèmes (réactifs, distribués ou hybrides). Ceci explique la diversification des générateurs de code source associés à Event-B. Techniquement, les générateurs de code source associés à Event-B se heurtent à la résolution de deux problèmes fondamentaux : les non déterminismes interne et externe.

Le travail décrit dans [52, 76] propose une collection de plug-ins sous Rodin permettant de générer du code séquentiel impératif à partir des modèles Event-B suffisamment raffinés dans divers langages C, C++, Java et C#. De même, l'outil EHDL [57] permet de générer un code VHDL à partir de modèles Event-B. Il est basé sur un algorithme de traduction qui admet un sous-ensemble réduit de la syntaxe Event-B et génère un code en langage de description de matériel. Le code VHDL obtenu est caractérisé par le même comportement que le modèle Event-B dont il est issu.

L'outil B2C [81] traduit un modèle Event-B en code C exécutable. Il a été développé dans le but de produire du code exécutable C pour la machine virtuel MIDAS [80] modélisée en Event-B. Malheureusement l'outil n'admet que des modèles Event-B conformes à un sous-ensemble réduit de la syntaxe Event-B. Ainsi, l'utilisateur est contraint à effectuer une étape de raffinement manuelle pour enlever les constantes symboliques et arranger les actions et les gardes du modèle à traduire.

L'outil EventB2Jml [79] génère un code JML de spécifications Java à partir de modèles décrits en Event-B. Il utilise le patron de conception visiteur pour parcourir l'arbre de syntaxe abstraite du modèle Event-B et génère des spécifications JML. Contrairement aux outils précédents, il n'exige pas à l'utilisateur une intervention de pré-traitement avant la traduction et il admet un sous-ensemble plus vaste de syntaxe Event-B. EventB2Jml a été étendu par l'outil EventB2Java [79] pour générer un code Java annoté par sa spécification JML.

L'outil Tasking Event-B [31, 33] permet de générer du code concurrent à partir de modèles Event-B. Le code concurrent produit est décrit dans divers langages de programmation à savoir C, Ada et Java. Contrairement aux autres outils de génération de code, l'outil Tasking Event-B permet d'annoter les modèles Event-B pour favoriser la génération de code. En particulier, l'utilisateur est censé ordonnancer les événements du modèle Event-B à traduire en utilisant un langage d'ordonnancement restreint.

2.4.6 Étude de cas en Event-B

Dans cette section, nous allons modéliser en Event-B l'application de l'Hôtel introduite dans la section 2.2.2.

2.4.6.1 Modèles Event-B proposés

Le modèle Event-B comportant le contexte `Chambre_C` et `Chambre_M` (voir le listing 2.19 et 2.20) modélise la notion de chambre dans l'application Hôtel. Il est similaire au composant `Chambre` décrit en BIP (voir le listing 2.5). Cependant, la machine `Chambre_M` apporte deux propriétés invariantes (`@inv4` et `@inv5`) permettant d'expliciter les liens entre les variables formant l'état de cette machine à savoir `cle_serrure`, `premier` et `second`. L'événement `passer_carte` qui est censé récupérer la carte à deux clés exige explicitement les conditions (`@grd3` et `@grd4`) afin de préserver `@inv4` et `@inv5` après avoir mis à jour `premier` et `second`. Contrairement à la transition `attribuer` du composant `Chambre` en BIP exigeant une garde (`cle_serrure==premier||cle_serrure==deuxieme`), dans l'événement `attribuer` de la machine `Chambre_M`, on déduit cette garde de l'invariant de la machine (voir la garde `@grd2` considéré comme théorème). L'action `@act2` de l'événement `attribuer` modélise en Event-B l'instruction if de BIP en couvrant les deux parties (then explicite et else implicite).

```
context Chambre_C
sets PLACE
constants dispo carte_valide chambre_attribuee

axioms
  @axm1 partition(PLACE,{dispo},{carte_valide},{chambre_attribuee})
end
```

Listing 2.19 – Contexte `Chambre_C`

```
machine Chambre_M sees Chambre_C

variables cle_serrure premier second place

invariants
  @inv1 cle_serrure ∈ ℕ
  @inv2 premier∈ℕ
  @inv3 second∈ℕ
  @inv4 premier≠second
  @inv5 cle_serrure ∈{premier,second}
  @inv6 place∈PLACE

events
  event INITIALISATION
    then
```

```
    @act1 cle_serrure , premier , second :| premier' ∈ℕ∧second' ∈ℕ∧
          premier' ≠second' ∧cle_serrure' ∈ℕ∧cle_serrure' ∈{premier',second'}
    @act2 place := dispo
  end

  event passer_carte
    any cle1 cle2
    where
      @grd1 cle1 ∈ℕ
      @grd2 cle2 ∈ℕ
      @grd3 cle1 ≠cle2
      @grd4 cle1 = cle_serrure
      @grd5 place = dispo
    then
      @act1 premier := cle1
      @act2 second := cle2
      @act3 place := carte_valide
  end

  event attribuer
    where
      @grd1 place = carte_valide
      @grd2 cle_serrure = premier ∨ cle_serrure = second
    then
      @act1 place := chambre_attribuee
      @act2 cle_serrure :| (( cle_serrure = premier) ⇒ (cle_serrure' = second)) ∧
            (( cle_serrure = second) ⇒ (cle_serrure' = cle_serrure))
  end

  event repasser_carte
    where
      @grd1 place = chambre_attribuee
  end

  event quitter
    where
      @grd1 place = chambre_attribuee
    then
      @act1 place := dispo
  end
end
```

Listing 2.20 – Machine `Chambre_M`

En outre, une machine Event-B peut être interprétée en tant qu'un système de transitions. La figure 2.14 représente le graphe de transitions associé à la machine `Chambre_M` dont les étiquettes des transitions sont les identifiants des évènements de la machine Event-B.

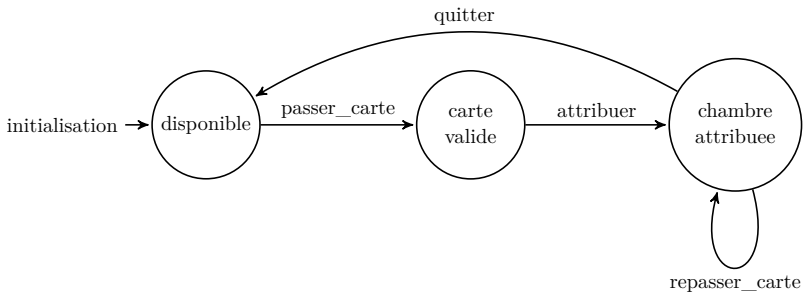

FIGURE 2.14 – Système de transitions associé à la machine `Chambre_M`

Le modèle Event-B permettant de modéliser la notion de client dans l'application Hôtel comporte un contexte `Client_C` et une machine `Client_M` (voir le listing 2.21 et 2.22). Vis-à-vis du composant `Client` en BIP (voir le listing 2.5), il apporte une propriété invariante (`@inv3`) permettant de décrire une contrainte primordiale entre les données `k1` et `k2`. De plus,

l'événement `prendre_carte` permet grâce à la garde (`@grd3`) de préserver l'invariant `@inv3`.

```
context Client_C
sets STATE
constants pas_de_carte  carte_initialisee  carte_a_valider

axioms
  @axm1 partition(STATE,{pas_de_carte},{carte_initialisee},{carte_a_valider})
end
```

Listing 2.21 – Contexte `Client_C`

```
machine Client_M sees Client_C

variables  k1 k2 s

invariants
  @inv1 k1∈ℕ
  @inv2 k2∈ℕ
  @inv3 k1≠k2
  @inv4 s∈STATE

events
  event INITIALISATION
    then
      @act1 k1,k2 :| k1'∈ℕ∧k2'∈ℕ∧k1'≠k2'
      @act2 s:=pas_de_carte
  end

  event passer_carte
    where
      @grd1 s= carte_initialisee
    then
      @act1 s:=carte_a_valider
  end

  event quitter
    where
      @grd1 s=carte_a_valider
    then
      @act1 s:=pas_de_carte
  end

  event repasser_carte
    where
      @grd1 s=carte_a_valider
    then
      @act1 s:=carte_a_valider
  end

  event prendre_carte
    any cle1  cle2
    where
      @grd1 cle1∈ℕ
      @grd2 cle2∈ℕ
      @grd3 cle1≠cle2
      @grd4 s=pas_de_carte
    then
      @act1 k1:=cle1
      @act2 k2:=cle2
      @act3 s:= carte_initialisee
  end
end
```

Listing 2.22 – Machine `Client_M`

La figure 2.15 montre le système de transitions qui interprète la machine Event-B `Client_M`.

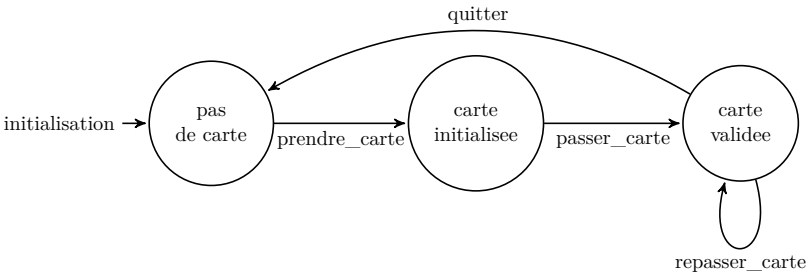

FIGURE 2.15 – Système de transitions associé à la machine `Client_M`

La machine `Hotel_CMP` (voir le listing 2.23) permet de composer les deux machines élémentaires `Chambre_M` et `Client_M` mentionnées dans la clause **includes**. L'architecture introduite par `Hotel_CMP` est décrite dans la clause **composes events**. Une telle architecture est la même que celle retenue en BIP en se limitant à deux instances (voir la figure 2.8).

```
composed machine Hotel_CMP
includes
[Hotel] Chambre_M (Invariant included)
[Hotel] Client_M (Invariant included)

composes events
INITIALISATION
combines event
[Hotel] Chambre_M·INITIALISATION | | [Hotel] Client_M·INITIALISATION

passer_carte
combines event
[Hotel] Chambre_M·passer_carte | | [Hotel] Client_M·passer_carte

attribuer
combines event
[Hotel] Chambre_M·attribuer

repasser_carte
combines event
[Hotel] Chambre_M·repasser_carte | | [Hotel] Client_M·repasser_carte

quitter
combines event
[Hotel] Chambre_M·quitter | | [Hotel] Client_M·quitter

prendre_carte
combines event
[Hotel] Client_M·prendre_carte

end
```

Listing 2.23 – Machine composée `Hotel_CMP`

La figure 2.16 représente le système de transitions qui interprète la machine de composition `Hotel_CMP`. Un tel système est obtenu par composition des deux systèmes de transitions associés respectivement aux machines Event-B `Chambre_M` et `Client_M`.

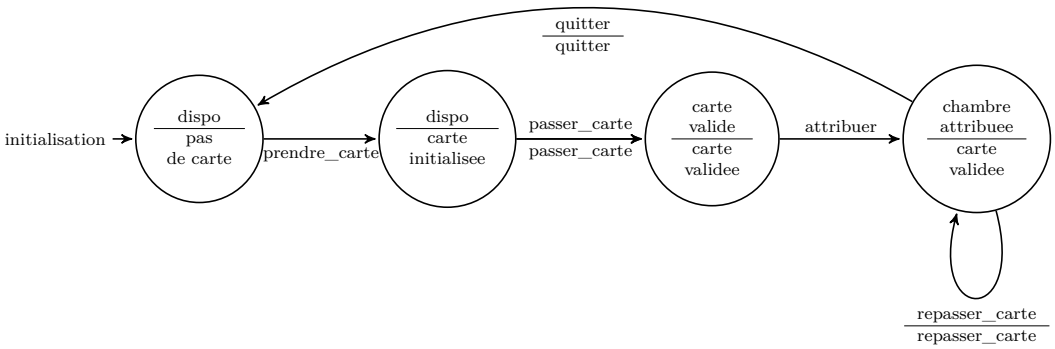

FIGURE 2.16 – Système de transitions associé à la machine composée `Hotel_CMP`

2.4.6.2 Preuve

Les obligations de preuve liées aux deux machines `Chambre_M` et `Client_M` ont été déchargées aussi bien par les prouveurs internes qu'externes de la plateforme Rodin. Ainsi, on peut dire que ces deux machines sont cohérentes vis-à-vis des critères de correction élaborés par Event-B tels que établissement de l'invariant par l'événement `initialisation`, préservation de l'invariant par les autres événements et la faisabilité des actions non déterministes. De

même, nous avons prouvé la cohérence de la machine composée `Hotel_CMP` expansée comme machine ordinaire en Event-B. Une telle machine est décrite dans l'annexe par le listing A.1.

2.4.6.3 Vérification comportementale

Les modèles Event-B cohérents ne sont pas forcément valides. En effet, les critères de correction établis par la théorie Event-B ne couvrent pas toutes les erreurs. En Event-B, l'activité de validation des modèles Event-B cohérents est supportée essentiellement par la boîte à outils ProB permettant : animation, model-checking des propriétés de sûreté et d'absence de blocage, vérification de la correction de la relation de raffinement, model-checking temporel et résolution des problèmes de satisfaction de contraintes (CSP : Constraint Satisfaction Problem). Nous avons animé les trois machines `Chambre_M`, `Client_M` et `Hotel_CMP_M0` avec succès. Ainsi, nous avons rejoué le scénario 1 (voir la section 2.3.4.2) sur la machine `Hotel_CMP_M0`. Également, nous avons vérifié l'absence de blocage par model-checking des trois machines `Client_M`, `Chambre_M` et `Hotel_CMP_M0`.

En outre, grâce au model-checker temporel de ProB, nous avons vérifié sur ces machines plusieurs propriétés de vivacité. Par exemple, nous avons vérifié, à l'aide du model-checker temporel ProB, sur la machine `Chambre_M` les propriétés dynamiques suivantes :

(P1) Après avoir passé la carte, la chambre peut être attribuée. Ceci est formalisé en LTL^e comme suit :

$$\boxed{\mathbf{G}\ ([passer\_carte] \Rightarrow \mathbf{X}(e(attribuer)))}$$

.

(P2) Une fois la chambre attribuée, on peut repasser la carte tant qu'on ne l'a pas jetée (Weak Until). Ceci est formalisé en LTL^e comme suit :

$$\boxed{\mathbf{G}\ ([passer\_carte] \Rightarrow \mathbf{X}\ (e(repasser\_carte)\ \mathbf{W}\ [quitter]))}$$

.

2.4.6.4 Discussion

La méthode formelle Event-B couplée à la boîte à outils ProB permet effectivement le développement de composants atomiques corrects et valides. En effet, les obligations de preuve induites par Event-B favorisent l'obtention de propriétés de sûreté et la correction du processus formel basé sur des raffinements successifs. De plus, le model-checker temporel ProB permet la vérification des propriétés de vivacité. Enfin, l'animateur de ProB permet de faire participer le client à la validation des modèles formels Event-B du futur système. La construction **composed machine** offre une structure d'accueil permettant la composition des machines Event-B. Cependant des efforts liés à l'adjonction d'un invariant de composition et la démonstration des obligations de preuve doivent être fournis par le modélisateur afin de finaliser le composant composite établi. Dans ce travail, nous allons apporter une méthode autour d'Event-B permettant la décomposition formelle des spécifications centralisées afin d'obtenir des sous-spécifications élémentaires correctes (voir le chapitre 3).

2.4.7 Évaluation

La méthode formelle Event-B permet le développement des systèmes **cohérents** et **valides**. Pour y parvenir, elle offre un cadre **homogène** favorisant trois activités : modélisation, vérification et validation. Ces activités sont **complémentaires**. L'activité de modélisation

s'appuie sur un langage formel (langage logico-ensembliste pour les données et langage d'actions simple pour les traitements) et un processus formel basé sur le concept de raffinement avec preuves mathématiques. Elle permet d'élaborer des modèles formels du futur système. L'activité de vérification a pour objectif de **vérifier formellement** la cohérence des modèles Event-B établis. Pour ce faire, Event-B propose des critères de correction dits **obligations de preuve** (lemmes mathématiques à démontrer dont les énoncés sont générés automatiquement à partir des modèles) permettant de garantir la correction des modèles abstraits et raffinés. Ainsi, Event-B supporte le paradigme **correct par construction** [8].

Le processus formel induit par la méthode Event-B est basé sur la technique de raffinement avec preuves mathématiques. Event-B supporte deux techniques de raffinement : horizontal et vertical. Le raffinement horizontal permet l'obtention d'une façon incrémentale d'une spécification **abstraite cohérente** et **valide** tandis que le raffinement vertical permet de concrétiser pas-à-pas la spécification du futur système issue de la phase de raffinement horizontal. La correction de la relation de raffinement est assurée par des obligations de preuve qui ne couvrent pas les propriétés de vivacité. Ceci peut être surmonté partiellement en se servant du model-checker temporel de ProB.

La plateforme Rodin qui implémente Event-B apporte des démonstrateurs automatiques des théorèmes de plus en plus puissants [5, 28]. L'activité de validation vise la vérification des propriétés de vivacité sur des modèles Event-B. Une telle activité s'appuie essentiellement sur la boîte à outils de ProB : animation, model-checking des propriétés d'invariance et de blocage, model-checking temporel des propriétés de vivacité et résolution des contraintes. D'une façon native, Event-B ne permet ni la composition ni la décomposition de modèles. Des travaux comme [11, 25, 24, 36, 73, 74, 75] ont permis d'ouvrir Event-B sur le paradigme composant. La composition et la décomposition par événements partagés (voir la section 2.4.4.2) entraînent des difficultés liées à l'adjonction d'un invariant de composition et à la démonstration des obligations de preuve de la machine composite et atomique. En outre, Event-B ne supporte pas l'instanciation. En effet, une machine (**machine**) ou une machine composée (**composed machine**) est considérée comme **module** (au sens Package Ada) et non pas comme **type** (design-time). Ceci ne permet pas de construire des configurations comportant plusieurs instances de même type. Enfin, la séparation entre les aspects comportementaux et architecturaux d'une architecture Event-B n'est pas nette.

2.5 Conclusion

Tout d'abord, nous avons souligné l'importance des systèmes distribués et identifié les difficultés inhérentes au développement de ces systèmes. Ensuite, nous avons présenté, expérimenté et évalué les aptitudes du langage BIP de modélisation et programmation des systèmes à base de composants. Pour y parvenir, nous avons programmé en BIP l'application de l'Hôtel [8, 45]. Enfin, nous avons exhibé les possibilités de modélisation formelle d'Event-B aussi bien native (**context**, **machine**) qu'augmentée (**composed machine**). De même, nous avons exploré les outils favorisant la correction de modèles Event-B issus de la plateforme Rodin : prouveurs, model-checkers, animateurs, simulateurs et générateurs de code. Également, nous avons développé en Event-B l'application de l'Hôtel en suivant une approche par composition d'événement partagé. Ceci nous a permis d'apprécier les primitives de composition fournies par Event-B.

BIP apporte un cadre homogène pour la modélisation et la programmation des systèmes à base de composants. Il supporte un modèle de composants qui distingue d'une façon nette les aspects comportementaux et architecturaux. Mais il retarde la vérification des programmes BIP a posteriori. Event-B supporte le paradigme correct par construction. Event-B couplée à la boîte à outils ProB couvre plusieurs classes de propriétés de sûreté, de vivacité, d'équité

et d'absence de blocage. Mais la composition et décomposition formelle en Event-B n'est pas une tâche facile. En outre, la génération de code à partir d'un modèle Event-B (simple ou composite) n'a pas de solution universelle. Dans le cadre d'Event-B couplée à BIP, nous allons adresser la décomposition formelle d'une spécification centralisée décrite en Event-B et la génération de code BIP à partir d'un modèle composite ultime Event-B. Pour y parvenir, nous allons entre autres utiliser avec profit des raffinements automatiques guidés par des DSL. Nous illustrons la démarche proposée sur l'application Hôtel introduite dans ce chapitre.

Chapitre 3

Démarche de développement de systèmes distribués

Sommaire

3.1 Introduction

Un système distribué est un ensemble d'entités autonomes de calcul interconnectées et qui peuvent communiquer. Concevoir de tels systèmes, les prouver, les valider, les implanter et les tester nécessite des plateformes de spécification formelle, d'implantation et des processus de développement adéquats. Dans le but d'apporter une solution aux concepteurs, nous avons élaboré une méthodologie de développement de systèmes distribués corrects par construction fondée sur des raffinements automatiques guidés par des DSLs [66, 65]. Cette méthodologie est bâtie sur un processus de développement **sûr** basé sur la méthode formelle Event-B [8] et la plateforme à base de composants BIP [70]. Event-B est utilisée pour la spécification et la décomposition formelles des systèmes distribués. BIP est utilisée pour l'implantation et le déploiement de systèmes distribués spécifiés, prouvés et validés en Event-B. Le passage d'Event-B vers BIP est assuré par un générateur de code. En effet, compte tenu de la proximité sémantique (voir les sections 2.3.2 et 2.4.1.3) et de la complémentarité méthodologique (approche descendante et approche ascendante) d'Event-B et de BIP, nous proposons leur couplage dans notre processus de développement. Le schéma général de notre processus de développement est donné par la figure 3.1.

Côté Event-B, on distingue deux activités : raffinement et décomposition symbolisées respectivement par $\dashrightarrow$ et $\otimes$. L'activité de raffinement permet d'introduire progressivement les exigences fonctionnelles du futur système. L'activité de décomposition assure pas à pas la prise en compte des aspects architecturaux du futur système. Les modèles Event-B intermédiaires comme `M0`, `N` et `P` ne sont pas traduits en BIP. Tandis que les modèles Event-B ultimes qui correspondent aux noeuds terminaux de l'arbre de développement en Event-B sont traduits en BIP comme des composants atomiques. Ces derniers sont regroupés au sein des composants composites BIP -d'une façon hiérarchique- en remontant les opérations de décomposition.

Dans cette optique, 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 model-checker où la preuve d'un jugement est automatique et un assistant de preuve où l'utilisateur doit composer des stratégies de base afin de démontrer un théorème. De même qu'un assistant de preuve aide à construire la preuve d'un théorème, notre objectif est d'aider les spécifieurs à l'élaboration par raffinements d'un modèle distribué. En effet, le processus de développement correct par construction de systèmes distribués préconisé [67, 69] combine des raffinements manuels et automatiques et se termine par la génération de code BIP. Les raffinements manuels horizontaux permettent l'introduction progressive de propriétés du futur système. Les raffinements manuels verticaux permettent

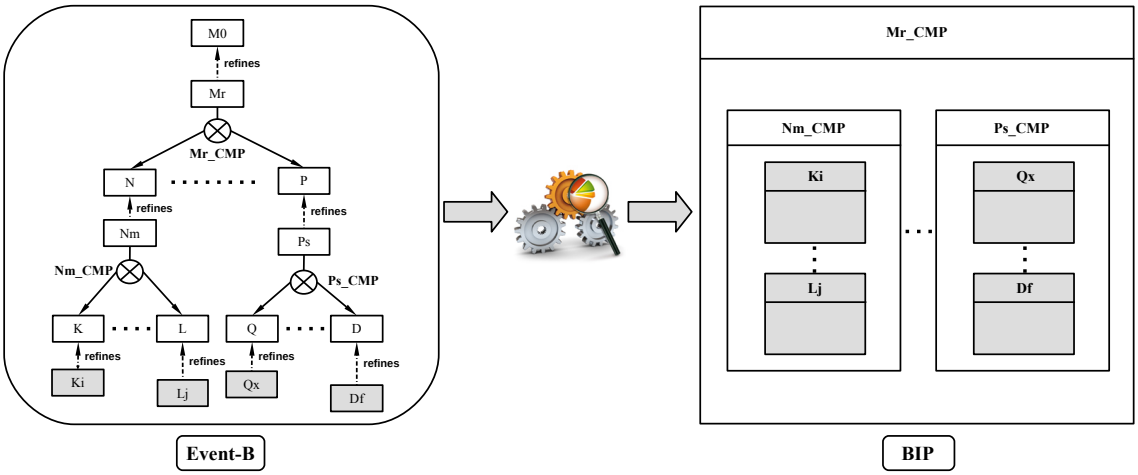

FIGURE 3.1 – Schéma général de notre processus de développement

l'obtention de modèles Event-B traduisibles vers BIP : déterminisation d'actions et concrétisation des données. Les raffinements automatiques seront guidés par l'utilisateur via des DSLs pour introduire les spécificités de l'architecture logicielle retenue. Ils permettent de prendre en considération les exigences de la Distribution.

Ce chapitre comporte six sections. La deuxième aborde, via des exemples bien ciblés, la représentation de la Distribution en Event-B et sa traduction en BIP. La troisième propose une méthodologie de développement de systèmes distribués en combinant Event-B et BIP. Les sections 3.4, 3.5 et 3.6 sont consacrées respectivement aux trois étapes formant notre méthodologie à savoir Fragmentation, Distribution et Génération de code BIP. Enfin, la section 3.7 fournit des éléments techniques liés à l'implémentation des plug-ins qui supportent notre méthodologie de développement des systèmes distribués.

3.2 Event-B distribuée et BIP

Dans cette section, nous présentons des schémas de définition d'événements Event-B modélisant la synchronisation d'événements exécutés sur les sous-composants. Ces schémas doivent correspondre au niveau BIP à des connecteurs n'effectuant que du transfert de données conformément à la décomposition par événement partagé (voir la section 2.4.4.2). Ils seront vus comme des éléments d'un langage `Event-B0` [2] ciblé par les étapes de raffinement automatiques ou manuelles.

3.2.1 Machine décomposable et événement de synchronisation

Notre objectif étant le développement de modèles distribués exécutables via leur transcription en BIP, nous devons être en mesure de représenter la répartition en Event-B. Ceci passe par la notion de décomposition avec synchronisation sur les événements partagés telle qu'introduite par Butler [74]. Les annotations de placement des variables sur les sous-composants permettent d'associer à une machine décomposable un ensemble de machines BIP composées via des connecteurs exprimant la sémantique des événements de synchronisation. Chaque événement correspond ainsi à un connecteur relié aux composants synchronisés via un port

2. À l'instar de `B0` impératif de la méthode B, `Event-B0` est notre sous-ensemble d'Event-B traduisible systématiquement en BIP.

spécifique ayant accès en lecture/écriture à certaines de leurs variables. Les événements projetés deviennent des transitions synchronisées sur ces mêmes ports. L'évaluation des gardes et l'exécution des actions de l'événement Event-B sont réalisés par le composant ou par le connecteur selon la localisation des données accédées. Les paramètres représentent soit un transfert de données soit du non-déterminisme qui devra être éliminé afin d'obtenir un modèle exécutable. Après avoir soulevé le problème de la décomposition d'événements non déterministes, nous détaillons les schémas de synchronisation autorisés.

3.2.2 Utilisation de connecteurs actifs

Nous présentons ici un premier schéma d'utilisation de BIP respectant la répartition des données. Un tel schéma suppose que les événements Event-B soient déterministes, c'est-à-dire ne comportent pas de paramètres locaux et que les actions soient des affectations (voir le listing 3.1). On introduit alors un composant BIP par composant Event-B. Chaque sous-composant BIP (C_i) contient les données (v_i) qui lui sont affectées et un port `ev` y donnant accès. A chaque événement Event-B est associé un connecteur BIP réalisant le test de la garde (clause `provided`) et l'exécution de l'action (clause `down`). Ce connecteur comporte n ports, un par composant, donnant accès en lecture/écriture aux variables de chaque composant (voir la figure 3.2).

```
machine distribution
variables
  v1 // sur C1
  ...
  vn // sur Cn
events
  event ev
  when
    @g P(v1,...,vn)
  then
    @a1 v1 := f1(v1,...,vn)
    ...
    @an vn := fn(v1,...,vn)
  end
end
```

Listing 3.1 – Machine annotée

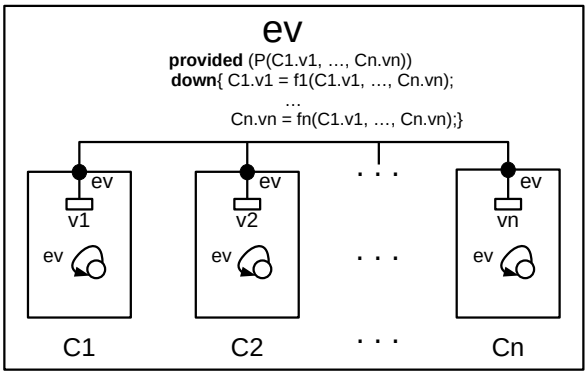

FIGURE 3.2 – Connecteur actif

Cette solution, bien que correcte du point de vue sémantique n'est pas retenue puisque nous souhaitons que les traitements soient réalisés par les composants, les connecteurs ne réalisant que de la synchronisation et du transfert de données. Ainsi comme dans les langages architecturaux (ADL), nous respectons la séparation entre les traitements et la synchronisation. Dans les propositions suivantes, les composants seront les éléments actifs du modèle et réaliseront les évaluations des gardes et des expressions contenues dans les actions. Il est à noter que nous ne distinguons pas ici composant logique et physique. Ainsi les traitements sont affectés aux composants logiques qui deviendront implicitement des composants physiques.

3.2.3 Décomposition d'un événement non déterministe

Une des caractéristiques d'une machine Event-B est d'exprimer le non-déterminisme. Ce non-déterminisme provient soit de l'existence de plusieurs événements déclenchables dans un état donné, soit d'un non-déterminisme interne à un événement. Considérons ce deuxième cas et le listing 3.2.

```
variables
  c1 // sur C1
  c2 // sur C2
event ndet
  any i
  where
    @i1 i < c1 // sur C1
    @i2 i < c2 // sur C2
  end
```

Listing 3.2 – Événement non déterministe

```
variables
  c1
event ndet
  any i
  where
    @i1 i < c1
  end
```

Listing 3.3 – Projection sur C1

```
variables
  c2
event ndet
  any i
  where
    @i2 i < c2
  end
```

Listing 3.4 – Projection sur C2

Cette machine est décomposable sur les composants `C1` et `C2`. L'événement `ndet` devient alors un événement de synchronisation non déterministe. Ces deux projections (voir le listing 3.3 et le listing 3.4) réalisent un choix non-déterministe de valeur pour le paramètre `i`. Comme en CSP, si les choix sont identiques, l'événement synchronisé sera déclenchable. Cependant, les projections étant non déterministes, le développement en principe séparé de chaque composant consiste à restreindre les choix de valeurs pour `i` et à proposer un algorithme déterminant une solution locale. La synchronisation peut alors devenir impossible et conduire à un interblocage. Afin d'éviter ce problème, les contraintes portant sur les paramètres d'un événement de synchronisation doivent être résolues sur un site unique. Nous proposons un outil d'aide à l'élimination du non déterminisme distribué : la transformation de Fragmentation (voir la section 3.4). Celle-ci permettra de supprimer les paramètres des événements synchronisés en extrayant les événements déterminant leurs valeurs.

3.2.4 Transfert de données

Ce premier schéma représente le transfert d'une donnée stockée dans la variable `x` du composant `C0` vers les variables `xi` des composants `Ci` (voir le listing 3.5 et la figure 3.3). L'événement centralisé exporte la valeur de `x` via le paramètre `vx`. Ce schéma est représenté en BIP par un connecteur n-aire dont l'action `down` recopie la valeur de `x` lue via le port `p0` dans chacune des variables `xi` modifiées via les ports `pi`. La forme de l'événement rend explicite le transfert de la donnée et la localisation des actions sur les composants, la partie droite des affectations ne faisant pas référence à des variables distantes.

```
event cnt
  any vx
  where
    @g1 vx=x // sur C0
  then
    @a1 x1:=vx // sur C1
      .
      .
      .
    @an xn:=vx // sur Cn
end
```

Listing 3.5 – Événement `cnt`

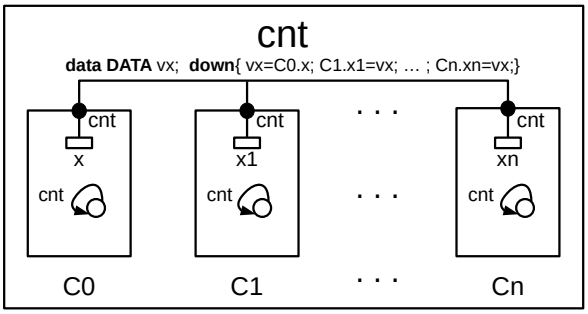

FIGURE 3.3 – Connecteur n-aire `cnt`

3.2.5 Utilisation de la valeur d'une variable distante dans une action

Le schéma introduit ici généralise le précédent en modélisant l'utilisation dans une expression quelconque de la valeur reçue afin de mettre à jour l'état des composants récepteurs.

Côté Event-B, le paramètre de synchronisation apparaît dans l'expression affectée. Côté BIP, une action doit être ajoutée dans chaque sous-composant récepteur pour exploiter la valeur transférée. Il est important ici de noter que les actions des sous-composants sont exécutées après les transferts de données. La sémantique du modèle BIP est donc conforme à celle du modèle Event-B.

```
event cnt
  any vx
  where
    @g1 vx=x // sur C0
  then
    @a1 a1:=a1+vx // sur C1
      .
      .
      .
    @an an:=an+vx // sur Cn
end
```

Listing 3.6 – Événement `cnt`

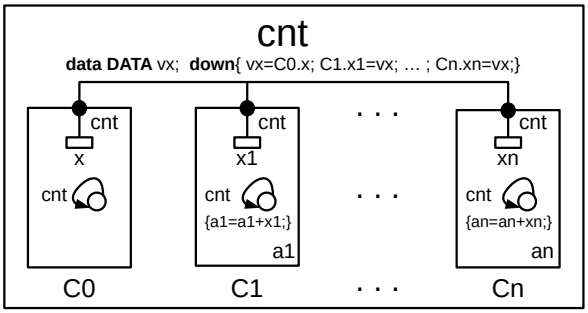

FIGURE 3.4 – Connecteur n-aire `cnt`

3.2.6 Utilisation de la valeur d'une variable distante dans une garde

Le schéma ci-dessous modélise une synchronisation n-aire dans laquelle chaque composant compare la valeur transmise avec une donnée locale.

```
event cnt
  any vx
  where
    @g vx=x // sur C0
    @g1 vx > a1 // sur C1
    ...
    @gn vx > an // sur Cn
  end
```

Listing 3.7 – Événement `cnt`

3.2.6.1 Tentatives de modélisation BIP

Une traduction directe en BIP (voir la figure 3.5) inspirée du paragraphe précédent serait incorrecte puisque le transfert de données est réalisé après le test (ou évaluation) des gardes des transitions des différents sous-composants. Le test serait donc réalisé avec les anciennes valeurs des variables `xi` recevant la copie de `x`.

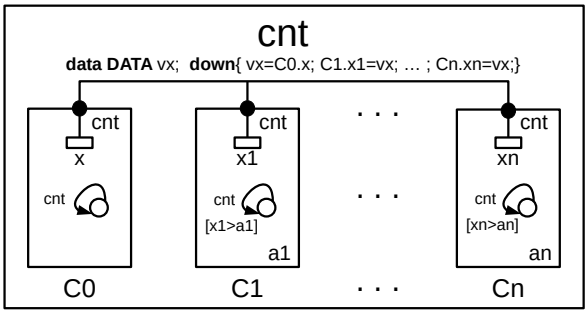

FIGURE 3.5 – Connecteur n-aire `cnt` incorrect

Afin de prendre en compte la valeur transmise, les gardes devraient être testées dans le connecteur qui devrait donc effectuer un accès aux `ai` puis le test des n gardes (voir la figure 3.6). Ayant restreint l'usage des connecteurs au transfert de données, ce schéma est donc aussi considéré comme incorrect.

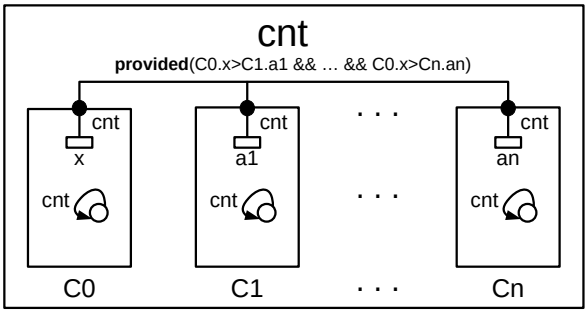

FIGURE 3.6 – Connecteur n-aire `cnt` interdit

Le modèle Event-B ne fait donc pas partie du sous-ensemble Event-B0 et doit être transformé.

3.2.6.2 Le modèle transformé

Les gardes ne pouvant utiliser que des données locales, des copies des données distantes sont introduites. Ici, nous déclarons les variables `xi` localisées sur les composants `Ci` et destinées à recevoir une copie de `x`. La variable booléenne `x_fresh` indique si les copies sont à jour. Cette mise-à-jour est réalisée par l'événement `share_x`. L'événement `cnt` compare maintenant chaque copie locale avec la donnée locale du composant lorsque les copies sont à jour (voir le listing 3.8). La section 3.5 généralisera cette transformation et l'introduira comme un raffinement du modèle initial afin de permettre d'en montrer la correction.

```
variables
  C1_x // copie de x sur C1
  ...
  Cn_x // copie de x sur Cn
  x_fresh // sur C0, indicateur de validite des copies
invariants
  @inv1 x_fresh = TRUE ⇒ C1_x = x
  ...
  @invn x_fresh = TRUE ⇒ Cn_x = x
events
  event share_x
  any vx
  where
    @vx vx = x // sur C0
    @nfr x_fresh = FALSE // sur C0
  then
    @C1_x C1_x := x // sur C1
    ...
    @Cn_x Cn_x := x // sur Cn
    @fri x_fresh := TRUE // sur C0
  end

  event cnt
  any vx
  where
    @fr x_fresh = TRUE // sur C0
    @g1 C1_x > a1 // sur C1
    ...
    @gn Cn_x > an // sur Cn
  end
```

Listing 3.8 – Raffinement introduisant des copies locales

Le modèle BIP (voir la figure 3.7) introduit deux connecteurs n-aires, un par événement. Le premier, `share_x` effectue la copie de la variable `x` sur chaque composant tandis que `cnt` synchronise les tests des gardes devenues locales.

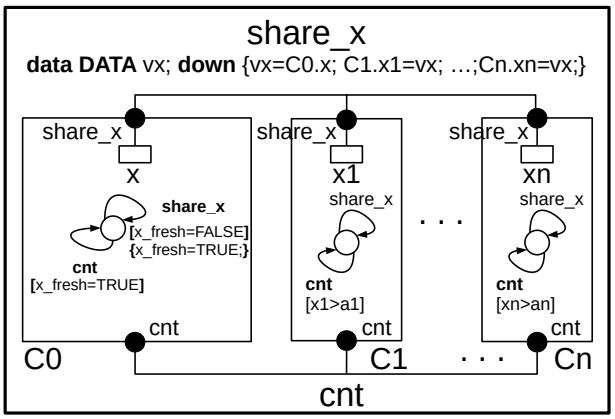

FIGURE 3.7 – Les deux connecteurs n-aire `cnt` et `share_x`

3.3 Méthodologie de développement

Dans cette section, nous proposons un processus de développement de systèmes distribués couplant Event-B et BIP.

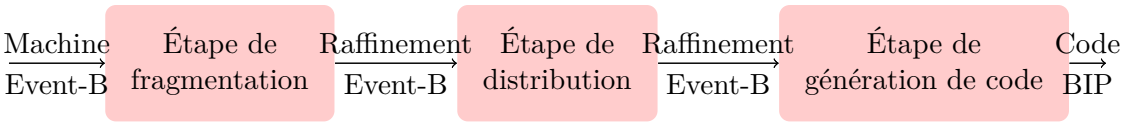

FIGURE 3.8 – Les étapes de notre méthodologie de développement

3.3.1 Vue d'ensemble

La figure 3.8 illustre notre méthodologie structurée en trois étapes : Fragmentation, Distribution et génération de code. En effet, 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 via deux langages dédiés. Ensuite, nous générons un modèle BIP qui peut être exécuté sur une plateforme distribuée via le moteur d'exécution de la chaîne d'outils BIP. Notons que la correction de cette méthode repose sur la correction des étapes de raffinement et de la traduction finale en code BIP.

L'application de notre méthodologie utilise et raffine la spécification Event-B du système à développer. Une telle spécification est obtenue par raffinements successifs en prenant en compte progressivement les exigences du système. Ces raffinements sont exprimés à l'aide de machines abstraites décrites en Event-B. Ensuite, nous proposons des schémas de raffinements destinés à prendre en compte la nature distribuée du système étudié. L'étape de

Fragmentation prend en entrée le modèle centralisé Event-B. 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. En se basant sur cette description et le modèle Event-B abstrait, l'étape de Fragmentation génère un modèle Event-B qui raffine son correspondant abstrait. Après l'étape de Fragmentation, l'étape de gestion de la Distribution prend en compte les particularités de l'architecture cible. Il s'agit 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). Une telle étape prend en entrée le modèle raffiné issu de l'étape de Fragmentation et une spécification de Distribution. Celle-ci est décrite par le spécifieur à l'aide d'un DSL. Elle répartit les variables, les actions et éventuellement les gardes du modèle Event-B sur les sous-composants. En conséquence, on obtient un ensemble de machines qui interagissent, dont la composition est prouvée correcte et conforme avec le niveau abstrait. L'ultime étape de notre méthodologie est l'étape de génération de code BIP. Elle prend en entrée les composants Event-B issus de l'étape de Distribution. Elle génère un modèle BIP. Les composants Event-B sont traduits en composants atomiques BIP. Les machines de composition sont représentées par une hiérarchie de connecteurs BIP.

3.3.2 Mérites

Notre méthodologie de développement combine plusieurs techniques afin de produire des systèmes distribués sûrs. Ces techniques englobent spécification, raffinement manuel, génération de code, langages dédiés et raffinement automatique. Les trois premières techniques sont couramment utilisées lors d'un processus formel de développement en Event-B. Nous avons combiné les deux techniques DSL et raffinement automatique afin d'apporter une assistance significative aux spécifieurs des systèmes distribués couplant Event-B et BIP. En effet, nous avons identifié deux sortes de raffinement automatique : Fragmentation (voir la section 3.4) et Distribution (voir la section 3.5). Ces deux étapes de raffinement sont récurrentes dans l'activité de développement descendant d'un système distribué : passage d'une spécification centralisée vers une spécification distribuée. Les besoins potentiels de deux étapes Fragmentation et Distribution sont exprimés par deux DSLs adéquats offerts au spécifieur afin d'exprimer ses exigences spécifiques. Le DSL de l'étape de Fragmentation permet de décrire l'ordre linéaire (penser au problème de tri topologique) des paramètres locaux d'un événement jugé comme événement de synchronisation dont les paramètres doivent être supprimés (voir la section 3.2.3). Le DSL de l'étape de Distribution permet de décrire l'architecture retenue par le spécifieur. Il a un pouvoir expressif supérieur à celui de l'outil SEDT développé par et autour de Butler (voir la section 2.4.4.4). Les aides à l'activité de raffinement sont souvent appréciées par les spécifieurs (voir la section 2.4.2.2).

C'est dans ce cadre que s'inscrivent les deux étapes de Fragmentation et de Distribution avec une originalité liée à la convivialité des DSLs et la clarté de l'objet des deux étapes de raffinement. Enfin Event-B n'est pas dotée d'un outil standard de génération de code (voir la section 2.4.5). Notre méthodologie de développement des systèmes distribués propose un générateur de code distribué BIP à partir d'un modèle Event-B distribué (voir la section 3.6).

3.4 L'étape de Fragmentation

Dans la section 3.2.3, nous avons souligné le besoin de supprimer les paramètres locaux d'un événement de synchronisation. Pour y parvenir, nous appliquons un schéma de raffinement appelé Fragmentation (voir la figure 3.9). Cette étape de raffinement automatique est guidée par le spécifieur. Celui-ci est censé décrire l'ordre de calcul linéaire des paramètres à travers un langage dédié (DSL). En se basant sur cette description et le modèle Event-B

abstrait, l'étape de Fragmentation génère un nouveau modèle Event-B. Intuitivement, cette étape de raffinement automatique s'appuie sur des règles simples assurant l'obtention d'un raffinement correct : 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.

La figure 3.9 illustre le schéma de la transformation de Fragmentation qui admet comme entrée une machine Event-B et une spécification de Fragmentation dont la structure est décrite par un langage dédié.

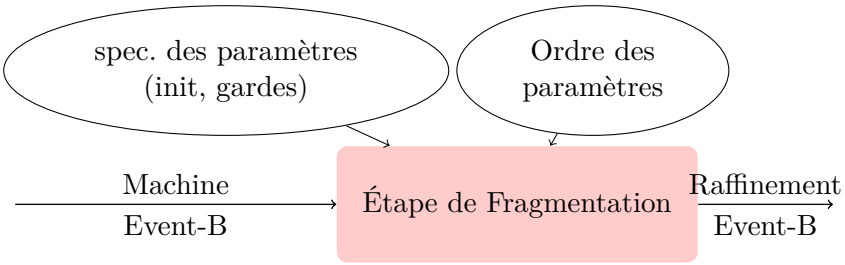

FIGURE 3.9 – L'étape de fragmentation

La transformation de Fragmentation est implantée par un plug-in Rodin (voir la section 3.7). Elle génère un raffinement de la machine Event-B fournie en entrée à partir d'une **spécification de la Fragmentation**. Celle-ci comporte des déclarations donnant pour un paramètre p d'un événement ev de la machine fournie les paramètres p_i (i allant de 1 à n) dont il dépend et les gardes g_k (k allant de 1 à m) le spécifiant. Une expression v est requise pour initialiser la variable globale introduite. La structure générale de ce DSL est donnée par le listing 3.9. Un tel DSL permet de décrire un ordre partiel sur les paramètres des événements concernés événement par événement. La clause `when` exprime les liens de précédence inter-paramètres.

```
splitting  output_machine refines input_machine
  events
    event ev
      [system] parameter p1 init  v1 with g1 ... gn
      ...
      when p1 ... pn [system] parameter p init v with gm ... gz
      ...
    event ev'
      ...
    ...
end
```

Listing 3.9 – DSL de Fragmentation

3.4.1 Schéma de transformation

La Fragmentation produit une machine Event-B dont le schéma général est donné par le listing 3.10. Elle implante les contraintes d'ordonnancement par l'introduction de deux variables d'état par paramètre `p` : la variable `ev_p` contient la valeur calculée pour le paramètre et la variable booléenne `ev_p_computed` indiquant si le paramètre a été calculé. L'invariant de la machine est étendu par les propriétés vérifiées par chaque paramètre : si un paramètre `p` a été calculé (`ev_p_computed = TRUE`), sa spécification, donnée par ses gardes `gi`, est alors satisfaite. Le calcul de la valeur d'un paramètre `p` est réalisé par un événement nouvellement

introduit (`compute_ev_p`). Celui-ci attend que les paramètres dont il dépend aient été calculés et sauvegarde dans la variable d'état `ev_p` une valeur satisfaisant les gardes spécifiant le paramètre. Les événements introduits par l'étape de Fragmentation -événement par paramètre local- de type `system`[3] sont déclarés convergents puisque l'événement abstrait `ev` doit finir par se produire. La progression du calcul des paramètres est assurée par un variant défini comme le nombre de paramètres restant à calculer. Lorsque tous les paramètres d'un événement abstrait ont été calculés, l'événement en question devient franchissable et nous savons que les gardes des anciens paramètres sont vérifiées par les variables globales qui leur sont associées. Cette information est ajoutée en tant que théorème afin de pouvoir être exploitée dans les projections sur les sous-composants. Dans la machine générée, les événements *transformés* n'ont plus de paramètres. Cependant, les événements introduits peuvent en comporter. Ils expriment alors un non-déterminisme local dont l'élimination est laissée à la charge des étapes ultérieures de raffinement.

```
machine output_machine refines input_machine
 variables
  ev_p  ev_p_computed // témoin et statut pour le paramètre 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) + · · · // pour chaque paramètre
 events
   event INITIALISATION extends INITIALISATION
   then
     @ev_p ev_p := v //v issue de la spécification de la fragmentation
     @ev_p_comp ev_p_computed := FALSE
   end
   convergent event compute_ev_p // calcule le  paramètre p de ev
   any p where
    @gi gi  // gardes spécifiant  p
    @pi ev_pi_computed = TRUE // les paramètres dont p dépend ont été calculés
    @p ev_p_computed = FALSE // p reste à calculer
   then
     @a ev_p := p // sauvegarde de p dans la variable d'état
     @computed ev_p_computed := TRUE // assure la décroissance du variant
   end
   event ev  refines  ev
   when
     @p_comp ev_p_computed = TRUE // si p a été calculé
     theorem @gi [p:=ev_p]gi  // les gardes sur p sont vérifiées par ev_p
   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
     // ev_pi_computed := FALSE pour chaque paramètre pi dont la valeur est
       impactée par les actions
   end
end
```

Listing 3.10 – Machine générée pour le raffinement de Fragmentation

Il est à noter que nous obtenons un raffinement de la machine d'entrée suite à la transformation de Fragmentation. La machine transformée est en principe correcte mais la technologie Event-B utilisée ne permet pas de générer facilement les preuves des propriétés exprimant cette correction. L'utilisateur doit donc décharger les obligations de preuve interactivement. Il pourra aussi vérifier que le raffinement n'introduit pas de blocages, mais cette propriété n'est pas garantie comme le montre l'exemple de la section 3.4.3.2.

Un autre point important concerne la réinitialisation des variables `ev_p_computed`. Il est en effet important de réactiver le calcul d'un paramètre lorsque les données dont il dépend évoluent. Ceci se produit lorsqu'un événement met à jour une variable utilisée par les gardes

3. Par défaut, les paramètres sont calculés par des événements `ordinaires`

spécifiant le paramètre `p`.

3.4.2 Paramètres système et environnementaux

La Fragmentation introduit de nouveaux événements calculant dans des variables globales la valeur des paramètres des événements de la machine raffinée. Nous avons déclaré ces événements convergents afin d'assurer qu'ils ne prennent pas indéfiniment le contrôle. Cette propriété se justifie lorsque le calcul effectué par ces événements est une action interne au **système**. La séquence d'actions doit se terminer pour assurer sa réactivité. Cependant, si ces événements concernent l'**environnement**, leur convergence ne se justifie plus. Il peut s'agir par exemple de proposer la valeur de paramètres. Ces valeurs peuvent être modifiées à volonté avant l'interaction avec le système. Il est donc nécessaire de déclarer les paramètres dont le calcul ne doit pas être relancé. On introduit pour cela le mot clé `system`. Seuls les paramètres `system` seront convergents. Les autres paramètres ne le seront pas et leur calcul ne sera pas conditionné par la garde `ev_p_computed = FALSE`.

3.4.3 Exemples

Afin d'illustrer la transformation de Fragmentation, nous allons considérer cinq exemples simples, le premier comportant des paramètres indépendants, le second comportant de multiples paramètres, le troisième comporte des événements multiples inter-dépendants, le quatrième comportant des événements multiples indépendants, et le cinquième ne comportant pas d'ordre linéaire. Dans ces exemples, les paramètres seront qualifiés `system` pour illustrer l'expression du variant.

3.4.3.1 Paramètres indépendants

La machine `param_indep` (voir le listing 3.11) comporte un événement `ev` ayant trois paramètres indépendants. Ainsi, l'ordre de calcul de ces paramètres `p`, `q` et `r` est sans importance (voir le listing 3.12).

```
machine param_indep
variables
  x
invariants
  @x_typ x ∈ ℕ
events
  event INITIALISATION
  then
    @x x :∈ℕ
  end
  event ev any p q r
  where
    @p p∈ℕ
    @q q∈ℕ
    @r r∈ℕ
  then
    @x x:=p+q+r
  end
end
```

Listing 3.11 – La machine source

```
splitting param_indep_frag refines param_indep
events
  event ev
    system parameter q init 0 with q
    system parameter p init 0 with p
    system parameter r init 0 with r
end
```

Listing 3.12 – Spécification de la Fragmentation

L'application de la transformation présentée construit le code suivant :

```
machine param_indep_frag refines param_indep
 variables
   // event parameters and computation states
   ev_p
   ev_p_computed
   ev_q
   ev_q_computed
   ev_r
   ev_r_computed
 invariants
   // specification of parameter values (guards of
       inherited events)
   // explicit typing is useless (Event−B type synthesis)
   @ev_p_p ev_p_computed = TRUE ⇒ ( ev_p∈ ℕ)
   @ev_q_q ev_q_computed = TRUE ⇒ ( ev_q∈ ℕ)
   @ev_r_r ev_r_computed = TRUE ⇒ ( ev_r∈ ℕ)
   // dependencies between computation states
 variant
   {FALSE ↦ 1, TRUE ↦ 0}(ev_p_computed)+{FALSE
       ↦ 1, TRUE ↦ 0}(ev_q_computed)+{FALSE ↦ 1,
       TRUE ↦ 0}(ev_r_computed)
 events
   event INITIALISATION extends INITIALISATION
   then
     // initialization of computation states
     @ev_p_not_computed ev_p_computed := FALSE
     @ev_q_not_computed ev_q_computed := FALSE
     @ev_r_not_computed ev_r_computed := FALSE
     // initialization of parameter variables
     @ev_p_init ev_p := 0
     @ev_q_init ev_q := 0
     @ev_r_init ev_r := 0
   end
   // computation of ev parameters
   // computation of p
   convergent event compute_ev_p
   any p
   where
     @ev_p_notcomputed ev_p_computed = FALSE
     @p p∈ ℕ
   then
     @ev_p_done ev_p_computed := TRUE
     @ev_p_value ev_p := p
   end
   // computation of q
   convergent event compute_ev_q
   any q
   where
     @ev_q_notcomputed ev_q_computed = FALSE
     @q q∈ ℕ
   then
     @ev_q_done ev_q_computed := TRUE
     @ev_q_value ev_q := q
   end
   // computation of r
   convergent event compute_ev_r
   any r
   where
     @ev_r_notcomputed ev_r_computed = FALSE
     @r r∈ ℕ
   then
     @ev_r_done ev_r_computed := TRUE
     @ev_r_value ev_r := r
   end
   event ev refines ev
   when
     // event parameters have been computed
     @ev_p_computed ev_p_computed = TRUE
     @ev_q_computed ev_q_computed = TRUE
     @ev_r_computed ev_r_computed = TRUE
     theorem @ev_p ev_p∈ ℕ
     theorem @ev_q ev_q∈ ℕ
     theorem @ev_q ev_r∈ ℕ
   with
     @p p = ev_p
     @q q = ev_q
     @r r = ev_r
   then
     @x x:=ev_p+ev_q+ev_r
     // reset computation state of current event
     // or of dependent parameters of other events
     @ev_q_init ev_q_computed := FALSE
     @ev_p_init ev_p_computed := FALSE
     @ev_r_init ev_r_computed := FALSE
   end
end
```

Listing 3.13 – La machine raffinée

On constate que les trois événements convergents introduits `compute_ev_p`, `compute_ev_q` et `compute_ev_r` sont bel et bien indépendants et peuvent être franchissables simultanément et sont donc en compétition. L'événement de synchronisation `ev` ne comporte plus des paramètres locaux.

3.4.3.2 Paramètres multiples

Dans ce deuxième exemple, nous considérons une machine comportant un événement possédant plusieurs paramètres. L'événement abstrait se déclenche si les trois paramètres satisfont les gardes de l'événement *ev*. La Fragmentation décrite par le listing 3.15 exprime que les paramètres `p` et `q` doivent être déterminés indépendamment et en utilisant respectivement les gardes nommées `p` et `q`. Ensuite, le paramètre `r` doit être déterminé pour satisfaire les gardes `r1` et `r2`. La figure 3.10 donne l'ordre partiel linéaire de trois paramètres `p`, `q` et `r` sous forme d'un graphe planaire dont les sommets sont les noms de paramètres et les arcs traduisent une relation de précédence entre deux sommets.

La stratégie adoptée peut être justifiée par le fait que le calcul des trois paramètres doit être réalisé sur trois sites différents, les valeurs de `p` et `q` étant communiquées au site

déterminant `r`. Elle n'est cependant pas complète et peut conduire à un interblocage si les choix de `p` et `q` sont inadéquats (par exemple p=q), mais il s'agit d'un choix guidé par le concepteur. Cependant un raffinement préalable peut permettre de résoudre ce problème (voir la section 3.4.4).

```
machine m_ev
 variables  x
 invariants
   @x_typ x∈ℕ
events
   event INITIALISATION
     then
       @x_init x:∈ℕ
   end

   event ev
     any p q r
     where
       @p p > x
       @q q > x
       @r1 r > p + q
       @r2 r < 2 ∗ p
     then
       @x x := x + r
   end
end
```

Listing 3.14 – La machine source

```
splitting  frag_m_ev refines m_ev
events
  event ev
    system parameter p init  0 with p
    system parameter q init  0 with q
    when p q system parameter r init  0 with r1  r2
end
```

Listing 3.15 – Spécification de la Fragmentation

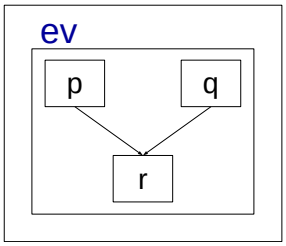

FIGURE 3.10 – Ordre partiel linéaire

3.4.3.3 Événements multiples dépendants

Nous considérons ici deux événements paramétrés modifiant chacun une variable globale. Afin d'illustrer un comportement spécifique du modèle transformé, la garde de chaque événement est supposée dépendre de la variable d'état modifiée par l'autre événement. Le modèle avant transformation est décrit par le listing 3.16. La transformation spécifiée par le listing 3.17 consiste à rendre global le paramètre `p` de chaque événement.

```
machine m_evts
 variables
   x y
 invariants
   @x_typ x ∈ ℕ
   @y_typ y ∈ ℕ
events
   event INITIALISATION
     then
       @x x:∈ℕ
       @y y:∈ℕ
   end
   event ev1 any p where @p p > y then @x x := x + p  end
   event ev2 any p where @p p > x then @y y := y + p  end
end
```

Listing 3.16 – La machine source

```
splitting  frag_m_evts refines m_evts
events
  event ev1
    system parameter p init  0 with p
  event ev2
    system parameter p init  0 with p
end
```

Listing 3.17 – Spécification de la Fragmentation

L'application de la transformation présentée construit le code suivant :

```
machine frag_m_evts
refines m_evts
variables
  // event parameters and computation states
  ev1_p ev1_p_computed
  ev2_p ev2_p_computed
invariants
  // specification of parameter values (guards of
     inherited events)
  @ev1_p_p ev1_p_computed = TRUE ⇒ ( ev1_p > y)
  @ev2_p_p ev2_p_computed = TRUE ⇒ ( ev2_p > x)
  // dependencies between computation states
variant
  {FALSE ↦ 1, TRUE ↦ 0}(ev1_p_computed)+
  {FALSE ↦ 1, TRUE ↦ 0}(ev2_p_computed)
events
  event INITIALISATION refines INITIALISATION
  then
    // initialization of computation states
    @ev1_p_not_computed ev1_p_computed := FALSE
    @ev2_p_not_computed ev2_p_computed := FALSE
    // initialization of parameter variables
    @ev1_p_init ev1_p := 0
    @ev2_p_init ev2_p := 0
  end
  // computation of p (event ev1)
  convergent event compute_ev1_p
  any p
  where
    @ev1_p_notcomputed ev1_p_computed = FALSE
    @p p > y
  then
    @ev1_p_done ev1_p_computed := TRUE
    @ev1_p_value ev1_p := p
  end
  event ev1 refines ev1
  when
    // event parameters have been computed
    @ev1_p_computed ev1_p_computed = TRUE
    theorem @ev1_p ev1_p > y
    // guards unused by event parameters
  with
    @p p = ev1_p
  then
    @x x := x + ev1_p
    // reset computation state of current event
    // or of dependent parameters of other events
    @ev1_p_init ev1_p_computed := FALSE
    @ev2_p_init ev2_p_computed := FALSE
  end
  // computation of p (event ev2)
  convergent event compute_ev2_p
  any p
  where
    @ev2_p_notcomputed ev2_p_computed = FALSE
    @p p > x
  then
    @ev2_p_done ev2_p_computed := TRUE
    @ev2_p_value ev2_p := p
  end
  event ev2 refines ev2
  when
    // event parameters have been computed
    @ev2_p_computed ev2_p_computed = TRUE
    theorem @ev2_p ev2_p > x
  with
    @p p = ev2_p
  then
    @y y := y + ev2_p
    // reset computation state of current event
    // or of dependent parameters of other events
    @ev1_p_init ev1_p_computed := FALSE
    @ev2_p_init ev2_p_computed := FALSE
  end
end
```

Listing 3.18 – La machine raffinée

Notons que la transformation proposée induit une perte d'atomicité : dans le modèle abstrait, la mise à jour des variables d'état (ici `x` ou `y`) suit immédiatement le choix de la valeur de `p`. Après transformation, les calculs des paramètres des événements peuvent être entrelacés, comme dans la trace suivante :

`compute_ev1_p`, `compute_ev2_p`, `ev1`

D'autre part, l'exécution d'un événement peut entraîner l'invalidation du calcul d'un paramètre d'un autre événement. Ainsi, l'exécution de `ev1` invalide le paramètre `ev2_p` puisque la variable `y` dont il dépend a été potentiellement modifiée. L'événement `ev2` ne peut donc pas suivre immédiatement `ev1` mais devra être précédé d'un nouveau calcul de `ev2_p` :

`compute_ev1_p`, `compute_ev2_p`, `ev1`, `compute_ev2_p`, `ev2`

La correction de la transformation n'est bien sûr pas remise en cause par cette observation, les événements intermédiaires de calcul étant nécessairement en nombre fini.

3.4.3.4 Événements multiples indépendants

Pour cet exemple, la machine `mevt_indep` (voir le listing 3.19) comporte deux événements paramétrés modifiant chacun une variable globale. La garde de chaque événement ne dépend que de la variable d'état modifiée par celui-ci. La transformation spécifiée par le listing 3.20 consiste à rendre global le paramètre `p` de chaque événement.

```
machine mevt_indep
 variables
   x y
 invariants
   @x_typ x ∈ ℕ
   @y_typ y ∈ ℕ
 events
   event INITIALISATION
   then
     @x x :∈ ℕ
     @y y :∈ ℕ
   end
   event ev1 any p where @p p > x then @x x := x + p end
   event ev2 any p where @p p > y then @y y := y + p end
end
```

Listing 3.19 – La machine source

```
splitting  frag_mevt_indep refines
      mevt_indep
events
  event ev1
    system parameter p init 0 with p
  event ev2
    system parameter p init 0 with p
end
```

Listing 3.20 – Spécification de la Fragmentation

L'application de la transformation présentée construit le code suivant :

```
machine frag_mevt_indep refines mevt_indep
 variables
   // event parameters and computation states
   ev1_p ev1_p_computed ev2_p ev2_p_computed
 invariants
   @ev1_p_p ev1_p_computed = TRUE ⇒ ( ev1_p > x)
   @ev2_p_p ev2_p_computed = TRUE ⇒ ( ev2_p > y)
 variant
   {FALSE ↦ 1, TRUE ↦ 0}(ev1_p_computed)+{FALSE
      ↦ 1, TRUE ↦ 0}(ev2_p_computed)
 events
   event INITIALISATION extends INITIALISATION
   then
     // initialization of computation states
     @ev1_p_not_computed ev1_p_computed := FALSE
     @ev2_p_not_computed ev2_p_computed := FALSE
     // initialization of parameter variables
     @ev1_p_init ev1_p := 0
     @ev2_p_init ev2_p := 0
   end
   // computation of ev1 parameters
   // computation of p
   convergent event compute_ev1_p
   any p
   where
     @ev1_p_notcomputed ev1_p_computed = FALSE
     @p p > x
   then
     @ev1_p_done ev1_p_computed := TRUE
     @ev1_p_value ev1_p := p
   end
   event ev1 refines ev1
   when
     // event parameters have been computed
     @ev1_p_computed ev1_p_computed = TRUE
     theorem @ev1_p ev1_p > x
   with
     @p p = ev1_p
   then
     @x x := x + ev1_p
     // reset computation state of current event
     // or of dependent parameters of other events
     @ev1_p_init ev1_p_computed := FALSE
   end
   // computation of ev2 parameters
   // computation of p
   convergent event compute_ev2_p
   any p
   where
     @ev2_p_notcomputed ev2_p_computed = FALSE
     @p p > y
   then
     @ev2_p_done ev2_p_computed := TRUE
     @ev2_p_value ev2_p := p
   end
   event ev2 refines ev2
   when
     // event parameters have been computed
     @ev2_p_computed ev2_p_computed = TRUE
     theorem @ev2_p ev2_p > y
   with
     @p p = ev2_p
   then
     @y y := y + ev2_p
     // reset computation state of current event
     // or of dependent parameters of other events
     @ev2_p_init ev2_p_computed := FALSE
   end
end
```

Listing 3.21 – La machine raffinée

3.4.3.5 Absence d'ordre linéaire

La machine `Pythagore` (voir le listing 3.22) modélise en Event-B l'équation de Pythagore $a^2+b^2 = c^2$. L'événement `triplet` comporte trois paramètres `x`, `y` et `z` et une contrainte (`xyz`) mettant en relation ces trois paramètres. Il n'est pas possible de séquentialiser le choix de valeurs pour `x`, `y` et `z` en ne prenant en compte que les gardes présentes. Sans transformation préalable (voir la section 3.4.4), un tel événement ne sera pas fragmenté et devra rester local

afin d'éviter un interblocage consécutif à un choix de valeurs ne pouvant être complété pour former un triplet solution (par exemple x=1, y=1).

```
machine Pythagore
variables
  a b c
invariants
  @a_typ a ∈ ℕ1
  @b_typ b ∈ ℕ1
  @c_typ c ∈ ℕ1
  @abc (a∗a)+(b∗b)=(c∗c)
events
  event INITIALISATION
  then
    @a a:=3
    @b b:=4
    @c c:=5
  end
  event triplet any x y z
  where
    @x x ∈ ℕ1
    @y y ∈ ℕ1
    @z z ∈ ℕ1
    @xyz (x∗x)+(y∗y)=(z∗z)
  then
    @x a:=x
    @y b:=y
    @z c:=z
  end
end
```

Listing 3.22 – Événement `triplet` à ne pas fragmenter

3.4.4 Préparation de la Fragmentation

Nous avons vu précédemment que la Fragmentation peut introduire des interblocages. L'utilisateur dispose cependant de moyens pour tenter d'y remédier. Il peut introduire un raffinement dans le but de résoudre le système de contraintes formé par les gardes. Ce raffinement peut introduire des théorèmes dérivés des gardes existantes ou proposer une famille de solutions. Ces deux méthodes sont illustrées ci-dessous.

3.4.4.1 Traitement de la machine `m_ev`

Pour résoudre le problème d'interblocage introduit dans la section 3.4.4.2, le concepteur a la possibilité d'aider la résolution des contraintes en ajoutant une garde à la machine qui est une conséquence des deux gardes `r1` et `r2` (voir le listing 3.23).

```
machine m_ev_r refines m_ev
 variables x
events
  event INITIALISATION extends INITIALISATION
  end

  event ev extends ev
    when
      theorem @pq q+1 < p // consequence de r1 et r2
  end
end
```

Listing 3.23 – La machine source avec contrainte dérivée

```
 splitting m_ev_r_frag refines m_ev_r
events
  event ev
    system parameter q init 0 with q
    when q system parameter p init 0 with p pq
    when p q system parameter r init 0 with r1 r2
end
```

Listing 3.24 – Fragmentation évitant un interblocage

Si les deux variables `p` et `q` satisfont cette nouvelle contrainte, il est toujours possible de trouver `r` satisfaisant les contraintes `r1` et `r2`. Quant à `p` et `q`, pour `q` donné, il est toujours

possible de trouver `p` satisfaisant `p` et `pq`. On arrive ainsi au nouveau schéma de Fragmentation du listing 3.24. Le listing 3.25 donne le résultat de la transformation. Des événements de calcul des trois paramètres sont introduits, ainsi que les variables booléennes contrôlant l'ordonnancement des calculs : la séquence `q`, `p`, `r` est imposée. On notera notamment que la garde `pq` est exploitée pour spécifier la valeur de `p`. Il ne s'agit plus d'un théorème puisque cette propriété ne se déduit pas des autres gardes de l'événement. L'événement `ev` n'a maintenant plus de paramètres. Il est actif lorsque les trois valeurs ont été calculées dans l'état global. Après l'exécution de l'événement, les trois paramètres doivent être recalculés. Les variables de contrôle sont donc remises à `FALSE`.

```
machine me_v_r_frag
refines m_ev_r
variables
  // event parameters and computation states
  ev_q ev_q_computed
  ev_p ev_p_computed
  ev_r ev_r_computed
invariants
  // typing invariants of control variables
  @TY_ev_q_computed ev_q_computed ∈ 𝔹
  @TY_ev_p_computed ev_p_computed ∈ 𝔹
  @TY_ev_r_computed ev_r_computed ∈ 𝔹
  // typing invariants of globalised parameters
  @TY_ev_p_pq  ev_q+1 < ev_p
  @TY_ev_r_r1  ev_r > ev_p + ev_q
  @TY_ev_r_r2  ev_r < 2 ∗ ev_p
  // specification of parameter values (guards of
     inherited events)
  // explicit typing is useless (Event−B type synthesis
     )
  @ev_q_q ev_q_computed = TRUE ⇒ ( ev_q > x)
  @ev_p_p ev_p_computed = TRUE ⇒ ( ev_p > x)
  @ev_p_pq ev_p_computed = TRUE ⇒ ( ev_q+1 <
     ev_p)
  @ev_r_r1 ev_r_computed = TRUE ⇒ ( ev_r > ev_p +
     ev_q)
  @ev_r_r2 ev_r_computed = TRUE ⇒ ( ev_r < 2 ∗
     ev_p)
  // dependencies between computation states
  @ev_q_p ev_q_computed = FALSE ⇒ ev_p_computed
     = FALSE
  @ev_p_r ev_p_computed = FALSE ⇒ ev_r_computed
     = FALSE
variant
  {FALSE ↦ 1, TRUE ↦ 0}(ev_p_computed)+{FALSE
     ↦ 1, TRUE ↦ 0}(ev_q_computed)+{FALSE ↦ 1,
     TRUE ↦ 0}(ev_r_computed)
events
  event INITIALISATION extends INITIALISATION
  then
    // initialization of computation states
    @ev_q_not_computed ev_q_computed := FALSE
    @ev_p_not_computed ev_p_computed := FALSE
    @ev_r_not_computed ev_r_computed := FALSE
    // initialization of parameter variables
    @ev_q_init ev_q := 0
    @ev_p_init ev_p := 0
    @ev_r_init ev_r := 0
  end
  convergent event compute_ev_q
  any q
  where
    @q  q > x
  then
    @ev_q_done ev_q_computed := TRUE
    @ev_q_value ev_q := q
    // reset dependent parameters
    @p_reset ev_p_computed := FALSE
    @r_reset ev_r_computed := FALSE
  end
  convergent event compute_ev_p
  any p
  where
    @ev_q_computed ev_q_computed = TRUE
    @p  p > x
    @pq  ev_q+1 < p // théorème transformé en garde
  then
    @ev_p_done ev_p_computed := TRUE
    @ev_p_value ev_p := p
    // reset dependent parameters
    @r_reset ev_r_computed := FALSE
  end
  convergent event compute_ev_r
  any r
  where
    @ev_p_computed ev_p_computed = TRUE
    @r1  r > ev_p + ev_q
    @r2  r < 2 ∗ ev_p
  then
    @ev_r_done ev_r_computed := TRUE
    @ev_r_value ev_r := r
  end
  event ev refines ev
  when
    // event parameters have been computed
    @ev_q_computed ev_q_computed = TRUE
    theorem @q  ev_q > x
    @ev_p_computed ev_p_computed = TRUE
    theorem @p  ev_p > x
    theorem @pq  ev_q+1 < ev_p
    @ev_r_computed ev_r_computed = TRUE
    theorem @r1  ev_r > ev_p + ev_q
    theorem @r2  ev_r < 2 ∗ ev_p
  with
    @q q = ev_q
    @p p = ev_p
    @r r = ev_r
  then
    @x  x := x + ev_r
    // reset computation state of current event
    // or of dependent parameters of other events
    @ev_q_init ev_q_computed := FALSE
    @ev_p_init ev_p_computed := FALSE
    @ev_r_init ev_r_computed := FALSE
  end
end
```

Listing 3.25 – La machine raffinée

Le comportement dynamique de ce raffinement de Fragmentation est modélisé par la

figure 3.11.

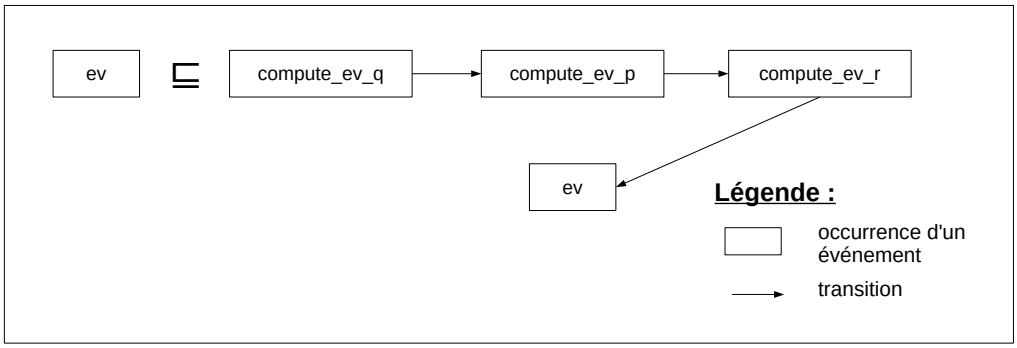

FIGURE 3.11 – Événement abstrait `ev` raffiné par Fragmentation

3.4.4.2 Application aux triplets de Pythagore

Dans le cas des triplets de Pythagore, l'ensemble des triplets (*irréductibles et rangés*) solutions peut être décrit comme la famille paramétrée décrite par le code ci-dessous. Le résultat mathématique affirme que toutes les solutions peuvent être obtenues ainsi mais nous ne cherchons qu'à obtenir un raffinement, c'est à dire que nous vérifions grâce à la construction `with` que les triplets indexés par `u` et `v` sont des solutions.

```
machine Pythagore_Solution refines Pythagore
variables a b c
events
  event INITIALISATION extends INITIALISATION
  end
  event triplet refines triplet
    any u v
    where
      @v_typ v>0
      @u_typ u>v
    with
      @x x=u∗u−v∗v
      @y y=2∗u∗v
      @z z=u∗u+v∗v
    then
      @x a:=u∗u−v∗v
      @y b:=2∗u∗v
      @z c:=u∗u+v∗v
  end
end
```

Il est maintenant possible d'en dériver un ordonnancement des calculs de `u` et `v` : la Fragmentation sélectionnera `v > 0` puis `u > v`.

3.5 L'étape de Distribution

Après l'étape de Fragmentation, 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. On considère qu'un connecteur ne réalise que des transferts de données sans traitement (voir les sections 3.2.4, 3.2.5 et 3.2.6). Les traitements internes aux connecteurs, possibles en BIP, sont donc exclus. L'ordonnancement des traitements effectués par le modèle BIP doit être pris en compte pour que la traduction vers la cible puisse être directe. En BIP, les gardes des transitions des composants atomiques – et éventuellement les gardes

des connecteurs – sont testées avant les transferts de données vers les composants connectés. Le modèle Event-B produit devra donc respecter cette contrainte.

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, décrite par un ensemble de sous-composants ainsi que la localisation des variables et éventuellement des gardes sur ses composants. La figure 3.12 montre les différentes phases de l'étape de Distribution.

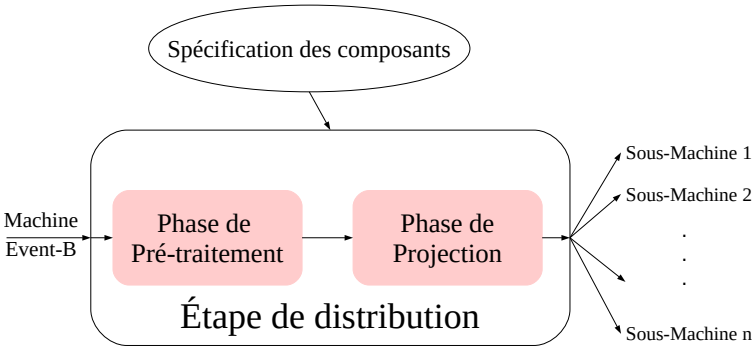

FIGURE 3.12 – L'étape de Distribution

Contrairement à l'outil SEDT (voir les sections 2.4.4.4 et 2.4.4.5), la transformation présentée ici comporte une phase de pré-traitement destinée à lever les restrictions de cet outil en permettant la projection de gardes et d'actions faisant référence à des variables destinées à être localisées sur des sites différents.

3.5.1 Spécification de la Distribution

Le listing 3.26 décrit la structure du DSL de spécification de la Distribution. Il reprend les informations présentes dans le plug-in SEDT [74] : introduction de sous-composants, Distribution des variables sur les sous-composants. L'outil SEDT supposant que les gardes ne font référence qu'à des variables d'un même sous-composant, ces informations étaient suffisantes pour déterminer le site de calcul d'une garde. Ici, nous levons cette restriction mais le concepteur a la possibilité d'indiquer le site sur lequel la garde sera calculée (mot-clé `guard` dans le listing 3.26). En l'absence de toute indication, une garde est calculée sur le site d'une variable globale à laquelle elle fait référence. Ce choix est arbitraire, éventuellement contre-intuitif comme le montrera l'étude de cas (voir le chapitre 4) et peut donc être modifié via le DSL. Si aucune variable globale n'est référencée, la garde est valide dans toute projection et sera donc conservée dans chaque composant. Lors du calcul, les valeurs des variables référencées par une garde doivent être accessibles localement. Si les variables sont distantes, des copies *faiblement cohérentes* de ces 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 méthodologie et étend strictement le plug-in SEDT de *décomposition par événement partagé* (voir la section 2.4.4.4).

```
shared event decomposition input_machine_decomposition refines input_machine
components C1 C2 ... Cn
mappings
```

```
  variables  V1 V2 V3 ↦ C1;
  variable  V4 ↦ C2;
  ...
  guard evt1·g1 ↦ C1;
  guard evt1·g2 ↦ C2;
  ...
end
```

Listing 3.26 – Spécification de Distribution (DSL)

3.5.2 Correction des projections : problématique de Distribution d'invariants et de gardes

La Distribution étant spécifiée, la projection d'une machine sur un composant est relativement directe : on conserve d'une part les variables associées à la machine cible et d'autre part, pour chaque événement, les gardes et les actions ne contenant pas de variables associées à un autre composant. Il manque cependant des informations pour obtenir une machine correcte : les variables et les paramètres doivent au moins être typés. Le plug-in SEDT propose un principe simple : il prend comme invariant le type faible des variables de la machine, c'est-à-dire celui construit à partir des ensembles de base et des opérations produit cartésien et ensemble de parties. Les propriétés ainsi obtenues sont préservées par les événements mais elles ne permettent pas d'assurer la bonne formation des termes. Considérons pour cela l'exemple suivant :

```
machine source
 variables  f
 invariants
   @f f∈ℤ→𝔹
events
   event ev
      any x where
         @x_typ x∈ℤ
         @f f(x)=TRUE
   end
end
```

La machine projetée sur un composant contenant la variable `f` aura pour seul invariant le type faible de `f` : $\mathcal{P}(\mathbb{Z} \times \mathbb{B})$. L'information selon laquelle `f` est une fonction, garantissant la bonne formation de `f(x)` est donc perdue.

```
machine Mxy
 variables  x y
 invariants
   @x_typ x>0
   @y_typ y>0
events
   event e
      any p where
         @p p∈ℤ
         @py p>y
      then
         @x x:=p
   end
end
```

Listing 3.27 – Machine source

```
machine C
 variables  x
 invariants
   @x x>0
events
   event e
      any p where
         @p p∈ℤ
      then
         @x x:=p
   end
end
```

Listing 3.28 – Projection sur x

Nous avons pris l'approche opposée consistant à conserver dans la projection toutes les propriétés s'exprimant avec les variables du composant. Ainsi, le type fort de `f` serait conservé. Cette solution ne garantit pas non plus l'obtention de projections correctes. Considérons l'exemple du listing 3.27.

La projection de cette machine sur le composant `C` ne contenant que la variable `x` donne la machine du listing 3.28 qui est incorrecte puisque l'invariant sur `x` n'est pas préservé par l'événement `e`. La propriété non projetable `p > y` permettait de déduire la préservation de l'invariant.

Dans [14], Banach propose une solution pour le problème de décomposition d'invariants en Event-B. Une telle solution est inspirée du travail décrit dans [30]. Elle repose sur l'introduction d'une notion de composant comportant une interface déclarant les variables exportées et les propriétés invariantes qui les relient, ainsi que les variables importées. Les notions de raffinement et de composition/décomposition sont définies dans ce nouveau formalisme. La solution proposée et difficilement transposable à notre contexte puisqu'elle repose sur la déclaration de l'interface et de divers liens entre machines et interfaces.

Nous proposons ici de traiter ce problème en complétant dans la machine source l'invariant et les gardes par des théorèmes pouvant être conservés par projection. Dans l'exemple donné par le listing 3.27, il suffit d'ajouter le théorème `p > 1` aux gardes de l'événement `e`. Ce théorème est une conséquence de l'invariant `@y_typ` (`y>0`) et de la garde `@py` (`p>y`) où la variable `y` a été éliminée via une quantification existentielle. Cette propriété sera préservée par projection et permettra de valider la préservation de l'invariant `x > 0`.

3.5.3 Conséquence méthodologique

L'absence de garantie sur la correction des projections induit un processus de développement cyclique passant par les étapes suivantes :

1. expression des mappings

2. appel du plug-in de Distribution

3. démonstration des obligations de preuve liées à la correction des projections

4. en cas d'échec,

(a) étude des propriétés invalides

(b) identification des hypothèses manquantes, non préservées par la projection

(c) ajout d'invariants ou de gardes (sous forme de théorème) au modèle centralisé

(d) ajustement des mappings et retour en **2.**

Notons que l'identification des hypothèses manquantes n'est pas une chose aisée. La méthode que nous avons utilisée consiste à éliminer via une quantification existentielle les variables distantes présentes dans une conjonction d'invariants ou de gardes. Ceci nous a ici permis de déduire une propriété conservée par projection.

3.5.4 La phase de pré-traitement

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

Dans le cadre d'une vue orientée composant, la figure 3.13 montre comment un modèle Event-B est transformé durant la phase de pré-traitement. En effet, l'événement `ev` du composant C_j comporte une garde qui lit la copie locale de vi et une action qui accède à la

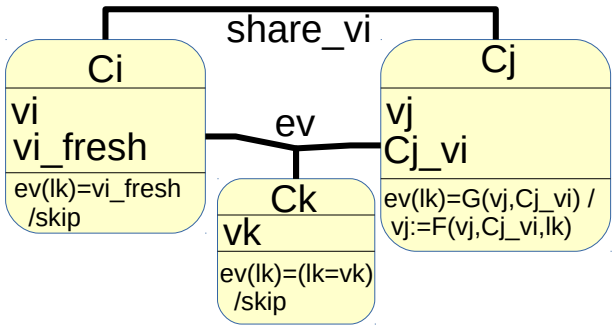

FIGURE 3.13 – Copies locales et accès distant

variable distante vk. La synchronisation des événements garantit que la copie locale de vi est à jour et donne accès à vk tout en contraignant le paramètre de l'événement lk.

La phase de pré-traitement produit une machine Event-B qui raffine la machine en entrée. Le listing 3.29 présente le schéma général d'une telle transformation pour un composant C_i. Notons que la fraicheur des variables copies est garantie par leur mise à jour à partir des variables sources dans les actions.

```
machine generated refines input_machine
 variables
   vi // variables héritées, sur Ci
   Cj_vi // copie sur Cj de vi ( utilis ée par une garde sur Cj)
   vi_fresh // ⊤ 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
     @t [vi :=local_vi](INV) // sur Cj
   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
     @t [vk :=local_vk](INV) // sur Cj
   then
     @a vj := [vi := Cj_vi;vk := local_vk]e // sur Cj, synchro avec Ck
   end
end
```

Listing 3.29 – Pré-traitement

Notons que les copies de données réalisées par ce pré-traitement peuvent se révéler incorrectes d'un point de vue métier. Par exemple, une garde effectuant la comparaison `x=y` entre deux variables localisées sur des composants différents va, pour être projetable, induire la recopie de l'une ou l'autre des variables sur le composant réalisant le test. Ce transfert de donnée peut être invalide si la donnée est supposée interne (masquage d'information) au composant. De telles considérations sont actuellement prises en compte par relecture du code projeté mais pourrait donner lieu à l'ajout d'annotations.

3.5.5 La phase de projection

Cette phase construit une machine pour chaque sous-composant. Les sites de projection des gardes et actions sont indiqués dans le listing 3.30. Notons que les invariants faisant référence à des variables distantes sont écartés.

```
machine Cj
variables vj Cj_vi
invariants  // Garder seulement ceux qui font référence à vj et Cj_vi
events
  event share_vi // synchronisation avec l'événement de Ci, importer vi
  any local_vi
  where
    @t [vi :=local_vi](INV) // sur Cj
  then
    @to_Cj Cj_vi := local_vi
  end

  event share_vj // synchronisation avec l'événement de Ci, exporter vj
  any local_vj then
    @to_Cl local_vj := vj
  end

  event ev
  any local_vk // accès synchronisé à vk
  where
    @vj_fresh vj_fresh = TRUE // la copie de vj doit etre à jour
    @g [vi := Cj_vi]g // accès à une copie de vi sur Cj
  @t [vk :=local_vk](INV) // sur Cj
  then
    @a vj := [vi := Cj_vi; vk :=local_vk]e // accès à une copie de vi, synchronisation sur vk
  end
end
```

Listing 3.30 – Phase de projection

Notons que l'utilisateur peut être amené à ajouter des invariants pour des besoins de vérification de la correction des composants issus de la phase de projection. D'autre part, les invariants écartés par la phase de projection doivent être ajoutés lors de la recomposition afin d'assurer que le produit des sous-composants, tel que défini dans [75], reconstruise la machine avant décomposition.

3.5.6 Exemples

Les sous-sections suivantes décrivent l'effet de la transformation selon la localisation des accès distants. La machine transformée est annotée afin d'indiquer les composants sur lesquels sont placés les variables et les invariants de la machine ainsi que les gardes et les actions des événements synchronisés. Le code des projections n'est pas donné mais s'en déduit directement.

3.5.6.1 Accès distant dans une action

L'exemple ci-dessous illustre l'accès d'une action à une variable distante. La machine initiale introduit un événement de mise à jour de la variable `x` et un événement ajoutant `x` à `s`. Ces événements n'étant pas synchronisés, certaines valeurs de `x` peuvent bien sûr ne pas être prises en compte.

```
machine dist_act
 variables
   x s
 invariants
   @x_typ x ∈ ℕ
   @s_typ s ∈ ℕ
events
   event incr
   then
     @x x := x + 1
   end
   event somme
   then
     @s s := s + x
   end
end
```

Listing 3.31 – Machine source

```
shared event decomposition dist_act_rep refines dist_act
components
   Cx Cs
mappings
   variable x ↦ Cx;
   variable s ↦ Cs;
end
```

Listing 3.32 – Spécification de la Distribution

La spécification de Distribution (voir le listing 3.32) introduit deux sous-composants (`Cx` et `Cs`) sur lesquels seront respectivement localisées les variables `x` et `s`. En conséquence, l'addition réalisée par l'événement `somme` sur le composant associé à la variable affectée devra accéder à la variable distante `x`.

Le modèle obtenu (voir le listing 3.33) est prêt à être projeté sur les deux sous-composants. La localisation de chaque élément est indiquée en commentaire. En particulier, l'événement `somme` sera présent sur les deux composants et exécutés de manière synchrone. La garde, exécutée sur `Cx`, fixe le paramètre `local_x` de l'événement et assure la communication de la valeur de `x` à `Cs`. L'action réalisée par `Cs` a ainsi accès à la valeur de la variable distante `x`.

```
machine dist_act_rep refines dist_act
 variables
   x // on Cx
   s // on Cs
 invariants
   @x_typ x ∈ ℕ // on Cx
   @s_typ s ∈ ℕ // on Cs
events
   event incr refines incr
   then
     @x x := x + 1 // on Cx
   end
   event somme refines somme
   any local_x
   where
     @x_access local_x = x // on Cx
   then
     @s s := s + local_x // on Cs
   end
end
```

Listing 3.33 – Machine transfomée

Ce protocole est réalisable en BIP via un connecteur assurant le transfert de `x` avant l'exécution de l'action.

3.5.6.2 Accès distant dans une garde

Le mécanisme précédent n'est pas directement utilisable lorsqu'une garde accède à une variable distante : en BIP, les gardes des transitions des composants sont testées avant le transfert de données sur les autres composants. Il serait nécessaire de déporter la garde dans le connecteur et donc d'utiliser le connecteur pour réaliser des traitements, ce que nous nous sommes interdits. L'exemple ci-dessous illustre le pré-traitement réalisé pour maintenir à

jour une copie locale des variables distantes utilisées dans des gardes. L'exemple précédent est adapté : la variable `x` peut être ajoutée à `s` lorsqu'elle lui est supérieure.

```
machine dist_grd
 variables
   x s
 invariants
   @x_typ x ∈ ℕ
   @s_typ s ∈ ℕ
events
   event incr
   then
     @x x := x+1
   end
   event somme
   when
     @sup x > s
   then
     @s s := s + x
   end
end
```

Listing 3.34 – Machine source

```
shared event decomposition dist_grd_rep refines dist_grd
components
   gCx gCs
mappings
    variable x ↦ gCx;
    variable s ↦ gCs;
   guard somme·sup ↦ gCs;
end
```

Listing 3.35 – Spécification de la Distribution

La spécification de Distribution (voir le listing 3.35) introduit toujours les deux sous-composants `gCx` et `gCs`. Nous ajoutons le fait que la garde sera calculée sur le composant `gCs`. L'accès à `x` sera donc distant.

Le code généré introduit une copie (`gCs_x`) de `x` sur `gCs` mise à jour par l'événement convergent `share_x` exécuté de manière synchrone par le site possédant la variable `x` originale et les sites possédant des copies (ici `gCs`). La variable booléenne `x_fresh`, localisée sur `gCx` indique si les copies sont à jour. Cette propriété est spécifiée par l'invariant `gCs_x_copy` dont l'annotation indique qu'il sera perdu lors de la projection. Il fait en effet référence à des variables localisées sur des sites différents.

```
machine dist_grd_rep refines dist_grd
 variables
   x // on gCx
   s // on gCs
   gCs_x // copie de x sur gCs
   x_fresh // on gCx
 invariants
   @x_typ x ∈ ℕ // on gCx
   @s_typ s ∈ ℕ // on gCs
   @gCs_x_copy x_fresh = TRUE ⇒ gCs_x = x // will be lost
 variant
   {FALSE ↦ 1, TRUE ↦ 0}(x_fresh)
 events
   convergent event share_x // réalise la copie x sur gCs
   any local_x
   where // on gCx
     @g x_fresh = FALSE
     @l local_x = x
   then
     @to_gCs gCs_x := local_x // on gCs
     @done x_fresh := TRUE // on gCx
   end
   event incr refines incr
   then
     @x x := x+1 // on gCx
     @x_reset x_fresh := FALSE // on gCx
   end
   event somme refines somme
   when
     @x_fresh x_fresh = TRUE // on gCx
```

```
    @sup  s < gCs_x // on gCs
  then
    @s s :=  s + gCs_x // on gCs
  end
end
```

Listing 3.36 – Machine transfomée

3.5.6.3 Accès distant dans une garde et dans une action

Cet exemple combine les deux schémas précédents. Le listing 3.37 introduit un événement `ev` dont la garde exécutée sur `D` (d'après la spécification 3.38) accède à la variable distante `v` (sur `C`) et l'action exécutée aussi sur `D` accède aux variables distantes `u` et `v`.

```
machine dist_grd_act
 variables
   v //sur C
   w //sur D
   u //sur E
 invariants
   @v_ty v ∈ ℕ
   @w_ty w ∈ ℕ
   @u_ty u ∈ ℕ
events
   event ev
   when
     @g v>0 // sur D
   then
     @a w := w+v+u //sur D
   end
end
```

Listing 3.37 – Machine `dist_grd_act`

```
shared event decomposition dist_grd_act_rep refines
      dist_grd_act
components
  C D E
mappings
   variable  v ↦ C;
   variable  w ↦ D;
   variable  u ↦ E;
  guard ev·g ↦ D;
end
```

Listing 3.38 – Spécification de la Distribution

```
machine dist_grd_act_rep refines dist_grd_act
 variables
   // inherited  variables
   v // on C
   w // on D
   u // on E
   // local  copies  and status  of  remote variables
   D_v // on D
   v_fresh  // on C
 invariants
   @v_ty  v ∈ ℕ // on C
   @w_ty  w ∈ ℕ // on D
   @u_ty  u ∈ ℕ // on E
   // local  copies  properties
   @D_v_copy v_fresh = TRUE ⇒ D_v = v // will be lost
   // add typing invariants  of  v
   @TY_D_v  D_v ∈ ℕ // on D component
 variant  {FALSE ↦ 1, TRUE ↦ 0}(v_fresh)
events
   convergent event share_v
   any local_v
   where // on C
     @g v_fresh = FALSE
     @l  local_v  = v
     // add typing invariants  of  v
     @TY_local_v  local_v ∈ ℕ
   then
     @to_D D_v := local_v  // on D
     @done v_fresh := TRUE  // on C
```

```
  end
  event ev refines ev
  any local_u where
    // access to remote variables used in actions
    @u_access local_u = u // on E
    // add typing invariants of u
    @TY_local_u local_u ∈ ℕ
    // access to copies of remote variables
    @v_fresh v_fresh = TRUE // on C
    // inherited guards
    @g D_v>0 // on D
  then
    @a w := w+D_v+local_u // on D
  end
end
```

Listing 3.39 – Machine raffinée

D'après l'annotation de placement du listing 3.37, le test de la valeur de la variable `v` doit être réalisé par le composant D et non pas par le composant C sur lequel se trouve la variable `v`. Il est donc nécessaire de disposer d'une copie de `v` sur le composant `D` avant de réaliser le test et de gérer la cohérence de cette copie. Ceci est réalisé par l'introduction de la variable `D_v` sur `D` et d'un booléen `v_fresh` sur `C` indiquant si la copie est à jour. L'événement `share_v`, partagé par `C` et `D`, réalise la mise à jour. Il effectue des tests locaux dans ses gardes. L'événement `ev`, partagé par `C`, `D` et `E` accède maintenant à la copie de `v` tout en vérifiant qu'elle est à jour. L'accès à `u` se fait via le paramètre `local_u`. Les gardes exécutées sur le composant spécifié accèdent à des données locales. La figure 3.14 illustre le connecteur ternaire BIP `ev` et le connecteur binaire `share_v`.

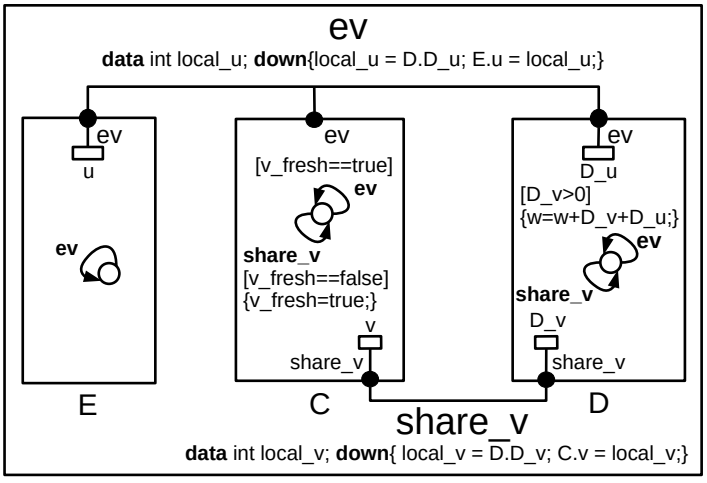

FIGURE 3.14 – Connecteur ternaire `ev` et binaire `share_v`

3.6 L'étape de génération de code

L'ultime étape de notre processus de développement est la génération de code distribué. Celle-ci consiste à projeter les composants Event-B, issus de l'étape de Distribution, vers un modèle de composants BIP. Durant cette phase de transformation, nous supposons que les modèles Event-B doivent être conformes à un sous-ensemble d'Event-B appelé Event-B0 à l'instar de B0 de B [7]. Un modèle décrit en Event-B0 doit pouvoir se traduire directement en un modèle à base de composants BIP. Le passage d'Event-B vers Event-B0 se fait par raffinements successifs en utilisant judicieusement le raffinement de données. La génération

de code prend en entrée le projet Event-B dans sa globalité. Chaque étape de décomposition se traduit en un connecteur. Les machines feuilles se traduisent en composants élémentaires BIP.

Nous allons cependant nous limiter à la dernière étape de décomposition et considérer que les projections sont exprimées dans le sous-langage Event-B0 directement traduisible en BIP. Le connecteur ne sera donc pas hiérarchique et les étapes de raffinement de données pourront être ignorées. Le générateur de code prend ainsi en entrée le modèle Event-B annoté par les informations de projection. Nous présentons dans ce qui suit l'architecture du code BIP généré. Nous avons établi des règles liées à la génération des types de port et de connecteurs, du squelette d'un sous-composant et du composant composite. Elles seront illustrées sur la machine `dist_grd_act_rep` (voir le listing 3.39) annotée par les projections sur les sous-composants `C`, `D` et `E`.

3.6.1 Types de port

En BIP, un type de port contient le type des variables d'un composant accessibles via un même port. En Event-B, l'accès en lecture aux variables d'un composant est représenté par un événement partagé dont les paramètres sont liés aux variables d'un composant par une contrainte d'égalité. L'application des transformations précédentes fixe la forme de tels événements : après la phase de Fragmentation, les événements partagés ne sont plus paramétrés. Après la phase de projection, des paramètres nommés `local_vi` sont introduits, contraints via une égalité et utilisés dans la partie action. Les égalités représentent les accès en lecture via un port.

```
event ev
any local_p1 ... local_pn
where
  @p1 local_p1 = v1 // sur C1
  ...
  @pn local_pn = vn // sur Cn
then
  @v v := ... // sur C
end
```

$\Longrightarrow$

```
port type ty_ev_C1(v1_type v1, ...)
 ...
port type ty_ev_Cn(vn_type vn, ...)
```

Les transformations fixent aussi le format des événements modélisant les accès en écriture via des ports : ce sont les événements de partage générés par la phase de projection. Ils permettent l'accès à des variables distantes dans les gardes. Chaque événement ne gérant qu'une variable, le type de port déclare le type de la variable partagée.

```
event share_v
any local_v
where
  @r local_v = v // lecture de C·v
  ...
then
  @w1 C1_v := local_v // écriture de C1·v
 ...
  @wn Cn_v := local_v // écriture de Cn·v
end
```

$\Longrightarrow$

```
port type ty_share_v_C(v_type v)
```

Application à l'exemple

Le schéma précédent appliqué à la machine `dist_grd_act_rep` donne les ports du listing 3.40. Notons que nous ajoutons un port spécial de synchronisation pure `ty_empty_port`

utilisé par les connecteurs n'effectuant aucun transfert de données sur un composant mais permettant au composant de tester une garde locale ou d'effectuer une action locale de manière synchronisée.

```
port type ty_empty_port() // porte sans donnees
port type ty_share_v_C(int v) // acces a C.v
port type ty_share_v_D(int v)   // acces a D.v
port type ty_ev_D(int u) // acces a D.u
port type ty_ev_E(int u) // acces a E.u
```

Listing 3.40 – Les types de portes BIP générés

3.6.2 Types de connecteur

Pour chaque événement faisant référence à des variables de plusieurs composants, nous générons un type de connecteur. Nous supposons que tous les connecteurs sont synchrones. Ils définissent une action `down` permettant le transfert de données (via les ports) vers les composants participant à l'interaction. Celle-ci admet comme paramètres des ports spécifiés par les types de port précédemment introduits. Comme précédemment, ces événements sont de deux types (voir la figure 3.14). Le premier (schéma ci-dessous) exprime l'accès à des variables distantes pour éventuellement mettre à jour des données locales. Le connecteur doit accéder aux ports des composants d'où proviennent les données distantes et met à jour des copies locales à `C` pour la mise à jour d'une variable locale.

```
event ev
any local_p1 ... local_pn
where
  @p1 local_p1 = v1 // sur C1
  ...
  @pn local_pn = vn // sur Cn
then
  @w w := f(local_p1, ..., local_pn) // sur C
end
```

$\Longrightarrow$

```
connector type ty_ev (ty_ev_C1 C1, ..., ty_ev_Cn
     Cn, ty_ev_C C)
  data v1_type local_p1  ...
  define C1  ...  Cn C
  on C1  ...  Cn
    down {
      local_p1 = C1.v1; C.v1 = local_p1;
     ...
    local_pn = Cn.vn; C.vn = local_pn;
    }
end
```

Le deuxième type d'événements est de la forme `share_v`. Ces événements sont aussi associés à des types de connecteurs spécifiques (voir le schéma ci-dessous). Ils expriment la copie d'une variable sur les sous-composants pouvant y accéder. On trouvera donc un port d'accès en lecture et n ports d'accès en écriture sur les réplications de la variable.

```
event share_v
any local_v
where
  @r local_v = v // lecture de C·v
  ...
then
  @w1 C1_v := local_v // écriture de C1·v
 ...
  @wn Cn_v := local_v // écriture de Cn·v
end
```

$\Longrightarrow$

```
connector type ty_share_v (ty_share_v_C C,
     ty_share_v_C C1, ..., ty_share_v_C Cn)
  data v_type local_v
  define C C1  ...  Cn
  on C C1  ...  Cn
    down {
    local_v = C.v;
    C1.v = local_v;
     ...
    Cn.v = local_v;
  }
end
```

Application à l'exemple

Nous obtenons deux types de connecteurs. Le premier est associé à l'événement `ev` et permet de transférer la valeur de `u` de `E` vers `D`. Aucun échange n'étant réalisé avec `C`, le port

`ty_empty_port` est utilisé pour la synchronisation avec ce composant. Elle permettra d'une part l'accès distant en lecture à la valeur de `u` depuis `D` et d'autre part le test de fraîcheur de la copie de `v`. Le deuxième type de connecteur `ty_share_v` réalise la copie de `v`.

```
connector type ty_ev (ty_empty_port C, ty_ev_D D,ty_ev_E E)
  data int local_u
  define C D E
  on C D E down { local_u = E.u; D.u = local_u; }
end
connector type ty_share_v (ty_share_v_C C,ty_share_v_D D)
  data int local_v
  define C D
  on C D down { local_v = D.v; C.v = local_v; }
end
```

Listing 3.41 – Les types de connecteurs BIP générés

3.6.3 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 référencées par ce sous-composant, les instances des types de ports référencés par les événements projetés sur ce composant et les transitions résultant des projections. Nous avons opté pour des composants BIP à une seule région de contrôle (`Place` en BIP) puisque cette notion n'est pas explicite en Event-B. Pour un composant `C`, nous obtenons donc le schéma d'atome BIP ci-dessous :

```
atom type C()
  data v_type v ... // variables locales de C
  data boolean v_fresh ... // état des copies de v
  data w_type C_w ... // variables distantes lues par C
  export port ty_share_v_C share_v(v) // variables accessibles à d'autres atomes
  export port ty_share_w_C share_w(C_w) // variables importées
  export port ty_ev_C ev(v) // variables lues par ev depuis d'autres atomes

  place S initial to S do { v_fresh = false; }
  on share_v from S to S // le connecteur lit v
    provided (v_fresh == false) // les copies ne sont pas à jour
    do { v_fresh = true; } // elles le sont après action du connecteur
  on share_w from S to S // le connecteur écrit C_w
  on ev from S to S
    provided (ev_garde // la garde de ev projetée sur C est vérifiée
        && v_fresh == true) // les autres atomes ont une copie à jour
    do { ev_action } // exécution de l'action de ev projetée sur C
end
```

Application à l'exemple

Le schéma précédent est appliqué aux trois composants de notre exemple. Le composant `C` exporte la variable `v` et gère la fraîcheur des copies via `v_fresh`. Il contraint l'événement `ev` en n'autorisant son déclenchement que si `v_fresh` est à `true`. Le composant `E` exporte la variable `u` via le connecteur associé à l'événement `ev`. Le composant `C` contient la variable locale `w` et les copies `D_u` et `D_v` mises à jour d'une part par la synchronisation sur `ev` pour une utilisation dans l'action et d'autre part par la synchronisation sur `share_v` pour une utilisation ultérieure dans la garde de `ev`.

```
atom type ty_C()
  /* state variables */
  data int v
  data bool v_fresh
  /* ports */
  export port ty_share_v_C share_v(v)
  export port ty_empty_port ev()
```

```
  place P0
  /* etat initial */
   initial to P0 do { v_fresh = false; }
  /* transitions */
  on share_v from P0 to P0
     provided(v_fresh == false) do { v_fresh = true; }

  on ev from P0 to P0 provided (v_fresh == true) do { }
end

atom type ty_D()
  /* state variables */
  data int w
  data int D_u
  data int D_v
  /* ports */
  export port ty_share_v_D share_v(D_v)
  export port ty_ev_D ev(D_u)

  place P0
  /* etat initial */
   initial to P0 do { }
  /* transitions */
  on share_v from P0 to P0

  on ev from P0 to P0 provided (D_v>0) do { w = w+D_v+D_u; }
end

atom type ty_E()
  /* state variables */
  data int u
  /* ports */
  export port ty_ev_E ev(u)

  place P0
   initial to P0 do { }
    /* transitions */
    on ev from P0 to P0
end
```

Listing 3.42 – Les types de sous-composants BIP générés

3.6.4 Le composant composite

Le composant composite 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.

```
compound type dist_grd_act_rep()
  component ty_C C()
  component ty_D D()
  component ty_E E()

  connector ty_share_v share_v(C.share_v,D.share_v)
  connector ty_ev ev(C.ev,D.ev,E.ev)
end
```

Listing 3.43 – Le composant composite BIP généré

3.6.5 Génération du code exécutable

L'architecture BIP que nous produisons est à compléter manuellement par les types de données et les comportements des composants atomiques. Pour automatiser cette phase,

nous comptons utiliser le plug-in Theory [32] de la plateforme Rodin. En effet, le composant Theory permet d'élaborer des théories mathématiques prouvées (types de données, opérateurs, des règles de réécritures, des règles d'inférence). Ceci autorise l'extension d'Event-B par des structures de données très utiles comme les tableaux, listes linéaires et tables de hachage pour lesquelles une implantation C++ pourrait être fournie et exploitée depuis le code BIP.

3.7 Outils supports

Les trois étapes Fragmentation, Distribution et Génération de code de notre démarche de développement des systèmes distribués sont considérées comme des opérations de transformation source-source. Les deux premières étapes sont deux opérations de transformation de nature endogène (Event-B vers Event-B) tandis que la troisième étape est une opération de transformation de nature exogène (Event-B vers BIP). Les deux opérations de transformation Fragmentation et Distribution sont guidées par des spécifications décrites à l'aide de DSL appropriés. Ces deux DSL ne sont pas des langages de programmation. Ils permettent de décrire des données complexes nécessaires au fonctionnement des deux plug-in Fragmentation et Distribution. Pour réaliser ces trois opérations de transformation, nous avons opté pour le framework d'ingéniérie des modèle standard de facto EMF [77]. En effet, EMF supporte deux langages Xtext [3] et Xtend [2] complémentaires permettant respectivement l'analyse syntaxique et la génération de code au format textuel.

3.7.1 Implantation des langages utilisés

La démarche de développement des systèmes distribués proposée utilise les langages suivants : DSL Fragmentation, DSL Distribution, Event-B et BIP. Les trois premiers langages sont utilisés en entrée des opérations de transformation Fragmentation, Distribution et Génération de code. Ainsi, ils sont reconnus via le framework Xtext, de même que le code Event-B des machines en entrée du processus. Xtend est utilisé pour générer les modèles Event-B raffinés ou projetés ainsi que le code BIP, produit par notre démarche de développement des systèmes distribués.

Les grammaires écrites sous Xtext s'appuient sur la forme étendue de Backus-Naur (EBNF). Nous avons décrit en Xtext les grammaires de trois langages DSL Fragmentation, DSL Distribution et Event-B. L'outil Xtext produit trois EDI (Environnement de développement Intégré) spécifiques respectivement pour le DSL Fragmentation (voir le listing 3.44), DSL Distribution (voir le listing B.11 dans l'annexe B) et Event-B (voir les deux listings B.9 et B.10 dans l'annexe B). Chaque langage traité est doté d'un éditeur de texte convivial permettant de vérifier le code saisi.

```
grammar fr. irit .eventb.dependency.Dependency with fr. irit .eventb.formulas.Formulas

generate dependency "http://www.irit. fr/eventb/dependency/Dependency"

import 'http://www.irit. fr/eventb/machine/Machine' as mch
import 'http://fr. irit .eventb.emf.ident' as id

Dependency:
  'dependency' name=ID
  'refines' refines=[mch::Machine]
  'events' (events+=EventSplitting)+
  'end';

EventSplitting:
  'event' event=[mch::Event] (parameters+=ParameterDependency)+;

ParameterDependency:
```

```
('when' (parameters+=[ id::ParameterIdentifierExpression ])+)?
'parameter' parameter=[ id::ParameterIdentifierExpression ]
'init ' init =expression
'with' (guards+=[mch::Guard])+;
```

Listing 3.44 – Grammaire Xtext de fragmentation

3.7.2 Transformation source-source en Xtend

Les trois transformations Fragmentation (Event-B2Event-B), Distribution (Event-B2Event-B) et Génération de code (Event-B2BIP) sont programmées en Xtend. Ce langage est adapté à la transformation source-source : il offre des facilités pour construire des chaînes de caractères incorporant des expressions à évaluer entre «...», des insertions conditionnelles (`IF ... ENDIF`) ou indexées sur des collections (`FOR ... ENDFOR`). Son intégration à Xtext permet la génération automatique du code lors de la mise à jour du source Event-B ou du DSL.

Le listing ci-dessous, extrait du générateur associé à la fragmentation, illustre la création d'une machine dont le nom est donné par la valeur de l'attribut `name` du DSL (voir la ligne 2 du listing 3.45). Le nom de la machine raffinée est indiqué dans le DSL via le mot clé `refines` (voir la ligne 3 du listing 3.45). Une clause `sees` est ajoutée si elle est présente dans la machine mentionnée par le DSL (voir les lignes de 4 à 6 du listing 3.45). Dans ce cas, les noms des contextes vus sont insérés (voir la ligne 5 du listing 3.45). Il en est de même pour les variables de la machine (voir les lignes de 7 à 11 du listing 3.45).

```
1  '''
2    machine «dep.name»
3    refines «dep.refines.name»
4    «IF !dep.refines.sees.empty»
5      sees «FOR c :dep.refines.sees» «c.name» «ENDFOR»
6    «ENDIF»
7    variables
8    // inherited variables
9    «FOR v : dep.refines.variables»
10     «v.name»
11   «ENDFOR»
12   end
13 '''
```

Listing 3.45 – Génération de code en Xtend

Sur le plan fonctionnel, les trois figures 3.15, 3.16 et 3.17 donnent respectivement les entrées et sorties des trois opérations de transformation Fragmentation, Distribution et Génération de code.

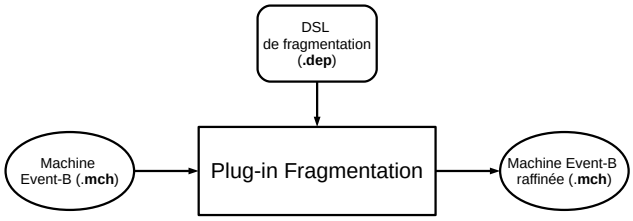

FIGURE 3.15 – Vue fonctionnelle du plug-in Fragmentation

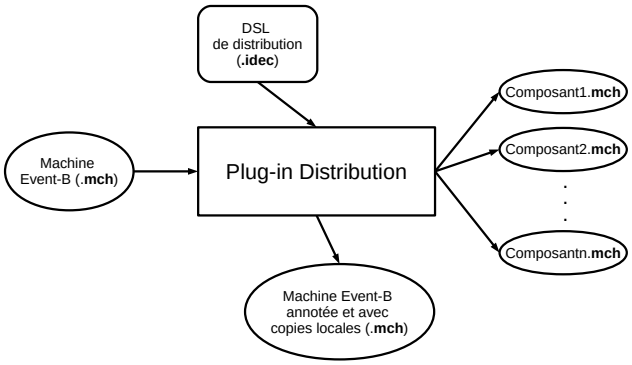

FIGURE 3.16 – Vue fonctionnelle du plug-in Distribution

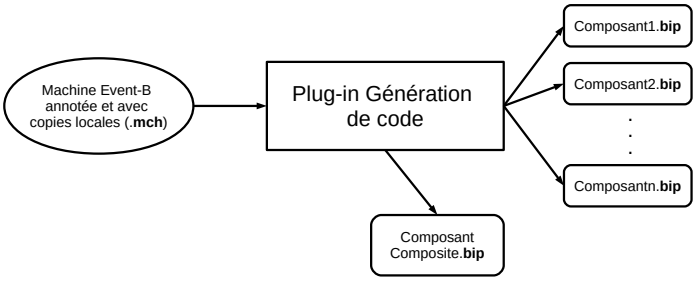

FIGURE 3.17 – Vue fonctionnelle du plug-in Génération de code BIP

3.7.3 Discussion

L'implantation en Xtext de deux DSL Fragmentation et Distribution ne couvre pas des aspects liés à leurs sémantiques statiques. Dans un futur proche, il serait bénéfique de définir la sémantique statique de ces deux DSL sous forme des contraintes exprimées sur les deux méta-modèles (en OCL dans le contexte d'UML ou en Xtend ici). Ceci favoriserait la cohérence des descriptions utilisant ces deux DSL. Par exemple, tous les paramètres d'un événement de synchronisation sont bel et bien initialisés dans une description utilisant le DSL Fragmentation. De même, l'ensemble des variables à distribuer forme une partition au sens mathématique dans une description utilisant le DSL Distribution.

La génération de code BIP fait partie intégrante techniquement de l'étape Distribution. Pour des raisons de réutilisabilité, il serait avantageux de restructurer (activité de refactoring) notre implémentation afin d'isoler la génération de code BIP dans un module autonome.

Les deux technologies complémentaires Xtext et Xtend ont été utilisées avec profit pour développer un **prototype** montrant la faisabilité de notre démarche de développement des systèmes distribués. A long terme, entre autres pour des raisons de diffusion, il serait intéressant d'intégrer totalement notre démarche au sein de la plateforme extensible Rodin.

3.8 Conclusion

Dans ce chapitre, nous avons montré la faisabilité et l'utilité de faire coopérer Event-B et BIP afin de développer des systèmes distribués de qualité : corrects par construction et exécutables sur divers plateformes matérielles. Pour y parvenir, nous avons élaboré une démarche utilisant judicieusement les concepts suivants : raffinement manuel, raffinement automatique et génération de code distribué. Le raffinement manuel est induit par le processus formel de développement supporté par Event-B. Il permet, pas-à-pas, l'obtention des spécifications

centralisées prouvées et validées en se servant des prouveurs internes et externes et de la boîte à outils de ProB accompagnant la plateforme Rodin. Le raffinement automatique vise la décomposition formelle c'est-à-dire le passage d'un modèle centralisé (spécification centralisée) en Event-B vers un modèle distribué (sous-spécifications qui communiquent par événement partagé) Event-B en tenant compte du modèle distribué BIP. Dans ce travail, nous avons identifié deux schémas de raffinement automatique appelés Fragmentation et Distribution. Le premier apporte une solution au problème de l'élimination des paramètres locaux d'un événement de synchronisation censé être partagé par plusieurs sous-spécifications. Une telle solution s'appuie sur la description d'un ordre partiel sur les paramètres locaux qui est fourni par le spécifieur moyennant un DSL adéquat. Le second (raffinement automatique Distribution) fournit une solution au problème de répartition des variables et gardes sur plusieurs composants Event-B moyennant l'utilisation d'un DSL approprié. Enfin, l'étape de Génération de code distribué permet de définir des règles de traduction systématique simples d'Event-B réduit -appelé Event-B0- vers BIP.

Les trois étapes Fragmentation, Distribution et Génération de code ont été implémentées en utilisant les deux technologies complémentaires Xtext et Xtend. Xtext est utilisé pour implanter le DSL de Fragmentation, le DSL de Distribution et le langage Event-B. Le langage Xtend est utilisé pour matérialiser les trois transformations M2M : Event-B vers Event-B (étape de Fragmentation), Event-B vers Event-B (étape de Distribution) et Event-B0 vers BIP (étape de génération de code).

Dans le chapitre suivant, nous allons appliquer, tester et évaluer notre démarche de développement des systèmes distribués combinant Event-B et BIP sur l'application de l'Hôtel introduite dans 2.2.2.

Chapitre 4

Étude de cas : Système de clés électroniques pour les hôtels

Sommaire

4.1 Introduction

Dans le chapitre précédent, nous avons préconisé une démarche de développement des systèmes distribués couplant Event-B et BIP. Dans ce chapitre, nous allons tester et évaluer empiriquement ladite démarche. Pour y parvenir, nous avons retenu l'application Hôtel [8, 45]. Dans un premier temps, nous étudierons les modélisations formelles existantes de l'application Hôtel. Dans un deuxième temps, nous restructurons le cahier des charges de l'application Hôtel explicitant les exigences fonctionnelles, environnementales, architecturales et de sécurité. Dans un troisième temps, nous allons établir une stratégie de raffinement adéquate à

l'application Hôtel. Enfin, nous appliquerons la stratégie de raffinement retenue. Ce chapitre contient plusieurs arguments informels et formels justifiant les choix de modélisation proposés. En outre, il apporte des pistes ou heuristiques qui facilitent l'application de deux étapes Fragmentation et Distribution de notre démarche de développement des systèmes distribués.

4.2 Modélisations formelles existantes

L'application Hôtel a été modélisée par divers langages formels. En effet, Jackson dans son livre [45] a établi une comparaison empirique des langages de spécification formelle Alloy, B, OCL, VDM et Z. Pour y parvenir, il a retenu l'application Hôtel. Celle-ci a été spécifiée par des experts connus et reconnus dans ces langages. La comparaison proposée par Jackson touche au pouvoir expressif de ces cinq langages. De plus, elle concerne les outils d'analyse et de vérification associés à ces langages. Pour décrire les données, les langages retenus s'appuient sur divers concepts : tout est relation pour Alloy, ensemble pour B et Z, type pour VDM et objet pour OCL. Pour décrire les traitements, ces langages utilisent la spécification Pré/Post pour Alloy, OCL, VDM et Z, et un langage de substitutions généralisées pour B. Les outils d'analyse et vérification associés aux langages comparés s'appuient sur deux grandes techniques : recherche dans un espace d'états fini (cas d'Alloy avec un outil natif appelé Analyseur d'Alloy et animateur pour OCL) et démonstrateurs de théorèmes (cas B, VDM et Z).

Le travail décrit dans [61] modélise l'application Hôtel afin de montrer la faisabilité d'une transformation d'Event-B vers UML/OCL.

Le travail décrit dans [29] propose un programme concurrent en Ada issu manuellement de l'application Hôtel spécifiée en Event-B.

En utilisant l'assistant de preuve Isabelle/HOL, le travail décrit dans [56] modélise l'application Hôtel afin de prouver formellement la propriété de sécurité suivante : "*only the owner of a room can be in a room*". Pour y parvenir, outre les variables fonctionnelles, il introduit deux variables de preuve : `isin` et `safe`. Ces variables sont mises à jour par quatre transitions : `init`, `check_in`, `enter_room` et `exit_room`. La sécurité de l'application Hôtel est formalisée grâce à neuf lemmes et un théorème.

En s'inspirant de ces travaux, nous allons proposer une spécification centralisée en Event-B de l'application Hôtel. Ensuite, en se servant de nos deux plug-ins Fragmentation et Distribution, nous allons produire un modèle distribué lié à cette application en Event-B prêt à être traduit systématiquement en BIP. Enfin, en utilisant notre plug-in de Génération de code, nous allons obtenir un squelette BIP quasi-exécutable sur divers plateformes matérielles telles que : mono-processeur, multi-processeurs et multi-coeurs.

4.3 Restructuration du cahier des charges : texte référentiel

En général, les cahiers des charges ne se prêtent pas à un développement formel par raffinements successifs prouvés mathématiquement. Pour remédier à ce problème, Abrial propose des guides méthodologiques permettant de restructurer un cahier des charges selon une démarche inspirée de la formulation des documents mathématiques [8]. Elle consiste à l'élaboration de deux textes dits référentiel et explicatif. Pour parvenir à restructurer le cahier des charges relatif à l'application Hôtel, nous avons appliqué avec profit cette démarche. Néanmoins, nous nous sommes limités à la rédaction du texte référentiel. Celui-ci comporte les exigences suivantes :

L'application Hôtel permet de gérer le contrôle d'accès aux chambres d'un hôtel.	**FUN-1**

Chaque chambre est dotée d'une serrure électronique autonome.	**ENV-1**
Une serrure électronique mémorise une clé électronique et est dotée d'un lecteur de cartes magnétiques.	**ENV-2**
Tout client ayant effectué une réservation à l'accueil, aura une carte magnétique lui permettant l'accès à sa chambre.	**ENV-3**
Une carte magnétique contient deux clés : cle1 et cle2.	**ENV-4**
Un client ne peut accéder à une chambre qu'en introduisant sa carte dans la serrure de la chambre.	**ENV-5**
La serrure ouvre la porte si l'une des clés de la carte introduite dans le lecteur de carte correspond à sa clé.	**FUN-2**
La serrure autonome peut mettre à jour sa clé avec la seconde clé de la carte.	**FUN-3**
Le système comporte trois types de composants qui coopèrent : Client, Chambre et Accueil.	**ARCH-1**
Un client dans une chambre doit être un client enregistré.	**SAF-1**
Une carte éditée demeure inchangée.	**SAF-2**

4.4 Stratégie de raffinement proposée

Le modèle que nous proposons a été développé en suivant une stratégie fondée sur des raffinements successifs. Une telle stratégie est formée de deux phases à savoir : phase de spécification et phase de distribution. La phase de spécification a été établie selon un raffinement horizontal du modèle initial. Elle comporte deux modèles : le modèle initial abstrait et le modèle initial raffiné. La phase de distribution consiste à fragmenter et à distribuer le modèle issu de la phase de spécification pour obtenir une spécification décentralisée. Les différentes phases sont donc les suivantes :

— **Modèle initial abstrait** : ce modèle est assimilé à une application de réservation classique auprès du bureau d'accueil. En outre, ce modèle permet aux clients résidents d'accéder à leurs chambres. Nous nous intéressons à l'exigence fonctionnelle **FUN-1** et de sécurité **SAF-1**.

— **Modèle initial raffiné** : ce modèle introduit les notions de cartes, de clés et de serrures électroniques. Nous nous intéressons aux exigences **FUN-2**, **FUN-3**, **ENV-1**, **ENV-2**, **ENV-3**, **ENV-4**, **ENV-5** et **SAF-2**.

— **Modèle fragmenté** : ce modèle est obtenu automatiquement par la fragmentation du modèle initial raffiné. Un tel raffinement consiste à réduire le non-déterminisme lié au calcul des paramètres locaux des événements partagés (événements de synchronisation) entre les composants. Nous nous intéressons à l'exigence architecturale **ARCH-1**.

— **Modèle distribué** : ce modèle résulte de la distribution du modèle issu de l'étape de fragmentation dans l'objectif d'obtenir une spécification distribuée ; modèle séparant les différentes entités, représentées dans le **premier raffinement** d'une façon centralisée, chacune à part entière.

La figure 4.1 illustre la stratégie de raffinement proposée.

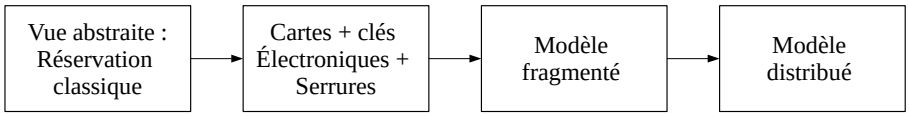

FIGURE 4.1 – Stratégie de raffinement adoptée pour l'application Hôtel

4.5 Développement en Event-B

Le développement en Event-B de l'application Hôtel comporte deux phases : spécification et distribution. La phase de spécification est formée des deux modèles à savoir : le modèle initial `Hotel_M0` et le modèle initial raffiné `Hotel_M1`. La phase de distribution est formée des cinq modèles `Hotel_dep`, `Hotel_split`, `Desk`, `Room` et `Guest`. Ces modèles résultent de la fragmentation suivie de la distribution du modèle issu de la phase de spécification et concernent les trois entités définies dans la section 4.4 (bureau d'accueil (`Desk`), les chambres (`Room`) et les clients (`Guest`)). La figure 4.2 illustre l'architecture générale de la modélisation en Event-B de l'application Hôtel.

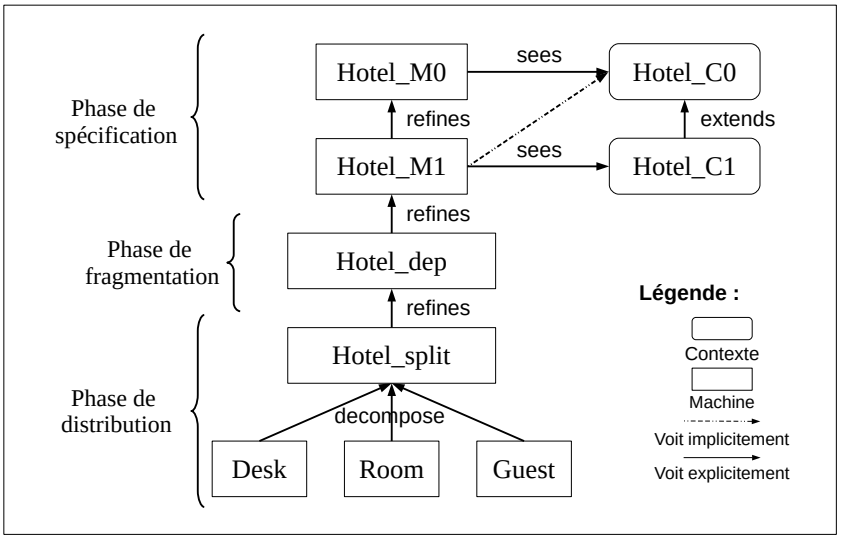

FIGURE 4.2 – Architecture générale du modèle Event-B proposé de l'application Hôtel

La machine `Hotel_M0` voit explicitement le contexte `Hotel_C0` et la machine `Hotel_M1` voit explicitement le contexte `Hotel_C1` et implicitement le contexte `Hotel_C0`.

4.5.1 Spécification centralisée

4.5.1.1 Modèle abstrait initial : réservation des chambres d'un hôtel

Dans cette section, nous spécifions notre application Hôtel par un modèle initial abstrait qui décrit une réservation classique des chambres d'un hôtel. En effet, un client peut se présenter à la réception de l'hôtel pour s'enregistrer. En plus, un client enregistré peut entrer ou sortir de sa chambre.

Modélisation

Notre modèle abstrait Event-B assure qu'un client enregistré accède à sa chambre d'une façon exclusive.

Le modèle (`Hotel_C0` et `Hotel_M0`) respecte les contraintes décrites par **FUN-1** et **SAF-1**(voir la section 4.3). Notre système est paramétré par deux ensembles abstraits `GUEST` et `ROOM` modélisant respectivement l'ensemble des clients et l'ensemble des chambres de l'hôtel. Ce paramétrage est localisé dans le contexte `Hotel_C0` qui est représenté par le listing 4.1.

```
context Hotel_C0 // parametres du modele Hotel_M0

sets GUEST // Ensemble des clients de l'hotel
     ROOM // Ensemble des chambres de l'hotel

end
```

Listing 4.1 – Le contexte Hotel_C0

Le modèle abstrait `Hotel_M0` (voir le listing C.12 dans l'annexe C) comporte deux variables d'état (voir le listing 4.2). La première, appelée `registered`, est une fonction partielle de l'ensemble `ROOM` vers l'ensemble `GUEST`. Elle mémorise des couples dont l'antécédent est une chambre disponible et l'image est un propriétaire. La deuxième, notée `isin`, est une variable liée à l'environnement. Elle est modélisée par une fonction totale de l'ensemble `ROOM` vers `ℙ(GUEST)`. Elle mémorise des couples dont l'antécédent est une chambre et l'image est l'ensemble de clients à l'intérieur de ladite chambre. L'invariant `safe` (voir le listing 4.2) stipule qu'un client dans une chambre est forcément un client enregistré conformément à l'exigence **SAF-1** (voir section 4.3).

```
variables   registered  // Variable systeme
            isin  // Variable d'environnement
invariants
  @typ_regi  registered ∈ROOM⇸GUEST // une chambre est allouée à au plus un client
  @typ_isin  isin ∈ROOM→ℙ(GUEST) // Contenu effectif d'une chambre
  @safe ∀r·(r∈ROOM⇒isin(r)⊆ registered[{r}]) // Un client dans la chambre est un client  enregistré SAF−1
```

Listing 4.2 – État de la Machine Hotel_M0

Le modèle `Hotel_M0` comporte six événements (autre que l'événement `INITIALISATION`). Deux événements, `check_in` et `forced_check_in`, qui permettent l'enregistrement d'un client, deux événements (`check_out` et `forced_check_out`) qui assurent les terminaisons des réservations, un événement `enter_room` qui autorise l'accès d'un client à sa chambre et un événement `exit_room` qui modélise le fait qu'un client peut quitter temporairement sa chambre.

- `INITIALISATION`
  L'événement particulier `INITIALISATION` comporte deux actions déterministes `@init_regi` et `@init_isin` qui indiquent respectivement qu'initialement aucune réservation n'a été établie et que toutes les chambres sont vides (voir le listing 4.3).

```
event INITIALISATION // Initialisation des variables
  then
    @init_regi   registered :=∅
    @init_isin   isin :=ROOM×{∅}
end
```

Listing 4.3 – Événement `INITIALISATION`

- **`check_in`** et **`forced_check_in`**
  Ces deux événements permettent l'enregistrement d'un client. Ils permettent d'effectuer des réservations selon la disponibilité des chambres. En effet, contrairement à l'événement `check_in`, qui permet d'attribuer une chambre non réservée à un client potentiel, l'événement `forced_check_in` permet d'attribuer une chambre réservée mais vide à un client. Ces événements nécessitent deux paramètres locaux g et r représentant respectivement un client et une chambre à réserver. Ces deux paramètres sont typés respectivement par `@typ_r` et `@typ_g`. La garde `@free_r` représente une contrainte sur la disponibilité de la chambre à réserver au niveau de l'événement `check_in` et représente une contrainte sur le fait que la chambre `r` est vide au niveau de l'événement `forced_check_in`. L'action `@booked_r` est la même dans les deux événements `check_in` et `forced_check_in`. Elle permet de mettre à jour la variable `registered` en ajoutant le couple (`r`,`g`) (voir le listing 4.4).

```
event check_in
/* un nouveau client peut être affecté
   à une chambre non reservée */
  any g r
  where
    @typ_g g∈GUEST
  @typ_r r∈ROOM
    @free_r r∉dom(registered)
  then
    @booked_r registered(r):=g
end

event forced_check_in
/* un nouveau client peut être affecté
   à une chambre reservée mais vide */
  any g r
  where
    @typ_g g∈GUEST
    @typ_r r∈ROOM
    @empty_r isin(r)=∅
  then
    @booked_r registered(r):=g
end
```

Listing 4.4 – Événements `check_in` et `forced_check_in`

- **`check_out`** et **`forced_check_out`**
  Ces deux événements permettent d'effectuer les terminaisons des réservations à la fin des séjours des clients préalablement enregistrés. Ils sont donc liés au fonctionnement du bureau d'accueil. En effet, l'événement `check_out`, modélise le fait qu'un client règle sa note volontairement auprès de la réception. Par contre, le bureau d'accueil peut contrôler la sortie d'un client en effectuant un "**check out**" forcé (événement `forced_check_out`) en expulsant le client de la chambre. Ces événements nécessitent deux paramètres locaux `g` et `r` représentant respectivement le client propriétaire de la chambre à désallouer et la chambre à libérer. La garde `@booked_r`, dans les deux événements, représente une contrainte sur l'appartenance de la chambre `r` au client `g`.

Une contrainte supplémentaire, représentée par la garde `@empty_r`, est ajoutée dans l'événement `check_out`. Elle exprime le fait que le client `g` n'est plus dans la chambre physiquement. L'action `@free_r` dans les deux événements est une substitution déterministe permettant de libérer la chambre `r` qui était réservée. L'action `@empty_r` propre à l'événement `forced_check_out` est aussi une substitution déterministe qui permet de sortir le client `g` de sa chambre `r` (voir le listing 4.5).

```
event check_out // Modélise le check out volontaire
  any g r
  where
    @typ_g g∈GUEST
    @typ_r r∈ROOM
    @booked_r r↦g∈registered
    @empty_r g∉isin(r)
  then
    @free_r  registered :={r}⩤registered
end

  event forced_check_out
/* Modélise le check out forcé
   (réalise par le bureau d'accueil) */
  any g r
  where
    @typ_g g∈GUEST
    @typ_r r∈ROOM
    @booked_r r↦g∈registered
    @busy_r g∈isin(r) // exclusion mutuelle avec check_out
  then
    @free_r  registered :={r}⩤registered
    @empty_r isin(r) := isin(r)\{g}
end
```

Listing 4.5 – Événements `check_out` et `forced_check_out`

- **`enter_room`**
  L'événement `enter_room` est lié au fonctionnement des chambres. Il s'agit d'un événement permettant l'accès de tout client ayant effectué une réservation à sa chambre. Un tel événement nécessite deux paramètres locaux : un client `g` et une chambre `r`. Il est conditionné par une garde `@booked_r` exprimant une contrainte sur le couple (`r`, `g`) : le client `g` doit être le propriétaire de `r`. Cet événement agit sur l'état de la machine en modifiant la variable `isin`. Ceci revient à dire que le client `g` est désormais dans la chambre `r` dont il est le propriétaire (voir le listing 4.6).

```
event enter_room
  any g r
  where
    @typ_g g∈GUEST
    @typ_r r∈ROOM
    @booked_r r↦g∈registered
    @empty_r g∉isin(r)
  then
    @busy_r isin(r) := isin(r)∪{g} // g est dans la chambre physiquement
end
```

Listing 4.6 – Événement `enter_room`

- **`exit_room`**
  L'événement `exit_room` modélise le fait qu'un client peut quitter la chambre dans laquelle il se trouve. Par conséquent, il s'agit d'un événement lié aux clients. La garde `@busy_r` représente une contrainte sur les deux paramètres `g` et `r` de l'événement. Elle indique la présence physique du client dans sa chambre. L'action `@empty_r` est une

substitution déterministe permettant de mettre à jour la variable `isin` pour exprimer le fait que le client `g` n'est plus dans la chambre `r` (voir le listing 4.7).

```
event exit_room // Un client peut quitter sa chambre
  any g r
  where
    @typ_g g∈GUEST
    @typ_r r∈ROOM
    @busy_r g∈isin(r)
  then
    @empty_r isin(r) ≔ isin(r)\{g} // g n'est plus dans la chambre physiquement
end
```

Listing 4.7 – Événement `exit_room`

Obligations de preuve standards et deadlock

Toutes les obligations de preuve relatives à la correction du modèle `Hotel_M0` ont été déchargées automatiquement ou interactivement par les prouveurs de la plateforme Rodin. Elles concernent les événements `INITIALISATION`, `check_in`, `enter_room`, `exit_room`, `check_out`, `forced_check_in` et `forced_check_out`. Ces OPs assurent l'établissement et la préservation des propriétés invariantes ainsi que la vérification de la bonne définition des formules (voir la figure 4.3).

Outre les obligations de preuve standards, il est parfois nécessaire de prouver d'autres propriétés comme la propriété de non blocage. Ainsi, pour que le modèle Event-B `Hotel_M0` soit vivace – autrement dit, à chaque instant stable, le modèle admet au moins un événement franchissable – nous sommes censés prouver qu'il évolue en permanence et qu'il ne subisse aucune situation de blocage. Ceci est traduit dans le modèle `Hotel_M0` par l'adjonction du théorème étiqueté `@DLF_0` (pour **D**ead**L**ock **F**reedom) dans la clause `invariants` (voir le listing 4.8).

```
theorem @DLF_0
  (∃r,g·r∈ROOM∧g∈GUEST∧r∉dom(registered)) ∨
  (∃r,g·r∈ROOM∧g∈GUEST∧r↦g∈registered∧g∉isin(r)) ∨
  (∃r,g·r∈ROOM∧g∈GUEST∧g∈isin(r)) ∨
  (∃r,g·r∈ROOM∧g∈GUEST∧r↦g∈registered∧g∉isin(r)) ∨
  (∃r,g·r∈ROOM∧g∈GUEST∧isin(r)=∅) ∨
  (∃r,g·r∈ROOM∧g∈GUEST∧r↦g∈registered∧g∈isin(r))
```

Listing 4.8 – Théorème `DLF_0` lié à l'absence de blocage

Un tel théorème est défini par la disjonction de la conjonction de toutes les gardes des événements définis au sein de la machine `Hotel_M0`. L'OP relative au théorème `@DLF_0` a été déchargée interactivement par les prouveurs de la plateforme Rodin (voir la figure 4.3).

Ainsi, le modèle `Hotel_M0` ne se bloque pas. Intuitivement, ceci est expliqué comme suit : dans le cas où l'hôtel est complet, un client peut entrer (`enter_room`) ou sortir (`exit_room`) sinon il peut s'enregistrer (`check_in` ou `forced_check_in`) entrer ou sortir. Dans le cas où l'hôtel n'est pas complet, l'enregistrement est possible car l'ensemble `KEY` est supposé infini.

Validation

Puisque les anomalies et les oublis liés à la modélisation d'un modèle Event-B donné ne peuvent pas être décelés en l'examinant d'une façon statique. Nous avons utilisé avec profit l'outil ProB pour animer le modèle `Hotel_M0`. Pour y parvenir, des scénarios ont été établis et le comportement du modèle est analysé vis-à-vis de ces scénarios.

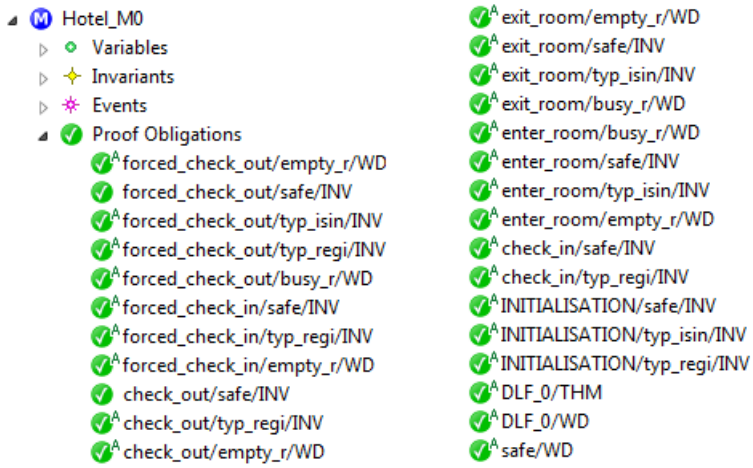

FIGURE 4.3 – Les obligations de preuve déchargées relatives à `Hotel_M0`

(Scénario 1) "Séjour avec réservation a priori : en passant par `check_in`"

Tout client qui veut séjourner dans l'hôtel doit s'enregistrer auprès du bureau d'accueil. L'enregistrement peut s'effectuer via les deux événements `check_in` et `forced_check_in` (voir scénario 2). En choisissant un enregistrement en passant par l'événement `check_in`, une chambre est réservée au client dont il peut entrer (événement `enter_room`) ou sortir (événement `exit_room`) autant de fois qu'il désire. Le comportement dynamique d'un tel scénario est vérifié avec ProB (voir la figure 4.4). Ainsi, en considérant un client `GUEST1` ayant réservé la chambre `ROOM2`, le franchissement de la séquence formée par les événements `check_in`(`GUEST1`, `ROOM2`) , (`enter_room`(`GUEST1`, `ROOM2`), `exit_room`(`GUEST1`, `ROOM2`))*, `check_out`(`GUEST1`, `ROOM2`) décrit le séjour du client `GUEST1`.

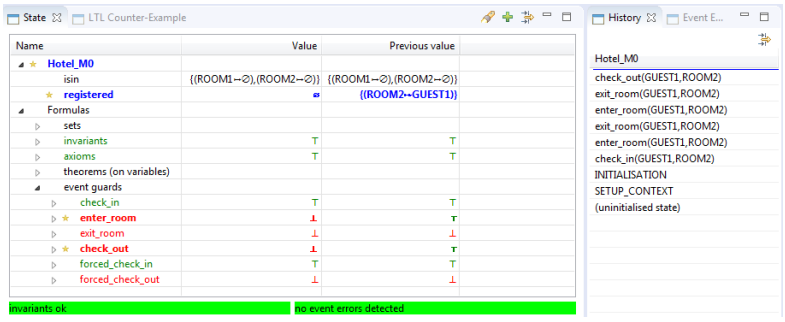

FIGURE 4.4 – Animation de la machine `Hotel_M0` suivant le scénario 1

(Scénario 2) "Séjour avec réservation a priori : en passant par `forced_check_in`"

A la différence du scénario 1, pour ce scénario nous considérons une réservation de séjour en passant par l'événement `forced_check_in`. La figure 4.5 -colonne History- montre la vérification de ce scénario à l'aide de l'animateur ProB.

(Scénario 3) "Séjour avec fin forcée"

Le bureau d'accueil peut contrôler la sortie d'un client en effectuant un "check out" forcé (événement `forced_check_out`) en expulsant le client de la chambre (voir la figure 4.6).

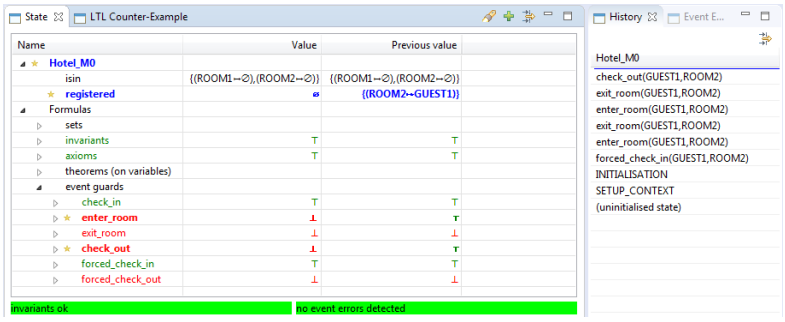

FIGURE 4.5 – Animation de la machine `Hotel_M0` suivant le scénario 2

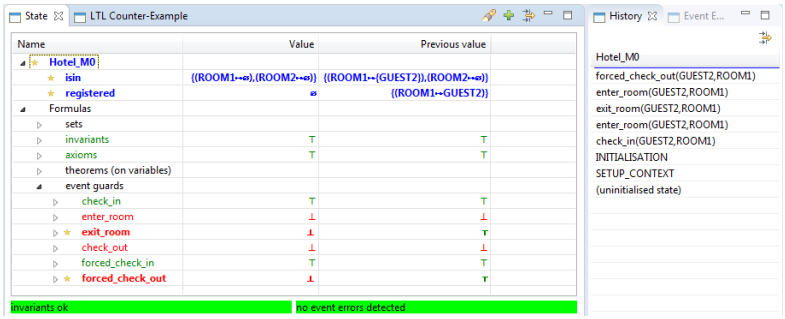

FIGURE 4.6 – Animation de la machine `Hotel_M0` suivant le scénario 3

(Scénario 4) "L'hôtel est complet"

L'état initial de la machine `Hotel_M0` modélise qu'aucune réservation n'est enregistrée. Moyennant un nombre de réservation égal au nombre de chambres de l'hôtel, l'événement `chech_in` devient non franchissable puisque l'hôtel est complet (voir la figure 4.7).

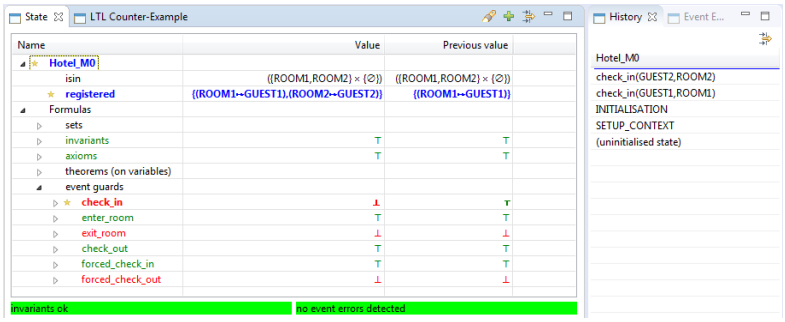

FIGURE 4.7 – Animation de la machine `Hotel_M0` suivant le scénario 4

(Scénario 5) "Les deux événements `enter_room` et `exit_room` sont déclenchables **séquentiellement avec répétition** "

La réservation d'une chambre par un client est un événement principal permettant au propriétaire d'entrer à sa chambre. Considérons un client `GUEST1` ayant réservé la chambre `ROOM1`. Ainsi le client peut entrer et sortir de sa chambre autant de fois qu'il désire durant son séjour. Dans la figure 4.8, le client `GUEST1` entre (événement `enter_room`

et sort (événement `exit_room`) de sa chambre `ROOM1` deux fois et de manière séquentielle.

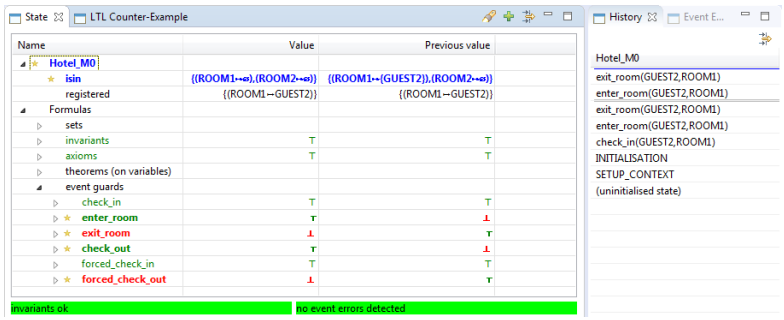

FIGURE 4.8 – Animation de la machine `Hotel_M0` suivant le scénario 5

(Scénario 6) "Les deux événements `chech_in` et `forced_check_in` **sont en compétition**"

La figure 4.9 montre comment les deux événements `chech_in` et `forced_check_in` **sont en compétition**. En effet, à un instant donné, ces deux événements sont franchissables.

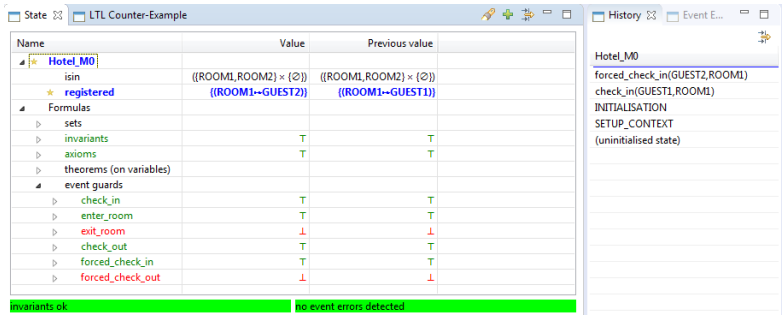

FIGURE 4.9 – Animation de la machine `Hotel_M0` suivant le scénario 6

(Scénario 7) "Les deux événements `chech_out` et `forced_check_out` **sont en exclusion mutuelle**"

D'après l'historique de l'animation fournie par la figure 4.10, nous avons constaté que soit l'événement `chech_out` est franchissable, soit l'événement `forced_check_out` est franchissable et jamais les deux à la fois.

Discussion

Tout d'abord, nous avons restructuré le cahier des charges de l'application Hôtel en énumérant ses différentes exigences fonctionnelles, matérielles et architecturales. Ceci nous a permis d'élaborer une stratégie de raffinement adéquate pour cette application. Ensuite, les modèles formels présentés de l'application Hôtel dans le livre de Jackson [45] (voir section 4.2) en utilisant les langages B, VDM et Z, comportent, en général, une variable d'état pour l'allocation des chambres et deux transitions `checkin` et `enter`. Vis-à-vis de ces modèles, notre modélisation en Event-B enrichit l'espace d'états par une variable d'environnement (`isin`) et apporte

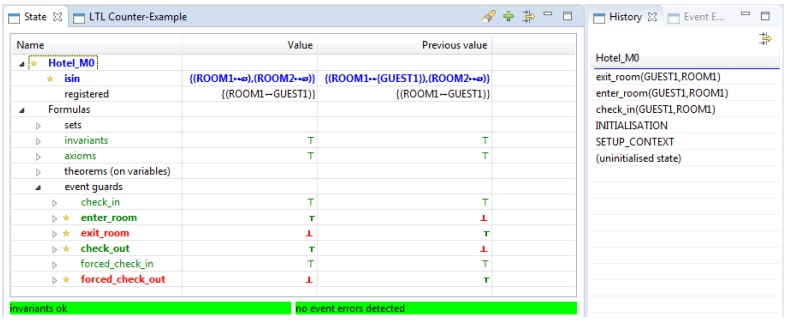

FIGURE 4.10 – Animation de la machine `Hotel_M0` suivant le scénario 7

quatre nouvelles transitions à savoir `forced_check_in`, `check_out`, `forced_check_out` et `exit_room`. Contrairement aux modèles décrits en B et Z, notre modèle en Event-B modélise explicitement la fin de séjour via deux transitions `check_out` et `forced_check_out`. En outre, il supporte plusieurs types de séjours. Sachant qu'un séjour démarre soit par `check_in`, soit par `forced_check_in` et se termine soit par `check_out`, soit par `force_check_out`. Entre le début et la fin de séjour, il peut y avoir plusieurs occurrences de deux événements `enter_room` et `exit_room`. Enfin, les modèles en Alloy et OCL decrits dans le livre de Jackson [45] sont plutôt concrets. Ils englobent les clés et cartes et supportent un seul type de séjour en proposant une interface comportant deux services `checkin` et `enter`.

4.5.1.2 Premier raffinement : introduction des clés, cartes et serrures électroniques

Dans cette section, nous allons présenter la machine concrète `Hotel_M1` qui raffine le modèle initial `Hotel_M0` sans le contredire. On va introduire les notions liées aux cartes, aux clés et aux serrures électroniques.

Modélisation

L'introduction de la notion de clés conduit à la définition d'un nouvel ensemble abstrait, nommé `KEY`, qui représente l'ensemble des clés potentielles. Cependant, une carte est modélisée par deux clés `k1` et `k2`. Nous définissons la constante `CARD` qui dénote l'ensemble de toutes les cartes électroniques. Un tel ensemble est considéré comme le produit cartésien de `KEY` × `KEY` privé des couples formés de la même clé (`k`, `k`).

Pour effectuer une distribution initiale de clés sur les différentes chambres de l'hôtel, nous définissons la constante `initk`. Il s'agit d'une fonction totale injective de l'ensemble abstrait `ROOM` vers l'ensemble abstrait `KEY`. Le caractère injectif de la fonction `initk` (voir axiome `@typ_initk`) assure que deux chambres différentes ne puissent pas avoir la même clé courante. Les trois ensembles `KEY`, `CARD` et `initk` sont localisés au sein du contexte `Hotel_C1` (voir le listing 4.9) qui étend le contexte `Hotel_C0` (voir le listing 4.1). Notons au passage que le cardinal de l'ensemble `KEY` est censé être largement supérieur à `ROOM` voire infini.

```
context Hotel_C1 // Parametres du modele Hotel_M1
 extends Hotel_C0

sets KEY // Ensemble des clés

constants CARD // Ensemble des cartes
          initk // Distribution initiale des clés courantessur les chambres de l'htel

axioms
```

```
  @typ_CARD CARD=(KEY×KEY)\{k↦k|k∈KEY} // Toutes les cartes ont deux clés différentes ENV−4
  @typ_initk  initk ∈ROOM⤖KEY // Deux chambres n'ont pas la meme clé courante

end
```

Listing 4.9 – Le contexte Hotel_C1

L'état du modèle concret `Hotel_M1`, qui étend l'état abstrait, est représenté par les six variables concrètes suivantes (voir le listing 4.10) :

- `owns` : elle associe des clients enregistrés aux chambres qu'ils ont réservées. Cette variable est similaire à la variable abstraite `registered` avec une différence de synchronisation. En effet, la mise à jour de la variable `owns` est synchronisée avec les nouvelles variables introduites : `currk`, `issued` et `cards` (voir l'événement `register` dans le listing 4.15). La variable `owns` est une fonction partielle de `ROOM` dans `GUEST`. Cette définition est assurée par l'invariant `@typ_owns`.
- `currk` : association entre les chambres et les clés courantes enregistrées auprès du bureau d'accueil. Il s'agit d'une fonction totale injective de `ROOM` vers `KEY` (voir `@typ_currk`).
- `issued` : ensemble de clés effectivement utilisées. Il s'agit d'un sous-ensemble, inclus au sens large, dans l'ensemble de clés potentielles `KEY` (voir `@typ_issued`).
- `cards` : association entre cartes et clients. C'est une fonction totale injective de `GUEST` vers $\mathbb{P}$(`CARD`) (voir `@typ_cards`).
- `roomk` : association entre les chambres et les clés courantes enregistrées par les serrures électroniques. C'est une fonction totale de `ROOM` vers `KEY` (voir `@typ_roomk`).
- `isin` : variable provenant de la machine abstraite.

Ces variables sont déclarées dans la clause `variables` et sont typées dans la clause `invariants`. En outre, d'autres propriétés sur ces variables peuvent être définies.

```
variables  owns currk  issued  cards  roomk  isin

invariants
  @typ_owns owns∈ROOM↔GUEST // Typage
  @typ_currk currk∈ROOM⤖KEY // Typage
  @typ_issued  issued⊆KEY // Typage
  @fin_issued  finite ( issued )
  @typ_cards cards∈GUEST→ℙ(CARD) // Typage
  @typ_roomk roomk∈ROOM→KEY // Typage ENV−2
  @key_room_desk ∀r·(r∈ROOM⇒currk(r)∈issued)
  @key1_card ∀g·(g∈GUEST⇒(∀c·(c∈cards(g)⇒prj1(c)∈issued))) // ENV−4
  @key2_card ∀g·(g∈GUEST⇒(∀c·(c∈cards(g)⇒prj2(c)∈issued))) // ENV−4
  @key_lock ∀r·(r∈ROOM⇒roomk(r)∈issued)
  @keys_card ∀r,g,k·(r∈ROOM∧g∈GUEST∧k∈KEY∧k∉issued⇒currk(r)↦k∉cards(g)) // ENV−4
  @uniq_card1 ∀g1,g2·(cards(g1)∩cards(g2) ≠ ∅ ⇒ g1=g2)
  @uniq_card2 ∀c1,c2,g1,g2·c1∈cards(g1) ∧ c2∈cards(g2) ∧ prj2(c1)=prj2(c2) ⇒ c1=c2
  @inv_Hotel_M1_safe ∀r,g·(r∈ROOM∧g∈GUEST∧g∈isin(r)⇒r↦g∈registered) // Tout client dans une
      chambre est un client enregistré
  @inv_coll  ∀c,r,g·(r∈ROOM∧g∈GUEST∧c∈cards(g)∧roomk(r)=prj2(c)⇒r↦g∈registered)
```

Listing 4.10 – État de la machine `Hotel_M1`

Les invariants étiquetés `@key_room_desk`, `@key1_card`, `@key2_card`, `@key_lock` formalisent des propriétés sur les clés effectivement utilisées. Ils indiquent respectivement que la clé courante associée à une chambre par le bureau d'accueil, la clé primaire et secondaire d'une carte utilisée et la clé portée par une serrure électronique d'une chambre sont des clés marquées comme utilisée (ensemble `issued`). L'invariant `@keys_card` exprime le fait qu'une carte dont la clé primaire correspond à une clé d'une serrure et la deuxième clé est une clé non utilisée est une carte qui pourrait être éditée. Les deux invariants `@uniq_card1` et `@uniq_card2` stipulent respectivement une carte appartient à un et un seul client et elle est unique. En

outre, nous proposons deux invariants de collage `@inv_coll` et `@inv_Hotel_M1_safe`. Ils explicitent les liens entre variables concrètes `cards`, `roomk` et `isin` et la variable abstraite `regsitered` supprimée. Ainsi, le contenu de `registered` est régi par ceux deux invariants.

Le modèle concret `Hotel_M1` comporte trois événements `get_room`, `enter_room` et `exit_room` qui raffinent respectivement les événements abstraits `forced_check_in`, `enter_room` et `exit_room`. De plus, deux nouveaux événements ordinaires (statut `ordinary`) sont ajoutés dans le modèle `register` et `unregister`. En effet, l'enregistrement d'un nouveau client qui réserve une chambre est modélisé par l'événement `register` et la terminaison de la réservation d'une chambre pour un client est modélisé par l'événement `unregister`.

Nous présentons dans la suite les événements concrets définis au sein de la machine `Hotel_M1`.

- `INITIALISATION`
  L'événement particulier `INITIALISATION` (voir le listing 4.11) initialise l'état de la machine `Hotel_M1`. Il fait appel aux éléments de modélisation, notamment à la fonction `initk`, introduits dans le contexte `Hotel_C1` (voir le listing 4.9).

```
event INITIALISATION // Initialisation des variables
  then
    @init_owns owns :=∅
    @init_currk currk :=initk
    @init_cards cards :=GUEST×{∅}
    @init_isin isin :=ROOM×{∅}
    @init_issued issued :=ran( initk )
    @init_roomk roomk :=initk
end
```

Listing 4.11 – Événement `INITIALISATION` de la machine `Hotel_M1`

L'ensemble de clés effectivement utilisées `issued` est initialisé par l'ensemble de clés préalablement utilisées pour la distribution initiale de clés courantes (`ran(initk)`). Les deux variables `currk` et `roomk` sont initialisées selon la fonction `initk`. Ainsi, les clés courantes des chambres enregistrées par les serrures électroniques et par le bureau d'accueil sont celles qui ont été distribuées initialement sur ces chambres. Initialement, toutes les chambres sont vides (voir `@init_isin`), aucun client n'est enregistré (voir `@init_owns`) et aucune carte n'est éditée (voir `@init_cards`).

- `exit_room`
  Il est identique à son homologue abstrait. Pour y parvenir, on utilise l'attribut `extended` avec une partie incrémentale vide (voir le listing 4.12).

```
event exit_room extends exit_room
end
```

Listing 4.12 – Événement `exit_room` de la machine `Hotel_M1`

- `get_room`
  L'événement concret `get_room` raffine l'événement abstrait `forced_check_in`. Il modélise l'opération d'insertion de la carte dans la serrure électronique d'une chambre. Dans le cas où la première clé de la carte correspond à la clé de la serrure (voir `@key_lock_c1`) et que la chambre est vide (voir `@empty_r`), la clé de la serrure est substituée par la deuxième clé de la carte (voir `@update_key_lock`). Ce raffinement introduit un nouvel paramètre `c` en plus. Ce paramètre représente la carte en possession du client `g` dont la clé primaire est une clé d'une serrure de la chambre `r`.
  L'événement concret `get_room` est représenté par le listing 4.13.

```
event get_room refines forced_check_in
  any c g r
  where
    @typ_c c∈CARD
    @card_g c∈cards(g)
    @typ_g g∈GUEST
    @typ_r r∈ROOM
    @key_lock_c1 roomk(r)=prj1(c)
    @empty_r isin(r)=∅
  then
    @update_key_lock roomk(r):=prj2(c)
end
```

Listing 4.13 – Événement `get_room` de la machine `Hotel_M1`

- **`enter_room`**
  C'est un événement qui raffine l'événement abstrait `enter_room`. Il nécessite les trois paramètres locaux `r`, `g`, et `c`. Il s'agit d'un événement qui permet au client `g` d'accéder à sa chambre sans aucune mise à jour de la clé mémorisée par la serrure. Ainsi, le système vérifie que la clé secondaire de la carte `c` correspond bien à celle mémorisée par la serrure électronique (voir `@key_lock`). La variable `isin` est mise à jour ayant pour signification que le client `g` est dans la chambre `r`.
  Le listing 4.14 décrit l'événement concret `enter_room`.

```
event enter_room // FUN−1
refines enter_room
  any c g r
  where
    @typ_c c∈CARD
    @card_g c∈cards(g)
    @typ_g g∈GUEST
    @typ_r r∈ROOM
    @empty_r g∉isin(r)
    @key_lock roomk(r)=prj2(c)
  then
    @busy_r isin(r):=isin(r)∪{g}
end
```

Listing 4.14 – Événement `enter_room` de la machine `Hotel_M1`

- **`register`**
  Il s'agit d'un événement nouvellement introduit qui raffine l'événement abstrait `skip`. Il permet la bonne gestion des réservations, la gestion des clés utilisées et l'attribution des cartes électroniques aux clients. Un tel événement nécessite les trois paramètres locaux `r`, `g`, et `k`. Les contraintes de typage et les propriétés sur ces paramètres sont données par les gardes `@typ_r`, `@typ_g`, `@typ_k`, `@free_k` et `@free_r`. Les actions sont `@key_r_desk`, `@used_k`, `@new_card` et `@booked_r`. La garde garantit la disponibilité de la chambre `r`. La nouvelle carte éditée est définie par `(currk(r), k)`. Sachant que `currk(r)` est la clé censée être portée par la serrure de la chambre `r`. En principe, `currk(r)` est différente de `k`. En effet, `currk(r)`∈ `issued` et `k`∉ `issued`.
  Le listing 4.15 décrit l'événement concret `register`.

```
event register
  any r g k
  where
    @typ_r r∈ROOM
    @typ_g g∈GUEST
    @typ_k k∈KEY
    @free_k k∉issued
```

```
    @free_r r∉dom(owns)
  then
    @key_r_desk currk(r) ≔ k
    @used_k issued ≔ issued∪{k}
    @new_card cards(g) ≔ cards(g)∪{currk(r)↦k}
    @booked_r owns(r) ≔ g
end
```

Listing 4.15 – Événement `register` de la machine `Hotel_M1`

- **`unregister`**
  Il s'agit d'un événement nouvellement introduit qui raffine l'événement abstrait `skip`. Il permet de désenregistrer un client (voir `@booked_r`) et d'invalider sa carte (voir `@update_card_g`) s'il n'est pas dans sa chambre (voir `@empty_r`). Ceci traduit le fait qu'un client a quitté sa chambre sans rendre sa carte. Le produit cartésien `KEY`×`currk(r)` fabrique toutes les cartes potentielles susceptibles d'être liées à la chambre `r`.
  Le listing 4.16 décrit l'événement concret `unregister`.

```
event unregister
  any g r
  where
    @typ_g g∈GUEST
    @typ_r r∈ROOM
    @booked_r_g r↦g∈owns
    @empty_r g∉isin(r)
  then
    @booked_r owns ≔ {r}⩤owns
    @update_card_g cards(g) ≔ cards(g)\(KEY×{currk(r)})
end
```

Listing 4.16 – Événement `unregister` de la machine `Hotel_M1`

Obligations de preuve standards et deadlock

Les obligations de preuve standards liées à la machine Event-B `Hotel_M1` ont été déchargées automatiquement ou interactivement par les prouveurs de la plateforme Rodin (voir la figure 4.11). Ceci prouve la correction de la relation de raffinement entre le modèle `Hotel_M1` et le modèle `Hotel_M0`.

En outre, nous avons réussi à démontrer la vivacité du modèle `Hotel_M1` en déchargeant par disjonction de cas et de manière interactive le théorème fourni par le listing 4.17.

```
theorem @DLF_1
(∃r,g,k·r∈ROOM∧g∈GUEST∧k∈KEY∧k∉issued∧r∉dom(owns)) ∨
(∃c,g,r·c∈CARD∧g∈GUEST∧c∈cards(g)∧r∈ROOM∧roomk(r)=prj1(c)∧isin(r)=∅) ∨
(∃r,g·g∈GUEST∧r∈ROOM∧g∈isin(r)) ∨
(∃r,g·r∈ROOM∧g∈GUEST∧r↦g∈owns∧g∉isin(r)) ∨
(∃c,g,r·c∈CARD∧g∈GUEST∧c∈cards(g)∧g∉isin(r)∧r∈ROOM∧roomk(r)=prj2(c))
```

Listing 4.17 – Théorème `DLF_1` lié à l'absence de blocage

Une telle preuve a nécessité l'adjonction de l'axiome `@fin_room` dans le contexte `Hotel_C0` (voir le listing 4.18) et l'axiome `@inf_key` dans le contexte `Hotel_C1` (voir le listing 4.19). Ces contraintes stipulent la finitude de l'ensemble `ROOM` et l'infinitude de l'ensemble `KEY`.

```
axioms
  ...
  @fin_room finite(ROOM)
```

Listing 4.18 – Axiomes ajoutés au contexte Hotel_C0

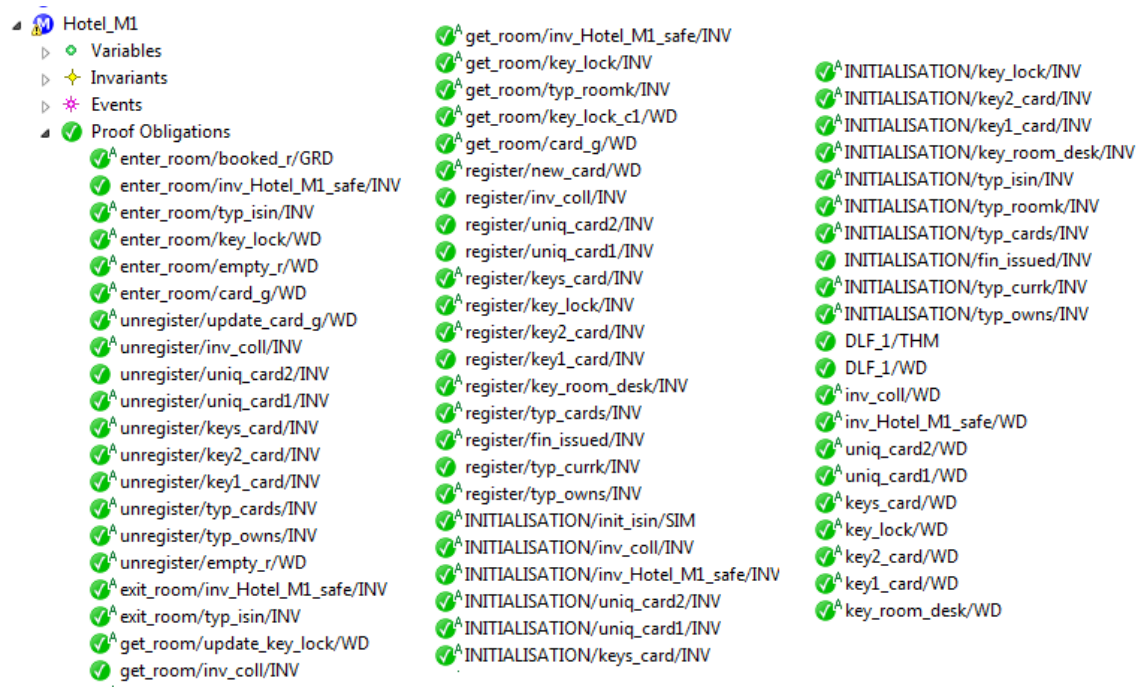

FIGURE 4.11 – Les obligations de preuve déchargées relatives à `Hotel_M1`

```
axioms
  ...
  @inf_key ¬ finite (KEY)
```

Listing 4.19 – Axiome ajouté au contexte Hotel_C1

Validation

Afin de compléter l'activité de preuve supportée par Event-B, nous utilisons l'outil ProB pour animer et valider le modèle formé par le contexte `C1` et la machine `Hotel_M1`. Pour y parvenir, nous proposons les scénarios suivants :

(Scénario 1) "Séjour dans l'hôtel"

L'historique -colonne History- (voir la figure 4.12) traduit un séjour pour le client `GUEST1` qui démarre avec `register` et finira par `unregister` en passant par `get_room`, `enter_room` et `exit_room`. La colonne History exprime les correspondances entre les événements concrets et abstraits. On constate que l'événement `register` n'a pas d'homologue abstrait (voir discussion ci-dessous).

(Scénario 2) "L'hôtel est complet"

A l'instar du modèle abstrait `Hotel_M0`, le modèle raffiné supporte naturellement le scénario exprimant un hôtel est complet en passant par le nouvel événement `register` n'ayant pas du correspondant abstrait (voir la figure 4.13).

(Scénario 3) "Les événements `enter_room` et `exit_room` sont déclenchables **séquentiellement avec répétition**"

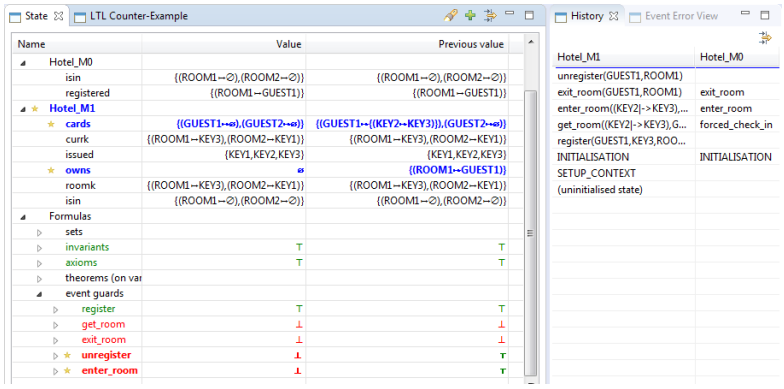

FIGURE 4.12 – Animation de la machine `Hotel_M1` suivant le scénario 1

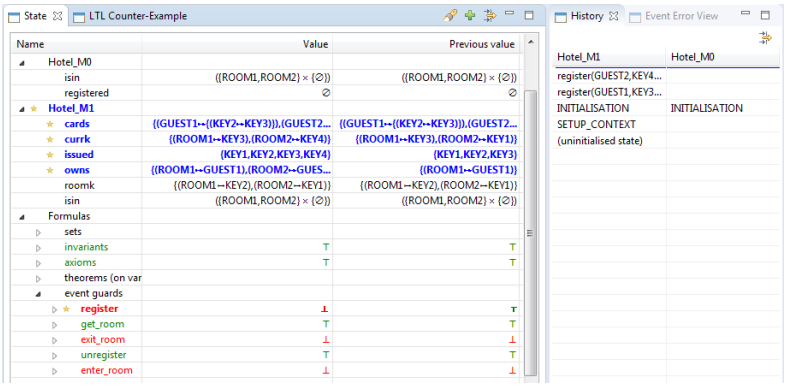

FIGURE 4.13 – Animation de la machine `Hotel_M1` suivant le scénario 2

Ce scénario est à la fois supporté par le modèle `Hotel_M0` et `Hotel_M1` en passant par un nouvel événement ordinaire introduit (voir la figure 4.14).

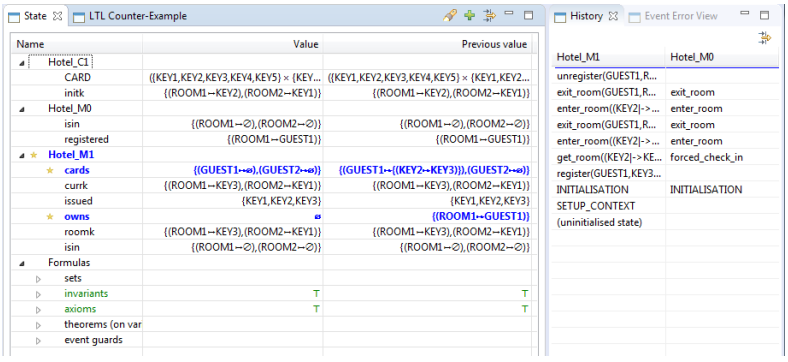

FIGURE 4.14 – Animation de la machine `Hotel_M1` suivant le scénario 3

La machine concrète `Hotel_M1` (voir le listing C.13 dans l'annexe C) présente une spécification centralisée de l'application Hôtel. En effet, son état et ses événements concernent les trois types d'acteurs opérant dans l'application Hôtel à savoir : bureau d'accueil, chambres et clients. Cette spécification centralisée permet de formaliser les interactions entre ces différents acteurs. Mais, elle ne peut pas être implémentée directement sous forme d'un système distribué. En effet, la machine `Hotel_M1` peut être perçue comme un composant composite à

décomposer en suivant une approche descendante de développement.

Discussion

Contrairement au modèle `Hotel_M0`, le modèle `Hotel_M1`, à l'instar des modèles Alloy, OCL, B, Z et VDM décrits dans [45], supporte un seul type de séjour. Pour ce faire, le modèle raffiné `Hotel_M1` introduit deux nouveaux événements ordinaires `register` et `unregister`. Ces deux événements sont appelés à faire partie de l'interface de l'ultime modèle de l'application Hôtel. De plus, pour imposer un seul protocole de séjour, le modèle `Hotel_M1` fait disparaître les événements abstraits : `check_in`, `check_out` et `forced_check_out`. Ainsi, le seul protocole de séjour supporté par `Hotel_M1`, pour un client donné, démarre avec `register` suivi impérativement par `get_room` et éventuellement par zéro ou plusieurs successions (`enter_room`;`exit_room`) et se termine par `unregister`.

L'introduction des nouveaux événements ordinaires et la disparition des événements abstraits caractérisent, entre autres, notre modélisation de l'application Hôtel. Sachant que, la disparition des événements abstraits n'est pas habituel dans un processus de développement basé sur des raffinements successifs en Event-B. En effet, la plateforme Rodin signale une erreur non bloquante : "*Abstract event check_in not refined, although not disabled*".

Dans la suite, nous allons appliquer sur le modèle centralisé `Hotel_M1` les deux étapes de notre démarche de développement des systèmes distribués à savoir Fragmentation et Distribution.

4.5.2 Fragmentation

Dans cette section, nous allons appliquer le schéma de raffinement automatique, appelé Fragmentation, sur le modèle Event-B `Hotel_M1` (voir le listing C.13 dans l'annexe C). Un tel raffinement permet de supprimer les paramètres locaux des événements de synchronisation. Pour y parvenir, le spécifieur est censé décrire l'ordre de calcul linéaire des paramètres à travers un langage dédié (DSL). En se basant sur ce DSL et le modèle Event-B abstrait, l'étape de Fragmentation génère un modèle Event-B qui raffine son correspondant abstrait.

4.5.2.1 Vers une spécification distribuée

En tenant compte de l'exigence architecturale (**ARCH-1**, voir section 4.3), le modèle `Hotel_M1` est décomposé en trois types de composants modélisant respectivement les trois entités bureau d'accueil (`Desk`), chambre (`Room`) et client (`Guest`). La décomposition par événements partagés proposée induit la répartition des variables d'état de la machine `Hotel_M1` sur les composants `Desk`, `Room` et `Guest` conformément à la configuration exprimée par la table 4.1.

Sous-composants	Variables d'état
Desk	owns currk issued
Guest	cards isin
Room	roomk

TABLE 4.1 – Configuration de décomposition de la machine `Hotel_M1`

Une telle configuration va guider le spécifieur pour identifier et fragmenter les événements de synchronisation non déterministes. En outre, elle arrête la distribution voulue des composants sur les sites d'exécution (voir section 4.5.3).

4.5.2.2 Événements de synchronisation non déterministes

Un événement de synchronisation non déterministe est un événement paramétré dont les contraintes sur ces paramètres sont potentiellement évaluées sur plusieurs sites, typiquement via des comparaisons avec des variables allouées sur des composants différents. Pour notre modèle `Hotel_M1` et en tenant compte de la configuration de décomposition (voir table 4.1), les événements de synchronisation non-déterministes sont : `register`, `get_room`, `unregister` et `enter_room`.

Sachant que l'événement `INITIALISATION` est un événement de synchronisation déterministe. Ainsi, il n'est pas concerné par l'étape de Fragmentation. En outre, l'événement `exit_room` n'est pas un événement de synchronisation. Il est local au composant `Guest`.

A titre d'exemple, l'événement `enter_room` (voir le listing 4.14) est un événement de synchronisation puisqu'il admet des variables d'état appartenant à deux sous-composants différents (les variables `cards` et `isin` appartiennent au composant `Guest` et la variable `roomk` est une variable d'état du composant `Room`). Un tel événement est non-déterministe puisqu'il est paramétré par `c`, `r` et `g`.

Une fois ces événements de synchronisation identifiés et afin de réduire voire supprimer le non déterminisme lié au calcul de leurs paramètres locaux, le spécifieur est censé décrire l'ordre de calcul linéaire des paramètres à travers un langage dédié (DSL). En se basant sur cette description et le modèle Event-B abstrait, l'étape de fragmentation génère un nouveau modèle Event-B qui raffine son correspondant abstrait.

4.5.2.3 Spécification de la fragmentation en DSL

Nous illustrons dans le listing 4.20 l'utilisation du DSL de fragmentation pour exprimer l'ordre de calcul des paramètres des événements de synchronisation et les gardes nécessaires à la spécification des opérations introduites. Ces gardes devront par la suite être allouées sur un même composant. Reprenons les événements de synchronisation identifiés :

— **register** : un client `g` et une chambre libre (`@free_r`) sont sélectionnés par l'environnement ; une clé n'ayant jamais été attribuée est choisie. Il faut cependant noter que le calcul de la clé est réalisé par le système en fonction des informations allouées sur le composant `Desk`. Cette opération doit donc terminer. Le paramètre `k` sera donc déclaré `system parameter` afin d'imposer la convergence de l'événement le calculant.

— **get_room** : un client `g` est tout d'abord choisi, puis une carte possédée par ce client. Si le client sélectionné n'a pas de carte, un nouveau client pourra être choisi, l'événement n'étant pas convergent. De manière indépendante, une chambre inoccupée est choisie. Ces choix, réalisés par l'environnement (le composant `Guest`) n'exploitent pas les gardes ne relevant pas du typage et pourront être remis en cause si la carte n'ouvre pas la chambre.

— **unregister** : on choisit de manière indépendante un client et une chambre. Le choix pourra être remis en cause si les conditions de retour de la chambre ne sont pas remplies.

— **enter_room** : on choisit de manière indépendante un client puis, si possible, une carte possédée par ce client, et une chambre. Ces choix pourront être remis en cause si le client ne peut pas utiliser la carte pour entrer dans la chambre.

Notons au passage que l'événement **exit_room** est un événement non déterministe mais n'est pas un événement de synchronisation puisqu'il n'accède qu'à une seule variable d'état (`isin`). Il n'a donc pas besoin d'être fragmenté.

```
dependency Hotel_dep refines Hotel_M1
events
  event register
    parameter g init guest0 with typ_g // select any guest
    system parameter r init room0 with typ_r free_r // select a free room
    system parameter k init key0 with typ_k free_k // build a key
  event get_room
    parameter g init guest0 with typ_g // select a guest
    when g parameter c init card0 with typ_c card_g // select a card owned by g, may
      fail if g has no card
    parameter r init room0 with typ_r empty_r // select an empty room
  event unregister
    parameter g init guest0 with typ_g // select a guest
    parameter r init room0 with typ_r // select a room
  event enter_room
    parameter g init guest0 with typ_g // select a guest
    when g parameter c init card0 with typ_c card_g // select a card owned by g
    parameter r init room0 with typ_r // select a room
end
```

Listing 4.20 – Spécification de la fragmentation du modèle `Hotel_M1`

En résumé, exception faite du paramètre `k` de `register`, les paramètres des différents événements sont extraits et calculés par de nouveaux événements relevant de l'environnement du système. Les choix effectués ne prennent pas en compte les données distantes (allouées aux chambres ou à l'accueil). Ils peuvent donc ne pas permettre le déclenchement de l'événement considéré mais les choix peuvent être remis en cause, ces nouveaux événements n'étant pas déclarés convergents. Il aurait été possible d'éliminer le non déterminisme distribué en allouant les contraintes sur un même composant mais ce choix n'aurait pas été réaliste : le système ne choisit pas le client, le client ne connait pas l'ensemble des clés allouées ou des chambres disponibles.

4.5.2.4 Utilisation du plug-in Fragmentation

En se basant sur DSL de fragmentation (voir le listing 4.20) et le modèle Event-B abstrait `Hotel_M1` (voir le listing C.13), le plug-in de fragmentation génère un modèle Event-B `Hotel_dep` (voir le listing C.14 dans l'annexe C) qui raffine son correspondant abstrait. Cette étape de raffinement automatique s'appuie sur des règles simples assurant l'obtention d'un raffinement correct.

A titre d'exemple, pour l'événement `enter_room`, le plug-in de Fragmentation introduit six variables d'état dans la machine `Hotel_dep`. Plus précisément, pour chaque paramètre de l'événement, l'espace d'état est augmenté avec deux variables : une qui contient la valeur calculée pour le paramètre et l'autre, de type booléen, qui indique si le paramètre a été calculé. Ces variables sont déclarées dans la clause `variables` et sont typées dans la clause `invariants` de la machine raffinée. De plus, l'invariant de la machine est étendu par les propriétés vérifiées par chaque paramètre : si un paramètre a été calculé, sa spécification, donnée par ses gardes, est alors satisfaite. Ceci est également vrai pour les propriétés de dépendance entre les différents paramètres de l'événement concerné (voir le listing 4.21).

```
machine Hotel_dep refines Hotel_M1 sees Hotel_C1
variables
  ...
  // event parameters and computation states
  enter_room_g
  enter_room_g_computed
```

```
  enter_room_c
  enter_room_c_computed
  enter_room_r
  enter_room_r_computed
invariants
  // inherited typing invariants
  ....
  // typing invariants of control variables
  @TY_enter_room_g_computed enter_room_g_computed ∈ ℙ
  @TY_enter_room_c_computed enter_room_c_computed ∈ ℙ
  @TY_enter_room_r_computed enter_room_r_computed ∈ ℙ
  // typing invariants of globalised parameters
  @TY_enter_room_g_typ_g  enter_room_g∈GUEST
  @TY_enter_room_c_typ_c  enter_room_c∈CARD
  @TY_enter_room_r_typ_r  enter_room_r∈ROOM
  // specification of parameter values (guards of inherited events)
  // explicit typing is useless (Event−B type synthesis)
  @enter_room_g_typ_g enter_room_g_computed = TRUE ⇒ ( enter_room_g∈GUEST)
  @enter_room_c_typ_c enter_room_c_computed = TRUE ⇒ ( enter_room_c∈CARD)
  @enter_room_c_card_g enter_room_c_computed = TRUE ⇒ ( enter_room_c∈cards(enter_room_g))
  @enter_room_r_typ_r enter_room_r_computed = TRUE ⇒ ( enter_room_r∈ROOM)
  // dependencies between computation states
  @enter_room_g_c enter_room_g_computed = FALSE ⇒ enter_room_c_computed = FALSE
  ....
```

Listing 4.21 – État de la machine `Hotel_dep` après la fragmentation de l'événement `enter_room`

L'initialisation des différentes variables d'état introduites suite à la Fragmentation de l'événement `enter_room` est assurée par l'adjonction (clause `extends`) des actions déterministes `@enter_room_g_not_computed`, `@enter_room_c_not_computed`, `@enter_room_r_not_computed`, `@enter_room_g_init`, `@enter_room_c_init` et `@enter_room_r_init`.

L'événement `INITIALISATION` est décrit par le listing 4.22. Sachant que les constantes `guest0`, `card0` et `room0` proviennent de la spécification de Fragmentation (voir le listing 4.20).

```
events
  event INITIALISATION extends INITIALISATION
  then
    // initialization of computation states related to enter_room
    @enter_room_g_not_computed enter_room_g_computed := FALSE
    @enter_room_c_not_computed enter_room_c_computed := FALSE
    @enter_room_r_not_computed enter_room_r_computed := FALSE
    // initialization of parameter variables related to enter_room
    @enter_room_g_init enter_room_g := guest0
    @enter_room_c_init enter_room_c := card0
    @enter_room_r_init enter_room_r := room0
    ...
  end
```

Listing 4.22 – Événement `INITIALISATION` de la machine `Hotel_dep` après la fragmentation de l'événement `enter_room`

Le calcul de la valeur de chaque paramètre de l'événement `enter_room` est réalisé par un événement nouvellement introduit dans la machine raffinée. Ces événements sont ordonnés suivant les dépendances des paramètres de l'événement `enter_room` exprimés dans la spécification de fragmentation (DSL). Ainsi, le plug-in de fragmentation génère trois événements ordinaires (statut par défaut `ordinary`) `compute_enter_room_g`, `compute_enter_room_c` et `compute_enter_room_r` qui calculent respectivement les paramètres `g`, `c` et `r` (voir le listing 4.23). Chaque calcul d'un paramètre attend que les paramètres dont il dépend aient été calculés et sauvegarde dans la variable d'état correspondante. Notons que ces événements de calcul sont déclarés ordinaires puisque les paramètres `g`, `c` et `r` sont des paramètres de l'environnement (clause par défaut dans le DSL de spécification de fragmentation voir le

listing 4.20).

```
// computation of enter_room parameters
// computation of g
event compute_enter_room_g
any g
where
  @typ_g  g∈GUEST
then
  @enter_room_g_done enter_room_g_computed := TRUE
  @enter_room_g_value enter_room_g := g
  // reset dependent parameters
  @c_reset enter_room_c_computed := FALSE
end
// computation of c
event compute_enter_room_c
any c
where
  @enter_room_g_computed enter_room_g_computed = TRUE
  @typ_c  c∈CARD
  @card_g  c∈cards(enter_room_g)
then
  @enter_room_c_done enter_room_c_computed := TRUE
  @enter_room_c_value enter_room_c := c
  // reset dependent parameters
end
// computation of r
event compute_enter_room_r
any r
where
  @typ_r  r∈ROOM
then
  @enter_room_r_done enter_room_r_computed := TRUE
  @enter_room_r_value enter_room_r := r
  // reset dependent parameters
end
```

Listing 4.23 – Événements de calcul des paramètres de l'événement `enter_room`

Enfin, la machine raffinée comporte l'événement `enter_room` qui raffine sont homologue abstrait qui n'est franchissable que lorsque tous les paramètres de l'événement abstrait ont été calculés.

Le listing 4.24 décrit l'événement raffiné `enter_room`.

```
event enter_room refines enter_room
when
  // event parameters have been computed
  @enter_room_g_computed enter_room_g_computed = TRUE
  @enter_room_c_computed enter_room_c_computed = TRUE
  @enter_room_r_computed enter_room_r_computed = TRUE
  // guards unused by event parameters
  @empty_r  enter_room_g∉isin(enter_room_r)
  @key_lock  roomk(enter_room_r)=prj2(enter_room_c)
with
  @g g = enter_room_g
  @c c = enter_room_c
  @r r = enter_room_r
then
  @busy_r  isin(enter_room_r):=isin(enter_room_r)∪{
   enter_room_g}
  // reset computation state of current event
  // or of dependent parameters of other events
  @get_room_r_init get_room_r_computed := FALSE
  @enter_room_c_init enter_room_c_computed := FALSE
  @enter_room_r_init enter_room_r_computed := FALSE
  @enter_room_g_init enter_room_g_computed := FALSE
end
```

Listing 4.24 – L'événement raffiné `enter_room`

Le comportement dynamique du raffinement de fragmentation de l'événement `enter_room` est décrit par la figure 4.15.

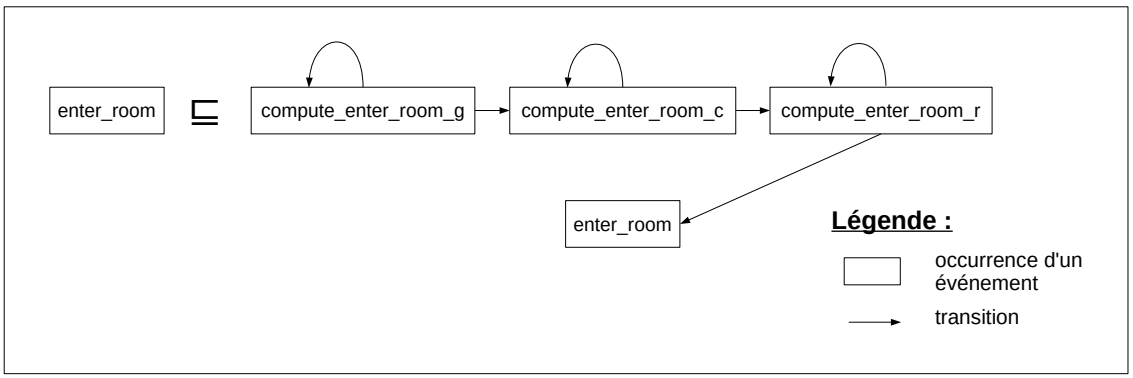

FIGURE 4.15 – Événement abstrait `enter_room` raffiné par fragmentation

4.5.2.5 Discussion

L'exigence architecturale issue du cahier des charges restructuré est une condition sine qua non de la bonne application de l'étape de Fragmentation. En effet, elle permet de répartir les variables d'état du modèle centralisé sur l'architecture à base de composants retenue. Par conséquent, l'identification des événements de synchronisation non déterministes devient un problème facile à résoudre par le spécifieur. Le DSL de Fragmentation proposé possède un pouvoir d'expression permettant de couvrir la plupart des besoins liés à la déterminisation des événements de synchronisation : ordonnancement des paramètres, attachement des propriétés aux paramètres, initialisation des variables destinées à remplacer les paramètres et distinction entre paramètres d'environnement versus paramètres du système. Les règles permettant l'écriture d'une spécification de Fragmentation en utilisant notre DSL sont explicites et systématiques. Elles sont basées essentiellement sur l'analyse des gardes associées à l'événement de synchronisation non déterministe concerné.

4.5.3 Distribution

Dans cette section, nous appliquons l'étape de gestion de la distribution. Elle prend en entrée le modèle issu de l'étape de fragmentation `Hotel_dep` et une spécification de distribution (DSL) pour générer un modèle raffiné `Hotel_split` (voir le listing C.15 dans l'annexe C). Ce modèle raffiné est ensuite distribué sur les différents composants : `Desk` (voir le listing C.16 dans l'annexe C), `Room` (voir le listing 4.26) et `Guest` (voir le listing C.17 dans l'annexe C). Le DSL de distribution est décrit par le spécifieur dont il définit la configuration retenue formée par un ensemble de composants ainsi que la localisation des variables et éventuellement des gardes sur ces composants.

4.5.3.1 Besoins de distribution informels

Nous analysons ici le placement des variables et des traitements sur les trois sites déjà identifiés (l'accueil, la chambre et le client, voir la section 4.5.2.1). Le placement des variables et le calcul des gardes seront explicitement indiqués tandis que les actions seront localisées implicitement sur le site possédant la variable (supposée unique) mise à jour. Après avoir placé les données présentes dans le modèle non fragmenté, nous examinons les variables introduites

par la fragmentation et les gardes. Les paramètres globalisés par la fragmentation seront surlignés.

— **les données initiales** : l'accueil (le composant `Desk`) gère la disponibilité des chambres et l'allocation des clés et reçoit donc les variables `owns` et `currk`. La chambre (le composant `Room`) gère les clés des serrures (`roomk`). Le client (composant `Guest`) gère ses cartes et sa position, donc les variables `cards` et `isin` (voir table 4.1).

— **register** : le composant `Guest` détermine un client `g`, `Desk` cherche une chambre `r` libre et fabrique la clé `k` associée. En conséquence, nous plaçons les variables et gardes comme suit (le préfixe `register_` des paramètres globalisés et remplacé par un surlignage) :

Composants	variables	gardes
Guest	$\overline{g}$ $\overline{g}$_computed	
Desk	$\overline{k}$ $\overline{r}$ $\overline{k}$_computed $\overline{r}$_computed	$\overline{k} \notin$ issued, $\overline{r} \notin$ **dom**(owns)

— **get_room** : un client `g` se trouve devant une chambre libre (`isin(r) = ∅`) et sort une carte. Les trois variables `g`, `r` et `c` sont ainsi localisées sur `Guest`. Le client essaie d'ouvrir : la chambre réalise le test `roomk(r)=prj1(c)` et en cas de réussite met à jour sa clé.

— **exit_room** : cet événement ne concerne que l'environnement et n'a pas été fragmenté. La garde `g ∈ isin(r)` est placée sur `Guest`.

Composants	variables	gardes
Guest		g ∈ isin(r)

— **unregister** : un client `g` désigne une chambre `r` qu'il n'occupe pas (`g ∉ isin(r)`) via le composant `Guest`. Le composant `Desk` vérifie l'appartenance de la chambre (`g ∈ ran(owns)`).

Composants	variables	gardes
Guest	$\overline{g}$ $\overline{g}$_computed $\overline{r}$ $\overline{r}$_computed	$\overline{g} \notin$ isin($\overline{r}$)
Desk		$\overline{r} \mapsto \overline{g} \in$ owns

— **enter_room** : un client `g` est sélectionné, puis une carte `c` possédée par ce client. Le composant `Guest` effectue le test correspondant (`c ∈ cards(g)`). Une chambre est ensuite choisie. La carte est vérifiée par la serrure (`roomk(r)=prj2(c)`) sur le composant `Room`, d'où les placements suivants :

Composants	variables	gardes
Guest	$\overline{g}$ $\overline{g}$_computed $\overline{r}$ $\overline{r}$_computed $\overline{c}$ $\overline{c}$_computed	$\overline{c} \in$ cards($\overline{g}$)
Room		roomk($\overline{r}$) = prj2($\overline{c}$)

4.5.3.2 Besoins de distribution en DSL

Tous les besoins informels de distribution du modèle `Hotel_dep` ont été formalisés par le DSL décrit par le listing 4.25.

Notons que tous les paramètres globalisés (c'est-à-dire remplacés par des variables d'état) sont placés naturellement, soit sur le composant `Desk`, soit sur le composant `Guest`. En effet, les événements de synchronisation du composant `Room` (`get_room` et `enter_room` voir le listing 4.26) exigent des paramètres d'entrée venant de son environnement (ici le composant `Guest`). Le placement explicite de deux grades `k_card` et `booked_r_g` respectivement sur les deux composants `Guest` et `Desk` est justifié ci-dessous.

```
shared event decomposition Hotel_split refines Hotel_dep
components
  Desk Guest Room
mappings
  variables owns currk issued ↦ Desk;
  variables cards isin ↦ Guest;
  variable roomk ↦ Room;
  variables register_k register_k_computed ↦ Desk;
  variables register_g register_g_computed ↦ Guest;
  variables register_r register_r_computed ↦ Desk;
  variables get_room_g get_room_g_computed ↦ Guest;
  variables get_room_c get_room_c_computed ↦ Guest;
  variables getRoom_r getRoom_r_computed ↦ Guest;
  variables unregister_g unregister_g_computed ↦ Guest;
  variables unregister_r unregister_r_computed ↦ Guest;
  variables enter_room_g enter_room_g_computed ↦ Guest;
  variables enter_room_c enter_room_c_computed ↦ Guest;
  variables enter_room_r enter_room_r_computed ↦ Guest;
  guard register ·k_card ↦ Guest;
  guard unregister ·booked_r_g ↦ Desk
end
```

Listing 4.25 – Spécification de la distribution du modèle `Hotel_dep`

4.5.3.3 Composants Event-B générés

La correction de l'étape de distribution repose sur la vérification de la correction des projections. Quelques essais ont été nécessaires pour d'une part corriger les mappings automatiques des gardes et d'autre part déterminer les propriétés nécessaires à la vérification des invariants projetés, et notamment des propriétés `uniq_card` de la machine `Guest`.

Le mapping automatique des gardes est guidé par le mapping des variables globales qu'elles contiennent. Si aucune variable globale n'est utilisée, la garde est projetée sur tous les composants. Si plusieurs composants sont référencés, l'un d'eux est choisi de manière arbitraire. Afin de corriger ce choix par défaut qui est contre-intuitif dans le cas ci-dessous, nous associons explicitement la garde de l'événement `unregister`

$$@\text{booked\_r\_g unregister\_r} \mapsto \text{unregister\_g} \in \text{owns}$$

au composant `Desk` propriétaire de la variable `owns` qui n'est a priori pas exportable. Les deux autres paramètres (`r` et `g`) seront donc copiés sur le composant `Desk` afin que le test puisse être réalisé. Le traitement automatique avait arbitrairement choisi le composant `Guest`, ce qui impliquait la copie de `owns` sur `Guest`, ce qui est peu réaliste.

L'intervention la plus délicate concerne l'identification des propriétés nécessaires à la vérification de la préservation des invariants. Il s'agissait en particulier de montrer la préservation de la propriété ci-dessous par l'ajout d'une nouvelle carte dans l'ensemble `cards` consécutif à l'exécution de l'événement `register` projeté dans `Guest` :

$$\forall g1, g2 \cdot (\text{cards}(g1) \cap \text{cards}(g2) \neq \emptyset \Rightarrow g1 = g2)$$

Il fallait pour cela exploiter le fait que la nouvelle carte n'utilisait pas de clés déjà distribuées, propriété exprimée de manière indirecte via la variable `issued` dans le modèle

centralisé. Ces informations sont perdues dans `Guest` puisque `issued` se situe sur `Desk`. Il a donc été nécessaire d'ajouter la propriété dérivée `k_card` (étiquetée `theorem`) aux gardes de l'événement `register` du modèle centralisé. Cette propriété est préservée par la projection puisqu'elle n'accède qu'à la variable `cards` locale à `Guest`.

theorem @k_card $k \notin \text{ran}(\text{union}(\text{ran}(\text{cards})))$

On obtient alors les composants ci-dessous vérifiés par Rodin. Ces composants générés peuvent comporter des variables d'état, propriétés et événements provenant du modèle centralisé et de deux plug-ins Fragmentation et Distribution. En outre, ils peuvent englober des propriétés ajoutées par le modélisateur pour des raisons de preuve.

Le plug-in Distribution appliqué sur le modèle `Hotel_dep` (voir le listing C.14 dans annexe C) et la spécification de la distribution du modèle `Hotel_dep` (voir le listing 4.25) génère trois composants Event-B : `Room` (voir le listing 4.26), `Desk` (voir le listing C.16 dans l'annexe C) et `Guest` (voir le listing C.17 dans l'annexe C).

La machine Event-B `Room` (voir le listing 4.26) modélise les chambres de l'application Hôtel. Elle comporte une variable `roomk` venant du modèle centralisé `Hotel_M1` (voir le listing C.13 dans l'annexe C). Les autres variables dont les noms sont préfixés par `Room_` proviennent du plug-in Distribution. Elles sont calculées par des événements issus de Distribution dont les noms sont préfixés par `share_`. Les deux événements `get_room` et `enter_room` figurent initialement dans le modèle centralisé `Hotel_M1`.

```
machine Room sees Hotel_C1

variables  roomk Room_enter_room_c Room_enter_room_r Room_get_room_r
      Room_get_room_c

invariants
  @typ_roomk roomk∈ROOM→KEY
  @TY_Room_enter_room_c Room_enter_room_c∈CARD
  @TY_Room_enter_room_r Room_enter_room_r∈ROOM
  @TY_Room_get_room_r Room_get_room_r∈ROOM
  @TY_Room_get_room_c Room_get_room_c∈CARD

events
  event share_enter_room_c
    any local_enter_room_c
    where
      @TY_local_enter_room_c local_enter_room_c∈CARD
    then
      @to_Room Room_enter_room_c := local_enter_room_c
  end

  event share_enter_room_r
    any local_enter_room_r
    where
      @TY_local_enter_room_r local_enter_room_r∈ROOM
    then
      @to_Room Room_enter_room_r := local_enter_room_r
  end

  event share_get_room_r
    any local_get_room_r
    where
      @TY_local_get_room_r local_get_room_r∈ROOM
    then
      @to_Room Room_get_room_r := local_get_room_r
  end

  event share_get_room_c
    any local_get_room_c
    where
      @TY_local_get_room_c local_get_room_c∈CARD
    then
```

```
    @to_Room Room_get_room_c := local_get_room_c
  end

  event INITIALISATION
    then
      @init_roomk roomk := initk
      @Room_enter_room_c_init Room_enter_room_c :∈ CARD
      @Room_enter_room_r_init Room_enter_room_r :∈ ROOM
      @Room_get_room_r_init Room_get_room_r :∈ ROOM
      @Room_get_room_c_init Room_get_room_c :∈ CARD
  end

  event get_room
    where
      @key_lock_c1 roomk(Room_get_room_r)=prj1(Room_get_room_c)
    then
      @update_key_lock roomk(Room_get_room_r) := prj2(Room_get_room_c)
  end

  event enter_room
    where
      @key_lock roomk(Room_enter_room_r)=prj2(Room_enter_room_c)
  end
end
```

Listing 4.26 – Composant Event-B `Room` généré

L'événement `get_room` met à jour `roomk` en tenant compte de deux variables `Room_get_room_r` et `Room_get_room_c`. Tandis que l'événement `enter_room` consulte `roomk` via `Room_enter_room_r` et `Room_enter_room_c`. Ces quatre variables nécessaires à la consultation et la mise à jour de la variable `roomk` sont déterminées par l'environnement du composant `Room` à savoir le composant `Guest`.

Les obligations de preuve associées au composant Event-B `Room` sont déchargées par la plateforme Rodin (voir la figure 4.16).

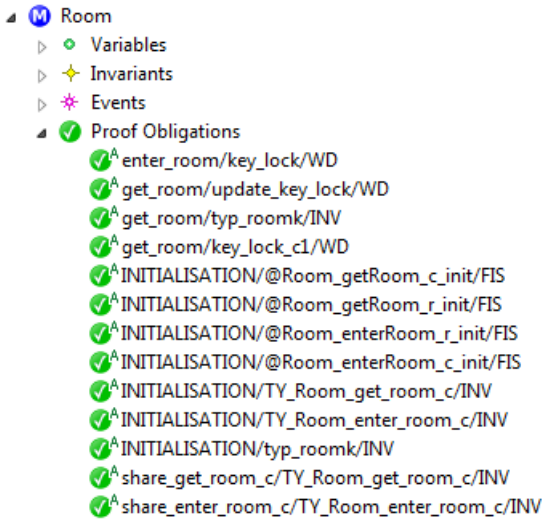

FIGURE 4.16 – Les obligations de preuve relatives au composant Event-B `Room`

En plus des machines `Room`, `Desk` et `Guest`, le plug-in Distribution génère le modèle `Hotel_split` (voir le listing C.15 dans l'annexe C) qui raffine le modèle `Hotel_dep`. Un tel raffinement permet d'introduire des copies des variables distantes lues par les gardes, des événements mettant à jour ces variables, et des paramètres recevant les valeurs des variables distantes lues par les actions. Notons que chaque élément de modélisation de la machine

`Hotel_split` est annoté par le composants concerné.

A titre d'exemple, l'événement `enter_room` (voir le listing 4.27) est extrait de la machine `Hotel_split`. Une telle machine est générée automatiquement par le plug-in Distribution et plus précisément par la phase de pré-traitement. Chaque garde et action de l'événement est annoté par le composant qui la définit (`Guest` ou `Room`). En effet, l'événement `enter_room` de la machine `Room` (voir le listing 4.26) ne comporte que la garde `@key_lock`.

```
event enter_room refines enter_room
when
  // access to remote variables used in actions
  // access to copies of remote variables
  @enter_room_r_fresh enter_room_r_fresh = TRUE // on Guest
  @enter_room_c_fresh enter_room_c_fresh = TRUE // on Guest
  // inherited guards
  @enter_room_g_computed  enter_room_g_computed = TRUE // on Guest
  @enter_room_c_computed  enter_room_c_computed = TRUE // on Guest
  @enter_room_r_computed  enter_room_r_computed = TRUE // on Guest
  @empty_r   enter_room_g∉isin(enter_room_r) // on Guest
  @key_lock   roomk(Room_enter_room_r)=prj2(Room_enter_room_c) // on
     Room
  then
  @busy_r isin(enter_room_r) := isin(enter_room_r)∪{enter_room_g} // on
     Guest
  @get_room_r_init get_room_r_computed :=  FALSE // on Guest
  @enter_room_c_init enter_room_c_computed :=  FALSE // on Guest
  @enter_room_r_init enter_room_r_computed :=  FALSE // on Guest
  @enter_room_g_init enter_room_g_computed :=  FALSE // on Guest
  // reset freshness state of copies of updated variables
end
```

Listing 4.27 – Événement `enter_room` avant la distribution

4.5.3.4 Discussion

L'utilisation du plug-in Distribution exige l'expression des mappings des éléments de modélisation provenant du modèle centralisé fragmenté sur les composants retenus. Ces éléments de modélisation englobent les variables d'état du modèle centralisé, les variables d'état des paramètres globalisés et les gardes des événements de synchronisation. Sachant que les actions sont placées automatiquement sur les composants retenus par le plug-in Distribution.

Le placement des variables d'état du modèle centralisé est suggéré de l'architecture à base de composants choisie. En outre, le placement des variables d'état des paramètres globalisés peut être déduit logiquement de celui des variables d'état du modèle centralisé.

Le plug-in Distribution propose un mécanisme systématique de placement des gardes. Mais, le spécifieur peut agir (construction guard) sur ce comportement par défaut en explicitant ses choix. Ceci peut être motivé par des raisons d'efficacité, environnementales, de masquage d'information ou de preuve (voir le listing 4.25).

Les composants Event-B générés par notre plug-in Distribution sont bien documentés. Ils englobent des éléments de modélisation provenant respectivement du modèle centralisé, de l'étape de Fragmentation et de l'étape de Distribution. La convention de nommage de ces éléments de modélisation est univoque. Ceci va faciliter l'enrichissement des ces composants.

4.6 Génération de Code BIP

Le code BIP généré devrait comporter des aspects architecturaux et comportementaux de l'application traitée : ici l'application Hôtel.

En ce qui concerne les aspects architecturaux, notre plug-in Génération de code produit avec succès l'architecture de l'application traitée : composants, portes et connecteurs. Ainsi,

l'architecture de l'application Hôtel générée par notre plug-in est fournie par le listing 4.28.

```
package root

/* portes */
port type ty_empty_port()
 ...

/* connecteurs */
connector type ty_get_room (ty_empty_port Guest,ty_empty_port Room)
  define Guest Room
  on Guest Room
  down { }
end

   ...

/* composants atomiques */
atom type ty_Desk()
   ...
end

atom type ty_Guest()
   ...
end

atom type ty_Room()
   ...
end

/* composant composite */
compound type ty_Hotel_split()
  component ty_Desk Desk()
  component ty_Guest Guest()
  component ty_Room Room()

  connector ty_get_room get_room(Guest.get_room,Room.get_room)
   ...
end
end /* package */
```

Listing 4.28 – Architecture du code BIP générée

En ce qui concerne les aspects comportementaux, le sous-ensemble d'Event-B retenu (dit Event-B0) permet une réalisation en BIP via le langage hôte C++ en s'appuyant sur des librairies génériques. Par exemple, le listing 4.29 regroupe les signatures des fonctions et des opérations offertes par Event-B0 en BIP. Sachant que `SET`, `PAIR` et `DATA` sont considérés comme types formels génériques.

```
// sets of machine context
extern data type GUEST as "int"
extern operator bool ==(GUEST, GUEST)
extern data type ROOM as "int"
extern operator bool ==(ROOM, ROOM)
extern data type KEY as "int"
extern operator bool ==(KEY, KEY)

// defined sets
extern data type CARD as "std::pair<KEY,KEY>"
extern function KEY prj1(CARD)
extern function KEY prj2(CARD)

extern data type MAP_ROOM_KEY as "std::map<ROOM,KEY>"
extern function KEY get(MAP_ROOM_KEY,ROOM)
extern function put(MAP_ROOM_KEY,ROOM,KEY)
```

Listing 4.29 – Signatures en BIP des surcharges des fonctions et des opérations générées

Ainsi, un composant atomique BIP peut réutiliser ces librairies génériques au niveau des transitions (gardes et actions) en invoquant les opérations souhaitées. Par exemple, le listing 4.30 montre un extrait du composant `ty_Room` qui réutilise les services génériques : `get`, `put`, `prj1`, `prj2` et `==`.

```
...
on get_room from P0 to P0
  provided ( (get(roomk,Room_get_room_r) == prj1(Room_get_room_c)) )
  do { put(roomk, Room_get_room_r, prj2(Room_get_room_c)); }

on enter_room from P0 to P0
  provided ( (get(roomk,Room_enter_room_r) == prj2(Room_enter_room_c)) )
...
```

Listing 4.30 – Extrait de code du composant `ty_Room`

Pour les aspects comportementaux, il reste un problème important à résoudre : la déterminisation des événements non déterministes internes par opposition aux événements de synchronisation non déterministes (voir section 4.5.2.2). Par raffinements successifs avec preuves mathématiques, de tels événements peuvent être concrétisés en utilisant un processus à la B. Pour y parvenir, tout d'abord, nous traduisons tous les éléments liés à un événement dans une machine abstraite B (construction `MACHINE`). Celle-ci est sans état et comporte une seule opération (`OPERATIONS`) correspondant à l'événement à déterminiser. Ensuite, nous appliquons le processus B allant de `MACHINE` vers `IMPLEMENTATION`. Enfin, nous obtenons un sous-programme C correct par construction qui peut être appelé depuis un code BIP.

4.7 Conclusion

Dans ce chapitre, nous avons montré la faisabilité de la démarche de développement des systèmes distribués introduite dans le chapitre précédent. En effet, nous avons appliqué avec succès notre démarche sur l'application Hôtel. Tout d'abord, nous avons établi un modèle centralisé en Event-B cohérent et valide en adoptant des raffinements horizontaux manuels. La cohérence et la validation du modèle centralisé sont vérifiées formellement à l'aide des prouveurs et de l'outil ProB de la plateforme Rodin. Ensuite, les exigences architecturales issues du cahier des charges restructuré de l'application Hôtel ont guidé le spécifieur afin d'élaborer la spécification DSL de l'étape de Fragmentation. Celle-ci est considérée comme une étape de raffinement automatique dirigée par le modélisateur. Les difficultés liées à l'application de l'étape Fragmentation sont maîtrisables moyennant la distinction des paramètres d'environnement par opposition aux paramètres du système et l'analyse des gardes exprimant les dépendances inter-paramètres d'un événement donné. De plus, l'étape de Distribution exige les placements des variables et gardes sur les sites formant l'architecture distribuée choisie. De tels placements sont plus ou moins faciles à faire par le modélisateur notamment celui des variables. La difficulté potentielle inhérente à l'étape de Distribution concerne la correction des composants Event-B générés par notre plug-in Distribution. En effet, le déchargement des OP associées à ces composants nécessite, parfois, l'adjonction de lemmes et théorèmes par le modélisateur pour combler les propriétés communes écartées. Ceci est dû au problème important de la distribution de l'invariant d'un modèle centralisé Event-B sur plusieurs composants Event-B [14]. Enfin, notre démarche produit un code BIP qui réutilise des librairie génériques C++. Celles-ci devraient implémenter les constructions Event-B retenues (Event-B0).

Chapitre 5

Conclusion et Perspectives

5.1 Bilan

Composition/Décomposition

En génie logiciel, les deux techniques de composition et de décomposition favorisent l'obtention de logiciels et systèmes de qualité. Elles sont applicables sur des entités de type spécification et également sur des entités de type implémentation. La composition des entités permet d'enrichir et par conséquent de changer le comportement du système global obtenu. La décomposition d'un système en plusieurs entités de tailles plus petites ne vise pas le changement du comportement du système à décomposer. Elle permet plutôt de changer la manière de préserver le comportement du système via ses entités. Dans ce travail, nous nous sommes intéressés aux entités de type spécification représentées par des modèles Event-B. Nous avons étudié le problème de décomposition des spécifications Event-B afin d'obtenir à terme des systèmes distribués sûrs et exécutables sur divers plateformes.

Démarche de développement des systèmes distribués

Dans cette thèse, nous avons établi, outillé et expérimenté une démarche de développement des systèmes distribués. Celle-ci comporte quatre étapes : Spécification, Fragmentation, Distribution et Génération de code. Techniquement, la démarche préconisée s'appuie sur la méthode formelle Event-B et le langage BIP de modélisation et programmation des applications à base de composants. L'étape de Spécification a pour objectif l'élaboration d'une spécification centralisée en utilisant un processus de développement formel basé sur Event-B. La Fragmentation et la Distribution sont deux concepts clefs, considérés comme deux sortes de raffinement automatique Event-B paramétrées à l'aide de deux DSL appropriés, introduits par cette thèse. Elles permettent la décomposition formelle d'une spécification centralisée Event-B en plusieurs composants Event-B traduisibles systématiquement en BIP comme système distribué grâce à l'étape Génération de code.

Boîte à outils

Nous avons conçu, réalisé et testé trois prototypes permettant d'outiller notre démarche de développement des systèmes distribués combinant Event-B et BIP. Pour y parvenir, nous avons utilisé avec profit les deux technologies IDM complémentaires Xtext et Xtend. Le premier prototype Fragmentation prend en entrée une spécification centralisée Event-B et une description DSL des événements de synchronisation à déterminiser et produit en sortie une

spécification Event-B fragmentée comportant des événements de synchronisation déterminisés c'est-à-dire sans paramètres locaux. Le second prototype Distribution prend en entrée une spécification Event-B fragmentée et une description DSL de la configuration architecturale retenue et produit en sortie un modèle Event-B distribué ayant plusieurs composants Event-B. Ces composants communiquent par événements partagés. Enfin, le dernier prototype Génération de code prend en entrée un modèle Event-B distribué et produit en sortie un programme BIP exécutable reposant sur des librairies génériques C++ implantant la théorie des ensembles d'Event-B.

Expérimentation

Nous avons expérimenté notre démarche outillée de développement des systèmes distribués sur l'application Hôtel permettant de gérer des hôtels ayant des chambres à clefs électroniques. Une telle expérimentation nous a permis de dégager des pistes ou heuristiques favorisant la bonne application notamment de deux étapes Fragmentation et Distribution : identification des exigences dédiées pour la répartition, identification des événements de synchronisation, distribution des variables, distribution des gardes, distribution de l'invariant et démonstration des OP des composants Event-B issus de l'étape Distribution.

5.2 Perspectives

Nous pourrions envisager les prolongements suivants :

Le premier prolongement concernerait l'amélioration des prototypes développés dans le cadre de cette thèse. Pour le prototype Fragmentation, on pourrait réfléchir sur la génération automatique du DSL Fragmentation à partir du modèle Event-B centralisé à fragmenter. Pour le prototype Génération de code BIP, on pourrait soigner davantage la modularité et l'efficacité des librairies génériques C++ implémentant le sous-ensemble d'Event-B retenu (Event-B0).

Le second prolongement aurait pour objectif d'apporter une démarche pour la concrétisation des événements non déterministes internes. Nous pensions passer par la méthode B. Mais l'extraction de la machine abstraite B (construction MACHINE) à partir du modèle Event-B contenant l'événement à concrétiser devrait être maîtrisée.

L'activité de décomposition formelle nécessite la distribution de l'invariant du modèle centralisé Event-B à décomposer. Il s'agit d'un problème fondamental et peu abordé [14, 72]. Notre prototype Distribution apporte une solution à ce problème. Mais, celle-ci ne garantit pas la correction des OP des composants Event-B. Comme troisième prolongement, il serait intéressant d'étudier davantage ce problème.

Actuellement, nous travaillons dans deux directions : la modélisation des aspects temps réel et temporels en Event-B [68] et la formalisation et la vérification de la technique de raffinement Event-B en servant d'Event-B elle-même [20, 21].

Bibliographie

[1] The Common Criteria Portal. http://www.commoncriteriaportal.org/cc/. 1

[2] Java 10, Today ! http://www.eclipse.org/xtend/. Acc : 16-01-06. 73

[3] Language Engineering For Everyone ! https://eclipse.org/Xtext. Acc : 16-01-06. 73

[4] Rigorous Design of Component-Based Systems - The BIP Component Framework. http://www-verimag.imag.fr/Rigorous-Design-of-Component-Based.html?lang=en. xi, 13, 19

[5] SMT Solvers Plug-in. http://wiki.event-b.org/index.php/SMT_Solvers_Plug-in. 39

[6] *BIP2 Documentation, Release 2015.04 (RC7)*. VERIMAG, Grenoble, France, April 30, 2015. 16, 17, 18, 19

[7] Jean-Raymond Abrial. *The B-book : Assigning Programs to Meanings*. Cambridge University Press, New York, NY, USA, 1996. 19, 68

[8] Jean-Raymond Abrial. *Modeling in Event-B : System and Software Engineering*. Cambridge University Press, New York, NY, USA, 1st edition, 2010. xi, 1, 4, 5, 19, 26, 27, 28, 39, 42, 77, 78

[9] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, and Laurent Voisin. *An Open Extensible Tool Environment for Event-B*, pages 588–605. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006. 4, 25, 27

[10] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, and Laurent Voisin. *A Roadmap for the Rodin Toolset*, pages 347–347. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. 2, 4, 25, 27

[11] Jean-Raymond Abrial and Stefan Hallerstede. Refinement, Decomposition, and Instantiation of Discrete Models : Application to Event-B. *Fundam. Inf.*, 77(1-2) :1–28, January 2007. 19, 31, 39

[12] Robert Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon, School of Computer Science, January 1997. Issued as CMU Technical Report CMU-CS-97-144. 5, 18

[13] Ralph-Johan Back and Kaisa Sere. Superposition Refinement of Reactive Systems. *Formal Asp. Comput.*, 8(3) :324–346, 1996. 25

[14] Richard Banach. *Invariant Guided System Decomposition*, pages 271–276. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014. 62, 107, 110

[15] Ananda Basu. *Component-based Modeling of Heterogeneous Real-time Systems in BIP*. Theses, Université Joseph-Fourier - Grenoble I, December 2008. 4, 6

[16] Ananda Basu, Saddek Bensalem, Marius Bozga, Jacques Combaz, Mohamad Jaber, Thanh-Hung Nguyen, and Joseph Sifakis. Rigorous Component-Based System Design Using the BIP Framework. *IEEE Software*, 28(3) :41–48, 2011. 1

[17] Saddek Bensalem, Marius Bozga, Thanh-Hung Nguyen, and Joseph Sifakis. *D-Finder : A Tool for Compositional Deadlock Detection and Verification*, pages 614–619. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. 12, 19

[18] Saddek Bensalem, Andreas Griesmayer, Axel Legay, Thanh-Hung Nguyen, Joseph Sifakis, and Rongjie Yan. *D-Finder 2 : Towards Efficient Correctness of Incremental Design*, pages 453–458. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. 12, 19

[19] Gérard Berry. Cours et Travaux du Collège de France. *Annuaire 108e année, Collège de France, Paris*, pages 821–846, décembre 2008. xi, 24

[20] Jean-Paul Bodeveix, Mamoun Filali, Mohamed Tahar Bhiri, and Badr Siala. An Event-B Framework for the Validation of Event-B Refinement Plugins. *CoRR*, abs/1701.00960, 2017. 110

[21] Jean-Paul Bodeveix, Mamoun Filali, Badr Siala, and Mohamed Tahar Bhiri. Towards an Event-B framework for Event-B Transformation Verification. In Dominique MERY Yamine AIT AMEUR, Shin NAKAJIMA, editor, *Implicit and explicit semantics integration in formal developments of discrete systems – Communications of NII Shonan Meetings*. Shonan publishing, 2018. **(Submitted)**. 110

[22] Borzoo Bonakdarpour, Marius Bozga, Mohamad Jaber, Jean Quilbeuf, and Joseph Sifakis. From High-level Component-based Models to Distributed Implementations. In *Proceedings of the Tenth ACM International Conference on Embedded Software*, EMSOFT '10, pages 209–218, New York, NY, USA, 2010. ACM. 12, 13

[23] Marius Bozga, Mohamad Jaber, and Joseph Sifakis. Source-to-Source Architecture Transformation for Performance Optimization in BIP. *IEEE Trans. Industrial Informatics*, 6(4) :708–718, 2010. 9, 10

[24] Michael Butler. An Approach to the Design of Distributed Systems with B AMN. In *ZUM '97 : The Z Formal Specification Notation, 10th International Conference of Z Users, Reading, UK, April 3-4, 1997, Proceedings*, pages 223–241, 1997. 19, 31, 39

[25] Michael Butler. Synchronisation-based Decomposition for Event B. In *RODIN Deliverable D19 Intermediate report on methodology*, 2006. 19, 39

[26] Michael Butler. *Decomposition Structures for Event-B*, pages 20–38. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. 4

[27] Clearsy. BART (B Automatic Refinement Tool). http://tools.clearsy.com/wp-content/ uploads/sites/8/resources/BART_GUI_User_Manual.pdf. 26

[28] David Déharbe, Pascal Fontaine, Yoann Guyot, and Laurent Voisin. *SMT Solvers for Rodin*, pages 194–207. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. 39

[29] Nouha Derbel. Construction Formelle de Programmes Séquentiels et Concurrents. Master's thesis, Faculté des Sciences de Sfax, Tunisie, Décembre 2013. 78

[30] John Derrick, John Fitzgerald, Stefania Gnesi, Sarfraz Khurshid, Michael Leuschel, Steve Reeves, and Elvinia Riccobene, editors. *Proceedings of the 3rd International Conference on Abstract State Machines, Alloy, B, VDM, and Z (ABZ2012)*, volume 7316 of *Lecture Notes in Computer Science*, Pisa, Italy, June 2012. Springer-Verlag. http://dx.doi.org/10.1007/978-3-642-30885-7. 62

[31] Andrew Edmunds and Michael Butler. Tasking Event-B : An Extension to Event-B for Generating Concurrent Code. Event Dates : 2nd April 2011, February 2011. 34

[32] Andrew Edmunds, Michael Butler, Issam Maamria, Renato Silva, and Chris Lovell. Event-B Code Generation : Type Extension with Theories. In *ABZ proceedings*, pages 365–368, 2012. 73

[33] Andrew Edmunds, Abdolbaghi Rezazadeh, and Michael Butler. Formal Modelling for Ada Implementations : Tasking Event-B. June 2012. 34

[34] Yliès Falcone, Mohamad Jaber, Thanh-Hung Nguyen, Marius Bozga, and Saddek Bensalem. *Runtime Verification of Component-Based Systems*, pages 204–220. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. 12

[35] Yliès Falcone, Mohamad Jaber, Thanh-Hung Nguyen, Marius Bozga, and Saddek Bensalem. Runtime Verification of Component-based Systems in the BIP Framework with Formally-proved Sound and Complete Instrumentation. *Software and System Modeling*, 14(1) :173–199, 2015. 12

[36] Asieh Salehi Fathabadi, Abdolbaghi Rezazadeh, and Michael Butler. Applying Atomicity and Model Decomposition to a Space Craft System in Event-B. In *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*, pages 328–342, 2011. 19, 39

[37] Andreas Fürst, Thai Son Hoang, David Basin, Naoto Sato, and Kunihiko Miyazaki. Formal System Modelling Using Abstract Data Types in Event-B. In *ABZ proceedings*, volume 8477 of *LNCS*, pages 222–237. Springer, 2014. 28

[38] David Garlan and Mary Shaw. An Introduction to Software Architecture. In V Ambriola and G Tortora, editors, *Advances in Software Engineering and Knowledge Engineering*, volume 2 of *Series on Software Engineering and Knowledge Engineering*. World Scientific Publishing Company, 1995. 4

[39] Stefan Hallerstede, Michael Leuschel, and Daniel Plagge. Refinement-Animation for Event-B - Towards a Method of Validation. In *Proceedings ABZ'2010*, volume 5977 of *Lecture Notes in Computer Science*, pages 287–301. Springer-Verlag, 2010. 27, 28

[40] Fei He, Liangze Yin, Bow-Yaw Wang, Lianyi Zhang, Guanyu Mu, and Wenrui Meng. *VCS : A Verifier for Component-Based Systems*, pages 478–481. Springer International Publishing, Cham, 2013. 12

[41] Thai Son Hoang and Jean-Raymond Abrial. *Reasoning about Liveness Properties in Event-B*, pages 456–471. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. 27

[42] Thai Son Hoang, Andreas Fürst, and Jean-Raymond Abrial. Event-B Patterns and Their Tool Support. *Software & Systems Modeling*, 12(2) :229–244, 2013. 26

[43] Charles A. R. Hoare. Communicating sequential processes. *ACM Trans. Program. Lang. Syst.*, 15(5) :795–825, November 1985. 29, 30

[44] Mohamad Jaber. *Centralized and Distributed Implementations of Correct-by-construction Component-based Systems by using Source-to-source Transformations in BIP*. Theses, Université Joseph-Fourier - Grenoble I, October 2010. 13

[45] Daniel Jackson. *Software Abstractions : Logic, Language, and Analysis*. The MIT Press, 2006. 2, 4, 5, 39, 77, 78, 87, 88, 95

[46] Leslie Lamport. The Temporal Logic of Actions. *ACM Trans. Program. Lang. Syst.*, 16(3) :872–923, May 1994. 27

[47] Thierry Lecomte. Return of Experience on Automating Refinement in B. *1st International Workshop about Sets and Tools (SETS 2014), Toulouse (France)*, June 3 2014. 26

[48] Thierry Lecomte, Dominique Méry, and Dominique Cansell. Patrons de Conception Prouvés. *Génie Logiciel - Magazine de l'ingéniérie du logiciel et des systèmes*, (81 (juin 2007)) :14–18, 2007. 26

[49] Michael Leuschel and Michael Butler. *ProB : A Model Checker for B*, pages 855–874. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003. 27, 28

[50] Radu Mateescu and Mihaela Sighireanu. Efficient On-the-fly Model-Checking for Regular Alternation-free Mu-Calculus. *Sci. Comput. Program.*, 46(3) :255–281, 2003. 12

[51] Radu Mateescu and Damien Thivolle. A Model Checking Language for Concurrent Value-Passing Systems. In *FM 2008 : Formal Methods, 15th International Symposium on Formal Methods, Turku, Finland, May 26-30, 2008, Proceedings*, pages 148–164, 2008. 12

[52] Dominique Méry and Neeraj Kumar Singh. Automatic Code Generation from event-B Models. In *Proceedings of the Second Symposium on Information and Communication Technology*, SoICT '11, pages 179–188, New York, NY, USA, 2011. ACM. 33

[53] Christophe Métayer. AnimB HomePage. http://www.animb.org/index.xml. 28

[54] Robert Monroe. Capturing Software Architecture Design Expertise with Armani. Technical Report CMU-CS-98-163, Carnegie-Mellon University.Computer science. Pittsburgh (PA US), Pittsburgh, 1998. 18

[55] Tobias Nipkow. Verifying a Hotel Key Card System. In *Theoretical Aspects of Computing (ICTAC 2006)*, volume 4281, pages 1–14, 2006. 2

[56] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL : A Proof Assistant for Higher-order Logic*. Springer-Verlag, Berlin, Heidelberg, 2002. 78

[57] Sergey Ostroumov and Leonidas Tsiopoulos. VHDL Code Generation from Formal Event-B Models. In *14th Euromicro Conference on Digital System Design, Architectures, Methods and Tools, DSD 2011, August 31 - September 2, 2011, Oulu, Finland*, pages 127–134, 2011. 33

[58] Daniel Plagge and Michael Leuschel. Seven at One Stroke : LTL Model Checking for High-level Specifications in B, Z, CSP, and More. *Int. J. Softw. Tools Technol. Transf.*, 12(1) :9–21, January 2010. 27

[59] Antoine Requet. *BART : A Tool for Automatic Refinement*, pages 345–345. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. 26

[60] Willem-Paul de Roever and Kai Engelhardt. *Data Refinement : Model-Oriented Proof Methods and Their Comparison*. Cambridge University Press, New York, NY, USA, 1st edition, 2008. 25

[61] Imen Sayar and Mohamed Tahar Bhiri. From an Abstract Specification in Event-b Toward an UML/OCL Model. In *Proceedings of the 2Nd FME Workshop on Formal Methods in Software Engineering*, FormaliSE 2014, pages 17–23, New York, NY, USA, 2014. ACM. 78

[62] Steve Schneider, Helen Treharne, and Heike Wehrheim. The Behavioural Semantics of Event-B Refinement. *Formal Aspects of Computing*, 26(2) :251–280, 2014. 25

[63] Steve Schneider, Helen Treharne, Heike Wehrheim, and David M. Williams. *Managing LTL Properties in Event-B Refinement*, pages 221–237. Springer International Publishing, Cham, 2014. 26

[64] Philippe Schnoebelen, Béatrice Bérard, Michel Bidoit, François Laroussinie, and Antoine Petit. *Vérification de Logiciels : Techniques et Outils du Model-Checking*. Vuibert, April 1999. 5

[65] Badr Siala, Mohamed Tahar Bhiri, Jean-Paul Bodeveix, and Mamoun Filali. An Event-B development process for the distributed BIP framework (FAC, Toulouse, 30/03/16-31/03/16). 2016. 42

[66] Badr Siala, Mohamed Tahar Bhiri, Jean-Paul Bodeveix, and Mamoun Filali. *Implicit and explicit semantics integration in formal developments of discrete systems – Communications of NII Shonan Meetings*, chapter An Event-B development process for the distributed BIP framework. Shonan publishing, 2018. **(Submitted)**. 42

[67] Badr Siala, Tahar Bhiri, Jean-Paul Bodeveix, and Mamoun Filali. An Event-B Development Process for the Distributed BIP Framework. (regular paper). In Kazuhiro Ogata, Mark Lawford, and Shaoying Liu, editors, *International Conference on Formal Engineering Methods (ICFEM), Tokyo Japan, 14/11/2016-18/11/2016*, volume 10009 of *Lecture Notes in Computer Science*, pages 313–328, http ://www.springerlink.com/, novembre 2016. Springer-Verlag. 42

[68] Badr Siala, Jean-Paul Bodeveix, Mamoun Filali, and Tahar Bhiri. Automatic Refinement for Event-B through Annotated Patterns (short paper). In Igor Kotenko, Yiannis Cotronis, and Masoud Daneshtalab, editors, *Euromicro International Conference on Parallel, Distributed and network-based Processing, St. Petersburg, Russia, 06/03/2017-08/03/2017*, pages 287–290, http ://www.ieee.org/, mars 2017. IEEE. 110

[69] Badr Siala, Mohamed Tahar Bhiri, Jean-Paul Bodeveix, and Mamoun Filali. Un Processus de Développement Event-B pour des Applications Distribuées (regular paper). In *Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL), Besançon, 07/06/2016-08/06/2016*, page (en ligne), http ://www.univ-fcomte.fr, juin 2016. Université de Franche-Comté. 42

[70] Joseph Sifakis. A Framework for Component-based Construction Extended Abstract. In Bernhard K. Aichernig and Bernhard Beckert, editors, *SEFM*, pages 293–300. IEEE Computer Society, 2005. 4, 6, 42

[71] Renato Silva. Towards the Composition of Specifications in Event-B. *Electronic Notes in Theoretical Computer Science*, 280(0) :81 – 93, 2011. Proceedings of the B 2011 Workshop, a satellite event of the 17th International Symposium on Formal Methods (FM 2011). 2, 29

[72] Renato Silva. *Supporting Development of Event-B Models*. PhD thesis, University of Southampton, May 2012. 2, 28, 29, 30, 31, 32, 110

[73] Renato Silva and Michael Butler. Supporting Reuse of Event-B Developments Through Generic Instantiation. In *ICFEM '09 Proceedings*, pages 466–484, 2009. 19, 39

[74] Renato Silva and Michael Butler. Shared Event Composition/Decomposition in Event-B. In *FMCO Formal Methods for Components and Objects*, November 2010. 2, 19, 28, 29, 30, 39, 43, 60

[75] Renato Silva, Carine Pascal, Thai Son Hoang, and Michael Butler. Decomposition Tool for Event-B. *Software : Practice and Experience*, 41(2) :199–208, 2011. 2, 19, 28, 31, 39, 64

[76] Neeraj Kumar Singh. EB2ALL : An Automatic Code Generation Tool. In *Using Event-B for Critical Device Software Systems*, pages 105–141. Springer London, 2013. 33

[77] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF : Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd edition, 2009. 73

[78] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems : Principles and Paradigms (2Nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006. 4

[79] Rivera Víctor. *Code Generation for Event-B*. PhD thesis, Universidade da Madeira, Centro de Ciencias Exactas e da Engenharia, June 2016. 34

[80] Steve Wright. Using Event-B to Create a Virtual Machine Instruction Set Architecture. In *Abstract State Machines, B and Z*, pages 265–279. Springer Berlin / Heidelberg, September 2008. 33

[81] Steve Wright. Automatic Generation of C from Event-B. In *Workshop on Integration of Model-based Formal Methods and Tools*. http ://www.lina.sciences.univ-nantes.fr/apcb/IM_FMT2009/im_fmt2009_proceedings.html, February 2009. 33

[82] Faqing Yang and Jean-Pierre Jacquot. JeB : un Environnement de Simulation en JavaScript pour B Événementiel. In *Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL)*, Nancy, France, April 2013. 28

[83] Faqing Yang, Jean-Pierre Jacquot, and Jeanine Souquières. The Case for Using Simulation to Validate Event-B Specifications. In *The 19th Asia-Pacific Software Engineering Conference*, pages 85–90, 2012. 28

A

Chapitre état de l'art

A.1 Machine composée expansée Hotel_CMP_M0

```
machine Hotel_CMP_M0 sees Chambre_C Client_C

variables  cle_serrure  premier second place  k1 k2 s

invariants
  @[Hotel]Chambre_M\inv1 cle_serrure∈ℕ
  @[Hotel]Chambre_M\inv2 premier∈ℕ
  @[Hotel]Chambre_M\inv3 second∈ℕ
  @[Hotel]Chambre_M\inv4 premier≠second
  @[Hotel]Chambre_M\inv5 cle_serrure∈{premier,second}
  @[Hotel]Chambre_M\inv6 place∈PLACE
  @[Hotel]Client_M\inv1_0 k1∈ℕ
  @[Hotel]Client_M\inv2_1 k2∈ℕ
  @[Hotel]Client_M\inv3_2 k1≠k2
  @[Hotel]Client_M\inv4_3 s∈STATE
  @[Hotel]Hotel_CMP\inv1_0 k1∈ℕ
  @[Hotel]Hotel_CMP\inv2_1 k2∈ℕ
  @[Hotel]Hotel_CMP\inv3_2 k1≠k2
  @[Hotel]Hotel_CMP\inv4_3 s∈STATE
  @[Hotel]Hotel_CMP\inv1_4 cle_serrure∈ℕ
  @[Hotel]Hotel_CMP\inv2_5 premier∈ℕ
  @[Hotel]Hotel_CMP\inv3_6 second∈ℕ
  @[Hotel]Hotel_CMP\inv4_7 premier≠second
  @[Hotel]Hotel_CMP\inv5_8 cle_serrure∈{premier,second}
  @[Hotel]Hotel_CMP\inv6_9 place∈PLACE

events
  event INITIALISATION
    then
      @[Hotel]Chambre_M\act1 cle_serrure,premier,second :|premier'∈ℕ∧second'∈ℕ∧premier'≠
     second'∧cle_serrure'∈ℕ∧cle_serrure'∈{premier',second'}
      @[Hotel]Chambre_M\act2 place :=dispo
      @[Hotel]Client_M\act1 k1,k2 :|k1'∈ℕ∧k2'∈ℕ∧k1'≠k2'
      @[Hotel]Client_M\act2 s :=pas_de_carte
  end

  event  passer_carte
    any  cle1  cle2
    where
      @[Hotel]Chambre_M\grd1 cle1∈ℕ
      @[Hotel]Chambre_M\grd2 cle2∈ℕ
      @[Hotel]Chambre_M\grd3 cle1≠cle2
      @[Hotel]Chambre_M\grd4 cle1=cle_serrure
      @[Hotel]Chambre_M\grd5 place=dispo
      @[Hotel]Client_M\grd1 s= carte_initialisee
    then
      @[Hotel]Chambre_M\act1 premier :=cle1
      @[Hotel]Chambre_M\act2 second :=cle2
      @[Hotel]Chambre_M\act3 place :=carte_valide
      @[Hotel]Client_M\act1 s :=carte_a_valider
```

```
  end

  event attribuer
    where
      @[Hotel]Chambre_M\grd1 place=carte_valide
      @[Hotel]Chambre_M\grd2 cle_serrure=premier ∨ cle_serrure=second
    then
      @[Hotel]Chambre_M\act1 place:=chambre_attribuee
      @[Hotel]Chambre_M\act2 cle_serrure:|((cle_serrure=premier)⇒(cle_serrure'=second))∧((
     cle_serrure=second)⇒(cle_serrure'=cle_serrure))
  end

  event repasser_carte
    where
      @[Hotel]Chambre_M\grd1 place=chambre_attribuee
      @[Hotel]Client_M\grd1 s=carte_a_valider
    then
      @[Hotel]Client_M\act1 s:=carte_a_valider
  end

  event quitter
    where
      @[Hotel]Chambre_M\grd1 place=chambre_attribuee
      @[Hotel]Client_M\grd1 s=carte_a_valider
    then
      @[Hotel]Chambre_M\act1 place:=dispo
      @[Hotel]Client_M\act1 s:=pas_de_carte
  end

  event prendre_carte
    any cle1 cle2
    where
      @[Hotel]Client_M\grd1 cle1∈ℕ
      @[Hotel]Client_M\grd2 cle2∈ℕ
      @[Hotel]Client_M\grd3 cle1≠cle2
      @[Hotel]Client_M\grd4 s=pas_de_carte
    then
      @[Hotel]Client_M\act1 k1:=cle1
      @[Hotel]Client_M\act2 k2:=cle2
      @[Hotel]Client_M\act3 s:= carte_initialisee
  end
end
```

Listing A.1 – Machine composée `Hotel_CMP_M0`

A.2 Modèle Essai

A.2.1 Contexte Essai_C0

```
context Essai_C0
constants d0  //valeur initiale
axioms
@data_typ d0∈ℕ
end
```

Listing A.2 – Contexte `Essai_C0`

A.2.2 Machine Essai_M0

```
machine Essai_M0 sees Essai_C0

variables  a // variable dont le contenu sera tansferee·
           b // variable qui recevra le contenu de a·
           m // la valeur intermediaire intervenant lors de l'echange du contenu de a et b·
           ctrl // variable de controle·
```

```
invariants
  @inv1 a∈ℕ
  @inv2 b∈ℕ
  @inv3 m∈ℕ
  @inv4  ctrl ∈𝔹

events
  event INITIALISATION // initialisation des variables
    then
      @act1 a:=d0
      @act2 b:∈ℕ
      @act3 m:∈ℕ
      @act4  ctrl :=FALSE
  end

  event Transfer_mid // evenement de transfert de contenue
    where
      @grd1  ctrl=FALSE // aucun transfert de contenue encours
    then
      @act1  ctrl  := TRUE // transfert de contenue encours
      @act2 m  := a
  end

  event Change // evenement d'echange de contenu
    any p
    where
      @grd1 p∈ℕ
      @grd2  ctrl  =TRUE // aucun echange de contenu encours
    then
      @act1  ctrl  := FALSE // echange de contenu encours
      @act2 a  := p
  end

  event Copy // evenement de copie de valeur·
    where
      @grd1  ctrl  = TRUE // possibilite de copier
    then
      @act1 b  := m
  end
end
```

Listing A.3 – Machine `Essai_M0`

A.2.3 Machine Essai_M1

```
machine Essai_M1 refines Essai_M0  sees Essai_C0

variables  a // variable dont le contenu sera tansferee·
           b // variable qui recevra le contenu de a·
           m // la valeur intermediaire intervenant lors de l'echange du contenu de a et b·
           ctrl  // variable de controle·


events
  event INITIALISATION // initialisation des variables
    then
      @act1 a:=d0
      @act2 b:∈ℕ
      @act3 m:∈ℕ
      @act4  ctrl :=FALSE
  end

  event Transfer_mid // evenement raffine
  refines  Transfer_mid
    any p
    where
      @grd1  ctrl=FALSE
      @grd2 p∈ℕ
      @grd3 p=a
```

```
    then
      @act1 ctrl := TRUE
      @act2 m := p
  end

  event Change // evenement d'echange de contenu
   refines Change
    any p
    where
      @grd1 p∈ℕ
      @grd2 ctrl =TRUE // aucun echange de contenu encours
    then
      @act1 ctrl := FALSE // echange de contenu encours
      @act2 a := p
  end

  event Copy // evenement raffine
   refines Copy
    any p // parametre de raffinement·
    where
      @grd1 ctrl = TRUE
      @grd2 p∈ℕ
      @grd3 p=m
    then
      @act1 b := p // b prends une valeur p dont le contenu est equivalent a celle de m
  end
end
```

Listing A.4 – Machine `Essai_M1`

A.2.4 Machine composée Essai_M1_CMP

```
composed machine Essai_M1_CMP
 refines Essai_M1 sees Essai_C0
includes
[Transf_Data] MA (Invariant ¬included)
[Transf_Data] MB (Invariant ¬included)
[Transf_Data] MM (Invariant ¬included)

composes events
INITIALISATION
combines event
[Transf_Data] MA·INITIALISATION | | [Transf_Data] MB·INITIALISATION | | [Transf_Data] MM
      ·INITIALISATION

Transfer_mid
combines event
[Transf_Data] MA·Transfer_mid | | [Transf_Data] MM·Transfer_mid
 refines Transfer_mid

Change
combines event
[Transf_Data] MA·Change | | [Transf_Data] MM·Change
 refines Change

Copy
combines event
[Transf_Data] MB·Copy | | [Transf_Data] MM·Copy
 refines Copy

END
```

Listing A.5 – Machine `Essai_M1_CMP`

A.2.5 Machines MA, MB et MM

```
machine MA sees Essai_C0

variables a

invariants
  theorem @typing_a a ∈ ℤ

events
  event INITIALISATION
    then
      @act1 a :=d0
  end

  event Transfer_mid
    any p
    where
      theorem @typing_p p ∈ ℤ
      @grd2 p∈ℕ
      @grd3 p=a
  end

  event Change
    any p
    where
      theorem @typing_p p ∈ ℤ
      @grd1 p∈ℕ
    then
      @act2 a := p
  end
end
```

Listing A.6 – Machine `MA`

```
machine MB sees Essai_C0

variables b

invariants
  theorem @typing_b b ∈ ℤ

events
  event INITIALISATION
    then
      @act2 b :∈ℕ
  end

  event Copy
    any p
    where
      theorem @typing_p p ∈ ℤ
      @grd2 p∈ℕ
    then
      @act1 b := p
  end
end
```

Listing A.7 – Machine `MB`

```
machine MM sees Essai_C0

variables m ctrl

invariants
  theorem @typing_ctrl ctrl ∈
    𝔹
  theorem @typing_m m ∈ ℤ

events
  event INITIALISATION
    then
      @act3 m :∈ℕ
      @act4 ctrl :=FALSE
  end

  event Transfer_mid
    any p
    where
      theorem @typing_p p ∈ ℤ
      @grd1 ctrl=FALSE
      @grd2 p∈ℕ
    then
      @act1 ctrl := TRUE
      @act2 m := p
  end

  event Change
    any p
    where
      theorem @typing_p p ∈ ℤ
      @grd1 p∈ℕ
      @grd2 ctrl =TRUE
    then
      @act1 ctrl := FALSE
  end

  event Copy
    any p
    where
      theorem @typing_p p ∈ ℤ
      @grd1 ctrl = TRUE
      @grd2 p∈ℕ
      @grd3 p=m
  end
end
```

Listing A.8 – Machine `MM`

B

Chapitre Démarche

B.1 Grammaire Xtext d'un contexte Event-B

```
grammar fr. irit .eventb.context.Context
  with fr. irit .eventb.formulas.Formulas

generate context "http://www.irit.fr/eventb/context/Context"

import 'http://fr. irit .eventb.emf.ident'

Context:
  'context' name=ID
  ('extends' (extends +=[Context]) (',' extends +=[Context])*)?
  ('sets' (sets+=Set)+)?
  ('constants' (constants+=Constant)+)?
  ('axioms' (axioms += Axiom)+)?
  'end';

Set returns SetIdentifierExpression: name=ID;
Constant returns ConstantIdentifierExpression: name=ID;

Axiom: '@' name=ID property=Predicate;
```

Listing B.9 – Grammaire Xtext d'un contexte Event-B

B.2 Grammaire Xtext d'une machine Event-B

```
grammar fr. irit .eventb.machine.Machine
  with fr. irit .eventb.formulas.Formulas

import 'http://www.irit.fr/eventb/context/Context' as ctx
import 'http://fr. irit .eventb.emf.ident' as id
import 'http://fr. irit .eventb.emf.formulas' as frm

generate machine "http://www.irit.fr/eventb/machine/Machine"

Machine:
  'machine' name=ID
  ('refines' (refines +=[Machine])+)?
  ('sees' (sees+=[ctx::Context])+)?
  ('variables' (variables +=Variable)+)?
  ('invariants' (invariants +=Invariant)+)?
  ('variant' variant=expression)?
  ('events' (events+=Event)+)?
  'end'
  ;

Variable returns id::VariableIdentifierExpression: name=ID;
```

```
Parameter returns id::ParameterIdentifierExpression: name=ID;

Event:
  (convergence=Convergence)?
  'event' name=ID
  eventRefinement=EventRefinement?
  (('any' (parameters+=Parameter)+
       'where' (guards+=Guard)+)
   | 'when' (guards+=Guard)+)?
  ('with' (witness+=Witness)+)?
  ('then' (actions+=LabeledAction)+)?
  'end'
;

enum Convergence:
    CONVERGENT = 'convergent'
 | ORDINARY = 'ordinary'
 | ANTICIPATED = 'anticipated';

EventRefinement:
   ' refines ' refines +=[Event] (',' ( refines += [Event]))*
 |  'extends' extends=[Event]
;

Guard:
  '@' name=ID predicate=Predicate;

Witness:
  '@' name=ID predicate=Predicate;

Invariant:
  '@' name=ID predicate=Predicate;

LabeledPredicate: Guard | Witness | Invariant ;

LabeledAction:
  '@' name=ID action=Action
;

VariableReference returns frm::IdentifierExpression:
  {VariableReference} reference=[ id::VariableIdentifierExpression ];

Action:
  ({BecomesEqualToAssignment} identifiers+=[ id::VariableIdentifierExpression ] (',' 
        identifiers +=[ id::VariableIdentifierExpression ])*
    => (':=' | ' :=') expressions +=expression (',' expressions +=expression)*)
 | ({BecomesMemberOfAssignment} identifiers+=VariableReference
    => ('::' | ' :: ') expression =expression)
 | ({BecomesSuchThatAssignment} identifiers+=[ id::VariableIdentifierExpression ] (',
       ' identifiers +=[ id::VariableIdentifierExpression ])*
    => ':|' predicate =Predicate)
 | ({FunctionUpdate} function=[ id::VariableIdentifierExpression ] => '('argument=
      expression')' (' :=' | ' :=') right =expression)
;
```

Listing B.10 – Grammaire Xtext d'une machine Event-B

B.3 Grammaire Xtext de distribution

```
grammar fr. irit .eventb.decomposition.Decomposition
 //with org. eclipse .xtext.common.Terminals
  with fr. irit .eventb.formulas.Formulas

generate decomposition "http://www.irit. fr/eventb/decomposition/Decomposition"

import 'http://www.irit. fr/eventb/machine/Machine' as mch
import 'http://fr. irit .eventb.emf.ident' as id
```

```
SharedEventDecomposition: {SharedEventDecomposition}
  'shared' 'event' 'decomposition' name=ID
  ' refines ' refines =[mch::Machine]
  'components' (components+=Component)+
  'mappings' (mappings+=Mapping) (';' mappings+=Mapping)*
  ('events' (events+=EventMapping)+)?
  ('connectors'
  (connectiors+=BinaryConnector) (';' connectiors+=BinaryConnector)*
  'end';

Step:
  '-[' event=ID ']->' component=[Component]
;
Route:
  name=ID ':'
  source=[Component]
  (step+=Step)+
;
Component: name=ID;
BinaryConnector:
  name=ID ':' end1= [Component] '<->' end2 = [Component]
  ;
Mapping:
  ' variable ' {VariableMapping} variables+=[ id::VariableIdentifierExpression ] ('' | '|->')
       component=[Component]
  | ' variables ' {VariableMapping} variables+=[ id::VariableIdentifierExpression ] ( variables
      +=[ id::VariableIdentifierExpression ])+ ('' | '|->') component=[Component]
  | 'guard' {GuardMapping} event=[mch::Event] '.' guard=[mch::Guard] '|->' component=[
      Component];

EventMapping: 'event' event=[mch::Event] (parameters+=ParameterMapping)+;

ParameterMapping:
  ('when' (parameters+=[ id::ParameterIdentifierExpression ])+)?
  'parameter' parameter=[ id::ParameterIdentifierExpression ]
  '->' component=[Component]
  ' init ' init =expression
  'with' (guards+=[mch::Guard])+
;
```

Listing B.11 – Grammaire Xtext de distribution

C

Chapitre Étude de cas

C.1 La Machine Hotel_M0

```
machine Hotel_M0 sees Hotel_C0

variables  registered  // Variable systeme
           isin  // Variable d'environnement
invariants
  @typ_regi  registered ∈ROOM⇸GUEST // une chambre est allouée à au plus
      un client
  @typ_isin  isin ∈ROOM→ℙ(GUEST) // Contenu effectif d'une chambre
  @safe ∀r·(r∈ROOM⇒isin(r)⊆ registered[{r}]) // Un client dans la chambre est
      un client enregistré SAF−1

events
  event INITIALISATION // Initialisation des variables
    then
      @init_regi  registered :=∅
      @init_isin  isin :=ROOM×{∅}
  end

  event check_in
  /* un nouveau client peut être affecté
     à une chambre non reservée */
    any g r
    where
      @typ_g g∈GUEST
    @typ_r r∈ROOM
      @free_r  r∉dom(registered)
    then
      @booked_r registered(r) :=g
  end

  event enter_room
    any g r
    where
      @typ_g g∈GUEST
      @typ_r r∈ROOM
      @booked_r r↦g∈registered
      @empty_r g∉isin(r)
    then
      @busy_r isin(r) :=isin(r)∪{g} // g est dans la chambre physiquement
  end

  event exit_room // Un client peut quitter sa chambre
    any g r
    where
      @typ_g g∈GUEST
      @typ_r r∈ROOM
      @busy_r g∈isin(r)
    then
```

```
    @empty_r isin(r) := isin(r)\{g} // g n'est plus dans la chambre
    physiquement
  end

  event check_out // Modélise le check out volontaire
    any g r
    where
      @typ_g g∈GUEST
      @typ_r r∈ROOM
      @booked_r r↦g∈registered
      @empty_r g∉isin(r)
    then
      @free_r  registered :={r}⩤registered
  end

  event forced_check_in
  /* un nouveau client peut être affecté
     à une chambre reservée mais vide */
    any g r
    where
      @typ_g g∈GUEST
      @typ_r r∈ROOM
      @empty_r isin(r)=∅
    then
      @booked_r registered(r) :=g
  end

  event forced_check_out
  /* Modélise le check out forcé
     (réalise par le bureau d'accueil) */
    any g r
    where
      @typ_g g∈GUEST
      @typ_r r∈ROOM
      @booked_r r↦g∈registered
      @busy_r g∈isin(r) // exclusion mutuelle avec check_out
    then
      @free_r  registered :={r}⩤registered
      @empty_r isin(r) := isin(r)\{g}
  end
end
```

Listing C.12 – Machine `Hotel_M0`

C.2 La machine Hotel_M1

```
machine Hotel_M1 refines Hotel_M0 sees Hotel_C1

variables  owns currk  issued  cards  roomk isin

invariants
  @typ_owns owns∈ROOM↠GUEST // Typage
  @typ_currk currk∈ROOM↣KEY // Typage
  @typ_issued issued⊆KEY // Typage
  @fin_issued  finite (issued)
  @typ_cards cards∈GUEST→ℙ(CARD) // Typage
  @typ_roomk roomk∈ROOM→KEY // Typage ENV−2
  @key_room_desk ∀r·(r∈ROOM⇒currk(r)∈issued)
  @key1_card ∀g·(g∈GUEST⇒(∀c·(c∈cards(g)⇒prj1(c)∈issued))) // ENV−4
  @key2_card ∀g·(g∈GUEST⇒(∀c·(c∈cards(g)⇒prj2(c)∈issued))) // ENV−4
  @key_lock ∀r·(r∈ROOM⇒roomk(r)∈issued)
  @keys_card ∀r,g,k·(r∈ROOM∧g∈GUEST∧k∈KEY∧k∉issued⇒currk(r)↦k∉cards(g)) // ENV
    −4
  @uniq_card1 ∀g1,g2·(cards(g1)∩cards(g2) ≠ ∅ ⇒ g1=g2)
  @uniq_card2 ∀c1,c2,g1,g2·c1∈cards(g1) ∧ c2∈cards(g2) ∧ prj2(c1)=prj2(c2) ⇒ c1=c2
  @inv_Hotel_M1_safe ∀r,g·(r∈ROOM∧g∈GUEST∧g∈isin(r)⇒r↦g∈registered) // Tout client
    dans une chambre est un client enregistré
```

```
  @inv_coll ∀c,r,g·(r∈ROOM∧g∈GUEST∧c∈cards(g)∧roomk(r)=prj2(c)⇒r↦g∈registered)
events
  event INITIALISATION // Initialisation des variables
    then
      @init_owns owns:=∅
      @init_currk currk:=initk
      @init_cards cards:=GUEST×{∅}
      @init_isin isin:=ROOM×{∅}
      @init_issued issued:=ran(initk)
      @init_roomk roomk:=initk
  end

  event register
    any r g k
    where
      @typ_r r∈ROOM
      @typ_g g∈GUEST
      @typ_k k∈KEY
      @free_k k∉issued
      @free_r r∉dom(owns)
    then
      @key_r_desk currk(r):=k
      @used_k issued:=issued∪{k}
      @new_card cards(g):=cards(g)∪{currk(r)↦k}
      @booked_r owns(r):=g
  end

  event get_room refines forced_check_in
    any c g r
    where
      @typ_c c∈CARD
      @card_g c∈cards(g)
      @typ_g g∈GUEST
      @typ_r r∈ROOM
      @key_lock_c1 roomk(r)=prj1(c)
      @empty_r isin(r)=∅
    then
      @update_key_lock roomk(r):=prj2(c)
  end

  event exit_room extends exit_room
  end

  event unregister
    any g r
    where
      @typ_g g∈GUEST
      @typ_r r∈ROOM
      @booked_r_g r↦g∈owns
      @empty_r g∉isin(r)
    then
      @booked_r owns:={r}⩤owns
      @update_card_g cards(g):=cards(g)\(KEY×{currk(r)})
  end

  event enter_room // FUN−1
   refines enter_room
    any c g r
    where
      @typ_c c∈CARD
      @card_g c∈cards(g)
      @typ_g g∈GUEST
      @typ_r r∈ROOM
      @empty_r g∉isin(r)
      @key_lock roomk(r)=prj2(c)
    then
      @busy_r isin(r):=isin(r)∪{g}
  end
end
```

Listing C.13 – Machine `Hotel_M1`

C.3 Machine obtenue après fragmentation

machine Hotel_dep
refines Hotel_M1
sees Hotel_C1
variables
// inherited variables
owns
currk
issued
cards
roomk
isin
// event parameters and computation states
register_g
register_g_computed
register_r
register_r_computed
register_k
register_k_computed
get_room_g
get_room_g_computed
get_room_c
get_room_c_computed
get_room_r
get_room_r_computed
unregister_g
unregister_g_computed
unregister_r
unregister_r_computed
enter_room_g
enter_room_g_computed
enter_room_c
enter_room_c_computed
enter_room_r
enter_room_r_computed
invariants
// inherited typing invariants
@TY_owns owns∈ROOM↔GUEST
@TY_currk currk∈ROOM↣KEY
@TY_issued issued⊆KEY
@TY_issued finite(issued)
@TY_cards cards∈GUEST→ℙ(CARD)
@TY_cards ∀g1,g2·(cards(g1)∩cards(g2) ≠ ∅ ⇒ g1=g2)
@TY_cards ∀c1,c2,g1,g2·c1∈cards(g1) ∧ c2∈cards(g2) ∧ prj_2(c1)=prj_2(c2) ⇒ c1=c2
@TY_roomk roomk∈ROOM→KEY
@TY_isin isin∈ROOM→ℙ(GUEST)
@TY_isin isin∈ROOM→ℙ(GUEST)
// typing invariants of control variables
@TY_register_g_computed register_g_computed ∈ ℙ
@TY_register_r_computed register_r_computed ∈ ℙ
@TY_register_k_computed register_k_computed ∈ ℙ
@TY_get_room_g_computed get_room_g_computed ∈ ℙ
@TY_get_room_c_computed get_room_c_computed ∈ ℙ
@TY_get_room_r_computed get_room_r_computed ∈ ℙ
@TY_unregister_g_computed unregister_g_computed ∈ ℙ
@TY_unregister_r_computed unregister_r_computed ∈ ℙ
@TY_enter_room_g_computed enter_room_g_computed ∈ ℙ
@TY_enter_room_c_computed enter_room_c_computed ∈ ℙ
@TY_enter_room_r_computed enter_room_r_computed ∈ ℙ
// typing invariants of globalised parameters
@TY_register_g_typ_g register_g∈GUEST
@TY_register_r_typ_r register_r∈ROOM
@TY_register_k_typ_k register_k∈KEY
@TY_get_room_g_typ_g get_room_g∈GUEST
@TY_get_room_c_typ_c get_room_c∈CARD
@TY_get_room_r_typ_r get_room_r∈ROOM
@TY_unregister_g_typ_g unregister_g∈GUEST
@TY_unregister_r_typ_r unregister_r∈ROOM
@TY_enter_room_g_typ_g enter_room_g∈GUEST

```
  @TY_enter_room_c_typ_c enter_room_c∈CARD
  @TY_enter_room_r_typ_r enter_room_r∈ROOM
  // specification of parameter values (guards of inherited events)
  // explicit typing is useless (Event−B type synthesis)
  @register_g_typ_g register_g_computed = TRUE ⇒ ( register_g∈GUEST)
  @register_r_typ_r register_r_computed = TRUE ⇒ ( register_r∈ROOM)
  @register_r_free_r register_r_computed = TRUE ⇒ ( register_r∉dom(owns))
  @register_k_typ_k register_k_computed = TRUE ⇒ ( register_k∈KEY)
  @register_k_free_k register_k_computed = TRUE ⇒ ( register_k∉issued)
  @get_room_g_typ_g get_room_g_computed = TRUE ⇒ ( get_room_g∈GUEST)
  @get_room_c_typ_c get_room_c_computed = TRUE ⇒ ( get_room_c∈CARD)
  @get_room_c_card_g get_room_c_computed = TRUE ⇒ ( get_room_c∈cards(get_room_g
     ))
  @get_room_r_typ_r get_room_r_computed = TRUE ⇒ ( get_room_r∈ROOM)
  @get_room_r_empty_r get_room_r_computed = TRUE ⇒ ( isin(get_room_r)=∅)
  @unregister_g_typ_g unregister_g_computed = TRUE ⇒ ( unregister_g∈GUEST)
  @unregister_r_typ_r unregister_r_computed = TRUE ⇒ ( unregister_r∈ROOM)
  @enter_room_g_typ_g enter_room_g_computed = TRUE ⇒ ( enter_room_g∈GUEST)
  @enter_room_c_typ_c enter_room_c_computed = TRUE ⇒ ( enter_room_c∈CARD)
  @enter_room_c_card_g enter_room_c_computed = TRUE ⇒ ( enter_room_c∈cards(
     enter_room_g))
  @enter_room_r_typ_r enter_room_r_computed = TRUE ⇒ ( enter_room_r∈ROOM)
  // dependencies between computation states
  @get_room_g_c get_room_g_computed = FALSE ⇒ get_room_c_computed = FALSE
  @enter_room_g_c enter_room_g_computed = FALSE ⇒ enter_room_c_computed =
     FALSE
variant
  {FALSE ↦ 1, TRUE ↦ 0}(register_k_computed) + {FALSE ↦ 1, TRUE ↦ 0}(
     register_r_computed)
events
  event INITIALISATION
  then
    // initialization of computation states
    @register_g_not_computed register_g_computed := FALSE
    @register_r_not_computed register_r_computed := FALSE
    @register_k_not_computed register_k_computed := FALSE
    @get_room_g_not_computed get_room_g_computed := FALSE
    @get_room_c_not_computed get_room_c_computed := FALSE
    @get_room_r_not_computed get_room_r_computed := FALSE
    @unregister_g_not_computed unregister_g_computed := FALSE
    @unregister_r_not_computed unregister_r_computed := FALSE
    @enter_room_g_not_computed enter_room_g_computed := FALSE
    @enter_room_c_not_computed enter_room_c_computed := FALSE
    @enter_room_r_not_computed enter_room_r_computed := FALSE
    // initialization of parameter variables
     @register_g_init register_g := guest0
     @register_r_init register_r := room0
     @register_k_init register_k := key0
    @get_room_g_init get_room_g := guest0
    @get_room_c_init get_room_c := card0
    @get_room_r_init get_room_r := room0
     @unregister_g_init unregister_g := guest0
     @unregister_r_init unregister_r := room0
    @enter_room_g_init enter_room_g := guest0
    @enter_room_c_init enter_room_c := card0
    @enter_room_r_init enter_room_r := room0
    // inherited
    @init_owns owns :=∅
        @init_currk currk :=initk
        @init_cards cards :=GUEST×{∅}
         @init_isin isin :=ROOM×{∅}
        @init_issued issued :=ran( initk )
        @init_roomk roomk :=initk

  end
  // computation of register parameters
  // computation of g
  event compute_register_g
  any g
  where
    @typ_g g∈GUEST
  then
```

```
    @register_g_done register_g_computed := TRUE
    @register_g_value register_g := g
    // reset dependent parameters
end
// computation of r
convergent event compute_register_r
any r
where
    @typ_r r∈ROOM
    @free_r r∉dom(owns)
then
    @register_r_done register_r_computed := TRUE
    @register_r_value register_r := r
    // reset dependent parameters
end
// computation of k
convergent event compute_register_k
any k
where
    @register_k_notcomputed register_k_computed = FALSE
    @typ_k k∈KEY
    @free_k k∉issued
then
    @register_k_done register_k_computed := TRUE
    @register_k_value register_k := k
    // reset dependent parameters
end

event register refines register
when
    // event parameters have been computed
    @register_g_computed register_g_computed = TRUE
    @register_r_computed register_r_computed = TRUE
    @register_k_computed register_k_computed = TRUE
    // guards unused by event parameters
with
    @g g = register_g
    @r r = register_r
    @k k = register_k
then
    @key_r_desk currk( register_r ) :=register_k
    @used_k issued :=issued∪{register_k}
    @new_card cards(register_g) :=cards( register_g )∪{currk( register_r )↦register_k }
    @booked_r owns(register_r) :=register_g
    // reset computation state of current event
    // or of dependent parameters of other events
    @register_g_init register_g_computed := FALSE
    @register_r_init register_r_computed := FALSE
    @register_k_init register_k_computed := FALSE
    @get_room_c_init get_room_c_computed := FALSE
    @enter_room_c_init enter_room_c_computed := FALSE
end
// computation of get_room parameters
// computation of g
event compute_get_room_g
any g
where
    @typ_g g∈GUEST
then
    @get_room_g_done get_room_g_computed := TRUE
    @get_room_g_value get_room_g := g
    // reset dependent parameters
    @c_reset get_room_c_computed := FALSE
end
// computation of c
event compute_get_room_c
any c
where
    @get_room_g_computed get_room_g_computed = TRUE
    @typ_c c∈CARD
    @card_g c∈cards(get_room_g)
then
```

```
  @get_room_c_done get_room_c_computed := TRUE
  @get_room_c_value get_room_c := c
  // reset dependent parameters
end
// computation of r
event compute_get_room_r
any r
where
  @typ_r  r∈ROOM
  @empty_r  isin(r)=∅
then
  @get_room_r_done get_room_r_computed := TRUE
  @get_room_r_value get_room_r := r
  // reset dependent parameters
end

event get_room refines get_room
when
  // event parameters have been computed
  @get_room_g_computed get_room_g_computed = TRUE
  @get_room_c_computed get_room_c_computed = TRUE
  @get_room_r_computed get_room_r_computed = TRUE
  // guards unused by event parameters
  @key_lock_c1  roomk(get_room_r)=prj1(get_room_c)
with
  @g g = get_room_g
  @c c = get_room_c
  @r r = get_room_r
then
  @update_key_lock  roomk(get_room_r):=prj2(get_room_c)
  // reset computation state of current event
  // or of dependent parameters of other events
  @get_room_g_init get_room_g_computed := FALSE
  @get_room_r_init get_room_r_computed := FALSE
  @get_room_c_init get_room_c_computed := FALSE
end
// computation of unregister parameters
// computation of g
event compute_unregister_g
any g
where
  @typ_g  g∈GUEST
then
  @unregister_g_done unregister_g_computed := TRUE
  @unregister_g_value unregister_g := g
  // reset dependent parameters
end
// computation of r
event compute_unregister_r
any r
where
  @typ_r  r∈ROOM
then
  @unregister_r_done unregister_r_computed := TRUE
  @unregister_r_value  unregister_r := r
  // reset dependent parameters
end

event unregister refines unregister
when
  // event parameters have been computed
  @unregister_g_computed unregister_g_computed = TRUE
  @unregister_r_computed unregister_r_computed = TRUE
  // guards unused by event parameters
  @booked_r_g  unregister_r↦unregister_g∈owns
  @empty_r  unregister_g∉isin( unregister_r )
with
  @g g = unregister_g
  @r r = unregister_r
then
  @booked_r  owns:={unregister_r}⩤owns
  @update_card_g  cards(unregister_g):=cards(unregister_g)\(KEY×{currk(unregister_r)})
```

```
    // reset computation state of current event
    // or of dependent parameters of other events
     @register_r_init  register_r_computed := FALSE
    @get_room_c_init get_room_c_computed := FALSE
     @unregister_r_init  unregister_r_computed := FALSE
    @unregister_g_init  unregister_g_computed := FALSE
    @enter_room_c_init enter_room_c_computed := FALSE
end
// computation of enter_room parameters
// computation of g
event compute_enter_room_g
any g
where
  @typ_g  g∈GUEST
then
  @enter_room_g_done enter_room_g_computed := TRUE
  @enter_room_g_value enter_room_g := g
  // reset dependent parameters
  @c_reset enter_room_c_computed := FALSE
end
// computation of c
event compute_enter_room_c
any c
where
  @enter_room_g_computed enter_room_g_computed = TRUE
  @typ_c  c∈CARD
  @card_g  c∈cards(enter_room_g)
then
  @enter_room_c_done enter_room_c_computed := TRUE
  @enter_room_c_value enter_room_c := c
  // reset dependent parameters
end
// computation of r
event compute_enter_room_r
any r
where
  @typ_r  r∈ROOM
then
  @enter_room_r_done enter_room_r_computed := TRUE
  @enter_room_r_value enter_room_r := r
  // reset dependent parameters
end

event enter_room refines enter_room
when
  // event parameters have been computed
  @enter_room_g_computed enter_room_g_computed = TRUE
  @enter_room_c_computed enter_room_c_computed = TRUE
  @enter_room_r_computed enter_room_r_computed = TRUE
  // guards unused by event parameters
  @empty_r  enter_room_g∉isin(enter_room_r)
  @key_lock  roomk(enter_room_r)=prj2(enter_room_c)
with
  @g g = enter_room_g
  @c c = enter_room_c
  @r r = enter_room_r
then
  @busy_r  isin (enter_room_r) :=isin(enter_room_r)∪{enter_room_g}
  // reset computation state of current event
  // or of dependent parameters of other events
  @get_room_r_init get_room_r_computed := FALSE
  @enter_room_c_init enter_room_c_computed := FALSE
  @enter_room_r_init enter_room_r_computed := FALSE
  @enter_room_g_init enter_room_g_computed := FALSE
end
// unfragmented events
event exit_room
  any g r
  where
    @typ_g g∈GUEST
    @typ_r r∈ROOM
    @busy_r g∈isin(r)
```

```
    then
      @empty_r isin(r) := isin(r)\{g}
      @get_room_r_init get_room_r_computed := FALSE
  end

end
```

Listing C.14 – Machine `Hotel_dep`

C.4 Machine obtenue après distribution

```
machine Hotel_split
// add parameters to non local events
// * non deterministic multi−updates on different components ¬allowed
// → should be transformed before dependency analysis
refines Hotel_dep
sees Hotel_C1
variables
  // inherited variables
  owns // on Desk
  currk // on Desk
  issued // on Desk
  cards // on Guest
  roomk // on Room
  isin // on Guest
  register_g // on Guest
  register_g_computed // on Guest
  register_r // on Desk
  register_r_computed // on Desk
  register_k // on Desk
  register_k_computed // on Desk
  get_room_g // on Guest
  get_room_g_computed // on Guest
  get_room_c // on Guest
  get_room_c_computed // on Guest
  get_room_r // on Guest
  get_room_r_computed // on Guest
  unregister_g // on Guest
  unregister_g_computed // on Guest
  unregister_r // on Guest
  unregister_r_computed // on Guest
  enter_room_g // on Guest
  enter_room_g_computed // on Guest
  enter_room_c // on Guest
  enter_room_c_computed // on Guest
  enter_room_r // on Guest
  enter_room_r_computed // on Guest
  // local copies and status of remote variables
  Guest_owns // on Guest
  owns_fresh // on Desk
  Room_get_room_c // on Room
  get_room_c_fresh // on Guest
  Room_get_room_r // on Room
  get_room_r_fresh // on Guest
  Room_enter_room_r // on Room
  enter_room_r_fresh // on Guest
  Room_enter_room_c // on Room
  enter_room_c_fresh // on Guest
invariants
  @TY_owns   owns∈ROOM↔GUEST // on Desk
  @TY_currk   currk∈ROOM⤔KEY // on Desk
  @TY_issued   issued⊆KEY // on Desk
  @TY_issued   finite(issued) // on Desk
  @TY_cards   cards∈GUEST→ℙ(CARD) // on Guest
  @TY_cards   ∀g1,g2·(cards(g1)∩cards(g2) ≠ ∅ ⇒ g1=g2) // on Guest
  @TY_cards   ∀c1,c2,g1,g2·c1∈cards(g1) ∧ c2∈cards(g2) ∧ prj2(c1)=prj2(c2) ⇒ c1=c2 // on
      Guest
```

@TY_roomk roomk∈ROOM→KEY // *on Room*
@TY_isin isin∈ROOM→ℙ(GUEST) // *on Guest*
@TY_isin isin∈ROOM→ℙ(GUEST) // *on Guest*
@TY_register_g_computed register_g_computed ∈ ℬ // *on Guest*
@TY_register_r_computed register_r_computed ∈ ℬ // *on Desk*
@TY_register_k_computed register_k_computed ∈ ℬ // *on Desk*
@TY_get_room_g_computed get_room_g_computed ∈ ℬ // *on Guest*
@TY_get_room_c_computed get_room_c_computed ∈ ℬ // *on Guest*
@TY_get_room_r_computed get_room_r_computed ∈ ℬ // *on Guest*
@TY_unregister_g_computed unregister_g_computed ∈ ℬ // *on Guest*
@TY_unregister_r_computed unregister_r_computed ∈ ℬ // *on Guest*
@TY_enter_room_g_computed enter_room_g_computed ∈ ℬ // *on Guest*
@TY_enter_room_c_computed enter_room_c_computed ∈ ℬ // *on Guest*
@TY_enter_room_r_computed enter_room_r_computed ∈ ℬ // *on Guest*
@TY_register_g_typ_g register_g∈GUEST // *on Guest*
@TY_register_r_typ_r register_r∈ROOM // *on Desk*
@TY_register_k_typ_k register_k∈KEY // *on Desk*
@TY_get_room_g_typ_g get_room_g∈GUEST // *on Guest*
@TY_get_room_c_typ_c get_room_c∈CARD // *on Guest*
@TY_get_room_r_typ_r get_room_r∈ROOM // *on Guest*
@TY_unregister_g_typ_g unregister_g∈GUEST // *on Guest*
@TY_unregister_r_typ_r unregister_r∈ROOM // *on Guest*
@TY_enter_room_g_typ_g enter_room_g∈GUEST // *on Guest*
@TY_enter_room_c_typ_c enter_room_c∈CARD // *on Guest*
@TY_enter_room_r_typ_r enter_room_r∈ROOM // *on Guest*
@register_g_typ_g register_g_computed = TRUE ⇒ (register_g∈GUEST) // *on Guest*
@register_r_typ_r register_r_computed = TRUE ⇒ (register_r∈ROOM) // *on Desk*
@register_r_free_r register_r_computed = TRUE ⇒ (register_r∉dom(owns)) // *on Desk*
@register_k_typ_k register_k_computed = TRUE ⇒ (register_k∈KEY) // *on Desk*
@register_k_free_k register_k_computed = TRUE ⇒ (register_k∉issued) // *on Desk*
@get_room_g_typ_g get_room_g_computed = TRUE ⇒ (get_room_g∈GUEST) // *on Guest*
@get_room_c_typ_c get_room_c_computed = TRUE ⇒ (get_room_c∈CARD) // *on Guest*
@get_room_c_card_g get_room_c_computed = TRUE ⇒ (get_room_c∈cards(get_room_g)) // *on Guest*
@get_room_r_typ_r get_room_r_computed = TRUE ⇒ (get_room_r∈ROOM) // *on Guest*
@get_room_r_empty_r get_room_r_computed = TRUE ⇒ (isin(get_room_r)=∅) // *on Guest*
@unregister_g_typ_g unregister_g_computed = TRUE ⇒ (unregister_g∈GUEST) // *on Guest*
@unregister_r_typ_r unregister_r_computed = TRUE ⇒ (unregister_r∈ROOM) // *on Guest*
@enter_room_g_typ_g enter_room_g_computed = TRUE ⇒ (enter_room_g∈GUEST) // *on Guest*
@enter_room_c_typ_c enter_room_c_computed = TRUE ⇒ (enter_room_c∈CARD) // *on Guest*
@enter_room_c_card_g enter_room_c_computed = TRUE ⇒ (enter_room_c∈cards(enter_room_g)) // *on Guest*
@enter_room_r_typ_r enter_room_r_computed = TRUE ⇒ (enter_room_r∈ROOM) // *on Guest*
@get_room_g_c get_room_g_computed = FALSE ⇒ get_room_c_computed = FALSE // *on Guest*
@enter_room_g_c enter_room_g_computed = FALSE ⇒ enter_room_c_computed = FALSE // *on Guest*
// *local copies properties*
@Guest_owns_copy owns_fresh = TRUE ⇒ Guest_owns = owns // *will be lost*
// *add typing invariants of owns*
@TY_Guest_owns Guest_owns∈ROOM↔GUEST // *on Guest_owns component*
@Room_get_room_c_copy get_room_c_fresh = TRUE ⇒ Room_get_room_c = get_room_c // *will be lost*
// *add typing invariants of get_room_c*
@TY_Room_get_room_c Room_get_room_c∈CARD // *on Room_get_room_c component*
@Room_get_room_r_copy get_room_r_fresh = TRUE ⇒ Room_get_room_r = get_room_r // *will be lost*
// *add typing invariants of get_room_r*
@TY_Room_get_room_r Room_get_room_r∈ROOM // *on Room_get_room_r component*
@Room_enter_room_r_copy enter_room_r_fresh = TRUE ⇒ Room_enter_room_r = enter_room_r // *will be lost*

```
  // add typing invariants of enter_room_r
  @TY_Room_enter_room_r  Room_enter_room_r∈ROOM // on Room_enter_room_r
      component
  @Room_enter_room_c_copy enter_room_c_fresh = TRUE ⇒ Room_enter_room_c =
      enter_room_c // will be lost
  // add typing invariants of enter_room_c
  @TY_Room_enter_room_c  Room_enter_room_c∈CARD // on Room_enter_room_c
      component
variant
  {FALSE ↦ 1, TRUE ↦ 0}(owns_fresh)+{FALSE ↦ 1, TRUE ↦ 0}(get_room_c_fresh)+{
      FALSE ↦ 1, TRUE ↦ 0}(get_room_r_fresh)+{FALSE ↦ 1, TRUE ↦ 0}(
      enter_room_r_fresh)+{FALSE ↦ 1, TRUE ↦ 0}(enter_room_c_fresh)
events
  // copy of remote variables
  // RQ∈ copy to all targets is useless (but would need to split event and status)
  convergent event share_owns
  any local_owns
  where // on Desk
    @g owns_fresh = FALSE
    @l local_owns = owns
    // add typing invariants of owns
    @TY_local_owns  local_owns∈ROOM↔GUEST
  then
    @to_Guest Guest_owns := local_owns  // on Guest
    @done owns_fresh := TRUE  // on Desk
  end
  convergent event share_get_room_c
  any local_get_room_c
  where // on Guest
    @g get_room_c_fresh = FALSE
    @l local_get_room_c = get_room_c
    // add typing invariants of get_room_c
    @TY_local_get_room_c  local_get_room_c∈CARD
  then
    @to_Room Room_get_room_c := local_get_room_c  // on Room
    @done get_room_c_fresh := TRUE  // on Guest
  end
  convergent event share_get_room_r
  any local_get_room_r
  where // on Guest
    @g get_room_r_fresh = FALSE
    @l local_get_room_r = get_room_r
    // add typing invariants of get_room_r
    @TY_local_get_room_r  local_get_room_r∈ROOM
  then
    @to_Room Room_get_room_r := local_get_room_r  // on Room
    @done get_room_r_fresh := TRUE  // on Guest
  end
  convergent event share_enter_room_r
  any local_enter_room_r
  where // on Guest
    @g enter_room_r_fresh = FALSE
    @l local_enter_room_r = enter_room_r
    // add typing invariants of enter_room_r
    @TY_local_enter_room_r  local_enter_room_r∈ROOM
  then
    @to_Room Room_enter_room_r := local_enter_room_r  // on Room
    @done enter_room_r_fresh := TRUE  // on Guest
  end
  convergent event share_enter_room_c
  any local_enter_room_c
  where // on Guest
    @g enter_room_c_fresh = FALSE
    @l local_enter_room_c = enter_room_c
    // add typing invariants of enter_room_c
    @TY_local_enter_room_c  local_enter_room_c∈CARD
  then
    @to_Room Room_enter_room_c := local_enter_room_c  // on Room
    @done enter_room_c_fresh := TRUE  // on Guest
  end
  // inherited events
  event INITIALISATION refines INITIALISATION
```

```
  // access to remote variables used in actions
  // access to copies of remote variables
  // inherited guards
then
  @register_g_not_computed register_g_computed := FALSE // on Guest
  @register_r_not_computed register_r_computed := FALSE // on Desk
  @register_k_not_computed register_k_computed := FALSE // on Desk
  @get_room_g_not_computed get_room_g_computed := FALSE // on Guest
  @get_room_c_not_computed get_room_c_computed := FALSE // on Guest
  @get_room_r_not_computed get_room_r_computed := FALSE // on Guest
  @unregister_g_not_computed unregister_g_computed := FALSE // on Guest
  @unregister_r_not_computed unregister_r_computed := FALSE // on Guest
  @enter_room_g_not_computed enter_room_g_computed := FALSE // on Guest
  @enter_room_c_not_computed enter_room_c_computed := FALSE // on Guest
  @enter_room_r_not_computed enter_room_r_computed := FALSE // on Guest
   @register_g_init register_g := guest0 // on Guest
   @register_r_init register_r := room0 // on Desk
   @register_k_init register_k := key0 // on Desk
  @get_room_g_init get_room_g := guest0 // on Guest
  @get_room_c_init get_room_c := card0 // on Guest
  @get_room_r_init get_room_r := room0 // on Guest
   @unregister_g_init unregister_g := guest0 // on Guest
   @unregister_r_init unregister_r := room0 // on Guest
  @enter_room_g_init enter_room_g := guest0 // on Guest
  @enter_room_c_init enter_room_c := card0 // on Guest
  @enter_room_r_init enter_room_r := room0 // on Guest
  @init_owns owns := ∅ // on Desk
   @init_currk currk := initk // on Desk
  @init_cards cards := GUEST×{∅} // on Guest
   @init_isin isin := ROOM×{∅} // on Guest
   @init_issued issued := ran( initk ) // on Desk
  @init_roomk roomk := initk // on Room
  // initialize copies and reset freshness state of copies
  @owns_reset owns_fresh := FALSE // on Desk
  @Guest_owns_init Guest_owns :| Guest_owns = owns // on Guest
  @get_room_c_reset get_room_c_fresh := FALSE // on Guest
  @Room_get_room_c_init Room_get_room_c :| Room_get_room_c = get_room_c // on
    Room
  @get_room_r_reset get_room_r_fresh := FALSE // on Guest
  @Room_get_room_r_init Room_get_room_r :| Room_get_room_r = get_room_r // on
   Room
  @enter_room_r_reset enter_room_r_fresh := FALSE // on Guest
  @Room_enter_room_r_init Room_enter_room_r :| Room_enter_room_r = enter_room_r
    // on Room
  @enter_room_c_reset enter_room_c_fresh := FALSE // on Guest
  @Room_enter_room_c_init Room_enter_room_c :| Room_enter_room_c =
   enter_room_c // on Room
end
event compute_register_g refines compute_register_g
any g
where
  // access to remote variables used in actions
  // access to copies of remote variables
  // inherited guards
  @typ_g g∈GUEST // on Guest
then
  @register_g_done register_g_computed := TRUE // on Guest
  @register_g_value register_g := g // on Guest
  // reset freshness state of copies of updated variables
end
event compute_register_r refines compute_register_r
any r
where
  // access to remote variables used in actions
  // access to copies of remote variables
  // inherited guards
  @typ_r r∈ROOM // on Guest
  @free_r r∉dom(owns) // on Desk
then
  @register_r_done register_r_computed := TRUE // on Desk
  @register_r_value register_r := r // on Desk
  // reset freshness state of copies of updated variables
```

```
end
event compute_register_k refines compute_register_k
any k
where
  // access to remote variables used in actions
  // access to copies of remote variables
  // inherited guards
  @register_k_notcomputed register_k_computed = FALSE // on Desk
  @typ_k k∈KEY // on Desk
  @free_k k∉issued // on Desk
then
  @register_k_done register_k_computed := TRUE // on Desk
  @register_k_value register_k := k // on Desk
  // reset freshness state of copies of updated variables
end
event register refines register
any local_register_k local_currk local_register_g local_register_r
where
  // access to remote variables used in actions
  @register_k_access local_register_k = register_k // on Desk
  // add typing invariants of register_k
  @TY_local_register_k local_register_k∈KEY
  @currk_access local_currk = currk // on Desk
  // add typing invariants of currk
  @TY_local_currk local_currk∈ROOM↦KEY
  @register_g_access local_register_g = register_g // on Guest
  // add typing invariants of register_g
  @TY_local_register_g local_register_g∈GUEST
  @register_r_access local_register_r = register_r // on Desk
  // add typing invariants of register_r
  @TY_local_register_r local_register_r ∈ROOM
  // access to copies of remote variables
  // inherited guards
  @register_g_computed register_g_computed = TRUE // on Guest
  @register_r_computed register_r_computed = TRUE // on Desk
  @register_k_computed register_k_computed = TRUE // on Desk
then
  @key_r_desk currk( local_register_r ) := local_register_k // on Desk
  @used_k issued := issued∪{ local_register_k } // on Desk
  @new_card cards(local_register_g) := cards( local_register_g )∪{local_currk(
    local_register_r )↦local_register_k } // on Guest
  @booked_r owns(local_register_r) := local_register_g // on Desk
  @register_g_init register_g_computed := FALSE // on Guest
  @register_r_init register_r_computed := FALSE // on Desk
  @register_k_init register_k_computed := FALSE // on Desk
  @get_room_c_init get_room_c_computed := FALSE // on Guest
  @enter_room_c_init enter_room_c_computed := FALSE // on Guest
  // reset freshness state of copies of updated variables
  @owns_reset owns_fresh := FALSE // on Desk
end
event compute_get_room_g refines compute_get_room_g
any g
where
  // access to remote variables used in actions
  // access to copies of remote variables
  // inherited guards
  @typ_g g∈GUEST // on Guest
then
  @get_room_g_done get_room_g_computed := TRUE // on Guest
  @get_room_g_value get_room_g := g // on Guest
  @c_reset get_room_c_computed := FALSE // on Guest
  // reset freshness state of copies of updated variables
end
event compute_get_room_c refines compute_get_room_c
any c
where
  // access to remote variables used in actions
  // access to copies of remote variables
  // inherited guards
  @get_room_g_computed get_room_g_computed = TRUE // on Guest
  @typ_c c∈CARD // on Guest
  @card_g c∈cards(get_room_g) // on Guest
```

```
then
  @get_room_c_done get_room_c_computed := TRUE // on Guest
  @get_room_c_value get_room_c := c // on Guest
  // reset freshness state of copies of updated variables
  @get_room_c_reset get_room_c_fresh := FALSE // on Guest
end
event compute_get_room_r refines compute_get_room_r
any r
where
  // access to remote variables used in actions
  // access to copies of remote variables
  // inherited guards
  @typ_r  r∈ROOM // on Guest
  @empty_r  isin(r)=∅ // on Guest
then
  @get_room_r_done get_room_r_computed := TRUE // on Guest
  @get_room_r_value get_room_r := r // on Guest
  // reset freshness state of copies of updated variables
  @get_room_r_reset get_room_r_fresh := FALSE // on Guest
end
event get_room refines get_room
when
  // access to remote variables used in actions
  // access to copies of remote variables
  @get_room_c_fresh get_room_c_fresh = TRUE // on Guest
  @get_room_r_fresh get_room_r_fresh = TRUE // on Guest
  // inherited guards
  @get_room_g_computed  get_room_g_computed = TRUE // on Guest
  @get_room_c_computed  get_room_c_computed = TRUE // on Guest
  @get_room_r_computed  get_room_r_computed = TRUE // on Guest
  @key_lock_c1  roomk(Room_get_room_r)=prj1(Room_get_room_c) // on Room
then
  @update_key_lock roomk(Room_get_room_r) := prj2(Room_get_room_c) // on Room
  @get_room_g_init get_room_g_computed := FALSE // on Guest
  @get_room_r_init get_room_r_computed := FALSE // on Guest
  @get_room_c_init get_room_c_computed := FALSE // on Guest
  // reset freshness state of copies of updated variables
end
event compute_unregister_g refines compute_unregister_g
any g
where
  // access to remote variables used in actions
  // access to copies of remote variables
  // inherited guards
  @typ_g  g∈GUEST // on Guest
then
  @unregister_g_done unregister_g_computed := TRUE // on Guest
  @unregister_g_value unregister_g := g // on Guest
  // reset freshness state of copies of updated variables
end
event compute_unregister_r refines compute_unregister_r
any r
where
  // access to remote variables used in actions
  // access to copies of remote variables
  // inherited guards
  @typ_r  r∈ROOM // on Guest
then
  @unregister_r_done unregister_r_computed := TRUE // on Guest
  @unregister_r_value unregister_r := r // on Guest
  // reset freshness state of copies of updated variables
end
event unregister refines unregister
any local_unregister_r local_currk
where
  // access to remote variables used in actions
  @unregister_r_access local_unregister_r = unregister_r // on Guest
  // add typing invariants of unregister_r
  @TY_local_unregister_r  local_unregister_r∈ROOM
  @currk_access local_currk = currk // on Desk
  // add typing invariants of currk
  @TY_local_currk  local_currk∈ROOM↦KEY
```

```
  // access to copies of remote variables
  @owns_fresh owns_fresh = TRUE // on Desk
  // inherited guards
  @unregister_g_computed  unregister_g_computed = TRUE // on Guest
  @unregister_r_computed  unregister_r_computed = TRUE // on Guest
  @booked_r_g   local_unregister_r↦unregister_g∈Guest_owns // on Guest
  @empty_r   unregister_g∉isin( local_unregister_r ) // on Guest
then
  @booked_r owns := {local_unregister_r}⩤owns // on Desk
  @update_card_g cards(unregister_g) := cards(unregister_g)\(KEY×{local_currk(
   local_unregister_r)}) // on Guest
   @register_r_init  register_r_computed :=  FALSE // on Desk
  @get_room_c_init get_room_c_computed :=  FALSE // on Guest
   @unregister_r_init  unregister_r_computed :=  FALSE // on Guest
  @unregister_g_init  unregister_g_computed :=  FALSE // on Guest
  @enter_room_c_init enter_room_c_computed :=  FALSE // on Guest
  // reset freshness state of copies of updated variables
  @owns_reset owns_fresh := FALSE // on Desk
end
event compute_enter_room_g refines compute_enter_room_g
any g
where
  // access to remote variables used in actions
  // access to copies of remote variables
  // inherited guards
  @typ_g   g∈GUEST // on Guest
then
  @enter_room_g_done enter_room_g_computed :=  TRUE // on Guest
  @enter_room_g_value enter_room_g :=  g // on Guest
  @c_reset enter_room_c_computed :=  FALSE // on Guest
  // reset freshness state of copies of updated variables
end
event compute_enter_room_c refines compute_enter_room_c
any c
where
  // access to remote variables used in actions
  // access to copies of remote variables
  // inherited guards
  @enter_room_g_computed  enter_room_g_computed = TRUE // on Guest
  @typ_c   c∈CARD // on Guest
  @card_g   c∈cards(enter_room_g) // on Guest
then
  @enter_room_c_done enter_room_c_computed :=  TRUE // on Guest
  @enter_room_c_value enter_room_c :=  c // on Guest
  // reset freshness state of copies of updated variables
  @enter_room_c_reset enter_room_c_fresh := FALSE // on Guest
end
event compute_enter_room_r refines compute_enter_room_r
any r
where
  // access to remote variables used in actions
  // access to copies of remote variables
  // inherited guards
  @typ_r   r∈ROOM // on Guest
then
  @enter_room_r_done enter_room_r_computed :=  TRUE // on Guest
  @enter_room_r_value enter_room_r :=  r // on Guest
  // reset freshness state of copies of updated variables
  @enter_room_r_reset enter_room_r_fresh := FALSE // on Guest
end
event enter_room refines enter_room
when
  // access to remote variables used in actions
  // access to copies of remote variables
  @enter_room_r_fresh enter_room_r_fresh = TRUE // on Guest
  @enter_room_c_fresh enter_room_c_fresh = TRUE // on Guest
  // inherited guards
  @enter_room_g_computed  enter_room_g_computed = TRUE // on Guest
  @enter_room_c_computed  enter_room_c_computed = TRUE // on Guest
  @enter_room_r_computed  enter_room_r_computed = TRUE // on Guest
  @empty_r   enter_room_g∉isin(enter_room_r) // on Guest
  @key_lock   roomk(Room_enter_room_r)=prj2(Room_enter_room_c) // on Room
```

```
  then
    @busy_r isin(enter_room_r) := isin(enter_room_r)∪{enter_room_g} // on Guest
    @get_room_r_init get_room_r_computed := FALSE // on Guest
    @enter_room_c_init enter_room_c_computed := FALSE // on Guest
    @enter_room_r_init enter_room_r_computed := FALSE // on Guest
    @enter_room_g_init enter_room_g_computed := FALSE // on Guest
    // reset freshness state of copies of updated variables
  end
  event exitRoom refines exitRoom
  any g r
  where
    // access to remote variables used in actions
    // access to copies of remote variables
    // inherited guards
    @typ_g g∈GUEST // on Guest
    @typ_r r∈ROOM // on Guest
    @busy_r g∈isin(r) // on Guest
  then
    @empty_r isin(r) := isin(r)\{g} // on Guest
    @get_room_r_init get_room_r_computed := FALSE // on Guest
    // reset freshness state of copies of updated variables
  end
end
```

Listing C.15 – Machine `Hotel_split` issue de la distribution

C.5 Les sous-composants

C.5.1 Sous-composant `Desk`

```
machine Desk
sees Hotel_C1
variables
  owns
  currk
  issued
  register_r
  register_r_computed
  register_k
  register_k_computed
  // local copies and status of remote variables
  register_k_fresh
  Desk_unregister_r
  register_r_fresh
  Desk_unregister_g
invariants
  @typ_owns owns∈ROOM↔GUEST
  @typ_currk currk∈ROOM⇸KEY
  @typ_issued issued⊆KEY
  @fin_issued finite(issued)
  @key_room_desk ∀r·(r∈ROOM⇒currk(r)∈issued)
  @TY_register_r_computed register_r_computed ∈ ℙ
  @TY_register_k_computed register_k_computed ∈ ℙ
  @TY_register_r_typ_r register_r∈ROOM
  @TY_register_k_typ_k register_k∈KEY
  @register_r_typ_r register_r_computed = TRUE ⇒ (register_r∈ROOM)
  @register_r_free_r register_r_computed = TRUE ⇒ (register_r∉dom(owns))
  @register_k_typ_k register_k_computed = TRUE ⇒ (register_k∈KEY)
  @register_k_free_k register_k_computed = TRUE ⇒ (register_k∉issued)
  @register_k_ty register_k_fresh ∈ ℙ
  @register_r_ty register_r_fresh ∈ ℙ
  // add typing invariants of unregister_r
  @TY_Desk_unregister_r Desk_unregister_r∈ROOM
  // add typing invariants of unregister_g
  @TY_Desk_unregister_g Desk_unregister_g∈GUEST
events
  event share_register_k
```

```
any local_register_k
where
  @g register_k_fresh = FALSE
  @l local_register_k = register_k
then
  @done register_k_fresh := TRUE
end
event share_unregister_r
any local_unregister_r
where
  @TY_local_unregister_r  local_unregister_r∈ROOM
then
  @to_Desk Desk_unregister_r := local_unregister_r
end
event share_register_r
any local_register_r
where
  @g register_r_fresh = FALSE
  @l local_register_r = register_r
then
  @done register_r_fresh := TRUE
end
event share_unregister_g
any local_unregister_g
where
  @TY_local_unregister_g  local_unregister_g∈GUEST
then
  @to_Desk Desk_unregister_g := local_unregister_g
end

event INITIALISATION
then
  @init_owns owns := ∅
  @init_currk currk := initk
  @init_issued issued := ran( initk )
  @register_r_not_computed register_r_computed := FALSE
  @register_k_not_computed register_k_computed := FALSE
  @register_r_init register_r := room0
  @register_k_init register_k := key0
  // initialize copies and reset freshness state of copies
  @register_k_reset register_k_fresh := FALSE
  @Desk_unregister_r_init Desk_unregister_r :∈ ROOM
  @register_r_reset register_r_fresh := FALSE
  @Desk_unregister_g_init Desk_unregister_g :∈ GUEST
end
event compute_register_r
any r
where
  @free_r  r∉dom(owns)
then
  @register_r_done register_r_computed := TRUE
  @register_r_value register_r := r
  // reset freshness state of copies of updated variables
  @register_r_reset register_r_fresh := FALSE
end
event compute_register_k
any k
where
  @register_k_notcomputed register_k_computed = FALSE
  @typ_k  k∈KEY
  @free_k  k∉issued
then
  @register_k_done register_k_computed := TRUE
  @register_k_value register_k := k
  // reset freshness state of copies of updated variables
  @register_k_reset register_k_fresh := FALSE
end
event register
any local_currk local_register_g
where
  @currk_access local_currk = currk
  // add typing invariants of register_g
```

```
    @TY_local_register_g   local_register_g∈GUEST
    // access to copies of remote variables
    @register_k_fresh  register_k_fresh  = TRUE
    @register_r_fresh  register_r_fresh  = TRUE
    // inherited guards
    @register_r_computed  register_r_computed = TRUE
    @free_r    register_r∉dom(owns)
    @register_k_computed  register_k_computed = TRUE
    @typ_k   register_k∈KEY
    @free_k    register_k∉issued
  then
    @key_r_desk currk(register_r) := register_k
    @used_k issued := issued∪{register_k}
    @booked_r owns(register_r) := local_register_g
    @register_r_init  register_r_computed := FALSE
    @register_k_init  register_k_computed := FALSE
  end
  event unregister
  any local_currk
  where
    @currk_access local_currk = currk
    // access to copies of remote variables
    // inherited guards
    @booked_r_g   Desk_unregister_r↦Desk_unregister_g∈owns
  then
    @booked_r owns := {Desk_unregister_r}⩤owns
    @register_r_init  register_r_computed := FALSE
  end
end
```

Listing C.16 – Machine `Desk`

C.5.2 Sous-composant `Guest`

```
machine Guest sees Hotel_C1

variables  cards  isin  register_g  register_g_computed get_room_g get_room_g_computed
      get_room_c get_room_c_computed get_room_r get_room_r_computed unregister_g
      unregister_g_computed unregister_r unregister_r_computed enter_room_g
      enter_room_g_computed enter_room_c enter_room_c_computed enter_room_r
           enter_room_r_computed // local copies and status of remote variables
           Guest_register_k enter_room_c_fresh unregister_r_fresh Guest_register_r
      enter_room_r_fresh get_room_r_fresh unregister_g_fresh get_room_c_fresh

invariants
  @typ_isin  isin ∈ROOM→ℙ(GUEST)
  @typ_cards cards∈GUEST→ℙ(CARD)
  @uniq_card1 ∀g1,g2·(cards(g1)∩cards(g2) ≠ ∅ ⇒ g1=g2)
  @uniq_card2 ∀c1,c2,g1,g2·c1∈cards(g1) ∧ c2∈cards(g2) ∧ prj2(c1)=prj2(c2) ⇒ c1=c2
  @TY_register_g_computed register_g_computed ∈ ℬ
  @TY_get_room_g_computed get_room_g_computed ∈ ℬ
  @TY_get_room_c_computed get_room_c_computed ∈ ℬ
  @TY_get_room_r_computed get_room_r_computed ∈ ℬ
  @TY_unregister_g_computed unregister_g_computed ∈ ℬ
  @TY_unregister_r_computed unregister_r_computed ∈ ℬ
  @TY_enter_room_g_computed enter_room_g_computed ∈ ℬ
  @TY_enter_room_c_computed enter_room_c_computed ∈ ℬ
  @TY_enter_room_r_computed enter_room_r_computed ∈ ℬ
  @TY_register_g_typ_g register_g∈GUEST
  @TY_get_room_g_typ_g get_room_g∈GUEST
  @TY_get_room_c_typ_c get_room_c∈CARD
  @TY_get_room_r_typ_r get_room_r∈ROOM
  @TY_unregister_g_typ_g unregister_g∈GUEST
  @TY_unregister_r_typ_r unregister_r∈ROOM
  @TY_enter_room_g_typ_g enter_room_g∈GUEST
  @TY_enter_room_c_typ_c enter_room_c∈CARD
  @TY_enter_room_r_typ_r enter_room_r∈ROOM
  @register_g_typ_g register_g_computed = TRUE ⇒ ( register_g∈GUEST)
```

```
  @get_room_g_typ_g get_room_g_computed = TRUE ⇒ ( get_room_g∈GUEST)
  @get_room_c_typ_c get_room_c_computed = TRUE ⇒ ( get_room_c∈CARD)
  @get_room_c_card_g get_room_c_computed = TRUE ⇒ ( get_room_c∈cards(get_room_g
     ))
  @get_room_r_typ_r get_room_r_computed = TRUE ⇒ ( get_room_r∈ROOM)
  @get_room_r_empty_r get_room_r_computed = TRUE ⇒ ( isin(get_room_r)=∅)
  @unregister_g_typ_g unregister_g_computed = TRUE ⇒ ( unregister_g∈GUEST)
  @unregister_r_typ_r unregister_r_computed = TRUE ⇒ ( unregister_r∈ROOM)
  @enter_room_g_typ_g enter_room_g_computed = TRUE ⇒ ( enter_room_g∈GUEST)
  @enter_room_c_typ_c enter_room_c_computed = TRUE ⇒ ( enter_room_c∈CARD)
  @enter_room_c_card_g enter_room_c_computed = TRUE ⇒ ( enter_room_c∈cards(
     enter_room_g))
  @enter_room_r_typ_r enter_room_r_computed = TRUE ⇒ ( enter_room_r∈ROOM)
  @get_room_g_c get_room_g_computed = FALSE ⇒ get_room_c_computed = FALSE
  @enter_room_g_c enter_room_g_computed = FALSE ⇒ enter_room_c_computed =
     FALSE
  @enter_room_c_ty enter_room_c_fresh ∈ ℙ
  @unregister_r_ty  unregister_r_fresh  ∈ ℙ
  @enter_room_r_ty enter_room_r_fresh ∈ ℙ
  @get_room_r_ty get_room_r_fresh ∈ ℙ
  @unregister_g_ty unregister_g_fresh  ∈ ℙ
  @get_room_c_ty get_room_c_fresh ∈ ℙ // add typing invariants of register_k
  @TY_Guest_register_k Guest_register_k∈KEY // add typing invariants of register_r
  @TY_Guest_register_r Guest_register_r∈ROOM

events
  event share_register_k
    any local_register_k
    where
      @TY_local_register_k local_register_k∈KEY
    then
      @to_Guest Guest_register_k := local_register_k
  end

  event share_enter_room_c
    any local_enter_room_c
    where
      @g enter_room_c_fresh = FALSE
      @l local_enter_room_c = enter_room_c
    then
      @done enter_room_c_fresh := TRUE
  end

  event share_unregister_r
    any local_unregister_r
    where
      @g unregister_r_fresh  = FALSE
      @l  local_unregister_r  = unregister_r
    then
      @done unregister_r_fresh  := TRUE
  end

  event share_register_r
    any local_register_r
    where
      @TY_local_register_r local_register_r∈ROOM
    then
      @to_Guest Guest_register_r := local_register_r
  end

  event share_enter_room_r
    any local_enter_room_r
    where
      @g enter_room_r_fresh = FALSE
      @l local_enter_room_r = enter_room_r
    then
      @done enter_room_r_fresh := TRUE
  end

  event share_get_room_r
    any local_get_room_r
    where
```

```
    @g get_room_r_fresh = FALSE
    @l local_get_room_r = get_room_r
  then
    @done get_room_r_fresh := TRUE
end

event share_unregister_g
  any local_unregister_g
  where
    @g unregister_g_fresh = FALSE
    @l local_unregister_g = unregister_g
  then
    @done unregister_g_fresh := TRUE
end

event share_get_room_c
  any local_get_room_c
  where
    @g get_room_c_fresh = FALSE
    @l local_get_room_c = get_room_c
  then
    @done get_room_c_fresh := TRUE
end

event INITIALISATION
  then
    @init_cards cards := GUEST×{∅}
    @init_isin isin := ROOM×{∅}
    @register_g_not_computed register_g_computed := FALSE
    @get_room_g_not_computed get_room_g_computed := FALSE
    @get_room_c_not_computed get_room_c_computed := FALSE
    @get_room_r_not_computed get_room_r_computed := FALSE
    @unregister_g_not_computed unregister_g_computed := FALSE
    @unregister_r_not_computed unregister_r_computed := FALSE
    @enter_room_g_not_computed enter_room_g_computed := FALSE
    @enter_room_c_not_computed enter_room_c_computed := FALSE
    @enter_room_r_not_computed enter_room_r_computed := FALSE
    @register_g_init register_g := guest0
    @get_room_g_init get_room_g := guest0
    @get_room_c_init get_room_c := card0
    @get_room_r_init get_room_r := room0
    @unregister_g_init unregister_g := guest0
    @unregister_r_init unregister_r := room0
    @enter_room_g_init enter_room_g := guest0
    @enter_room_c_init enter_room_c := card0
    @enter_room_r_init enter_room_r := room0
    @enter_room_c_reset enter_room_c_fresh := FALSE
    @unregister_r_reset unregister_r_fresh := FALSE
    @enter_room_r_reset enter_room_r_fresh := FALSE
    @get_room_r_reset get_room_r_fresh := FALSE
    @unregister_g_reset unregister_g_fresh := FALSE
    @get_room_c_reset get_room_c_fresh := FALSE
end

event compute_register_g
  any g
  where
    @typ_g g∈GUEST
  then
    @register_g_done register_g_computed := TRUE
    @register_g_value register_g := g
end

event register
  any local_currk local_register_g
  where
    @register_g_access local_register_g = register_g
    @TY_local_currk local_currk∈ROOM↦KEY
    @register_g_computed register_g_computed = TRUE
    @typ_g local_register_g∈GUEST
    @typ_r Guest_register_r∈ROOM
    @k_card Guest_register_k ∉ ran(union(ran(cards)))
```

```
  then
    @new_card cards(local_register_g) := cards( local_register_g )∪{local_currk(
    Guest_register_r)↦Guest_register_k}
     @register_g_init register_g_computed := FALSE
    @get_room_c_init get_room_c_computed := FALSE
    @enter_room_c_init enter_room_c_computed := FALSE
end

event compute_get_room_g
  any g
  where
    @typ_g g∈GUEST
  then
    @get_room_g_done get_room_g_computed := TRUE
    @get_room_g_value get_room_g := g
    @c_reset get_room_c_computed := FALSE
end

event compute_get_room_c
  any c
  where
    @get_room_g_computed get_room_g_computed = TRUE
    @card_g c∈cards(get_room_g)
  then
    @get_room_c_done get_room_c_computed := TRUE
    @get_room_c_value get_room_c := c
    @get_room_c_reset get_room_c_fresh := FALSE
end

event compute_get_room_r
  any r
  where
    @typ_r r∈ROOM
    @empty_r isin(r)=∅
  then
    @get_room_r_done get_room_r_computed := TRUE
    @get_room_r_value get_room_r := r
    @get_room_r_reset get_room_r_fresh := FALSE
end

event get_room
  where
    @get_room_r_fresh get_room_r_fresh = TRUE
    @get_room_c_fresh get_room_c_fresh = TRUE
    @get_room_g_computed get_room_g_computed = TRUE
    @typ_g get_room_g∈GUEST
    @get_room_c_computed get_room_c_computed = TRUE
    @typ_c get_room_c∈CARD
    @card_g get_room_c∈cards(get_room_g)
    @get_room_r_computed get_room_r_computed = TRUE
    @typ_r get_room_r∈ROOM
    @empty_r isin(get_room_r)=∅
  then
    @get_room_g_init get_room_g_computed := FALSE
    @get_room_c_init get_room_c_computed := FALSE
    @get_room_r_init get_room_r_computed := FALSE
end

event compute_unregister_g
  any g
  where
    @typ_g g∈GUEST
  then
    @unregister_g_done unregister_g_computed := TRUE
    @unregister_g_value unregister_g := g
    @unregister_g_reset unregister_g_fresh := FALSE
end

event compute_unregister_r
  any r
  where
    @typ_r r∈ROOM
```

```
  then
    @unregister_r_done unregister_r_computed := TRUE
    @unregister_r_value unregister_r := r
    @unregister_r_reset unregister_r_fresh := FALSE
end

event unregister
  any local_currk
  where
    @TY_local_currk local_currk∈ROOM↦KEY
    @unregister_r_fresh unregister_r_fresh = TRUE
    @unregister_g_fresh unregister_g_fresh = TRUE
    @unregister_g_computed unregister_g_computed = TRUE
    @typ_g unregister_g∈GUEST
    @unregister_r_computed unregister_r_computed = TRUE
    @typ_r unregister_r∈ROOM
    @empty_r unregister_g∉isin(unregister_r)
  then
    @update_card_g cards(unregister_g) := cards(unregister_g)\(KEY×{local_currk(
   unregister_r)})
    @get_room_c_init get_room_c_computed := FALSE
    @unregister_g_init unregister_g_computed := FALSE
    @unregister_r_init unregister_r_computed := FALSE
    @enter_room_c_init enter_room_c_computed := FALSE
end

event compute_enter_room_g
  any g
  where
    @typ_g g∈GUEST
  then
    @enter_room_g_done enter_room_g_computed := TRUE
    @enter_room_g_value enter_room_g := g
    @c_reset enter_room_c_computed := FALSE
end

event compute_enter_room_c
  any c
  where
    @enter_room_g_computed enter_room_g_computed = TRUE
    @card_g c∈cards(enter_room_g)
  then
    @enter_room_c_done enter_room_c_computed := TRUE
    @enter_room_c_value enter_room_c := c
    @enter_room_c_reset enter_room_c_fresh := FALSE
end

event compute_enter_room_r
  any r
  where
    @typ_r r∈ROOM
  then
    @enter_room_r_done enter_room_r_computed := TRUE
    @enter_room_r_value enter_room_r := r
    @enter_room_r_reset enter_room_r_fresh := FALSE
end

event enter_room
  where
    @enter_room_c_fresh enter_room_c_fresh = TRUE
    @enter_room_r_fresh enter_room_r_fresh = TRUE
    @enter_room_g_computed enter_room_g_computed = TRUE
    @typ_g enter_room_g∈GUEST
    @enter_room_c_computed enter_room_c_computed = TRUE
    @typ_c enter_room_c∈CARD
    @card_g enter_room_c∈cards(enter_room_g)
    @enter_room_r_computed enter_room_r_computed = TRUE
    @typ_r enter_room_r∈ROOM
    @empty_r enter_room_g∉isin(enter_room_r)
  then
    @busy_r isin(enter_room_r) := isin(enter_room_r)∪{enter_room_g}
    @get_room_r_init get_room_r_computed := FALSE
```

```
      @enter_room_c_init enter_room_c_computed := FALSE
      @enter_room_r_init enter_room_r_computed := FALSE
      @enter_room_g_init enter_room_g_computed := FALSE
  end

  event exit_room
    any g r
    where
      @typ_g g∈GUEST
      @typ_r r∈ROOM
      @busy_r g∈isin(r)
    then
      @empty_r isin(r) := isin(r)\{g}
      @get_room_r_init get_room_r_computed := FALSE
  end
end
```

Listing C.17 – Machine `Guest`

C.6 Code BIP généré

```
package root

// sets of machine context
extern data type GUEST as "int"
extern operator bool ==(GUEST, GUEST)
extern data type ROOM as "int"
extern operator bool ==(ROOM, ROOM)
extern data type KEY as "int"
extern operator bool ==(KEY, KEY)

// defined sets
extern data type CARD as "std::pair<KEY,KEY>"
extern function KEY prj1(CARD)
extern function KEY prj2(CARD)
extern function CARD pair(KEY, KEY)
extern operator bool ==(CARD, CARD)
extern data type SETOF_ROOM as "std::set<ROOM>"
extern function SETOF_ROOM set_union(SETOF_ROOM,SETOF_ROOM)
extern function SETOF_ROOM set_diff(SETOF_ROOM,SETOF_ROOM)
extern function bool set_in(ROOM,SETOF_ROOM)
extern function bool set_included(SETOF_ROOM,SETOF_ROOM)
extern function SETOF_ROOM set_sing(ROOM)
extern function clear(SETOF_ROOM)
extern operator bool ==(SETOF_ROOM, SETOF_ROOM)
extern data type SETOF_GUEST as "std::set<GUEST>"
extern function SETOF_GUEST set_union(SETOF_GUEST,SETOF_GUEST)
extern function SETOF_GUEST set_diff(SETOF_GUEST,SETOF_GUEST)
extern function bool set_in(GUEST,SETOF_GUEST)
extern function bool set_included(SETOF_GUEST,SETOF_GUEST)
extern function SETOF_GUEST set_sing(GUEST)
extern function clear(SETOF_GUEST)
extern operator bool ==(SETOF_GUEST, SETOF_GUEST)
extern data type PAIR_ROOM_GUEST as "std::pair<ROOM,GUEST>"
extern function ROOM prj1(PAIR_ROOM_GUEST)
extern function GUEST prj2(PAIR_ROOM_GUEST)
extern function PAIR_ROOM_GUEST pair(ROOM, GUEST)
extern operator bool ==(PAIR_ROOM_GUEST, PAIR_ROOM_GUEST)
extern data type MAP_ROOM_GUEST as "std::map<ROOM,GUEST>"
extern function GUEST get(MAP_ROOM_GUEST,ROOM)
extern function put(MAP_ROOM_GUEST,ROOM,GUEST)
extern function SETOF_ROOM dom(MAP_ROOM_GUEST)
extern function SETOF_GUEST ran(MAP_ROOM_GUEST)
extern function MAP_ROOM_GUEST dom_sub(SETOF_ROOM, MAP_ROOM_GUEST)
extern function MAP_ROOM_GUEST ran_sub(MAP_ROOM_GUEST, SETOF_GUEST)
extern operator bool ==(MAP_ROOM_GUEST, MAP_ROOM_GUEST)
extern function MAP_ROOM_GUEST set_union(MAP_ROOM_GUEST,MAP_ROOM_GUEST)
```

```
extern function MAP_ROOM_GUEST set_diff(MAP_ROOM_GUEST,MAP_ROOM_GUEST)
extern function bool set_in(PAIR_ROOM_GUEST,MAP_ROOM_GUEST)
extern function bool set_included(MAP_ROOM_GUEST,MAP_ROOM_GUEST)
extern function MAP_ROOM_GUEST set_sing(PAIR_ROOM_GUEST)
extern function clear(MAP_ROOM_GUEST)
extern data type SETOF_KEY as "std::set<KEY>"
extern function SETOF_KEY set_union(SETOF_KEY,SETOF_KEY)
extern function SETOF_KEY set_diff(SETOF_KEY,SETOF_KEY)
extern function bool set_in(KEY,SETOF_KEY)
extern function bool set_included(SETOF_KEY,SETOF_KEY)
extern function SETOF_KEY set_sing(KEY)
extern function clear(SETOF_KEY)
extern operator bool ==(SETOF_KEY, SETOF_KEY)
extern data type PAIR_ROOM_KEY as "std::pair<ROOM,KEY>"
extern function ROOM prj1(PAIR_ROOM_KEY)
extern function KEY prj2(PAIR_ROOM_KEY)
extern function PAIR_ROOM_KEY pair(ROOM, KEY)
extern operator bool ==(PAIR_ROOM_KEY, PAIR_ROOM_KEY)
extern data type MAP_ROOM_KEY as "std::map<ROOM,KEY>"
extern function KEY get(MAP_ROOM_KEY,ROOM)
extern function put(MAP_ROOM_KEY,ROOM,KEY)
extern function SETOF_ROOM dom(MAP_ROOM_KEY)
extern function SETOF_KEY ran(MAP_ROOM_KEY)
extern function MAP_ROOM_KEY dom_sub(SETOF_ROOM, MAP_ROOM_KEY)
extern function MAP_ROOM_KEY ran_sub(MAP_ROOM_KEY, SETOF_KEY)
extern operator bool ==(MAP_ROOM_KEY, MAP_ROOM_KEY)
extern function MAP_ROOM_KEY set_union(MAP_ROOM_KEY,MAP_ROOM_KEY)
extern function MAP_ROOM_KEY set_diff(MAP_ROOM_KEY,MAP_ROOM_KEY)
extern function bool set_in(PAIR_ROOM_KEY,MAP_ROOM_KEY)
extern function bool set_included(MAP_ROOM_KEY,MAP_ROOM_KEY)
extern function MAP_ROOM_KEY set_sing(PAIR_ROOM_KEY)
extern function clear(MAP_ROOM_KEY)
extern data type SETOF_CARD as "std::map<KEY,KEY>"
extern function KEY get(SETOF_CARD,KEY)
extern function put(SETOF_CARD,KEY,KEY)
extern function SETOF_KEY dom(SETOF_CARD)
extern function SETOF_KEY ran(SETOF_CARD)
extern function SETOF_CARD dom_sub(SETOF_KEY, SETOF_CARD)
extern function SETOF_CARD ran_sub(SETOF_CARD, SETOF_KEY)
extern operator bool ==(SETOF_CARD, SETOF_CARD)
extern function SETOF_CARD set_union(SETOF_CARD,SETOF_CARD)
extern function SETOF_CARD set_diff(SETOF_CARD,SETOF_CARD)
extern function bool set_in(CARD,SETOF_CARD)
extern function bool set_included(SETOF_CARD,SETOF_CARD)
extern function SETOF_CARD set_sing(CARD)
extern function clear(SETOF_CARD)
extern data type SETOF_SETOF_CARD as "std::set<SETOF_CARD>"
extern function SETOF_SETOF_CARD set_union(SETOF_SETOF_CARD,SETOF_SETOF_CARD)
extern function SETOF_SETOF_CARD set_diff(SETOF_SETOF_CARD,SETOF_SETOF_CARD)
extern function bool set_in(SETOF_CARD,SETOF_SETOF_CARD)
extern function bool set_included(SETOF_SETOF_CARD,SETOF_SETOF_CARD)
extern function SETOF_SETOF_CARD set_sing(SETOF_CARD)
extern function clear(SETOF_SETOF_CARD)
extern function SETOF_CARD set_kunion(SETOF_SETOF_CARD)
extern operator bool ==(SETOF_SETOF_CARD, SETOF_SETOF_CARD)
extern data type PAIR_GUEST_SETOF_CARD as "std::pair<GUEST,SETOF_CARD>"
extern function GUEST prj1(PAIR_GUEST_SETOF_CARD)
extern function SETOF_CARD prj2(PAIR_GUEST_SETOF_CARD)
extern function PAIR_GUEST_SETOF_CARD pair(GUEST, SETOF_CARD)
extern operator bool ==(PAIR_GUEST_SETOF_CARD, PAIR_GUEST_SETOF_CARD)
extern data type MAP_GUEST_SETOF_CARD as "std::map<GUEST,SETOF_CARD>"
extern function SETOF_CARD get(MAP_GUEST_SETOF_CARD,GUEST)
extern function put(MAP_GUEST_SETOF_CARD,GUEST,SETOF_CARD)
extern function SETOF_GUEST dom(MAP_GUEST_SETOF_CARD)
extern function SETOF_SETOF_CARD ran(MAP_GUEST_SETOF_CARD)
extern function MAP_GUEST_SETOF_CARD dom_sub(SETOF_GUEST, MAP_GUEST_SETOF_CARD
    )
extern function MAP_GUEST_SETOF_CARD ran_sub(MAP_GUEST_SETOF_CARD,
    SETOF_SETOF_CARD)
extern operator bool ==(MAP_GUEST_SETOF_CARD, MAP_GUEST_SETOF_CARD)
extern function MAP_GUEST_SETOF_CARD set_union(MAP_GUEST_SETOF_CARD,
    MAP_GUEST_SETOF_CARD)
```

```
extern function MAP_GUEST_SETOF_CARD set_diff(MAP_GUEST_SETOF_CARD,
    MAP_GUEST_SETOF_CARD)
extern function bool set_in(PAIR_GUEST_SETOF_CARD,MAP_GUEST_SETOF_CARD)
extern function bool set_included(MAP_GUEST_SETOF_CARD,MAP_GUEST_SETOF_CARD)
extern function MAP_GUEST_SETOF_CARD set_sing(PAIR_GUEST_SETOF_CARD)
extern function clear(MAP_GUEST_SETOF_CARD)
extern data type SETOF_SETOF_GUEST as "std::set<SETOF_GUEST>"
extern function SETOF_SETOF_GUEST set_union(SETOF_SETOF_GUEST,SETOF_SETOF_GUEST)
extern function SETOF_SETOF_GUEST set_diff(SETOF_SETOF_GUEST,SETOF_SETOF_GUEST)
extern function bool set_in(SETOF_GUEST,SETOF_SETOF_GUEST)
extern function bool set_included(SETOF_SETOF_GUEST,SETOF_SETOF_GUEST)
extern function SETOF_SETOF_GUEST set_sing(SETOF_GUEST)
extern function clear(SETOF_SETOF_GUEST)
extern function SETOF_GUEST set_kunion(SETOF_SETOF_GUEST)
extern operator bool ==(SETOF_SETOF_GUEST, SETOF_SETOF_GUEST)
extern data type PAIR_ROOM_SETOF_GUEST as "std::pair<ROOM,SETOF_GUEST>"
extern function ROOM prj1(PAIR_ROOM_SETOF_GUEST)
extern function SETOF_GUEST prj2(PAIR_ROOM_SETOF_GUEST)
extern function PAIR_ROOM_SETOF_GUEST pair(ROOM, SETOF_GUEST)
extern operator bool ==(PAIR_ROOM_SETOF_GUEST, PAIR_ROOM_SETOF_GUEST)
extern data type MAP_ROOM_SETOF_GUEST as "std::map<ROOM,SETOF_GUEST>"
extern function SETOF_GUEST get(MAP_ROOM_SETOF_GUEST,ROOM)
extern function put(MAP_ROOM_SETOF_GUEST,ROOM,SETOF_GUEST)
extern function SETOF_ROOM dom(MAP_ROOM_SETOF_GUEST)
extern function SETOF_SETOF_GUEST ran(MAP_ROOM_SETOF_GUEST)
extern function MAP_ROOM_SETOF_GUEST dom_sub(SETOF_ROOM,
    MAP_ROOM_SETOF_GUEST)
extern function MAP_ROOM_SETOF_GUEST ran_sub(MAP_ROOM_SETOF_GUEST,
    SETOF_SETOF_GUEST)
extern operator bool ==(MAP_ROOM_SETOF_GUEST, MAP_ROOM_SETOF_GUEST)
extern function MAP_ROOM_SETOF_GUEST set_union(MAP_ROOM_SETOF_GUEST,
    MAP_ROOM_SETOF_GUEST)
extern function MAP_ROOM_SETOF_GUEST set_diff(MAP_ROOM_SETOF_GUEST,
    MAP_ROOM_SETOF_GUEST)
extern function bool set_in(PAIR_ROOM_SETOF_GUEST,MAP_ROOM_SETOF_GUEST)
extern function bool set_included(MAP_ROOM_SETOF_GUEST,MAP_ROOM_SETOF_GUEST)
extern function MAP_ROOM_SETOF_GUEST set_sing(PAIR_ROOM_SETOF_GUEST)
extern function clear(MAP_ROOM_SETOF_GUEST)

extern function MAP_GUEST_SETOF_CARD PAIR_GUEST_set_sing_empty()
extern function MAP_ROOM_SETOF_GUEST PAIR_ROOM_set_sing_empty()

// constants of machine context
extern function MAP_ROOM_KEY initk()
extern function GUEST guest0()
extern function ROOM room0()
extern function KEY key0()
extern function CARD card0()

/* ports */
port type ty_empty_port()
port type ty_share_get_room_r_Guest(ROOM get_room_r)
port type ty_share_get_room_r_Room(ROOM get_room_r)
port type ty_share_enter_room_c_Guest(CARD enter_room_c)
port type ty_share_enter_room_c_Room(CARD enter_room_c)
port type ty_share_register_k_Desk(KEY register_k)
port type ty_share_register_k_Guest(KEY register_k)
port type ty_share_enter_room_r_Guest(ROOM enter_room_r)
port type ty_share_enter_room_r_Room(ROOM enter_room_r)
port type ty_share_get_room_c_Guest(CARD get_room_c)
port type ty_share_get_room_c_Room(CARD get_room_c)
port type ty_share_unregister_g_Desk(GUEST unregister_g)
port type ty_share_unregister_g_Guest(GUEST unregister_g)
port type ty_share_unregister_r_Desk(ROOM unregister_r)
port type ty_share_unregister_r_Guest(ROOM unregister_r)
port type ty_register_Desk(MAP_ROOM_KEY currk,GUEST register_g,ROOM register_r)
port type ty_register_Guest(MAP_ROOM_KEY currk,GUEST register_g,ROOM register_r)
port type ty_unregister_Desk(MAP_ROOM_KEY currk)
port type ty_unregister_Guest(MAP_ROOM_KEY currk)

/* connecteurs */
```

```
connector type ty_share_get_room_r (ty_share_get_room_r_Guest Guest,ty_share_get_room_r_Room
    Room)
  define Guest Room
  on Guest Room
  down { Room.get_room_r = Guest.get_room_r; }
end
connector type ty_share_enter_room_c (ty_share_enter_room_c_Guest Guest,
    ty_share_enter_room_c_Room Room)
  define Guest Room
  on Guest Room
  down { Room.enter_room_c = Guest.enter_room_c; }
end
connector type ty_share_register_k (ty_share_register_k_Desk Desk,ty_share_register_k_Guest Guest)
  define Desk Guest
  on Desk Guest
  down { Guest.register_k = Desk.register_k; }
end
connector type ty_share_enter_room_r (ty_share_enter_room_r_Guest Guest,
    ty_share_enter_room_r_Room Room)
  define Guest Room
  on Guest Room
  down { Room.enter_room_r = Guest.enter_room_r; }
end
connector type ty_share_get_room_c (ty_share_get_room_c_Guest Guest,ty_share_get_room_c_Room
    Room)
  define Guest Room
  on Guest Room
  down { Room.get_room_c = Guest.get_room_c; }
end
connector type ty_share_unregister_g (ty_share_unregister_g_Desk Desk,ty_share_unregister_g_Guest
    Guest)
  define Desk Guest
  on Desk Guest
  down { Desk.unregister_g = Guest.unregister_g; }
end
connector type ty_share_unregister_r (ty_share_unregister_r_Desk Desk,ty_share_unregister_r_Guest
    Guest)
  define Desk Guest
  on Desk Guest
  down { Desk.unregister_r = Guest.unregister_r; }
end
connector type ty_compute_register_g (ty_empty_port Guest)
  define Guest
  on Guest
end
connector type ty_compute_register_r (ty_empty_port Desk)
  define Desk
  on Desk
end
connector type ty_compute_register_k (ty_empty_port Desk)
  define Desk
  on Desk
end
connector type ty_register (ty_register_Desk Desk,ty_register_Guest Guest)
  define Desk Guest
  on Desk Guest
  down { Desk.register_g = Guest.register_g;
    Guest.currk = Desk.currk;
    Guest. register_r = Desk.register_r;
  }
end
connector type ty_compute_get_room_g (ty_empty_port Guest)
  define Guest
  on Guest
end
connector type ty_compute_get_room_c (ty_empty_port Guest)
  define Guest
  on Guest
end
connector type ty_compute_get_room_r (ty_empty_port Guest)
  define Guest
  on Guest
```

```
end
connector type ty_get_room (ty_empty_port Guest,ty_empty_port Room)
  define Guest Room
  on Guest Room
end
connector type ty_compute_unregister_g (ty_empty_port Guest)
  define Guest
  on Guest
end
connector type ty_compute_unregister_r (ty_empty_port Guest)
  define Guest
  on Guest
end
connector type ty_unregister (ty_unregister_Desk Desk,ty_unregister_Guest Guest)
  define Desk Guest
  on Desk Guest
  down { Guest.currk = Desk.currk; }
end
connector type ty_compute_enter_room_g (ty_empty_port Guest)
  define Guest
  on Guest
end
connector type ty_compute_enter_room_c (ty_empty_port Guest)
  define Guest
  on Guest
end
connector type ty_compute_enter_room_r (ty_empty_port Guest)
  define Guest
  on Guest
end
connector type ty_enter_room (ty_empty_port Guest,ty_empty_port Room)
  define Guest Room
  on Guest Room
end
connector type ty_exit_room (ty_empty_port Guest)
  define Guest
  on Guest
end

atom type ty_Desk()
  /* state variables */
  data MAP_ROOM_GUEST owns
  data MAP_ROOM_KEY currk
  data SETOF_KEY issued
  data ROOM register_r
  data bool register_r_computed
  data KEY register_k
  data bool register_k_computed
  data bool register_k_fresh
  data GUEST Desk_unregister_g
  data ROOM Desk_unregister_r
  /* temporary variables updated by connectors */
  data GUEST register_g
  data GUEST unregister_g
  data ROOM unregister_r
  /* ports */
  export port ty_share_register_k_Desk share_register_k( register_k )
  export port ty_share_unregister_g_Desk share_unregister_g(Desk_unregister_g)
  export port ty_share_unregister_r_Desk share_unregister_r(Desk_unregister_r)
  export port ty_empty_port compute_register_r()
  export port ty_empty_port compute_register_k()
  export port ty_register_Desk register (currk , register_g , register_r )
  export port ty_unregister_Desk unregister (currk)

  place P0
  /* etat initial */
   initial to P0
  do {
     clear (owns);
     currk = initk ();
     issued = ran( initk ());
     register_r_computed = false;
```

```
    register_k_computed = false;
    register_r  = room0();
    register_k  = key0();
    register_k_fresh  = false;
  }

  /* transitions */
  on compute_register_r from P0 to P0
  provided ( (register_r_computed == false) &&
    !set_in(room0(),dom(owns)) )
  do { register_r_computed = true;
    register_r  = room0(); }

  on compute_register_k from P0 to P0
  provided ( (register_k_computed == false) &&
    !set_in(key0(), issued) )
  do { register_k_computed = true;
    register_k  = key0();
    register_k_fresh  = false; }

  on register from P0 to P0
  provided ( (register_k_fresh == true) &&
    (register_r_computed == true) &&
    !set_in( register_r ,dom(owns)) &&
    (register_k_computed == true) &&
    !set_in( register_k , issued) )
  do { put(currk, register_r, register_k);
    issued = set_union(issued, set_sing( register_k ));
    put(owns, register_r, register_g);
    register_k_computed = false;
    register_r_computed = false; }

  on unregister from P0 to P0
  provided ( set_in( pair(Desk_unregister_r, Desk_unregister_g),owns) )
  do { owns = dom_sub(set_sing(Desk_unregister_r),owns);
    register_r_computed = false; }

  /* synchro on share_ events */
  on share_register_k from P0 to P0 do { register_k_fresh = true; }
  on share_unregister_g from P0 to P0
  on share_unregister_r from P0 to P0
end
atom type ty_Guest()
  /* state variables */
  data MAP_GUEST_SETOF_CARD cards
  data MAP_ROOM_SETOF_GUEST isin
  data GUEST register_g
  data bool register_g_computed
  data GUEST get_room_g
  data bool get_room_g_computed
  data CARD get_room_c
  data bool get_room_c_computed
  data ROOM get_room_r
  data bool get_room_r_computed
  data GUEST unregister_g
  data bool unregister_g_computed
  data ROOM unregister_r
  data bool unregister_r_computed
  data GUEST enter_room_g
  data bool enter_room_g_computed
  data CARD enter_room_c
  data bool enter_room_c_computed
  data ROOM enter_room_r
  data bool enter_room_r_computed
  data bool get_room_r_fresh
  data bool enter_room_c_fresh
  data KEY Guest_register_k
  data bool enter_room_r_fresh
  data bool get_room_c_fresh
  data bool unregister_g_fresh
  data bool unregister_r_fresh
  /* temporary variables updated by connectors */
```

```
data MAP_ROOM_KEY currk
data KEY register_k
data ROOM register_r
/* ports */
export port ty_share_get_room_r_Guest share_get_room_r(get_room_r)
export port ty_share_enter_room_c_Guest share_enter_room_c(enter_room_c)
export port ty_share_register_k_Guest share_register_k(Guest_register_k)
export port ty_share_enter_room_r_Guest share_enter_room_r(enter_room_r)
export port ty_share_get_room_c_Guest share_get_room_c(get_room_c)
export port ty_share_unregister_g_Guest share_unregister_g(unregister_g)
export port ty_share_unregister_r_Guest share_unregister_r( unregister_r )
export port ty_empty_port compute_register_g()
export port ty_register_Guest  register (currk, register_g , register_r )
export port ty_empty_port compute_get_room_g()
export port ty_empty_port compute_get_room_c()
export port ty_empty_port compute_get_room_r()
export port ty_empty_port get_room()
export port ty_empty_port compute_unregister_g()
export port ty_empty_port compute_unregister_r()
export port ty_unregister_Guest  unregister (currk)
export port ty_empty_port compute_enter_room_g()
export port ty_empty_port compute_enter_room_c()
export port ty_empty_port compute_enter_room_r()
export port ty_empty_port enter_room()
export port ty_empty_port exit_room()

place P0
/* etat  initial  */
 initial  to P0
do { cards = PAIR_GUEST_set_sing_empty();
   isin = PAIR_ROOM_set_sing_empty();
  register_g_computed = false;
  get_room_g_computed = false;
  get_room_c_computed = false;
  get_room_r_computed = false;
  unregister_g_computed = false;
  unregister_r_computed = false;
  enter_room_g_computed = false;
  enter_room_c_computed = false;
  enter_room_r_computed = false;
   register_g = guest0();
  get_room_g = guest0();
  get_room_c = card0();
  get_room_r = room0();
   unregister_g = guest0();
   unregister_r = room0();
  enter_room_g = guest0();
  enter_room_c = card0();
  enter_room_r = room0();
  get_room_r_fresh = false;
  enter_room_c_fresh = false;
  enter_room_r_fresh = false;
  get_room_c_fresh = false;
   unregister_g_fresh = false;
   unregister_r_fresh = false; }

/* transitions */
on compute_register_g from P0 to P0
do { register_g_computed = true;
   register_g = guest0(); }

on register  from P0 to P0
provided ( (register_g_computed == true) &&
   !set_in(Guest_register_k,ran(set_kunion(ran(cards)))) )
do { put(cards, register_g, set_union(get(cards, register_g), set_sing(pair(get(currk, register_r),
      Guest_register_k))));
  register_g_computed = false;
  get_room_c_computed = false;
  enter_room_c_computed = false; }

on compute_get_room_g from P0 to P0
do { get_room_g_computed = true;
```

```
  get_room_g = guest0();
  get_room_c_computed = false; }

on compute_get_room_c from P0 to P0
provided ( (get_room_g_computed == true) &&
  set_in(card0(),get(cards,get_room_g)) )
do { get_room_c_computed = true;
  get_room_c = card0();
  get_room_c_fresh = false; }

on compute_get_room_r from P0 to P0
provided ( (get( isin ,room0()) == set_empty()) )
do { get_room_r_computed = true;
  get_room_r = room0();
  get_room_r_fresh = false; }

on get_room from P0 to P0
provided ( (get_room_r_fresh == true) &&
     (get_room_c_fresh == true) &&
     (get_room_g_computed == true) &&
     (get_room_c_computed == true) &&
  set_in(get_room_c,get(cards,get_room_g)) &&
     (get_room_r_computed == true) &&
     (get( isin ,get_room_r) == set_empty()) )
do { get_room_r_computed = false;
  get_room_g_computed = false;
  get_room_c_computed = false; }

on compute_unregister_g from P0 to P0
do { unregister_g_computed = true;
  unregister_g = guest0();
  unregister_g_fresh = false ; }

on compute_unregister_r from P0 to P0
do { unregister_r_computed = true;
  unregister_r = room0();
  unregister_r_fresh = false ; }

on unregister from P0 to P0
provided ( (unregister_g_fresh == true) &&
     (unregister_r_fresh == true) &&
     (unregister_g_computed == true) &&
     (unregister_r_computed == true) &&
  !set_in(unregister_g,get( isin , unregister_r)) )
do { put(cards, unregister_g, ran_sub(get(cards,unregister_g),set_sing(get(currk, unregister_r))));
  get_room_c_computed = false;
  unregister_g_computed = false;
  unregister_r_computed = false;
  enter_room_c_computed = false; }

on compute_enter_room_g from P0 to P0
do { enter_room_g_computed = true;
  enter_room_g = guest0();
  enter_room_c_computed = false; }

on compute_enter_room_c from P0 to P0
provided ( (enter_room_g_computed == true) &&
  set_in(card0(),get(cards,enter_room_g)) )
do { enter_room_c_computed = true;
  enter_room_c = card0();
  enter_room_c_fresh = false; }

on compute_enter_room_r from P0 to P0
do { enter_room_r_computed = true;
  enter_room_r = room0();
  enter_room_r_fresh = false; }

on enter_room from P0 to P0
provided ( (enter_room_c_fresh == true) &&
     (enter_room_r_fresh == true) &&
     (enter_room_g_computed == true) &&
     (enter_room_c_computed == true) &&
```

```
    set_in(enter_room_c,get(cards,enter_room_g)) &&
      (enter_room_r_computed == true) &&
    !set_in(enter_room_g,get(isin,enter_room_r)) )
  do { put( isin , enter_room_r, set_union(get( isin ,enter_room_r), set_sing(enter_room_g)));
    get_room_r_computed = false;
    enter_room_g_computed = false;
    enter_room_r_computed = false;
    enter_room_c_computed = false; }

  on exit_room from P0 to P0
  provided ( set_in(guest0(),get( isin ,room0())) )
  do { put( isin , room0(), set_diff(get( isin ,room0()),set_sing(guest0())));
    get_room_r_computed = false; }

  /* synchro on share_ events */
  on share_get_room_r from P0 to P0 do { get_room_r_fresh = true; }
  on share_enter_room_c from P0 to P0 do { enter_room_c_fresh = true; }
  on share_register_k from P0 to P0
  on share_enter_room_r from P0 to P0 do { enter_room_r_fresh = true; }
  on share_get_room_c from P0 to P0 do { get_room_c_fresh = true; }
  on share_unregister_g from P0 to P0 do { unregister_g_fresh = true; }
  on share_unregister_r from P0 to P0 do { unregister_r_fresh = true; }
end
atom type ty_Room()
  /* state variables */
  data MAP_ROOM_KEY roomk
  data ROOM Room_get_room_r
  data CARD Room_enter_room_c
  data ROOM Room_enter_room_r
  data CARD Room_get_room_c
  /* temporary variables updated by connectors */
  data ROOM enter_room_r
  data CARD enter_room_c
  data CARD get_room_c
  data ROOM get_room_r
  /* ports */
  export port ty_share_get_room_r_Room share_get_room_r(Room_get_room_r)
  export port ty_share_enter_room_c_Room share_enter_room_c(Room_enter_room_c)
  export port ty_share_enter_room_r_Room share_enter_room_r(Room_enter_room_r)
  export port ty_share_get_room_c_Room share_get_room_c(Room_get_room_c)
  export port ty_empty_port get_room()
  export port ty_empty_port enter_room()

  place P0
  /* etat initial */
   initial to P0
  do { roomk = initk(); }

  /* transitions */
  on get_room from P0 to P0
  provided ( (get(roomk,Room_get_room_r) == prj1(Room_get_room_c)) )
  do { put(roomk, Room_get_room_r, prj2(Room_get_room_c)); }

  on enter_room from P0 to P0
  provided ( (get(roomk,Room_enter_room_r) == prj2(Room_enter_room_c)) )

  /* synchro on share_ events */
  on share_get_room_r from P0 to P0
  on share_enter_room_c from P0 to P0
  on share_enter_room_r from P0 to P0
  on share_get_room_c from P0 to P0
end

compound type ty_Hotel_split()
  component ty_Desk Desk()
  component ty_Guest Guest()
  component ty_Room Room()

  connector ty_share_get_room_r share_get_room_r(Guest.share_get_room_r,Room.share_get_room_r)
  connector ty_share_enter_room_c share_enter_room_c(Guest.share_enter_room_c,Room.
      share_enter_room_c)
  connector ty_share_register_k share_register_k(Desk.share_register_k,Guest.share_register_k)
```

```
  connector ty_share_enter_room_r share_enter_room_r(Guest.share_enter_room_r,Room.
      share_enter_room_r)
  connector ty_share_get_room_c share_get_room_c(Guest.share_get_room_c,Room.share_get_room_c)
  connector ty_share_unregister_g share_unregister_g(Desk.share_unregister_g,Guest.share_unregister_g)
  connector ty_share_unregister_r share_unregister_r(Desk.share_unregister_r,Guest.share_unregister_r)
  connector ty_compute_register_g compute_register_g(Guest.compute_register_g)
  connector ty_compute_register_r compute_register_r(Desk.compute_register_r)
  connector ty_compute_register_k compute_register_k(Desk.compute_register_k)
  connector ty_register register (Desk.register ,Guest.register )
  connector ty_compute_get_room_g compute_get_room_g(Guest.compute_get_room_g)
  connector ty_compute_get_room_c compute_get_room_c(Guest.compute_get_room_c)
  connector ty_compute_get_room_r compute_get_room_r(Guest.compute_get_room_r)
  connector ty_get_room get_room(Guest.get_room,Room.get_room)
  connector ty_compute_unregister_g compute_unregister_g(Guest.compute_unregister_g)
  connector ty_compute_unregister_r compute_unregister_r(Guest.compute_unregister_r)
  connector ty_unregister unregister (Desk.unregister ,Guest.unregister )
  connector ty_compute_enter_room_g compute_enter_room_g(Guest.compute_enter_room_g)
  connector ty_compute_enter_room_c compute_enter_room_c(Guest.compute_enter_room_c)
  connector ty_compute_enter_room_r compute_enter_room_r(Guest.compute_enter_room_r)
  connector ty_enter_room enter_room(Guest.enter_room,Room.enter_room)
  connector ty_exit_room exit_room(Guest.exit_room)

   priority pr_compute_register_g compute_register_g:Guest.compute_register_g < register:*
   priority pr_compute_register_r compute_register_r:Desk.compute_register_r < register:*
   priority pr_compute_register_k compute_register_k:Desk.compute_register_k < register:*
   priority pr_compute_get_room_g compute_get_room_g:Guest.compute_get_room_g < get_room:*
   priority pr_compute_get_room_c compute_get_room_c:Guest.compute_get_room_c < get_room:*
   priority pr_compute_get_room_r compute_get_room_r:Guest.compute_get_room_r < get_room:*
   priority pr_compute_unregister_g compute_unregister_g:Guest.compute_unregister_g < unregister:*
   priority pr_compute_unregister_r compute_unregister_r:Guest.compute_unregister_r < unregister:*
   priority pr_compute_enter_room_g compute_enter_room_g:Guest.compute_enter_room_g <
      enter_room:*
   priority pr_compute_enter_room_c compute_enter_room_c:Guest.compute_enter_room_c <
      enter_room:*
   priority pr_compute_enter_room_r compute_enter_room_r:Guest.compute_enter_room_r <
      enter_room:*

end
end /* package */
```

Listing C.18 – Code BIP généré

Résumé

Cette thèse a pour cadre scientifique la décomposition formelle des spécifications centralisées Event-B appliquée aux systèmes distribués BIP. Elle propose une démarche descendante de développement des systèmes distribués corrects par construction en combinant judicieusement Event-B et BIP. La démarche proposée comporte trois étapes : Fragmentation, Distribution et Génération de code BIP. Les deux concepts clefs Fragmentation et Distribution, considérés comme deux sortes de raffinement automatique Event-B paramétrées à l'aide de deux DSL appropriés, sont introduits par cette thèse. Cette thèse apporte également une contribution au problème de la génération de code à partir d'un modèle Event-B issu de l'étape de distribution. Nous traitons aussi bien les aspects architecturaux que comportementaux. Un soin particulier a été accordé à l'outillage et l'expérimentation de cette démarche. Pour y parvenir, nous avons utilisé l'approche IDM pour l'outillage et l'application Hôtel à clés électroniques pour l'expérimentation.

Mots-clés: Méthodes formelles, Décomposition formelle, Systèmes distribués, Raffinement automatique, Génération de code, IDM, DSL, Event-B, BIP

Abstract

The scientific framework of this thesis is the formal decomposition of the centralized specifications Event-B applied to distributed systems based on the BIP (Behavior, Interaction, Priority) component framework. It suggets a top-down approach to the development of correct by construction distributed systems by judiciously combining Event-B and BIP. The proposed approach consists in three steps : Fragmentation, Distribution and Generation of BIP code. We introduce two key concepts, Fragmentation and Distribution, which are considered as two kinds of automatic refinement of Event-B models. They are parameterized using two appropriate DSL. This thesis also contributes to the problem of code generation from Event-B models resulting from the Distribution step. Accordingly, we deal with both architectural and behavioral aspects. A special care has been devoted to the implementation and the experimentation of this approach. To achieve this, we have used the IDM approach for tooling and the Electronic Hotel Key System for experimentation.

Keywords: Formal Methods, Formal Decomposition, Distributed Systems, Automatic Refinement, Code Generation, IDM, DSL, Event-B, BIP