Vérification par Model-Checking Modulaire de Propriétés Dynamiques PLTL exprimées dans le cadre de Spécifications B événementielles

Pierre-Alain Masson

N° d'ordre 874

THÈSE

présentée à

L'UFR des Sciences et Techniques
Université de Franche-Comté

pour obtenir le

GRADE DE DOCTEUR DE L'UNIVERSITÉ DE FRANCHE-COMTÉ
Spécialité Automatique et Informatique

Vérification par model-checking modulaire de propriétés dynamiques PLTL exprimées dans le cadre de spécifications B événementielles

par
Pierre-Alain MASSON

Soutenue le 21 décembre 2001 devant la Commission d'Examen

Rapporteurs	M. Saddek Bensalem, HDR, Maître de Conférences à l'Université Joseph Fourier de Grenoble
	M. Didier Bert, DE, Chargé de recherches CNRS, IMAG de Grenoble
	M. Philippe Schnoebelen, HDR, Chargé de recherches CNRS, ENS de Cachan
Directeur de thèse	M. Jacques Julliand, Professeur à l'Université de Franche-Comté
Examinateurs	Mme Françoise Bellegarde, Professeur à l'Université de Franche-Comté
	M. Hassan Mountassir, Professeur à l'Université de Franche-Comté

Remerciements

Je tiens ici à remercier

- Messieurs Saddek Bensalem, Didier Bert et Philippe Schnoebelen, de m'avoir fait l'honneur d'accepter d'être rapporteurs de cette thèse. Les corrections et les précisions qu'ils ont apportées à ce document m'ont été de la plus grande utilité,

- Monsieur Jacques Julliand, qui a dirigé cette thèse, pour sa disponibilité et son soutien permanents. La patience avec laquelle il a suivi et corrigé mon travail, la direction qu'il a su lui donner, ainsi que ses enseignements, m'ont été inestimablement précieux,

- Monsieur Hassan Mountassir, qui a co-encadré cette thèse, pour la confiance qu'il m'a accordée, pour la pertinence de ces conseils, et pour le suivi au quotidien qu'il fait de mon travail depuis le DEA,

- Madame Françoise Bellegarde, pour la rigueur et la précision de son travail et de ses conseils,

Je tiens également à remercier l'ensemble des membres du LIFC, avec qui travailler fut toujours un plaisir en raison de leur disponibilité et de leur chaleur.

Merci enfin à tous mes proches, pour leur soutien, leurs encouragements et bien plus ...

Table des matières

Table des figures

Liste des définitions

Introduction

1 Enjeux de la vérification formelle

Si le cerveau d'un être humain est capable de concevoir et d'écrire un algorithme, il est en revanche incapable de l'exécuter mentalement. L'ordinateur, à qui l'on soumet cet algorithme, exécute fidèlement les instructions qui y sont décrites, y compris lorsque celles-ci conduisent à une situation que l'utilisateur n'a pas souhaité, mais dont il n'a pas su détecter l'éventualité.

Cette difficulté à comprendre ce que fait vraiment un programme dès que celui-ci devient de taille non négligeable explique pourquoi toute application informatique contient un certain nombre de *bugs*, c'est à dire d'erreurs programmées de manière involontaire.

Le constat de cette situation inquiète lorsque l'on sait que l'informatique est appliquée à la plupart des activités humaines, y compris les plus critiques en termes de risques humains (centrales nucléaires, transports, ...) ou économiques (aérospatiale, communications, systèmes bancaires, ...).

Si l'on souhaite améliorer cette situation, il est nécessaire de mettre en œuvre des techniques destinées à fiabiliser le processus de développement des applications informatiques, tant du point de vue de leur conception que de leur vérification.

Les techniques dites de spécification et de vérification formelles permettent une approche fiabilisée de ces problèmes, et elles constituent un domaine de recherche important et très actif. La *spécification formelle* a pour objectif de proposer une méthodologie du développement visant à aider l'utilisateur à mieux maîtriser les différents aspects de l'application qu'il développe, en l'obligeant à suivre un certain nombre de règles. La *vérification formelle* a pour but de réussir à prouver qu'un programme réalise bien ce qu'on a imaginé qu'il allait faire.

2 Techniques de vérification formelle

Tester un programme pour s'assurer dans un certain nombre de cas « sensibles » qu'il réagit conformément à ce qu'on attendait est une méthode importante et incontournable de *validation*, mais elle n'est pas suffisante pour réellement prouver de manière exhaustive que le programme est correct. Accumuler les présomptions n'est pas prouver.

Le point de vue adopté par les techniques de vérification formelle est de formaliser en les spécifiant d'une part le système conçu (le programme), et les comportements que l'on souhaite qu'il ait. Une fois qu'un modèle M d'un programme a été donné, ainsi qu'un modèle P des comportements souhaités (on parle de *propriétés*), on cherche à répondre à la question suivante :

$$M \models P \ ?$$

c'est à dire que l'on cherche à prouver que le modèle du système satisfait bien le modèle des propriétés.

Ces modèles sont décrits en utilisant des langages formels de spécification tels que les systèmes d'actions pour les systèmes, et les logiques temporelles pour les propriétés. Ces modèles permettent de lever toute les ambiguïtés de l'expression informelle (en langage naturel) des besoins telle qu'on la trouve dans le cahier des charges. La vérification formelle montre que tous les comportements du système satisfont les propriétés.

Il faut bien avoir conscience que même en faisant cela, on n'aura pas garantit que le programme est totalement exempt de défauts. En effet, on aura « seulement » prouvé que des *modèles* sont compatibles. Cependant, cela constitue déjà une avancée considérable vers l'objectif « zéro défaut ». L'application d'une méthodologie rigoureuse de spécification est nécessaire et complémentaire à l'utilisation des techniques formelles de vérification, afin de s'assurer que les modèles constituent une bonne interprétation des exigences informelles exprimées dans le cahier des charges.

Deux techniques principales permettent de prouver formellement qu'un modèle du système satisfait bien un modèle des propriétés, il s'agit de la *preuve* et du *model-checking*.

2.1 Vérification formelle par preuve

Les techniques de preuve sont basées sur des démonstrations mathématiques de la compatibilité des modèles. Un démonstrateur de théorèmes est un outil qui permet, à partir d'un certain nombre d'axiomes et de règles d'inférence, d'aboutir à des conclusions du type $M \models P$.

Les techniques de preuve ont l'avantage de ne pas reposer sur une construction explicite d'un modèle de comportement du type états/transitions, très gourmand en mémoire, puisqu'elles sont capables d'inférer des conclusions directement à partir d'une description d'événements ou d'opérations permettant de faire évoluer le système. En général, le nombre d'opérations autorisées sur un système est petit en comparaison de la représentation explicite des états auxquels l'application de ces opérations permet d'aboutir.

L'utilisation des techniques de preuve est cependant rendue difficile par le fait que, même en utilisant des outils informatiques de démonstration tels que les démonstrateurs de théorèmes, leur automatisation complète est rarement possible. En effet, l'utilisateur est souvent obligé de *guider* la preuve en interagissant avec le démonstrateur de théorèmes. De plus, la préparation de la preuve exige souvent de la part de l'utilisateur la spécification d'éléments permettant de mener cette preuve, mais qui sortent du cadre direct de l'expression des besoins décrite dans le cahier des charges.

2.2 Vérification formelle par model-checking

Le model-checking est une autre technique de vérification formelle qui repose sur une idée simple : si on énumère toutes les situations possibles auxquelles peut mener le programme, on saura s'assurer qu'aucune de ces situations n'est en contradiction avec les comportements que l'on désirait. Le model-checking ne requiert aucune interaction avec l'utilisateur, qui soumet à la fois les modèles du système et des propriétés à un *model-checker*, et qui attend que celui-ci ait terminé l'examen de toutes les situations possibles. Pour cette raison, le model-checking est souvent présenté comme une technique « presse bouton » : une fois les modèles soumis, on presse sur un bouton de type « Vérifier » et on attend le résultat. Cet aspect ne doit pas cacher que la construction des modèles reste une tâche difficile et incontournable.

Les modèles de système utilisés par les model-checkers sont généralement des modèles de type états/transitions, tandis que les modèles des propriétés sont généralement des formules de logique temporelle.

S'il est simple à utiliser (une fois les modèles construits), le model-checking a le désavantage de n'adresser que les problèmes où le nombre d'états est fini. Il est en effet impossible de faire une énumération exhaustive d'un nombre d'états infinis. Et surtout, la représentation de toutes les situations possibles conduit rapidement à un dépassement des capacités de l'ordinateur à stocker toutes ces informations en mémoire. C'est notamment le cas quand on tente d'appliquer le model-checking à des systèmes industriels non triviaux. Ce phénomène est connu sous le nom d'*explosion combinatoire*, et la recherche de solutions à ce problème est un domaine important de l'informatique.

3 Contexte du travail

Notre approche consiste à rédiger des spécifications sous la forme de systèmes d'événements B, en spécifiant les comportements dynamiques en PLTL. Nous utilisons la technique du model-checking pour vérifier ces propriétés.

3.1 Spécification du système : systèmes d'événements B

La méthode B est une méthode de spécification et de vérification définie par Jean-Raymond Abrial, qui avait déjà participé dans les années 80 au développement de la notation Z [Spi88]. Le B-book [Abr96a] est l'ouvrage de référence de la méthode.

Cette méthode met en œuvre des notions de la théorie des ensembles, et elle utilise un formalisme mathématique de haut niveau. Elle est non seulement une méthode de spécification puisqu'elle propose une démarche de développement basée sur la notion de raffinement (un modèle initial abstrait est progressivement enrichi jusqu'à obtenir une spécification directement implantable), mais elle est aussi une méthode de vérification car elle permet, à l'origine, de spécifier des propriétés invariantes devant être respectées à tout moment par le système.

Des développements de la méthode B [AM98] permettent de spécifier des comportements dynamiques sous la forme de contraintes dynamiques. L'une des grandes forces de la méthode est d'associer étroitement les activités d'écriture de spécifications et de preuve de celles-ci.

La notion de systèmes d'événements B que nous adoptons pour la spécification de systèmes réactifs est décrite dans [Abr96b].

La technique de vérification adoptée par la méthode B est une technique de preuve.

3.2 Spécification des propriétés : PLTL

Nous souhaitons utiliser la méthode B pour spécifier les systèmes, mais nous préférons vérifier les propriétés de ce système par une technique de model-checking plutôt que par les techniques de preuve de B.

Pour cela, nous avons choisi de ne pas spécifier les comportements dynamiques désirés dans le formalisme proposé par B (les contraintes dynamiques), et nous spécifions ces comportements sous la forme de formules de la logique temporelle linéaire propositionnelle (PLTL) [Pnu81], dont le pouvoir d'expression est plus riche que celui des contraintes dynamiques B.

Les contraintes dynamiques, comme les formules de logique temporelle, permettent de spécifier des comportements liés à l'évolution dans le temps d'un système, comme par exemple « quand on se retrouve dans la situation A, on devra obligatoirement dans le futur se retrouver dans la situation B ».

La PLTL ne permet pas, contrairement à des logiques temporelles telles que CTL ou CTL* [CE81, EH82], de s'intéresser à l'aspect arborescent des comportements d'un système, mais elle est compatible avec le raffinement B dans le sens où une propriété établie à un niveau donné du raffinement reste vraie dans les raffinements suivants.

4 Problématique

Le travail présenté dans ce document a pour objectif de contribuer à repousser les limites des méthodes de vérification en facilitant leur utilisation et en augmentant la taille des systèmes traitables.

4.1 Vérification par preuve difficile et non automatique

En B, la vérification des contraintes dynamiques d'un système est effectuée par une technique de preuve, ce qui requiert que l'utilisateur, en plus d'être un expert en spécification, soit également un expert dans l'utilisation de cette technique. En effet, la vérification des contraintes dynamiques est confiée à un outil informatique de démonstration de théorèmes, mais dès que cet outil n'arrive plus à progresser seul vers la solution, il fait appel à l'utilisateur pour continuer à avancer. En pratique, la vérification par preuve de contraintes dynamiques est rarement obtenue de manière automatique.

De plus, le démonstrateur de théorèmes est confronté d'une part à la vérification de la correction partielle d'une propriété, et d'autre part à une vérification de terminaison pour certains schémas de contrainte dynamique. Pour effectuer cette vérification de terminaison, l'utilisateur doit exhiber, en plus de la propriété elle même, un invariant de boucle ainsi qu'une fonction décroissante appelée variant. Ces deux éléments ne sont pas directement liés aux propriétés dynamiques décrites dans le cahier des charges. Ils doivent plutôt être considérés comme des « eurêkas » nécessaires à la preuve.

4.2 Utilisation du model-checking limitée par l'explosion combinatoire

Ces raisons (preuve non automatique et surtout nécessité de spécifier un variant et un invariant) motivent notre choix de la technique du model-checking plutôt que celle de la preuve pour vérifier les comportements dynamiques d'un système, dans le but de faciliter la tâche du spécifieur.

Les premiers algorithmes de model-checking [QS82, LP85, CES86, VW86] de propriétés de logique temporelle sont apparus dans les années 80. Depuis, en raison de sa facilité d'utilisation, cette technique est devenue très populaire tant au niveau des milieux académiques que des milieux industriels.

Le model-checking a pour avantage de fournir en général un contre exemple à l'utilisateur lorsqu'il détecte la violation d'une propriété par le système. Ceci facilite grandement la compréhension et la correction des erreurs. Cependant, comme nous avons noté que l'utilisation de cette technique était limitée par le phénomène d'explosion combinatoire, il est nécessaire de trouver des solutions permettant de contourner ce problème afin de pouvoir l'appliquer en pratique.

Plusieurs réponses ont déjà été apportées à ce problème qui s'attaquent soit au problème de l'optimisation en mémoire du codage de la représentation du système, soit à l'optimisation du modèle lui-même. Parmi les premières, citons les techniques de compression mémoire [Hol97b], et les techniques de représentation symbolique des ensembles d'états par des BDD [BCMD90]. Parmi les secondes, citons les techniques de vérification partielle fondées sur des heuristiques [Hol91], les techniques d'ordre partiel [WG93, Pel94, God96] qui suppriment les entrelacements inutiles lors de la modélisation de l'évolution parallèle de processus concurrents, et les techniques d'abstraction du vecteur d'état [CWB94].

5 Contributions

Notre approche propose également un réponse au problème de l'explosion combinatoire, fondée sur la démarche de spécification par raffinement de la méthode B.

Puisque le modèle du système est trop gros pour pouvoir tenir d'un seul bloc dans la mémoire d'un ordinateur, nous proposons de ne jamais construire ce modèle en entier, mais d'en construire successivement plusieurs parties, appelées *modules*, suffisamment petites pour

pouvoir être stockées en mémoire le temps de leur vérification. Chacun de ses modules sera vérifié par model-checking de manière indépendante des autres, et lorsqu'une propriété aura été prouvée vraie sur chacun des modules, nous en conclurons qu'elle est également vraie sur l'intégralité du modèle. Nous appelons cette méthode la *vérification modulaire*. Pour ne pas que certaines transitions d'un modèle états/transitions soient « oubliées » lors de la vérification, nous proposons que l'ensemble des modules constitue une partition ou un recouvrement du modèle initial.

La technique de vérification modulaire s'adresse à une classe de propriétés de la PLTL, que nous appelons les propriétés *vérifiables modulairement*. Une propriété est vérifiable modulairement quand le fait qu'elle est vraie sur tous les modules nous autorise à conclure qu'elle est vraie sur l'intégralité du modèle. Nous définissons une classe de propriétés PLTL vérifiables modulairement.

Nous établissons également une procédure de décision qui permet de prouver qu'une propriété appartient à la classe des propriétés vérifiables modulairement. Cette procédure repose sur l'analyse de l'automate de Büchi qu'il est possible d'associer à toute propriété de la PLTL.

Une fois que l'on a remarqué qu'une propriété est vérifiable modulairement, il faut encore produire un découpage du modèle global en modules qui soient tels qu'ils satisfont tous la propriété à vérifier. Nous définissons une méthode de découpage modulaire qui est basée sur le processus de raffinement B.

Nous proposons également d'augmenter l'efficacité de la méthode en définissant une méthode de vérification modulaire combinant les techniques de preuve et de model-checking. Nous appelons cette technique *méthode d'élimination des modules* car elle permet de vérifier certains modules par preuve sans avoir à leur appliquer le model-checking.

Publications

Ce travail a donné lieu à quatre publications :

- dans [JMM99], nous présentons le découpage modulaire que l'on peut obtenir grâce au raffinement B,
- dans [MBJM00], nous présentons la méthode d'élimination des modules,
- dans [MMJ00], nous présentons la démonstration de la condition suffisante de vérifiabilité modulaire,
- [JMM01] est un article de fond présentant l'ensemble de la méthode, à l'exception de l'élimination des modules.

6 Plan du document

Contexte scientifique

Le chapitre 1 est consacré à la description des principaux concepts de la méthode B et de ses extensions, c'est à dire les contraintes dynamiques et les systèmes d'événements.

Au chapitre 2, nous présentons deux exemples nous servant à illustrer l'ensemble des notions présentées dans le document. Le premier exemple est celui d'un robot industriel de transport de pièces, tandis que le second est un protocole de communication, le BRP. Ces deux exemples sont spécifiés en B et sont accompagnés de la description de propriétés dynamiques.

Au chapitre 3, nous présentons les formalismes que nous utilisons pour spécifier les modèles en vue de leur vérification par model-checking. Pour le système, le modèle utilisé est celui des systèmes de transitions, que nous voyons comme la sémantique des systèmes d'événements décrits au chapitre 1. Pour les propriétés, le modèle utilisé est celui de la logique temporelle, et de la PLTL en particulier.

Le chapitre 4 présente la technique du model-checking PLTL, après avoir introduit quelques notions de la théorie des langages, ainsi que les automates de Büchi. Nous décrivons également dans ce chapitre quelques approches permettant de répondre au problème de l'explosion combinatoire.

Contributions

Au chapitre 5, nous présentons la technique de vérification modulaire, puis nous définissons une classe de propriétés de la PLTL, dont nous démontrons qu'elles sont vérifiables modulairement.

Le chapitre 6 est consacré au problème de la recherche du découpage permettant de mettre en œuvre la vérification modulaire, et nous présentons la technique consistant à guider la construction des modules par le raffinement.

Nous observons les résultats fournis par la méthode de vérification modulaire sur les exemples au chapitre 7. Ces résultats permettent de révéler certaines limites de la technique, et nous proposons dans ce chapitre un autre découpage modulaire que celui du chapitre 6, qui permet de repousser ces limites.

Enfin, dans la conclusion de ce document, après avoir dressé un bilan de la méthode de vérification modulaire, nous positionnons notre approche par rapport à d'autre travaux de vérification par découpage. Nous présentons également en conclusion un ensemble de perspectives dégagées par ce travail.

Première partie

Contexte scientifique, préliminaires

Chapitre 1

La méthode B et les systèmes d'événements B

1.1 Présentation

1.1.1 Démarche de spécification B

La méthode B est une méthode de spécification qui a été proposée et définie par Jean-Raymond Abrial, et qui est décrite dans le livre *The B Book* [Abr96a]. Elle utilise une notation fondée sur les concepts mathématiques de la théorie des ensembles. Elle propose une démarche de spécification qui recouvre toutes les étapes de développement d'un logiciel, depuis la spécification abstraite jusqu'à l'implantation. Le passage de l'une à l'autre se fait progressivement, par le biais des raffinements successifs du modèle initial.

Le cahier des charges constitue l'expression initiale des besoins, et il est généralement rédigé en langue naturelle. Il peut parfois être complété en faisant appel à des modèles de description tels que les graphes, les automates, ou encore des méthodes dites « semi-formelles » telles que UML [RJB98].

La première étape de la spécification consiste à construire un modèle abstrait décrivant les principales variables d'état du système, les propriétés que celles-ci devront satisfaire en permanence (ces propriétés constituent l'*invariant* du système), ainsi que les *opérations* permettant de modifier ces variables. Cette première spécification se situe en principe à un haut niveau d'abstraction, en fournissant une vision globale et peu détaillée du système à concevoir.

Les étapes suivantes ont alors pour but de raffiner cette première spécification en introduisant pas à pas des précisions sur les détails de fonctionnement du système, ce qui permet d'aboutir au final à une spécification directement implantable (voir la figure 1.1).

A chaque étape du raffinement, la cohérence de la spécification raffinée par rapport à la spécification du niveau précédent est garantie par des preuves mathématiques. La vérification du maintien de l'invariant est également effectuée par preuve. En pratique, deux logiciels

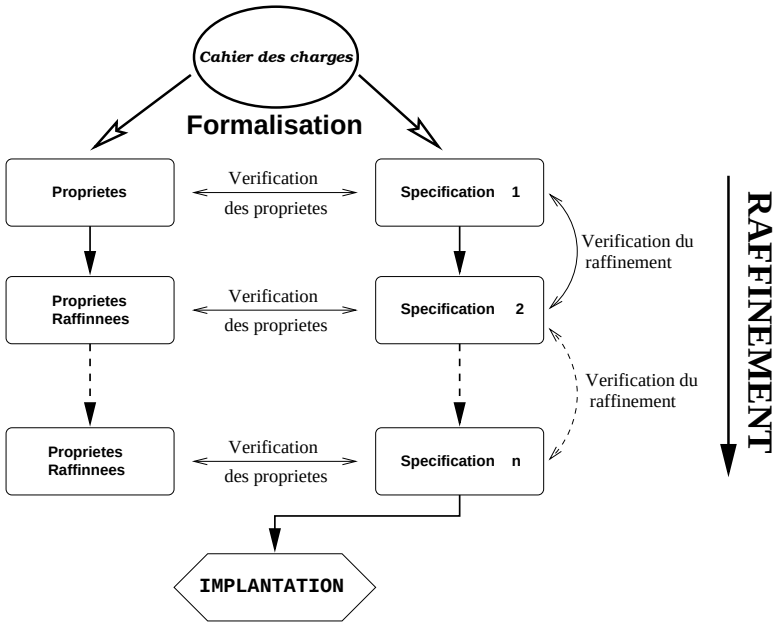

FIG. 1.1 – Démarche de spécification B

(`Atelier B`[1] et `B Tool Kit`[2]) proposent des outils visant à établir ces preuves de manière automatique. Toutefois, en pratique, le spécifieur est souvent amené à interagir avec ces outils afin de tenter d'établir les preuves non résolues automatiquement.

1.1.2 B événementiel

La méthode B, telle qu'elle est présentée dans [Abr96a], repose sur les concepts de machine abstraite et d'opérations. Une machine abstraite peut être vue comme un système ouvert sur son environnement dont l'état interne est susceptible d'évoluer par l'application des opérations. En 1996, Jean-Raymond ABRIAL a proposé dans [Abr96b] une extension de B appelée *B événementiel*, permettant la spécification de systèmes réactifs.

Dans ce développement, les concepts de machine abstraite et d'opérations sont respectivement remplacés par ceux de *système abstrait* et d'*événements*. Ces systèmes abstraits peuvent maintenant être vus comme des systèmes fermés, qui modélisent le système d'intérêt et son environnement, et dont l'état interne peut évoluer par l'application des événements. Ceux-ci sont décrits en termes d'actions gardées, et ils sont susceptibles d'être déclenchés quand leur garde devient vraie. L'un des événements déclenchables peut alors être appliqué, et l'état du système change en fonction de l'action appliquée.

Comme précédemment, ces systèmes abstraits peuvent être raffinés. De plus, Jean-

[1] développé en France par la société ClearSY (anciennement STERIA)
[2] développé au Royaume-Uni par la société B-Core

Raymond ABRIAL et Louis MUSSAT ont proposé dans [AM98] d'introduire dans les spécifications la description de contraintes dynamiques permettant d'exprimer des propriétés qui ne peuvent être décrites en termes d'invariant. A chaque étape du raffinement, de nouvelles contraintes dynamiques peuvent être introduites, qui sont observables grâce à l'introduction des nouveaux événements.

La vérification de ces contraintes dynamiques est elle aussi assurée par une technique de preuve.

Ces deux extensions de B (événements et contraintes dynamiques), constituent le cadre dans lequel s'inscrit notre travail.

1.1.3 Structure d'une spécification B

Il faut tout d'abord noter que, contrairement au langage B présenté dans [Abr96a], le B événementiel n'a pas fait l'objet d'une implantation particulière[3]. Nous reprenons donc ici la structure d'une machine abstraite B classique, dans laquelle nous remplaçons le mot `MACHINE` par `ABSTRACT SYSTEM`. Nous remplaçons aussi les définitions d'opérations par celles d'événements gardés.

Un système d'événements B est composé de deux parties :

1. une spécification descriptive contenant les éléments qui permettent de décrire l'état interne du système et ses propriétés,
2. une spécification opérationnelle indiquant comment l'état interne du système est susceptible d'évoluer.

Dans la spécification descriptive, une section intitulée `VARIABLES` regroupe la liste des variables d'état du système. L'invariant du système est donné dans la section `INVARIANT`. Il permet de typer les variables et d'exprimer des propriétés de sûreté que le système devra respecter à tout moment. Cet invariant a pour effet de limiter le nombre d'états valides du système. On trouve aussi dans cette spécification descriptive d'éventuelles définitions de constantes (section `CONSTANTS`), d'ensembles (section `SETS`), de contraintes (section `CONSTRAINTS`), ...

C'est également dans la spécification descriptive que peuvent être décrites les contraintes dynamiques, dans la section `DYNAMIC CONSTRAINTS`.

La spécification opérationnelle regroupe quant à elle toutes les définitions des événements du système dans une section `EVENTS`, ainsi que la définition de l'initialisation dans la section `INITIALIZATION`.

La structure générale d'une spécification B avec ses principales sections est résumée dans la figure 1.2.

[3] ClearSY a annoncé récemment (c'est à dire en 2001), que la prochaine version d'`Atelier B` permettrait de spécifier des systèmes d'événements B.

	SECTION	DÉFINITION
Spécification descriptive	ABSTRACT SYSTEM	Identificateur *ou* Identificateur(liste d'identificateurs)
	CONSTRAINTS	Prédicat
	SETS	Ensembles
	CONSTANTS	Liste d'identificateurs
	PROPERTIES	Prédicat
	VARIABLES	Liste d'identificateurs
	INVARIANT	Prédicat
Spécification opérationnelle	INITIALIZATION	Substitution généralisée
	EVENTS	Substitution généralisée

FIG. 1.2 – Structure d'un système d'événements B

1.2 Systèmes d'événements B

1.2.1 Définitions

Nous formalisons dans cette section les notions d'événements, de systèmes d'événements et de contraintes dynamiques.

Dans la définition 1, nous définissons les événements en termes de substitutions généralisées, et nous définissons la garde associée à chaque type d'événement.

Définition 1 (Langage des événements).

Soit x une variable, expr une expression et p un prédicat. Un événement est une action gardée définie par la grammaire suivante des substitutions :

$$\begin{aligned} Sub ::= \quad & x := expr \quad | \quad Sub \parallel Sub \\ & | \ \texttt{select}\ p\ \texttt{then}\ Sub\ \texttt{end} \\ & | \ \texttt{any}\ x\ \texttt{where}\ p\ \texttt{then}\ Sub\ \texttt{end} \\ & | \ \texttt{choice}\ Sub\ \texttt{or}\ Sub\ \texttt{end} \end{aligned}$$

La garde d'une substitution généralisée définissant un événement est définie ainsi :

- $g(x := expr) = \texttt{TRUE}$
- $g(Sub_1 \parallel Sub_2) = g(Sub_1) \wedge g(Sub_2)$
- $g(\texttt{select}\ p\ \texttt{then}\ Sub\ \texttt{end}) = p \wedge g(Sub)$
- $g(\texttt{any}\ x\ \texttt{where}\ p\ \texttt{then}\ Sub\ \texttt{end}) = \exists x \cdot p \wedge g(Sub)$
- $g(\texttt{choice}\ Sub_1\ \texttt{or}\ Sub_2\ \texttt{end}) = g(Sub_1) \vee g(Sub_2)$

La définition 2 décrit formellement un système d'événements B à travers un ensemble de variables, un invariant, des propriétés dynamiques, une initialisation et des événements.

Définition 2 (Système d'événements B).

Un système d'événements B est un 6-uplet $\mathcal{S} = \langle X, I, F, Init, A, E_A \rangle$ composé de :

- *un ensemble X de variables,*
- *un prédicat I appelée* invariant *typant chaque variable dans un domaine particulier,*
- *un ensemble F de formules décrivant des propriétés dynamiques,*
- *une initialisation Init des variables par des valeurs initiales,*
- *un ensemble A de noms d'événements,*
- *un ensemble E_A de définitions de chaque événement $e_i \in A$; chacun est défini par $e_i \mathrel{\widehat{=}} sub_i$ où $sub_i \in Sub$ est une substitution généralisée B.*

Soit $x_i \in X$. On note $Domaine(x_i)$ le domaine dans lequel la variable x_i prend ses valeurs. Si $Domaine(x_i)$ est fini pour tout $x_i \in X$, alors on parle de *système à états finis.*

La cohérence d'un système d'événements B est garantie par la vérification d'un ensemble d'obligations de preuve (voir la section 1.2.2).

Définition 3 (Contraintes dynamiques).

Soit $\mathcal{S} = \langle X, I, F, Init, A, E_A \rangle$ un système d'événements. Soit X', un ensemble de variables obtenu en « primant » chaque variable de X. Soient p, q et J des prédicats définis sur des variables de X, soit $e_i \in A$ et soit Va une formule arithmétique entière. Les formules de F décrivant les propriétés dynamiques sont :

- *soit un invariant dynamique défini par une formule de la forme*

  $$\texttt{Dynamics}\ \mathcal{P}(X, X')$$

  où $\mathcal{P}(X, X')$ est un prédicat portant sur des variables de X et de X',
- *soit une modalité définie par une formule de la forme*

  $$\texttt{Select}\ p\ \texttt{Leadsto}\ q\ \texttt{While}\ e_1, \ldots, e_n\ \texttt{Invariant}\ J\ \texttt{Variant}\ Va\ \texttt{End}$$

  ou de la forme

  $$\texttt{Select}\ p\ \texttt{Until}\ q\ \texttt{While}\ e_1, \ldots, e_n\ \texttt{Invariant}\ J\ \texttt{Variant}\ Va\ \texttt{End}.$$

La définition 3 permet d'exprimer des contraintes dynamiques sur les événements, soit sous forme d'invariants dynamiques, soit sous forme de modalités.

Un invariant dynamique est un prédicat *avant-après* $\mathcal{P}(X, X')$ permettant de décrire comment les variables sont autorisées à évoluer quand elles sont modifiées par l'application d'un événement. X désigne l'état des variables *avant* l'application de l'événement, tandis que X' désigne l'état des variables *après* l'application de l'événement.

Les deux schémas de modalité permettent d'exprimer des propriétés dynamiques de vivacité sur les séquences d'application des événements. La modalité p `Leadsto` q permet d'exprimer qu'il est toujours le cas que quand p est vérifié, alors q le sera inévitablement. La modalité p `Until` q permet d'exprimer que à la fois, la modalité p `Leadsto` q est vérifiée, et p reste vérifié tant que q ne l'est pas.

Les sections `While`, `Invariant` et `Variant` qui apparaissent dans l'expression des modalités servent à établir la preuve que le système respecte bien ces contraintes dynamiques, comme expliqué dans la section suivante.

1.2.2 Vérification de la cohérence du système

Vérifier la cohérence d'un système d'événements consiste à vérifier que l'espace d'états du système satisfait l'invariant. De plus, si des contraintes dynamiques ont été spécifiées, il faut également vérifier que celles-ci sont satisfaites.

Ainsi, étant donné un système d'événements $\mathcal{S} = \langle X, I, F, Init, A, E_A \rangle$, le spécifieur est amené à vérifier les obligations de preuve présentées ci-dessous.

Vérification de l'invariant

Afin de prouver que l'espace d'états du système satisfait l'invariant, il appartient au spécifieur de vérifier les obligations de preuve suivantes :

- $[Init]I$; cette obligation prouve que les états initiaux du système satisfont l'invariant,
- $I \Rightarrow [sub_i]I$ pour tout événement $e_i \in A$ défini par $e_i \mathrel{\widehat{=}} sub_i$; celle-ci assure que tout événement déclenchable maintient le système dans un état satisfaisant l'invariant.

Vérification d'un invariant dynamique

On donne l'obligation de preuve à vérifier pour assurer qu'un invariant dynamique $\mathcal{P}(X, X')$ est satisfait par le système. Dans cette expression, $\mathtt{prd}_X(e_i)$ désigne le prédicat avant-après (*before-after* en anglais) associé à l'événement e_i, tel que défini dans le *B book* [Abr96a] :

$$I \wedge \mathtt{prd}_X(e_i) \Rightarrow \mathcal{P}(X, X') \text{ pour tout événement } e_i \in A.$$

Vérification des contraintes dynamiques

Afin de vérifier les modalités exprimant des contraintes dynamiques, l'utilisateur doit exhiber, en plus de la propriété, un *invariant de boucle* et une fonction décroissante appelée *variant*. Le variant décrit la raison pour laquelle les événements du système conduisent inévitablement à un état qui satisfait un prédicat donné (le prédicat noté q, dans la définition 3). L'invariant de boucle est une propriété complétant l'invariant du système qui est nécessaire à la démonstration de la terminaison des portions de chemins conduisant à un état où q est vérifié.

L'utilisateur a également la possibilité de définir une liste d'événements intitulée `While`, facultative. Cette liste sert à restreindre le nombre d'événements intervenant dans la preuve d'une contrainte dynamique, ce qui facilite sa vérification en réduisant le nombre d'obligations de preuve. Ainsi, la signification intuitive d'une modalité p `Leadsto` q incluant une liste `While` $e_1, \ldots, e_n$, est qu'un état où p est vérifié n'est obligé de mener à un état où q est vérifié que si les événements $e_1, \ldots, e_n$ sont appliqués. Dans le cas où l'on veut vérifier qu'un état où p est vérifié mène à un état où q est vérifié quels que soient les événements appliqués, on peut omettre la liste `While`, ce qui revient à inclure tous les événements du système dans cette liste.

On considère que chaque événement e_i est défini par $e_i \mathrel{\widehat{=}} sub_i$ et que $g_i = g(sub_i)$. Soit x, un n-uplet des variables d'états modifiées par les événements de la liste `While`. Les cinq obligations de preuve à vérifier pour satisfaire une modalité sont données ci-dessous.

1. $I \wedge p \Rightarrow J$
2. $I \wedge p \Rightarrow \forall x \cdot (I \wedge J \Rightarrow Va \in \mathbb{N})$
3. $I \wedge p \Rightarrow \forall x \cdot (I \wedge J \wedge \neg q \Rightarrow [sub_i]J)$ pour chaque événement de la liste `While`
4. $I \wedge p \Rightarrow \forall x \cdot (I \wedge J \wedge \neg q \Rightarrow [va := Va][sub_i](Va < va)$ pour chaque événement de la liste `While`
5. $I \wedge p \Rightarrow \forall x \cdot (I \wedge J \wedge \neg q \Rightarrow g_1 \vee g_2 \vee \ldots \vee g_n)$

Dans [AM98], les auteurs présentent la séquence des événements appliqués à partir de l'état où p est vérifié jusqu'à l'état où q est vérifié comme une boucle, dont la garde est $\neg\, q$.

Ces obligations signifient respectivement que :

1. l'invariant J est satisfait au début de la boucle,
2. le variant Va est un entier naturel,
3. chaque événement maintient l'invariant J dans la boucle,
4. chaque événement de la liste `While` qui est applicable quand p ou $\neg\, q$ est satisfait fait décroître le variant,
5. tant que q n'est pas satisfait, il y a au moins un événement déclenchable parmi ceux de la liste `While`.

1.3 Le raffinement de systèmes d'événements

Nous avons dit que la méthode B proposait de construire progressivement les spécifications dans une démarche de raffinement. Raffiner une spécification consiste à remplacer progressivement les éléments de haut niveau d'abstraction par des constructions propres aux langages de programmation afin d'aboutir à une spécification implantable. Plus précisément, raffiner une spécification consiste à :

- remplacer peu à peu les structures de données abstraites (ensembles, relations, fonctions, ...) par des données concrètes (variables scalaires, tableaux),
- lever progressivement le niveau d'indéterminisme des substitutions,
- remplacer les substitutions abstraites (parallélisme, choix) par des substitutions concrètes (séquences, conditions, boucles).

Ainsi, le raffinement porte à la fois sur les données et sur les algorithmes.

Le raffinement de systèmes d'événements permet également d'introduire la notion de raffinement temporel. L'idée est que par unité de temps, il ne peut se produire qu'un seul événement, de telle sorte que l'action associée à chaque événement est *atomique*. Le niveau

de détail d'observation du système est donc fonction de la granularité du temps choisie par le spécifieur. Plus le système est abstrait, plus les unités de temps sont grandes. Raffiner le système d'événements consiste alors à « raccourcir » les unités de temps, de telle sorte que des événements auparavant cachés deviennent maintenant observables.

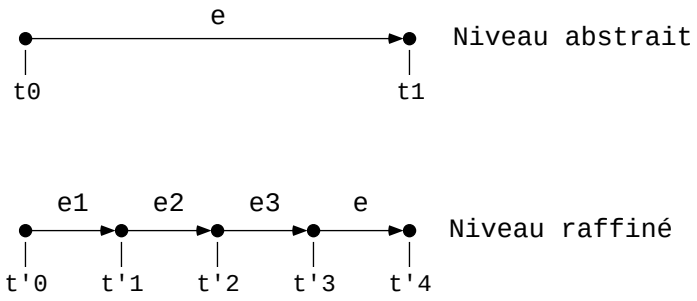

FIG. 1.3 – Le système est observé plus souvent

La figure 1.3 représente un événement e atomique au niveau abstrait, qui se décompose en une succession d'événements e_1, e_2, e_3, e au niveau raffiné. Dans cette figure, il n'y a qu'une seule unité de temps (de t_0 à t_1) au niveau abstrait : tout ce qui survient entre ces deux instants est masqué. Au niveau raffiné, avec une granularité du temps plus fine, on peut observer les événements e_1 (entre t'_0 et t'_1), e_2 (entre t'_1 et t'_2), et e_3 (entre t'_2 et t'_3), qu'on appelle les *nouveaux événements*. Cette succession de nouveaux événements se termine par un événement e (entre t'_3 et t'_4) qui porte le même nom que l'événement du niveau abstrait, et qu'on appelle *ancien événement*.

Ainsi, on introduit des nouveaux événements à chaque étape du raffinement. Ils raffinent la substitution qui ne fait rien, c'est à dire la substitution `skip`.

A partir d'une spécification abstraite, plusieurs raffinements sont possibles. On note $\mathcal{S}_1 \sqsubseteq \mathcal{S}_2$ le fait qu'un système d'événements $\mathcal{S}_1$ est raffiné par un système d'événements $\mathcal{S}_2$. Le système $\mathcal{S}_1$ est appelé le *système abstrait* par rapport à $\mathcal{S}_2$ qui est appelé le *système raffiné*. Afin de prouver qu'un système d'événements $\mathcal{S}_1$ est raffiné par un système d'événements $\mathcal{S}_2$, on doit vérifier l'ensemble des obligations de preuve que présente la définition 4.

Le système raffiné contient les mêmes éléments que le système abstrait, mais on doit ajouter à $\mathcal{S}_2$ la définition d'un variant Va_2, qui est une fonction entière décroissante. Ce variant sert à établir la preuve que les nouveaux événements spécifiés dans $\mathcal{S}_2$ n'introduisent pas de nouveaux cycles, en s'assurant qu'ils font tous décroître le variant. Les variables de $\mathcal{S}_2$ sont collées à celles de $\mathcal{S}_1$ (éventuellement par un simple renommage), et de nouvelles variables peuvent être définies dans $\mathcal{S}_2$. Les relations entre les variables de $\mathcal{S}_2$ et celles de $\mathcal{S}_1$ sont données par l'invariant du système raffiné, appelé *invariant de collage*. Celui-ci donne aussi le typage des nouvelles variables. Enfin, le système raffiné définit à nouveau les anciens événements en les raffinant, et il introduit les nouveaux événements.

Définition 4 (Raffinement de systèmes d'événements).

Soit $\mathcal{S}_1 = \langle X_1, I_1, F_1, Init_1, A_1, E_{A_1} \rangle$ *et* $\mathcal{S}_2 = \langle X_2, I_2, F_2, Init_2, A_2, E_{A_2} \rangle$, *deux systèmes*

d'événements. Soit Va_2, une fonction entière décroissante appelée variant ajoutée à la définition de $\mathcal{S}_2$. Soit $e \in A_1$ et $e \in A_2$ tel que $e \mathrel{\widehat{=}} sub_1 \in E_{A_1}$ et $e \mathrel{\widehat{=}} sub_2 \in E_{A_2}$. Soit $A_3 = A_2 \setminus A_1$, soit $e_3 \in A_3$ tel que $e_3 \mathrel{\widehat{=}} sub_3 \in E_{A_3}$, et soit $e_{3_i} \in A_3$ tel que $e_{3_i} \mathrel{\widehat{=}} sub_{3_i} \in E_{A_3}$. Soit enfin $g_1 = g(sub_1)$, $g_2 = g(sub_2)$ et $g_{3_i} = g(sub_{3_i})$.

On a $\mathcal{S}_1 \sqsubseteq \mathcal{S}_2$ si et seulement si les obligations de preuve énoncées ci-dessous sont vérifiées :

1. $X_1 \cap X_2 = \emptyset$,
2. $A_1 \subset A_2$,
3. $[Init_2]\neg[Init_1]\neg I_2$,
4. $\forall e \cdot ((e \in A_1 \wedge e \in A_2) \Rightarrow I_1 \wedge I_2 \Rightarrow [sub_2]\neg\, [sub_1]\neg\, I_2)$,
5. $\forall e_3 \cdot (e_3 \in A_3 \Rightarrow I_1 \wedge I_2 \Rightarrow [sub_3](I_2 \wedge I_1))$,
6. $\forall e \cdot ((e \in A_1 \wedge e \in A_2) \Rightarrow (I_1 \wedge I_2 \Rightarrow g_1 \Rightarrow g_2 \bigvee_{i=1}^{n} g_{3_i}))$,
7. $\forall e_3 \cdot (e_3 \in A_3 \Rightarrow I_1 \wedge I_2 \Rightarrow [va := Va_2][sub_3](Va_2 < va))$.

La signification des obligations de preuve présentées dans la définition 4 est la suivante :

1. les ensembles de variables des systèmes abstraits et raffinés sont disjoints,
2. tous les événements abstraits sont raffinés,
3. l'action initiale raffinée n'est pas contradictoire avec l'action initiale abstraite,
4. les actions de tous les événements qui raffinent des événements abstraits ne sont pas contradictoires avec les actions abstraites ; notons que cette condition permet de diminuer le non-déterminisme interne aux anciens événements,
5. tous les nouveaux événements satisfont l'invariant $I_1 \wedge I_2$,
6. les nouveaux événements n'introduisent pas de nouveaux blocages (*deadlocks* en anglais),
7. les nouveaux événements n'introduisent pas de nouveaux cycles (*livelocks* en anglais).

1.4 Conclusion

Nous avons présenté dans ce chapitre la méthode B, qui est une méthode de spécification proposant de passer étape par étape d'une description abstraite d'un système à une description proche de l'implantation. Chaque étape, appelée raffinement, décrit de nouveaux détails de fonctionnement du système.

Les systèmes d'événements B permettent de spécifier de cette manière des systèmes réactifs. Ces systèmes d'événements peuvent être enrichis de la description de contraintes dynamiques permettant de spécifier les comportements dynamiques que l'on souhaite que le système respecte.

A chaque étape du raffinement, la cohérence du système ainsi que le respect des contraintes dynamiques sont établis par une technique de preuve.

Le chapitre suivant illustre ces concepts en proposant deux exemples de systèmes spécifiés en B par une démarche de raffinement, et accompagnés de contraintes dynamiques.

Chapitre 2

Deux exemples

Nous présentons dans ce chapitre deux exemples de systèmes spécifiés en B événementiel, afin d'illustrer les concepts présentés au chapitre précédent. Le premier de ces systèmes est un robot industriel de transport de pièces, le second est un protocole de communication (le BRP).

Pour chacun de ces deux exemples, la présentation est structurée en trois parties :

1. présentation informelle,
2. spécification abstraite et raffinement(s) de celle-ci,
3. spécification de contraintes dynamiques devant être respectées par le système.

Un exemple de vérification par preuve d'une contrainte dynamique est développé dans le cas du robot.

Ces deux exemples nous serviront tout au long de ce document à illustrer les différents concepts qui y sont présentés.

2.1 Un robot industriel

Nous souhaitons spécifier le comportement d'un robot industriel destiné à transporter des pièces. Ce système a été présenté et spécifié en B par Jean-Raymond Abrial dans [Abr97]. La version que nous présentons ici est une adaptation.

2.1.1 Présentation

Principalement, ce système est composé de trois dispositifs : un dispositif d'arrivée des pièces, un dispositif d'évacuation des pièces, et un dispositif de transport des pièces. Les pièces sont transportées du dispositif d'arrivée vers le dispositif d'évacuation au moyen du dispositif de transport.

Les dispositifs d'arrivée et d'évacuation sont vus comme des plateaux sur lesquels des pièces sont susceptibles d'arriver (pour le dispositif d'arrivée) et d'être emmenées (pour le dispositif d'évacuation). Pour la modélisation de ce système, on ne « voit » pas ce qui amène les pièces ou ce qui les évacue. Ainsi, tout se passe comme si une pièce était susceptible d'être amenée spontanément sur le dispositif d'arrivée, de même qu'une pièce sur le dispositif d'évacuation est susceptible d'être évacuée spontanément. Le dispositif de transport est vu comme une pince qui saisit les pièces sur le dispositif d'arrivée et va les déposer sur le dispositif d'évacuation.

Les dispositifs d'arrivée et d'évacuation sont immobiles. Par contre, la pince est susceptible de se déplacer dans deux directions de l'espace : de bas en haut et vice versa (pour emmener une pièce d'un dispositif à l'autre et retourner en chercher une autre), et de gauche à droite et vice-versa (la pince s'avance pour aller saisir une pièce ou la déposer, et recule avant d'effectuer un mouvement vertical).

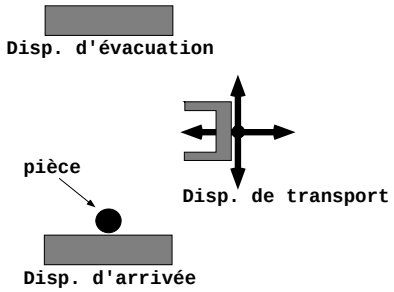

FIG. 2.1 – Schéma de fonctionnement du robot

La figure 2.1 schématise le fonctionnement de ce système industriel simple. Nous avons arbitrairement placé le dispositif d'arrivée en bas et le dispositif d'évacuation en haut.

Les contraintes suivantes sont imposées au système :

- chaque dispositif ne peut contenir qu'une seule pièce à un moment donné,
- la pince ne peut pas monter à vide,
- la pince ne peut pas descendre chargée,

Nous spécifions ce système pas à pas, en introduisant à chaque étape un nouvel élément permettant de décrire le système. Chaque spécification constitue un raffinement de la précédente.

2.1.2 Spécification abstraite du robot

Pour la première spécification de ce système, nous ne nous préoccupons que du comportement du dispositif de transport, qui peut être soit vide, soit occupé (par une seule pièce).

Cette première spécification est donnée dans la figure 2.2. Une seule variable appelée dt_0 (pour dispositif de transport) suffit à décrire l'état de ce système.

```
SYSTEM Robot0
VARIABLES dt0
INVARIANT
  dt0 ∈ {vid, occ}
INITIALISATION
  dt0 := vid
EVENTS
  chgt  ≙ select dt0 = vid then dt0 := occ end
  dchgt ≙ select dt0 = occ then dt0 := vid end
END Robot0
```

FIG. 2.2 – Spécification abstraite du robot

L'invariant indique que dt_0 peut prendre deux valeurs `vid` et `occ`, selon que la pince est vide ou occupée.

L'initialisation place le système dans l'état où la pince est vide. Deux événements permettent de faire évoluer le système : *chgt*, qui décrit le chargement d'une pièce dans la pince, et *dchgt*, qui décrit le déchargement d'une pièce.

2.1.3 Premier raffinement de la spécification du robot

Nous introduisons lors de ce premier raffinement, au travers de la variable da_1 (pour *dispositif d'arrivée*), le dispositif d'arrivée des pièces qui peut être soit vide soit occupé. La spécification correspondante est donnée par la figure 2.3.

Remarque. Nous rappelons qu'un raffinement de spécification doit être accompagné d'un variant pour pouvoir faire la preuve que les nouveaux événements n'introduisent pas de *livelocks* (voir la définition 4, dans la section 1.3).

L'invariant donne le typage de da_1 et indique également que la variable dt_1 est identique à son abstraction. L'initialisation place les deux dispositifs dans l'état vide.

Un événement *arr_p* est ajouté, permettant de modéliser l'arrivée d'une pièce sur le dispositif d'arrivée.

2.1.4 Deuxième raffinement de la spécification du robot

Au cours de ce deuxième raffinement, présenté dans la figure 2.4, nous introduisons le dispositif d'évacuation, qui peut être vide ou occupé.

Le dispositif d'évacuation est modélisé par la variable de_2 (pour *dispositif d'évacuation*), dont le typage est donné dans l'invariant. Celui-ci stipule également que les variables dt_2 et da_2 sont identiques à leur abstraction, c'est à dire aux variables dt_1 et da_1 du premier raffinement.

L'initialisation place les trois dispositifs dans l'état vide. Un événement *evac* est ajouté, qui permet de décrire l'évacuation d'une pièce.

```
SYSTEM Robot_1 refines Robot_0
VARIABLES dt_1, da_1
INVARIANT
  dt_1 = dt_0 ∧ da_1 ∈ {vid, occ}
INITIALISATION
  dt_1, da_1 := vid, vid
EVENTS
  chgt   ≙ select da_1 = occ ∧ dt_1 = vid
           then da_1, dt_1 := vid, occ end
  dchgt  ≙ select dt_1 = occ
           then dt_1 := vid end
  arr_p  ≙ select da_1 = vid
           then da_1 := occ end
VARIANT
  f_1(da_1) where f_1(occ) = 0 and f_1(vid) = 1
END Robot_1
```

FIG. 2.3 – Premier raffinement du robot

```
SYSTEM Robot_2 refines Robot_1
VARIABLES dt_2, da_2, de_2
INVARIANT
  dt_2 = dt_1 ∧ da_2 = da_1 ∧ de_2 ∈ {vid, occ}
INITIALISATION
  dt_2, da_2, de_2 := vid, vid, vid
EVENTS
  chgt   ≙ select da_2 = occ ∧ dt_2 = vid
           then dt_2, da_2 := occ, vid end
  dchgt  ≙ select dt_2 = occ ∧ de_2 = vid
           then dt_2, de_2 := vid, occ end
  arr_p  ≙ select da_2 = vid
           then da_2 := occ end
  evac   ≙ select de_2 = occ
           then de_2 := vid end
VARIANT
  f_2(de_2) where f_2(occ) = 1 and f_2(vid) = 0
END Robot_2
```

FIG. 2.4 – Deuxième raffinement du robot

2.1.5 Troisième raffinement de la spécification du robot

Lors du troisième raffinement, donné par la figure 2.5, nous modélisons le mouvement ascendant ou descendant de la pince. Pour cela, une variable appelée $posdt_3$ (pour position du dispositif de transport) est introduite. Cette variable peut prendre deux valeurs `haut` et `bas` selon que le dispositif de transport est en haut (près du dispositif d'évacuation), ou en bas (près du dispositif d'arrivée).

```
SYSTEM Robot3 refines Robot2
VARIABLES dt3, da3, de3, posdt3
INVARIANT
  dt3 = dt2 ∧ da3 = da2 ∧ de3 = de2 ∧ posdt3 ∈ {haut, bas}
INITIALISATION
  dt3, da3, de3, posdt3 := vid, vid, vid, bas
EVENTS
  chgt   ≙ select dt3 = vid ∧ da3 = occ ∧ posdt3 = bas
           then dt3, da3 := occ, vid end
  dchgt  ≙ select dt3 = occ ∧ de3 = vid ∧ posdt3 = haut
           then dt3, de3 := vid, occ end
  arr_p  ≙ select da3 = vid
           then da3 := occ end
  evac   ≙ select de3 = occ
           then de3 := vid end
  mont   ≙ select dt3 = occ ∧ posdt3 = bas
           then posdt3 := haut end
  desc   ≙ select dt3 = vid ∧ posdt3 = haut
           then posdt3 := bas end
VARIANT
  f3(dt3, posdt3) where
        f3(occ, haut) = 0 and f3(occ, bas) = 1 and
        f3(vid, haut) = 1 and f3(vid, bas) = 0
END Robot3
```

FIG. 2.5 – Troisième raffinement du robot

L'invariant donne le typage de cette nouvelle variable et indique que dt_3, da_3 et de_3 sont identiques à leur abstraction.

L'initialisation place le système dans un état où les trois dispositifs sont vides, et où la pince est en bas.

Deux nouveaux événements sont introduits : *mont* permettant de modéliser la montée de la pince vers le dispositif d'évacuation, et *desc* permettant de modéliser la descente de la pince vers le dispositif d'arrivée.

2.1.6 Expression des propriétés dynamiques du robot

Dans cette section, nous exprimons des propriétés dynamiques que l'on souhaite que le système respecte, sous forme de contraintes dynamiques. A chaque niveau du raffinement, de nouvelles propriétés sont introduites qui correspondent à des comportements qui sont devenus observables grâce à l'introduction de nouveaux événements.

Afin de s'assurer que le système respecte bien les contraintes spécifiées, il faut vérifier l'ensemble des obligations de preuve générées pour chaque contrainte [AM98]. La vérification de ces obligations est effectuée à titre d'exemple pour une contrainte dynamique dans la section suivante.

Nous avons vu dans la section 1.2.2 que la spécification de propriétés sous forme de contraintes dynamiques nécessitait de la part du spécifieur l'introduction d'un invariant *local* et d'un variant, dans le but de pouvoir vérifier les obligations de preuve. La définition de ces deux éléments, qui ne sont pas issus directement de l'expression informelle des besoins du système (le cahier des charges), exige souvent une grande compétence de la part du spécifieur.

Nous souhaitons que ces propriétés soient vérifiées par rapport à l'occurrence de n'importe quel événement du système, aussi nous ne définissons pas de liste `While` dans l'expression de ces propriétés.

Propriétés dynamiques de la spécification abstraite du robot

A ce niveau de la spécification, nous exprimons les deux propriétés suivantes :

(P1) si le dispositif de transport est occupé, alors il deviendra inévitablement vide,

(P2) si le dispositif de transport est vide, alors il deviendra inévitablement occupé.

La propriété P1 s'exprime en B par la contrainte dynamique donnée dans la figure 2.6. La propriété P2 s'exprime par la contrainte dynamique de la figure 2.7.

```
Select
  dt0 = occ
Leadsto
  dt0 = vid
Invariant
  TRUE
Variant
  f(dt0) where f(occ) = 1 and f(vid) = 0
End
```

FIG. 2.6 – Expression B de la propriété P1

```
Select
  dt0 = vid
Leadsto
  dt0 = occ
Invariant
  TRUE
Variant
  f(dt0) where f(vid) = 1 and f(occ) = 0
End
```

FIG. 2.7 – Expression B de la propriété P2

Propriétés dynamiques du premier raffinement du robot

Les propriétés dynamiques que l'on définit lors du premier raffinement du robot sont les suivantes :

(P3) le chargement ne peut se faire que dans une pince vide,

(P4) le dispositif d'arrivée reste occupé jusqu'à ce qu'inévitablement la pince soit vide,

(P5) la pince ne peut pas décharger sur le dispositif d'arrivée.

```
Dynamics
  da1 = occ ∧ da1' = vid ⇒ dt1 = vid
```

FIG. 2.8 – Expression B de la propriété P3

```
Select
  da1 = occ
Until
  dt1 = vid
Invariant
  TRUE
Variant
  f(dt1) where f(occ) = 1 and f(vid) = 0
End
```

FIG. 2.9 – Expression B de la propriété P4

Les propriétés P3 et P5 s'expriment par des invariants dynamiques tandis que la propriété P4 s'exprime sous la forme d'une modalité `Until`. Les figures 2.8, 2.9 et 2.10 donnent respectivement l'expression B de ces trois propriétés.

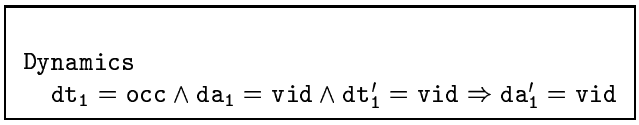

FIG. 2.10 – Expression B de la propriété P5

Remarque. Nous rappelons que dans l'expression des invariants dynamiques, une variable sans prime indique l'état de celle-ci *avant* l'application d'un événement, tandis qu'une variable avec prime indique l'état de celle-ci *après* l'application d'un événement.

Propriétés dynamiques du deuxième raffinement du robot

Lors du deuxième raffinement du robot, on spécifie les propriétés dynamiques suivantes :

(P6) la pince ne peut décharger que sur un dispositif d'évacuation vide,

(P7) la pince occupée le reste jusqu'à ce qu'inévitablement le dispositif d'évacuation se libère,

(P8) si une pièce est placée sur le dispositif d'évacuation alors elle sera évacuée.

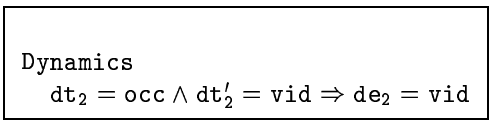

FIG. 2.11 – Expression B de la propriété P6

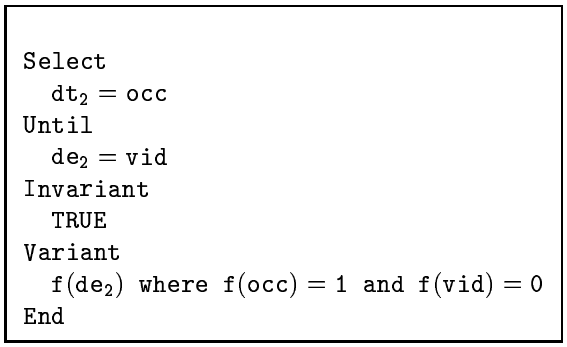

FIG. 2.12 – Expression B de la propriété P7

L'invariant dynamique exprimant la propriété P6 est donné par la figure 2.11. Les contraintes dynamiques permettant d'exprimer les propriétés P7 et P8 sont données respectivement dans les figures 2.12 et 2.13.

```
Select
  de2 = occ
Leadsto
  de2 = vid
Invariant
  TRUE
Variant
  f(de2) where f(occ) = 1 and f(vid) = 0
End
```

FIG. 2.13 – Expression B de la propriété P8

Propriétés dynamiques du troisième raffinement du robot

Les propriétés dynamiques suivantes sont exprimées lors du troisième raffinement du robot :

(P9) la pince vide en position haute redescend inévitablement,

(P10) la pince chargée en position basse remonte inévitablement,

(P11) la pince ne peut pas descendre chargée,

(P12) la pince haute et occupée le reste jusqu'à ce qu'inévitablement elle se vide,

(P13) la pince ne peut pas monter à vide,

(P14) la pince basse et vide le reste jusqu'à ce qu'inévitablement elle se charge.

```
Select
  posdt3 = haut ∧ dt3 = vid
Leadsto
  posdt3 = bas
Invariant
  TRUE
Variant
  f(posdt3) where f(haut) = 1 and f(bas) = 0
End
```

FIG. 2.14 – Expression B de la propriété P9

Les propriétés P9, P10, P12 et P14 sont exprimées par des contraintes dynamiques décrites respectivement par les figures 2.14, 2.15, 2.17 et 2.19. Les propriétés P11 et P13 sont exprimées par des invariants dynamiques décrits par les figures 2.16 et 2.18.

```
Select
  posdt3 = bas ∧ dt3 = occ
Leadsto
  posdt3 = haut
Invariant
  TRUE
Variant
  f(posdt3) where f(bas) = 1 and f(haut) = 0
End
```

FIG. 2.15 – Expression B de la propriété P10

```
Dynamics
  posdt3 = haut ∧ posdt3' = bas ⇒ dt3 = vid
```

FIG. 2.16 – Expression B de la propriété P11

```
Select
  posdt3 = haut ∧ dt3 = occ
Until
  dt3 = vid
Invariant
  TRUE
Variant
  f(dt3) where f(occ) = 1 and f(vid) = 0
End
```

FIG. 2.17 – Expression B de la propriété P12

```
Dynamics
  posdt3 = bas ∧ posdt3' = haut ⇒ dt3 = occ
```

FIG. 2.18 – Expression B de la propriété P13

```
Select
  posdt3 = bas ∧ dt3 = vid
Until
  dt3 = occ
Invariant
  J
Variant
  f(dt3) where f(vid) = 1 and f(occ) = 0
End
```

FIG. 2.19 – Expression B de la propriété P14

2.1.7 Vérification par preuve d'une contrainte dynamique

On montre ici un exemple pratique de la vérification des obligations de preuve pour une contrainte dynamique de la forme

Select p **Leadsto** q **While** $e_1, \ldots, e_n$ **Invariant** J **Variant** Va **End**.

La contrainte dynamique exprimant la propriété P1 est de cette forme. L'expression B de cette contrainte a déjà été donnée dans la figure 2.6, mais nous la redonnons ici dans la figure 2.20 afin de faciliter la lecture.

Notons que dans cette figure, une liste `While` incluant les deux événements du système abstrait est définie, de manière à rendre explicite le fait que nous vérifions les obligations de preuve par rapport à *tous* les événements.

```
Select
  dt = occ
Leadsto
  dt = vid
While
  chgt, dchgt
Invariant
  J = VRAI
Variant
  Va = f(dt) where f(occ) = 1 and f(vid) = 0
End
```

FIG. 2.20 – Une contrainte dynamique du robot à vérifier

Dans ce qui suit, ce que nous appelons *boucle* désigne (pour cette forme de contrainte dynamique) un chemin d'exécution formé d'une séquence d'états $s_1, \ldots, s_n$ telle que : s_1 satisfait p, pour tout i tel que $1 \leq i \leq n-1$, s_i satisfait $\neg\, q$, et s_n satisfait q. Dans l'exemple

de la figure 2.20, p est représenté par l'expression $\texttt{dt} = \texttt{occ}$, et q est représenté par l'expression $\texttt{dt} = \texttt{vid}$.

Les obligations de preuve générées pour la vérification de cette contrainte dynamique sont les suivantes (voir la section 1.2.2) :

$$I \wedge dt = \texttt{occ} \Rightarrow J \tag{2.1}$$
$$I \wedge dt = \texttt{occ} \Rightarrow \forall y \cdot (I \wedge J \Rightarrow Va \in \mathbb{N}) \tag{2.2}$$
$$I \wedge dt = \texttt{occ} \Rightarrow \forall y \cdot (I \wedge J \wedge \neg\,(dt = \texttt{vid}) \Rightarrow [chgt]J) \tag{2.3}$$
$$I \wedge dt = \texttt{occ} \Rightarrow \forall y \cdot (I \wedge J \wedge \neg\,(dt = \texttt{vid}) \Rightarrow [dchgt]J) \tag{2.4}$$
$$I \wedge dt = \texttt{occ} \Rightarrow \forall y \cdot (I \wedge J \wedge \neg\,(dt = \texttt{vid}) \Rightarrow [va := Va][chgt](Va < va)) \tag{2.5}$$
$$I \wedge dt = \texttt{occ} \Rightarrow \forall y \cdot (I \wedge J \wedge \neg\,(dt = \texttt{vid}) \Rightarrow [va := Va][dchgt](Va < va)) \tag{2.6}$$
$$I \wedge dt = \texttt{occ} \Rightarrow \forall y \cdot (I \wedge J \wedge \neg\,(dt = \texttt{vid}) \Rightarrow g(chgt) \vee g(dchgt)) \tag{2.7}$$

Dans ces expressions, la variable y représente toutes les variables d'états modifiées par les événements de la liste `While` et I correspond à l'invariant de la spécification `B` donnée dans la figure 2.2. Ici, on a donc $y = dt$ et $I = (dt \in \{\texttt{vid}, \texttt{occ}\})$, qui vaut `VRAI` par définition.

Remarque. Pour ne pas alourdir la lecture des expressions, nous ne représentons pas l'indice 0 normalement associé aux variables d'état de la spécification abstraite du robot.

Nous rappelons la signification intuitive de ces obligations :

- l'obligation 2.1 vérifie que l'invariant local est satisfait au début de la boucle,
- l'obligation 2.2 vérifie que le variant est un entier naturel,
- les obligations 2.3 et 2.4 vérifient que l'invariant local est préservé,
- les obligations 2.5 et 2.6 vérifient que les événements *chgt* et *dchgt* font décroître le variant,
- l'obligation 2.7 vérifie qu'il y a toujours un événement déclenchable tant que la condition inévitable n'est pas atteinte.

Le prédicat J (invariant de boucle) et l'expression Va (variant de boucle) sont définies dans l'expression de la contrainte dynamique donnée par la figure 2.20 : J a pour valeur `VRAI` et Va est défini par une fonction f de la variable dt qui compte le nombre de pièces présentes à un instant donné dans le dispositif de transport. Enfin, l'expression $\neg\,(dt = \texttt{vid})$ (correspondant à $\neg\, q$) peut être remplacée par l'expression équivalente $dt = \texttt{occ}$.

Les preuves de 2.1, 2.2, 2.3 et 2.4 sont triviales car pour chacune d'elles, la partie droite de l'implication a la valeur `VRAI`.

Vérification de l'obligation de preuve 2.5

On veut vérifier l'obligation de preuve suivante :

$$I \wedge dt = \texttt{occ} \Rightarrow \forall y \cdot (I \wedge J \wedge dt = \texttt{occ} \Rightarrow [va := Va][chgt](Va < va)).$$

Intuitivement, celle-ci est satisfaite car l'événement *chgt* n'est pas déclenchable dans la boucle, c'est à dire quand p ou $\neg\, q$ sont satisfaits.

En supprimant I et J qui valent `VRAI` et en remplaçant Va par $f(dt)$, y par dt et *chgt* par son expression, on obtient :

$$dt = \texttt{occ} \Rightarrow \forall dt \cdot (dt = \texttt{occ} \Rightarrow [va := f(dt)] \\ [\texttt{select}\ \ dt = \texttt{vid}\ \texttt{then}\ \ dt := \texttt{occ}\ \texttt{end}](f(dt) < va)).$$

Sachant d'après [Abr96a] qu'un prédicat de la forme [`select` a `then` b `end`]pr est équivalent au prédicat $a \Rightarrow [b]pr$ (où pr est un prédicat), on a :

$$dt = \texttt{occ} \Rightarrow \forall dt \cdot (dt = \texttt{occ} \Rightarrow [va := f(dt)](dt = \texttt{vid} \Rightarrow [dt := \texttt{occ}](f(dt) < va)))$$

ce qui donne, par calcul de la substitution $[dt := \texttt{occ}]$ avec $f(\texttt{occ}) = 1$:

$$dt = \texttt{occ} \Rightarrow \forall dt \cdot (dt = \texttt{occ} \Rightarrow [va := f(dt)](dt = \texttt{vid} \Rightarrow 1 < va)).$$

Afin de ne pas confondre la variable libre dt instanciée à `occ` avec la variable dt quantifiée par $\forall dt$, on renomme cette dernière en dt_r. On obtient alors :

$$dt = \texttt{occ} \Rightarrow \forall dt_r \cdot (dt_r = \texttt{occ} \Rightarrow [va := f(dt_r)](dt_r = \texttt{vid} \Rightarrow 1 < va)).$$

Deux cas doivent être distingués, celui où $dt_r = \texttt{vid}$ et celui où $dt_r = \texttt{occ}$.

Premier cas : $dt_r = \texttt{vid}$

On obtient, par calcul de la substitution $[va := f(dt_r)]$ avec $f(\texttt{vid}) = 0$:

$$dt = \texttt{occ} \Rightarrow (\underbrace{\underbrace{\texttt{vid} = \texttt{occ}}_{\texttt{FAUX}} \Rightarrow (\underbrace{\underbrace{\texttt{vid} = \texttt{vid}}_{\texttt{VRAI}} \Rightarrow \underbrace{1 < 0}_{\texttt{FAUX}}}_{\texttt{FAUX}})}_{\texttt{VRAI}})$$

qui est `VRAI`.

Deuxième cas : $dt_r = \texttt{occ}$

On obtient, par calcul de la substitution $[va := f(dt_r)]$ avec $f(\texttt{occ}) = 1$:

$$dt = \texttt{occ} \Rightarrow (\underbrace{\underbrace{\texttt{occ} = \texttt{occ}}_{\texttt{VRAI}} \Rightarrow (\underbrace{\underbrace{\texttt{occ} = \texttt{vid}}_{\texttt{FAUX}} \Rightarrow \underbrace{1 < 1}_{\texttt{FAUX}}}_{\texttt{VRAI}})}_{\texttt{VRAI}})$$

qui est `VRAI`.

Vérification de l'obligation de preuve 2.6

On veut vérifier l'obligation de preuve

$$I \wedge dt = \texttt{occ} \Rightarrow \forall y \cdot (I \wedge J \wedge \neg\,(dt = \texttt{vid}) \Rightarrow [va := Va][dchgt](Va < va)).$$

Intuitivement, cette obligation est satisfaite car l'événement *dchgt* fait décroître le variant.

Cette obligation est équivalente à :

$$dt = \texttt{occ} \Rightarrow \forall dt \cdot (dt = \texttt{occ} \Rightarrow [va := f(dt)] \\ [\texttt{select}\ \ dt = \texttt{occ}\ \texttt{then}\ \ dt := \texttt{vid}\ \texttt{end}](f(dt) < va))$$

qui est équivalente à :

$$dt = \texttt{occ} \Rightarrow \forall dt \cdot (dt = \texttt{occ} \Rightarrow [va := f(dt)](dt = \texttt{occ} \Rightarrow [dt := \texttt{vid}](f(dt) < va)))$$

qui est équivalente à :

$$dt = \texttt{occ} \Rightarrow \forall dt_r \cdot (dt_r = \texttt{occ} \Rightarrow [va := f(dt_r)](dt_r = \texttt{occ} \Rightarrow 0 < va)).$$

Deux cas doivent être distingués, celui où $dt_r = \texttt{vid}$ et celui où $dt_r = \texttt{occ}$.

Premier cas : $dt_r = \texttt{vid}$

On obtient

$$dt = \texttt{occ} \Rightarrow (\texttt{vid} = \texttt{occ} \Rightarrow (\texttt{vid} = \texttt{occ} \Rightarrow 0 < 0))$$

qui est `VRAI`.

Deuxième cas : $dt_r = \texttt{occ}$

On obtient

$$dt = \texttt{occ} \Rightarrow (\texttt{occ} = \texttt{occ} \Rightarrow (\texttt{occ} = \texttt{occ} \Rightarrow 0 < 1))$$

qui est `VRAI`.

Vérification de l'obligation de preuve 2.7

L'obligation de preuve

$$I \wedge dt = \texttt{occ} \Rightarrow \forall y \cdot (I \wedge J \wedge \neg\,(dt = \texttt{vid}) \Rightarrow g(chgt) \vee g(dchgt))$$

est équivalente à

$$dt = \texttt{occ} \Rightarrow \forall dt_r \cdot (dt_r = \texttt{occ} \Rightarrow dt_r = \texttt{vid} \vee dt_r = \texttt{occ})$$

qui a trivialement la valeur `VRAI` car $dt_r = \texttt{occ} \Rightarrow dt_r = \texttt{vid} \vee dt_r = \texttt{occ}$ est une tautologie.

Conclusion de la preuve

Chacune des 7 obligations de preuve étant vérifiée, on en déduit que la contrainte dynamique exprimée dans la figure 2.20 est respectée par le système.

2.2 Le protocole BRP

2.2.1 Présentation

Le protocole BRP (Bounded Retransmission Protocol) a été défini et utilisé par la société *Philips*. Cette variante du protocole de bit alterné est un protocole de copie d'un fichier depuis un site appelé le *site émetteur* vers un autre appelé le *site récepteur*. Le fichier est organisé en paquets qui sont copiés un par un. La copie est totale en cas de bon fonctionnement et partielle en cas de rupture des communications.

Ce protocole a fait l'objet de nombreuses études (parmi lesquelles [GV96, HS96, dKRT96, DG97]) et il a notamment été spécifié en B dans [AM97].

Il fait ici l'objet de deux spécifications : une spécification abstraite et une spécification raffinée.

Principe de fonctionnement

L'émetteur envoie un premier paquet à destination du récepteur qui expédie en retour un accusé de réception. L'émetteur envoie ensuite le deuxième paquet, et ainsi de suite jusqu'à réception par l'émetteur de l'accusé de réception correspondant au dernier paquet à transmettre. La communication est réalisée au travers de deux canaux de communication : le canal de données et le canal d'acquittement. La figure 2.21 illustre le fonctionnement du BRP.

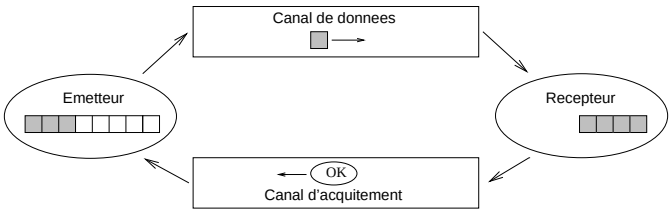

FIG. 2.21 – Communication entre émetteur et récepteur

Les canaux de communication pouvant être défectueux, certaines données (paquets ou accusés) peuvent se perdre. Pour prendre en compte de telles pertes, l'émetteur déclenche une horloge à chaque fois qu'il envoie un paquet. Si, passé un délai D, l'émetteur n'a toujours pas reçu l'accusé de réception, alors il réexpédie le même paquet. Après N tentatives infructueuses (pas d'accusé de réception) d'émission du même paquet, l'émetteur considère que les communications sont définitivement rompues et abandonne l'échange.

Envisageons maintenant les choses du point de vue du récepteur. A chaque réception d'un nouveau paquet, le récepteur déclenche une horloge. La réception répétée d'un même paquet en provenance de l'émetteur indique que ce dernier ne reçoit pas les accusés de réception et qu'il réexpédie donc le même paquet. Si, passé un délai supérieur à $N \times D$, le récepteur n'a toujours pas reçu de nouveau paquet, alors il est certain que l'émetteur a déjà abandonné et il abandonne à son tour.

Situations finales du protocole

Après une exécution complète du protocole, on se retrouve dans l'une des trois situations finales possibles :

1. le protocole s'est bien déroulé et le fichier a été entièrement copié d'un site à l'autre ;
2. le fichier a été entièrement copié sur le récepteur. Le récepteur envoie le dernier accusé de réception et termine normalement. Cet accusé se perd et l'émetteur, ne le recevant jamais, abandonne ;
3. les deux sites ont abandonné et le récepteur ne possède qu'une copie partielle du fichier à copier.

2.2.2 Spécification abstraite du BRP

Dans cette première approche de la spécification formelle, nous ne nous intéressons qu'à la terminaison du protocole sur chacun des sites. On considère à ce niveau que la copie du fichier, qu'elle soit partielle ou totale, s'effectue en une seule fois.

La spécification abstraite est décrite dans la figure 2.22 par un système d'événements B. Nous considérons dans notre spécification des fichiers à deux paquets, un fichier étant une séquence soit vide (notée `[]`), soit complète (notée `[1,2]`), soit partielle (notée `[1]`).

Parmi les variables utilisées dans la spécification abstraite présentée sur la figure 2.22, celles commençant par la lettre s représentent les variables liées au site émetteur (*sender* en anglais), et celles commençant par la lettre r représentent les variables liées au site récepteur (en anglais, *receiver*). Elles ont la signification suivante :

- sf_0 et rf_0 désignent respectivement les fichiers de l'émetteur et du récepteur,
- sa_0 et ra_0 indiquent respectivement si l'émetteur et le récepteur sont actifs (valeur à `act`) ou inactifs (valeur à `inact`),
- st_0 et rt_0 désignent le statut final du transfert du point de vue respectivement de l'émetteur et du récepteur. Ce transfert est considéré comme `complet` ou `partiel`.

La propriété invariante *(1)* donne le typage de chacune des variables. Le fichier source est une séquence de deux paquets. De plus, l'invariant établit que :

(2) le fichier reçu constitue un préfixe du fichier source,
(3) si l'émetteur a terminé en considérant que le fichier a été transmis en entier, alors le récepteur a terminé et le fichier a été entièrement copié,

```
SYSTEM BRP0
SETS Fichier0 = {[ ], [1], [1, 2]}
VARIABLES sf0, rf0, sa0, ra0, st0, rt0
INVARIANT
  sf0 = [1, 2] ∧ rf0 ∈ Fichier0 ∧ sa0 ∈ {act, inact} ∧ ra0 ∈ {act, inact} ∧
  st0 ∈ {complet, partiel} ∧ rt0 ∈ {complet, partiel} ∧                    (1)
  rf0 ⊆ sf0 ∧                                                              (2)
  ((sa0 = inact ∧ st0 = complet) ⇒ (ra0 = inact ∧ rt0 = complet)) ∧        (3)
  ((ra0 = inact ∧ rt0 = partiel) ⇒ (sa0 = inact ∧ st0 = partiel)) ∧        (4)
  (ra0 = inact ⇒ (rt0 = complet ⟺ (sf0 = rf0)))                           (5)
INITIALIZATION
  sf0 := [1, 2] || rf0 := [ ] || sa0, ra0 := act, act || st0, rt0 := partiel, partiel
EVENTS
  End_s     ≙ select sa0 = act ∧ ra0 = inact ∧ rt0 = complet
              then sa0, st0 := inact, complet end
  End_r     ≙ select ra0 = act ∧ sa0 = act
              then ra0, rt0, rf0 := inact, complet, sf0 end
  Abort_s   ≙ select sa0 = act
              then sa0, st0 := inact, partiel end
  Abort_r   ≙ select ra0 = act ∧ sa0 = inact ∧ st0 = partiel
              then ra0, rt0 := inact, partiel ||
                   any f' where f' ⊂ sf0 then rf0 := f' end
               end
END BRP0
```

FIG. 2.22 – Spécification abstraite du BRP

(4) si le récepteur a terminé en considérant que le fichier a été partiellement transmis, alors l'émetteur a terminé en sachant que le transfert n'a été que partiel,

(5) si le récepteur a terminé alors il considère que le fichier est entièrement copié si et seulement si il l'est.

Une représentation graphique des comportements de ce protocole d'après sa spécification abstraite est donnée au chapitre suivant par la figure 3.1 page 63.

L'initialisation correspond à la mise en activité du protocole sur chacun des sites.

Les événements `End_s` et `End_r` provoquent respectivement l'arrêt normal du protocole sur les sites émetteur et récepteur après le transfert du fichier. Les événements `Abort_s` et `Abort_r` provoquent respectivement l'abandon du transfert sur les sites émetteur et récepteur. Notons pour finir que l'on pourrait définir un événement appelé `Reinit_BRP` dont le rôle serait de réinitialiser le protocole en vue du transfert d'un nouveau fichier.

2.2.3 Spécification raffinée du BRP

Nous procédons maintenant au raffinement de la première spécification (la spécification abstraite) du protocole. La figure 2.23 donne cette spécification dans laquelle les deux points suivants sont détaillés :

```
SYSTEM BRP_1 refines BRP_0
CONSTANTS MAX = 2
SETS Fichier_1 = Fichier_0 ∪ {[2]}
VARIABLES sf_1, rf_1, sa_1, ra_1, st_1, rt_1, src_1
INVARIANT
  sa_1 = sa_0 ∧ ra_1 = ra_0 ∧ st_1 = st_0 ∧ rt_1 = rt_0 ∧                    (6)
  sf_1 ∈ Fichier_1 ∧ rf_1 ∈ Fichier_1 ∧
  src_1 ∈ [0 ··· MAX + 1] ∧                                                 (7)
  rf_1 ⌢ sf_1 = sf_0 ∧                                                      (8)
  ra_1 = act ⇒ rf_1 ≠ sf_0 ∧                                                (9)
  ra_1 = inact ⇒ rf_1 = rf_0 ∧                                              (10)
  (src_1 = MAX + 1 ⟺ sa_1 = inact ∧ st_1 = partiel) ∧                       (11)
  (src_1 = 0 ∧ sa_1 = inact ⟺ st_0 = complet)                               (12)
INITIALIZATION
  sf_1 = [1, 2] ‖ rf_1 = [ ] ‖ sa_1, ra_1 := act, act ‖
  src_1 := 0 ‖ st_1, rt_1 := partiel, partiel
EVENTS
  End_s      ≙ select sa_1 = act ∧ ra_1 = inact ∧ rt_1 = complet
               then sa_1, st_1, src_1 := inact, complet, 0 end
  End_r      ≙ select ra_1 = act ∧ sa_1 = act ∧ tail(sf_1) = [ ]
               then rf_1 := rf_1←first(sf_1) ‖ sf_1 := [ ]
                    ‖ ra_1, rt_1 := inact, complet end
  Abort_s    ≙ select sa_1 = act ∧ src_1 = MAX
               then sa_1, st, src_1 := inact, partiel, MAX + 1 end
  Abort_r    ≙ select ra_1 = act ∧ src_1 = MAX + 1
               then ra_1, rt_1 := inact, partiel end
  Resend_s   ≙ select sa_1 = act ∧ src_1 < MAX
               then src_1 := src_1 + 1 end
  Receive_r  ≙ select ra_1 = act ∧ tail(sf_1) ≠ [ ] ∧ sa_1 = act
               then rf_1, sf_1 := rf_1←first(sf_1), tail(sf_1) end
VARIANT f(sf_1, src_1) = size(sf_1) + MAX − src_1
END BRP_1
```

FIG. 2.23 – Spécification raffinée du BRP

1. la copie du fichier ne s'effectue plus en une seule fois mais paquet par paquet,
2. on introduit un compteur src_1 (pour *sender's retransmission counter*) au niveau du site émetteur ; il permet de borner le nombre de retransmissions d'un même paquet à destination du site récepteur.

Le compteur src_1 constitue la seule nouvelle variable introduite puisque sa_1, ra_1, st_1 et rt_1 sont identiques respectivement à sa_0, ra_0, st_0 et rt_0 (elles sont simplement renommées). Par contre, les variables sf_1 et rf_1 sont vues cette fois comme des séquences transmises par paquet.

On note MAX le nombre maximum de retransmissions autorisées d'un même paquet avant que l'émetteur n'abandonne.

L'invariant de collage *(6)* établit que les variables sa_1, ra_1, st_1 et rt_1 sont identiques à leur abstraction. L'invariant indique également :

(7) le typage de la nouvelle variable src_1,
(8) que sf_1 représente maintenant la partie du fichier source qui n'a pas encore été copiée sur le site récepteur,
(9) que tant que le récepteur est actif, alors le fichier n'a pas été entièrement transmis,
(10) que dans un état final, le fichier reçu (au niveau raffiné) est identique à son abstraction,
(11) que l'émetteur abandonne si et seulement si le nombre maximum de retransmissions infructueuses a été atteint,
(12) que s'il n'y a pas de retransmissions en cours alors que l'émetteur n'est plus actif, alors l'émetteur sait que le transfert a été complet.

Une représentation graphique du comportement du BRP décrit par la spécification raffinée est donnée au chapitre suivant dans la figure 3.2 page 65.

Deux nouveaux événements sont introduits :

- l'événement `Resend_s` réalise la retransmission par l'émetteur d'un paquet dont l'accusé de réception n'a pas été reçu,
- l'événement `Receive_r` réalise la réception et l'enregistrement par le récepteur d'un paquet en provenance de l'émetteur.

2.2.4 Expression des propriétés dynamiques du BRP

Comme dans la section 2.1.6, les contraintes dynamiques présentées ici n'incluent pas de liste `While` car on souhaite que les propriétés soient respectées quels que soient les événements appliqués.

Propriété dynamique de la spécification abstraite du BRP

Nous voulons, lors de la spécification abstraite du BRP, exprimer la propriété suivante :

(P15) lorsque le protocole est actif sur les deux sites, alors il termine inévitablement sur les deux sites, et dans l'une des trois situations finales présentées dans la section 2.2.1.

L'expression B de cette contrainte dynamique est donnée par la figure 2.24.

```
Select
  sa0 = act ∧ ra0 = act
Leadsto
  sa0 = inact ∧ ra0 = inact ∧
  ((st0 = complet ∧ rt0 = complet ∧ rf0 = sf0) ∨
  (st0 = partiel ∧ rt0 = complet ∧ rf0 = sf0) ∨
  (st0 = partiel ∧ rt0 = partiel ∧ rf0 ⊂ sf0))
Invariant
  TRUE
Variant
  length(sf0) − length(rf0) + f(ra0) + f(sa0) where f(act) = 1 and f(inact) = 0
End
```

FIG. 2.24 – Expression B de la propriété P15

Propriétés dynamiques de la spécification raffinée du BRP

Les 4 propriétés dynamiques suivantes sont introduites lors du premier raffinement du BRP :

(P16) un même paquet, même s'il est retransmis par l'émetteur, ne peut jamais être enregistré deux fois sur le récepteur,

(P17) le compteur de retransmissions prendra inévitablement la valeur 0 ou la valeur $MAX+1$,

(P18) si l'émetteur peut envoyer un paquet, alors ce paquet sera inévitablement reçu par le récepteur ou réexpédié MAX fois,

(P19) si les deux sites sont actifs, alors un paquet situé sur le site émetteur y reste jusqu'à ce qu'inévitablement soit l'émetteur abandonne, soit le paquet est reçu par le récepteur.

```
Dynamics
  Any A, B, d where
    A ∈ Fichier1 ∧ B ∈ Fichier1 ∧ d ∈ {[1], [2]}
  Then
    sf1 = d→A ∧ rf1 = B ∧ rf1' = B←d) ⇒ sf1' = A
```

FIG. 2.25 – Expression B de la propriété P16

La propriété P16 est modélisée par l'invariant dynamique donné dans la figure 2.25, tandis que les propriétés P17, P18 et P19 sont modélisées par les contraintes dynamiques données respectivement dans les figures 2.26, 2.27 et 2.28.

Pour formaliser la propriété P16, nous la réexprimons de la manière suivante : si un paquet se trouve en tête du fichier émetteur et qu'après application d'un événement il se retrouve en queue du fichier récepteur, alors il a été supprimé du fichier émetteur après l'application de cet événement.

```
Select
  TRUE
Leadsto
  src_1 = 0 ∨ src_1 = MAX + 1
Invariant
  TRUE
Variant
  2 × (length(sf_1) − length(rf_1)) + MAX + 1 − src_1 + 3 × (f(sa_1) + f(ra_1)) where
              f(act) = 1 and f(inact) = 0
End
```

FIG. 2.26 – Expression B de la propriété P17

```
Any A, B, d where
  A ∈ Fichier_1 ∧ B ∈ Fichier_1 ∧ d ∈ {[1], [2]}
Then Select
  sf_1 = d→A ∧ rf_1 = B
Leadsto
  rf_1 = B←d ∨ src_1 = MAX + 1
Invariant
  TRUE
Variant
  2 × (length(sf_1) − length(rf_1)) + MAX + 1 − src_1
End
```

FIG. 2.27 – Expression B de la propriété P18

```
Any A, B, d where
  A ∈ Fichier_1 ∧ B ∈ Fichier_1 ∧ d ∈ {[1], [2]}
Then Select
  sa_1 = act ∧ ra_1 = act ∧ sf_1 = d→A ∧ rf_1 = B
Until
  rf_1 = B←d ∨ src_1 = MAX + 1
Invariant
  TRUE
Variant
  2 × (length(sf_1) − length(rf_1)) + MAX + 1 − src_1 + f(sa_1) + f(ra_1) where
              f(act) = 1 and f(inact) = 0
End
```

FIG. 2.28 – Expression B de la propriété P19

2.3 Conclusion

Ces deux exemples nous ont permis d'illustrer le fait que l'expression et la vérification de propriétés dynamiques dans le formalisme B pouvaient se révéler des tâches ardues, dans la mesure où le spécifieur se voit dans l'obligation de fournir un variant et un invariant local pour chaque contrainte dynamique exprimée. De plus, la démonstration des obligations de preuve générées pour la vérification de chaque propriété est rarement établie de manière automatique, c'est à dire que le spécifieur est souvent amené à interagir avec les outils que sont les *theorem provers* (démonstrateurs de théorèmes) pour mener à bien les démonstrations.

Ces deux raisons motivent le fait que l'on puisse avoir envie de recourir à d'autres méthodes pour exprimer et vérifier des propriétés dynamiques dans des spécifications B. Un autre formalisme (la PLTL) permettant d'exprimer des propriétés dynamiques est présenté au chapitre 3, tandis qu'une méthode de vérification automatique de ces propriétés (le model-checking) est présenté au chapitre 4.

Chapitre 3

Systèmes de transitions et logiques temporelles

Dans ce chapitre, nous présentons la notion de système de transitions étiquetées. Nous utilisons cette notion pour définir la sémantique des systèmes d'événements présentés au chapitre 1. Un système de transitions étiquetées à nombre d'états finis permet la représentation graphique des comportements du système modélisé. Les systèmes de transitions donnent une définition des comportements par des chemins d'exécution, finis ou infinis.

Nous présentons également dans ce chapitre quelques unes des logiques temporelles, qui sont des formalismes plus expressifs que celui présenté au chapitre 1 pour l'expression des propriétés dynamiques que doit respecter un système. Nous présentons plus particulièrement la PLTL, dont la sémantique se définit par rapport à un chemin d'exécution d'un système de transitions étiquetées, car nos travaux, exposés dans la deuxième partie de ce document, reposent sur l'utilisation de la PLTL. Nous présentons également CTL* et CTL de manière succincte, sachant que la PLTL est une restriction de CTL*. C'est aussi le cas de CTL, mais les pouvoirs d'expression de CTL et de PLTL sont différents.

3.1 Systèmes de transitions

Les systèmes de transitions permettent d'exprimer la sémantique des systèmes d'événements, à partir de la partie opérationnelle de ceux-ci. Un système de transitions étiquetées est constitué d'un ensemble d'états, d'un ensemble de transitions et d'un alphabet permettant d'étiqueter les transitions.

Chaque état d'un système de transitions exprimant la sémantique d'un système d'événements correspond à une instanciation particulière des variables du système, appelée *interprétation* ou *décor*. Les transitions, quant à elles, représentent les occurrences des événements, et sont étiquetées par leurs noms.

Dans la section 3.1.1, la définition 5 définit la notion de système de transitions étiquetées. Les définitions 6, 7 et 8 définissent les notions de chemins d'exécution (respectivement fini, infini et maximal) issus de ces systèmes de transitions.

Dans la section 3.1.2, nous montrons quel système de transitions est associé à un système d'événements donné.

3.1.1 Définitions

Nous introduisons un ensemble fini de variables $X = \{x_1, \ldots, x_n\}$, chacune de ces variables étant respectivement typée sur un domaine fini $D_1, \ldots, D_n$. Pour chaque variable x_i, nous définissons l'ensemble des propositions atomiques

$$PA_i = \{x_i = v \ / \ v \in D_i\}.$$

Nous définissons $\mathbb{D}_X$, l'ensemble des propositions de la forme

$$\bigwedge_{i=1}^{n} x_i = v$$

pour toute proposition atomique $x_i = v$ de PA_i. L'ensemble $\mathbb{D}_X$ est appelé l'ensemble des décors des états pour un système dont l'ensemble de variables est X.

Nous définissons également $\mathbb{P}_X$, l'ensemble des propositions d'état défini par la grammaire suivante :

$$p ::= pa \mid p_1 \vee p_2 \mid \neg\, p$$

où $pa \in PA(= \bigcup_{i=1}^{n} PA_i)$ et où $p, p_1, p_2 \in \mathbb{P}_X$.

Remarque. L'ensemble des décors des états est inclus dans l'ensemble de propositions d'état : $\mathbb{D}_X \subset \mathbb{P}_X$.

Définition 5 (Système de transitions étiquetées).

Soit $\mathbb{D}_X$ un ensemble de décors d'état. Un système de transitions étiquetées *(STE en abrégé) est un quintuplet $ST = \langle S_0, S, A, T, \mathcal{L} \rangle$ où :*

- *S est un ensemble d'états,*
- *S_0 est l'ensemble des états initiaux : $S_0 \subseteq S$,*
- *A est l'alphabet des étiquettes des transitions,*
- *T est l'ensemble des transitions étiquetées : $T \subseteq S \times A \times S$,*
- *$\mathcal{L}$ est une application injective de S dans $\mathbb{D}_X$, $\mathcal{L} : S \to \mathbb{D}_X$.*

Un système de transitions étiquetées est *fini* quand l'ensemble S est fini. Une transition (s_i, e, s_j) de T est notée par $s_i \xrightarrow{e} s_j$.

Remarque. Dans une structure de Kripke, l'application $\mathcal{L}$ est une application de S dans l'ensemble 2^{PA_X}. Elle associe à un état s un ensemble de propositions atomiques vraies. Nous

avons choisi d'y associer une seule proposition, qui est en quelque sorte la conjonction de toutes les propositions atomiques vraies. Alors, l'état s satisfait la proposition p, et l'on note $s \models p$, si et seulement si $\mathcal{L}(s) \Rightarrow p$, tandis que dans une structure de Kripke, ceci s'exprimerait ainsi :

- $s \models pa$ si et seulement si $pa \in \mathcal{L}(s)$,
- $s \models \neg\, p$ si et seulement si $s \not\models p$,
- $s \models p_1 \vee p_2$ si et seulement si $s \models p_1$ ou $s \models p_2$.

Notre choix d'une telle fonction d'interprétation $\mathcal{L}$ a été fait par souci de compatibilité avec les notations B, pour les substitutions $[sub]\mathcal{L}(s)$ ou $\langle sub\rangle\mathcal{L}(s)$.

Définition 6 (Chemin d'exécution fini).

Soit $ST = \langle S_0, S, A, T, \mathcal{L}\rangle$, un système de transitions étiquetées. Un chemin d'exécution fini *de ST est une séquence d'états $\sigma = s_0 s_1 \cdots s_n$ telle que :*

$$s_0 \in S_0 \wedge \forall i \cdot i \in [1..n] \Rightarrow s_{i-1} \xrightarrow{e} s_i \in T.$$

Définition 7 (Chemin d'exécution infini).

Soit $ST = \langle S_0, S, A, T, \mathcal{L}\rangle$, un système de transitions étiquetées. Un chemin d'exécution infini *de ST est une séquence d'états $\sigma = s_1 s_2 \cdots s_i \cdots$ telle que :*

$$s_0 \in S_0 \wedge \forall i \cdot i > 0 \Rightarrow s_{i-1} \xrightarrow{e} s_i \in T.$$

La définition 8 introduit la notion de chemin d'exécution *maximal* d'un système de transitions ST, qui est soit un chemin d'exécution infini de ST, soit un chemin d'exécution fini dont le dernier état n'a de successeur par aucune transition de ST.

Définition 8 (Chemin d'exécution maximal).

Soit $ST = \langle S_0, S, A, T, \mathcal{L}\rangle$, un système de transitions étiquetées. Un chemin d'exécution maximal *de ST est une séquence d'états $\sigma = s_0 s_1 \cdots$ telle que :*

- *soit σ est un chemin d'exécution infini de ST,*
- *soit $\sigma = s_0 s_1 \cdots s_n$ est un chemin d'exécution fini de ST et on a :*

$$\forall s \cdot \forall e \cdot s \in S \wedge e \in A \Rightarrow \neg\, (s_n \xrightarrow{e} s \in T).$$

Notation. On note $\Sigma(ST)$ l'ensemble des chemins d'exécution maximaux d'un système de transitions ST.

Remarque. Dans la suite de ce document, un chemin d'exécution est appelé plus simplement une *exécution*.

3.1.2 Sémantique d'un système d'événements

La définition 9 définit la sémantique d'un système d'événements sous forme d'un système de transitions étiquetées. Dans cette thèse, on fait l'hypothèse que la condition de terminaison des événements est vraie. Sous cette hypothèse, dans [But99, BC00], les auteurs définissent la notion de « plus faible pré-condition conjuguée » de la manière suivante :

$$\langle sub\rangle p \mathrel{\widehat{=}} \neg\, [sub] \neg\, p.$$

L'expression $\langle sub \rangle p$ représente la plus faible pré-condition conjuguée qui garantit qu'il est possible pour un événement e défini par $e \mathrel{\widehat{=}} sub$ d'établir une proposition p.

Définition 9 (Sémantique d'un système d'événements).

Soit un système d'événements $\mathcal{S} = \langle X, I, F, Init, A, E_A \rangle$, *fini. Soit* $\mathbb{D}_X$, *l'ensemble des décors des états de* $\mathcal{S}$ *défini par* $\mathbb{D}_X = \{ \bigwedge_{i=1}^{n} (x_i = v_j) \ / \ x_i \in X \wedge v_j \in Domaine(x_i) \}$. *La sémantique de* $\mathcal{S}$ *est un système de transitions étiquetées* $ST = \langle S_0, S, A, T, \mathcal{L} \rangle$ *défini de la manière suivante :*

1. $S_0 = \{ s \ / \ \mathcal{L}(s) = \bigwedge_{i=1}^{n} (x_i = v_j) \wedge \langle Init \rangle \bigwedge_{i=1}^{n} (x_i = v_j) \}$,
2. $T = \{ s_1 \xrightarrow{e} s_2 \ / \ s_1 \in S \wedge (e \mathrel{\widehat{=}} sub) \in E_A \wedge \mathcal{L}(s_1) = \bigwedge_{i=1}^{n} (x_i = v_j) \wedge \mathcal{L}(s_2) = \bigwedge_{i=1}^{n} (x_i = v'_j) \wedge I \wedge \mathcal{L}(s_1) \Rightarrow \langle sub \rangle \mathcal{L}(s_2) \}$,

Remarque.

- $[sub]p$ est la notation ensembliste de la *weakest precondition* $wp(sub, p)$ (la plus faible pré-condition de la substitution *sub* qui assure que p est vrai après exécution de *sub*). Par exemple, la weakest precondition qui assure que $y = x$ après exécution de

  `CHOICE x := 4 OR x := 5 END`

  est $false$ ($y = 4 \wedge y = 5$) d'après le calcul des substitutions B ;
- $\langle sub \rangle p$ est la *conjugate weakest precondition* $\neg\, wp(sub, \neg p)$ (la plus faible pré-condition de la substitution *sub* qui donne la possibilité d'atteindre p après exécution de *sub*). Par exemple, la plus faible pré-condition conjuguée qui donne la possibilité de satisfaire $y = x$ après exécution de

  `CHOICE x := 4 OR x := 5 END`

  est $y = 4 \vee y = 5$.

La construction d'un STE à partir d'un système d'événements peut par exemple être obtenue en appliquant la méthode suivante (construction en largeur d'abord) : à partir de l'ensemble des états initiaux, on construit l'ensemble des sommets du graphe. Par application de toutes les transitions issues des états initiaux, on construit l'ensemble des successeurs de ces sommets, ainsi que les arcs qui les relient. Le même processus réitéré à partir des successeurs permet, génération après génération, de construire le STE complet.

Un tel STE ne contient que les nœuds correspondant à des états *accessibles*.

3.1.3 Application aux exemples

Nous présentons dans cette section les STEs correspondant aux systèmes de transitions qui sont les sémantiques des spécifications des exemples présentés au chapitre 2. A titre d'exemple, la correspondance entre la spécification et le STE est détaillée pour le cas de la spécification abstraite du BRP.

Le BRP

La figure 3.1 représente la sémantique de la spécification abstraite du BRP, donnée dans la figure 2.22 de la page 53. La direction horizontale correspond au déroulement normal du protocole, tandis que la direction verticale correspond au cas où les communications sont rompues.

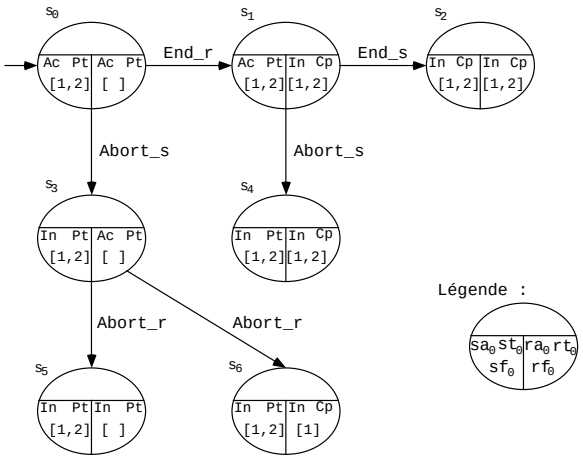

FIG. 3.1 – Le STE associé à la spécification abstraite du BRP

L'état initial s_0 est l'état obtenu par application de la partie `INITIALIZATION` de la spécification de la figure 2.22. L'ensemble S des états est $\{s_0, s_1, s_2, s_3, s_4, s_5, s_6\}$. Le décor des états est défini par l'application $\mathcal{L}$. Sur la figure 3.1, comme l'indique la légende, chaque état est décoré par la valeur des variables de la manière suivante : dans la partie gauche figurent respectivement les valeurs des variables de l'émetteur sa_0, st_0 et sf_0. Dans la partie droite on trouve les valeurs de ra_0, rt_0 et rf_0 pour le récepteur. Les valeurs de ces variables sont codées ainsi : `Ac` pour un site actif et `In` pour un site inactif ; `Pt` pour un transfert partiel et `Cp` pour un transfert complet ; `[]`, `[1]` et `[1,2]` pour des séquences respectivement vide, partielle et complète. L'application $\mathcal{L}$ est alors définie ainsi :

$$
\begin{array}{lll}
\mathcal{L}: & s_0 & \mapsto \ \mathtt{sa_0} = \mathtt{act} \wedge \mathtt{st_0} = \mathtt{partiel} \wedge \mathtt{sf_0} = [1,2] \\
 & & \quad \wedge\ \mathtt{ra_0} = \mathtt{act} \wedge \mathtt{rt_0} = \mathtt{partiel} \wedge \mathtt{rf_0} = [\,] \\
 & s_1 & \mapsto \ \mathtt{sa_0} = \mathtt{act} \wedge \mathtt{st_0} = \mathtt{partiel} \wedge \mathtt{sf_0} = [1,2] \\
 & & \quad \wedge\ \mathtt{ra_0} = \mathtt{inact} \wedge \mathtt{rt_0} = \mathtt{complet} \wedge \mathtt{rf_0} = [1,2] \\
 & s_2 & \mapsto \ \mathtt{sa_0} = \mathtt{inact} \wedge \mathtt{st_0} = \mathtt{complet} \wedge \mathtt{sf_0} = [1,2] \\
 & & \quad \wedge\ \mathtt{ra_0} = \mathtt{inact} \wedge \mathtt{rt_0} = \mathtt{complet} \wedge \mathtt{rf_0} = [1,2] \\
 & s_3 & \mapsto \ \mathtt{sa_0} = \mathtt{inact} \wedge \mathtt{st_0} = \mathtt{partiel} \wedge \mathtt{sf_0} = [1,2] \\
 & & \quad \wedge\ \mathtt{ra_0} = \mathtt{act} \wedge \mathtt{rt_0} = \mathtt{partiel} \wedge \mathtt{rf_0} = [\,] \\
 & s_4 & \mapsto \ \mathtt{sa_0} = \mathtt{inact} \wedge \mathtt{st_0} = \mathtt{partiel} \wedge \mathtt{sf_0} = [1,2] \\
 & & \quad \wedge\ \mathtt{ra_0} = \mathtt{inact} \wedge \mathtt{rt_0} = \mathtt{complet} \wedge \mathtt{rf_0} = [1,2] \\
 & s_5 & \mapsto \ \mathtt{sa_0} = \mathtt{inact} \wedge \mathtt{st_0} = \mathtt{partiel} \wedge \mathtt{sf_0} = [1,2] \\
 & & \quad \wedge\ \mathtt{ra_0} = \mathtt{inact} \wedge \mathtt{rt_0} = \mathtt{partiel} \wedge \mathtt{rf_0} = [\,]
\end{array}
$$

$$s_6 \mapsto \mathtt{sa}_0 = \mathtt{inact} \wedge \mathtt{st}_0 = \mathtt{partiel} \wedge \mathtt{sf}_0 = [1,2] \\ \wedge\ \mathtt{ra}_0 = \mathtt{inact} \wedge \mathtt{rt}_0 = \mathtt{partiel} \wedge \mathtt{rf}_0 = [1]$$

L'ensemble des transitions étiquetées est le suivant :

$$\{s_0 \xrightarrow{\texttt{End\_r}} s_1,\ s_1 \xrightarrow{\texttt{End\_s}} s_2,\ s_0 \xrightarrow{\texttt{Abort\_s}} s_3,\ s_3 \xrightarrow{\texttt{Abort\_r}} s_5,\ s_3 \xrightarrow{\texttt{Abort\_r}} s_6,\ s_1 \xrightarrow{\texttt{Abort\_s}} s_4\}.$$

Notons que les états notés s_2, s_4, s_5 et s_6 correspondent respectivement aux trois situations finales du protocole décrites dans la section 2.2.1 (s_5 et s_6 correspondent à la même situation finale).

La figure 3.2 montre le STE issu de la spécification raffinée du BRP présentée dans la figure 2.23 de la page 54. Le nombre d'états de ce graphe varie en fonction du nombre de paquets à transmettre et du nombre maximum de retransmissions autorisées pour un même paquet. Pour cette figure et les suivantes, afin de permettre une représentation compacte, nous avons arbitrairement donné la valeur 2 à la fois au nombre de paquets du fichier à transmettre et au nombre maximum de retransmissions autorisées (`MAX = 2`). Un plus grand nombre de paquets augmenterait le nombre de transitions `Receive_r` avant la transition `End_r` correspondant à la réception du dernier paquet (un nombre N de paquets à transmettre correspond à $N-1$ transitions `Receive_r`). Un plus grand nombre de retransmissions autorisées augmenterait le nombre de transitions `Resend_s` (un nombre N de retransmissions autorisées correspond à N transitions `Resend_s`). Notons que pour un nombre non fixé de retransmissions et de paquets, le STE serait à nombre d'états infini.

Le décor des états est représenté de la même manière que dans la figure 3.1, mais les notations `[]`, `[1]`, `[2]` et `[1,2]` représentent les fichiers respectivement vide, avec un paquet (`[1]` ou `[2]`), et plein. On a ajouté au dessus de la barre horizontale la valeur du compteur de retransmissions src_1.

Le robot

Nous donnons ici les STEs correspondant à la spécification abstraite du robot et à ses trois raffinements. Dans toutes les représentations graphiques de ces STEs, le décor des états indique de manière schématique l'état des variables du système. Le plateau du bas représente le dispositif d'arrivée (da), et la présence d'une pièce ou non indique si celui-ci est occupé ou vide ; le plateau du haut représente le dispositif d'évacuation (de), et la présence d'une pièce ou non indique si celui-ci est occupé ou vide. La pince (dt) est représentée au milieu des deux plateaux lorsque sa position ($posdt$) est indéterminée (cas de la spécification abstraite du robot et de ses deux premiers raffinements) ; elle est représentée en bas (vers le dispositif d'arrivée) lorsque $posdt$ = `bas` et en haut (vers le dispositif d'évacuation) lorsque $posdt$ = `haut` (cas du troisième raffinement du robot). Là encore, la présence d'une pièce ou non dans la pince indique si celle-ci est occupée ou vide.

La figure 3.3 représente le STE associé à la spécification abstraite du robot, donnée dans la figure 2.2 de la page 39.

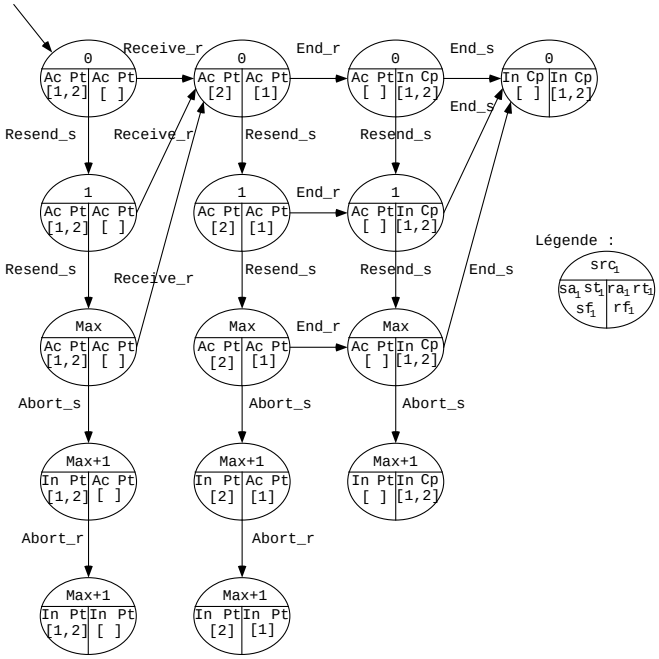

FIG. 3.2 – Le STE associé à la spécification raffinée du BRP

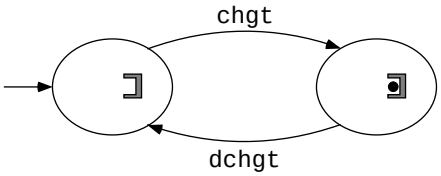

FIG. 3.3 – Le STE associé à la spécification abstraite du robot

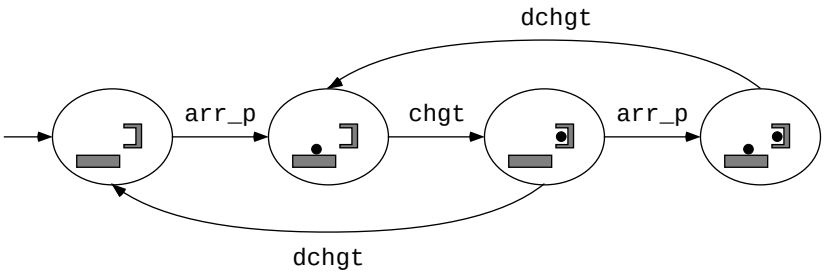

FIG. 3.4 – Le STE associé au premier raffinement du robot

Le STE associé au premier raffinement du robot (voir la figure 2.3, page 40), est donné par la figure 3.4.

Le STE associé au deuxième raffinement du robot (voir la figure 2.4, page 40), est donné par la figure 3.5.

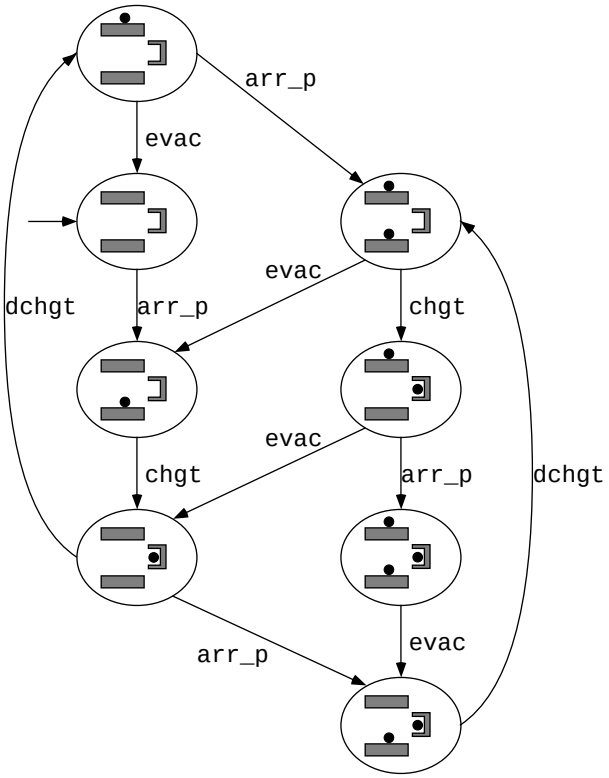

FIG. 3.5 – Le STE associé au deuxième raffinement du robot

Le STE associé au troisième raffinement du robot (voir la figure 2.5, page 41), est donné par la figure 3.6.

3.2 La logique temporelle linéaire propositionnelle

3.2.1 Le concept de logique temporelle

Les logiques temporelles sont utilisées dans la description de systèmes informatiques afin de spécifier des propriétés concernant le comportement dynamique du système. En ajoutant aux connecteurs et aux opérateurs de la logique classique des *opérateurs temporels*, elles permettent d'énoncer formellement des propriétés sur les enchaînements d'états, c'est à dire sur les exécutions du système étudié.

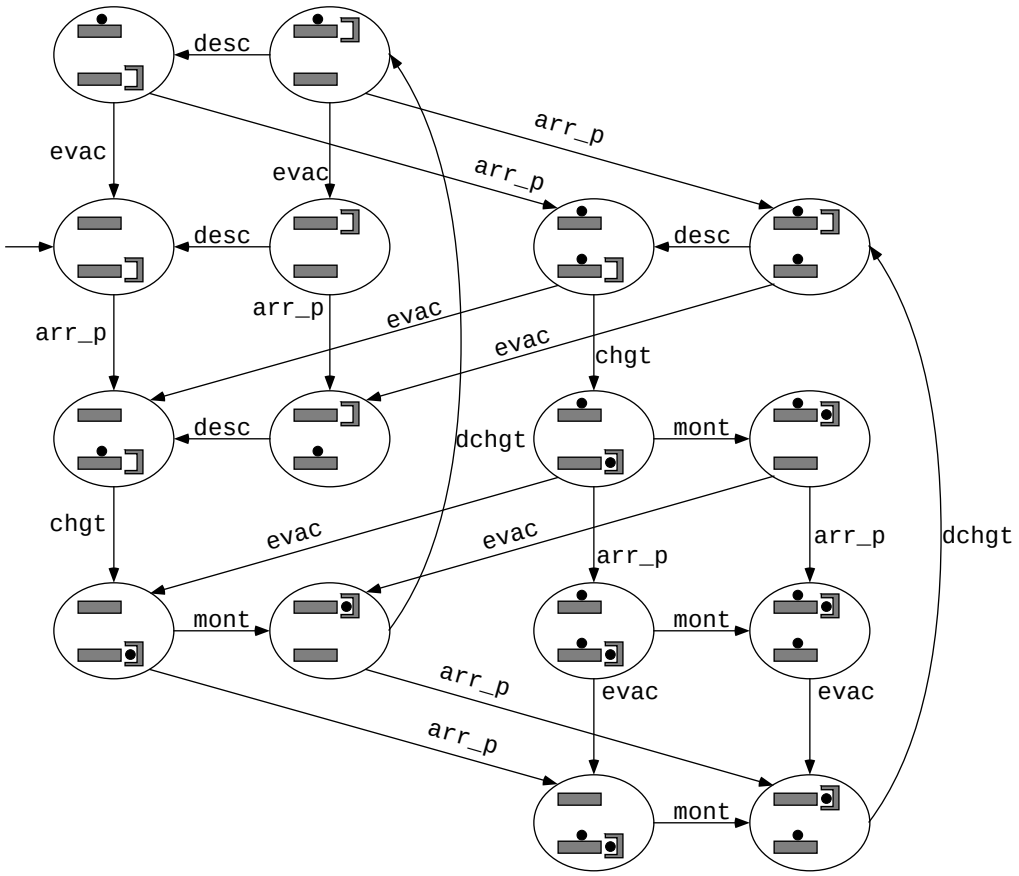

FIG. 3.6 – Le STE associé au troisième raffinement du robot

Ces logiques permettent de découper le temps en une suite d'instants, de telle sorte que le comportement du système pourra être observé le long d'une ou plusieurs successions de ces instants, chaque état du système lors d'une exécution correspondant alors à un instant particulier.

On peut ainsi exprimer les notions d'antériorité ou de postériorité d'un état par rapport à un autre. Par exemple, une propriété telle que « l'envoi de telle requête est toujours suivie de la réception de telle réponse » s'exprime facilement en logique temporelle.

Parmi les logiques temporelles, on distingue les logiques du temps linéaire (LTL pour *Linear Temporal Logic*) [Pnu81] et les logiques du temps arborescent (CTL pour *Computation Tree Logic*) [CE81, EH82]. Chacune a un pouvoir d'expression différent. Les premières spécifient des propriétés sur l'ensemble des exécutions d'un système considérées individuellement, c'est à dire qu'elles ne considèrent pas la façon dont les exécutions sont organisées en arbre. Dans une exécution, tout instant possède alors un et un seul successeur immédiat. En introduisant des quantificateurs universels et existentiels sur les chemins, les secondes permettent d'exprimer des propriétés prenant en compte l'aspect arborescent des exécutions. La figure 3.7 montre deux STEs dont les exécutions ne peuvent être distinguées par la logique temporelle linéaire. La logique temporelle arborescente permet de distinguer ces deux systèmes dont les comportements diffèrent par le moment où est effectué le choix non déterministe qui conduit à un état R ou à un état R'.

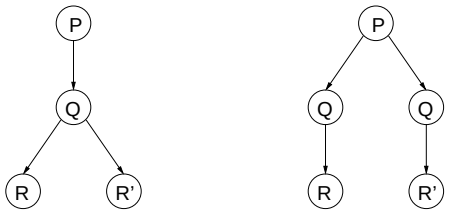

FIG. 3.7 – Deux STEs identiques pour la LTL

La logique que nous utilisons dans ce travail pour spécifier les propriétés dynamiques d'un système est la logique temporelle linéaire propositionnelle (PLTL pour *Propositional Linear Temporal Logic*), qui restreint la construction de formules LTL à l'utilisation de *propositions* reliées par des connecteurs et des opérateurs de la logique temporelle. La PLTL ne permet donc pas d'introduire de formules du premier ordre. Elle a un pouvoir d'expression suffisant pour exprimer les modalités B, les invariants dynamiques, ainsi que certaines classes de propriétés dynamiques couramment rencontrées dans la pratique sur des systèmes à états finis.

Notre choix de la PLTL plutôt qu'une autre logique temporelle pour exprimer les propriétés dynamiques est justifié dans la section 3.3.3.

Notons enfin que la PLTL que nous présentons dans la section suivante n'utilise que des opérateurs du futur. Il existe également des opérateurs du passé. Nous ne les présentons pas ici car les outils de vérification que sont les *model-checkers* ne les utilisent pas en général.

3.2.2 Définition et sémantique de la PLTL

La PLTL utilise les connecteurs $\neg$, $\vee$, $\wedge$, $\Rightarrow$, ... du calcul des propositions. Elle utilise également les opérateurs temporels[1] du futur suivants : $\Box$ *(toujours)*, $\bigcirc$ *(suivant)*, $\Diamond$ *(inévitablement)*, $\mathcal{U}$ *(jusqu'à)* et $\mathcal{W}$ *(à moins que)*. Leur signification intuitive est la suivante (p et q étant des propositions d'état) :

- $\Box\, p$: p est vraie dans tous les états futurs,
- $\bigcirc\, p$: p est vraie à l'état suivant,
- $\Diamond\, p$: p sera inévitablement vraie dans le futur,
- $p\, \mathcal{U}\, q$: p est vraie dans tous les états du futur jusqu'à un état où q est vraie, celui-ci étant inévitable,
- $p\, \mathcal{W}\, q$: p est vraie dans tous les états du futur jusqu'à un éventuel état où q est vraie.

Remarque. La lettre $\mathcal{U}$ est l'abréviation du terme anglais *until* signifiant *jusqu'à*. La lettre $\mathcal{W}$ est parfois présentée comme l'abréviation de *waiting for* (*en attendant*, ou *à moins que*) mais elle peut également être lue comme l'abréviation de *weak until* (*jusqu'à faible*).

La figure 3.8 illustre l'utilisation de ces opérateurs temporels par des exemples de séquences d'états satisfaisant les propriétés $\Box\, p$, $\Box\, (p \Rightarrow \bigcirc\, q)$, $\Box\, (p \Rightarrow \Diamond\, q)$, $p\, \mathcal{U}\, q$ et $p\, \mathcal{W}\, q$, où p et q sont des propositions d'état.

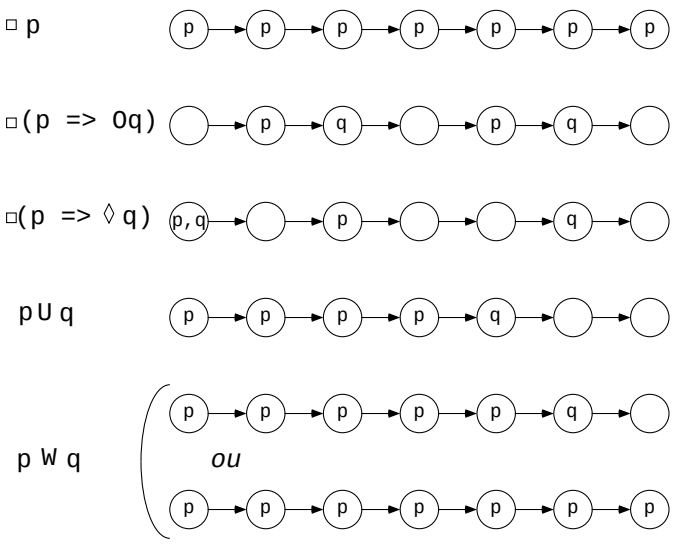

FIG. 3.8 – Exemples de propriétés PLTL

Définition 10 (Formule du futur de la PLTL).

Soit pa, une proposition atomique de l'ensemble $\bigcup_{i=1}^{n} PA_i$. *Une formule du futur* P *de la*

[1] Les notations $\Box$, $\bigcirc$ et $\Diamond$ de ces opérateurs sont couramment utilisées. Une autre notation de ces opérateurs est respectivement G, X et F.

PLTL est définie inductivement par la grammaire suivante :

$$P ::= pa \mid \neg P \mid P_1 \vee P_2 \mid \bigcirc P \mid P_1 \,\mathcal{U}\, P_2$$

où P, P_1 et P_2 sont des formules du futur de la PLTL.

Les autres opérateurs propositionnels, ainsi que les autres opérateurs temporels du futur, se redéfinissent à partir des 4 opérateurs $\neg$, $\vee$, $\bigcirc$ et $\mathcal{U}$ de la manière suivante :

- $P \Rightarrow Q \equiv \neg P \vee Q$
- $P \wedge Q \equiv \neg (\neg P \vee \neg Q)$
- $\Diamond P \equiv \texttt{VRAI}\, \mathcal{U}\, P$
- $\Box P \equiv \neg \Diamond \neg P$
- $P \,\mathcal{W}\, Q \equiv (P \,\mathcal{U}\, Q) \vee \Box P$

Les opérateurs du futur de la PLTL sont interprétés sur des chemins d'exécution de systèmes de transitions, comme indiqué dans la définition 11. Cette définition établit la validité d'une formule PLTL P sur un chemin d'exécution σ à l'état i, noté $(\sigma, i) \models P$. La notation $|\sigma|$ désigne la *longueur* d'un chemin d'exécution σ. La longueur d'un chemin d'exécution fini $\sigma = s_0 s_1 \cdots s_n$ est le nombre d'états qui le composent, c'est à dire $|\sigma| = n + 1$. La longueur d'un chemin d'exécution infini est ∞.

La définition 12 définit la validité d'une formule PLTL sur l'intégralité d'un système de transitions, en disant qu'un STE satisfait une propriété si tous ses chemins d'exécutions maximaux la satisfont.

Définition 11 (Sémantique de la PLTL).

Soit $\sigma = s_0 s_1 \cdots s_i \cdots$ un chemin d'un système de transitions $ST = \langle S_0, S, A, T, \mathcal{L} \rangle$. La sémantique de la PLTL est définie de la manière suivante :

- $(\sigma, i) \models p$ *si et seulement si* $\mathcal{L}(s_i) \Rightarrow p$,
- $(\sigma, i) \models \neg P$ *si et seulement si* $(\sigma, i) \not\models P$,
- $(\sigma, i) \models \bigcirc P$ *si et seulement si* $|\sigma| > i \wedge (\sigma, i+1) \models P$,
- $(\sigma, i) \models P_1 \,\mathcal{U}\, P_2$ *si et seulement si* $\exists j \cdot i \leq j \leq |\sigma| \wedge (\sigma, j) \models P_2 \wedge \forall k \cdot i \leq k < j \Rightarrow (\sigma, k) \models P_1$,
- $(\sigma, i) \models P_1 \vee P_2$ *si et seulement si* $(\sigma, i) \models P_1 \vee (\sigma, i) \models P_2$,
- $(\sigma, i) \models \Box P$ *si et seulement si* $\forall k \cdot i \leq k \leq |\sigma| \Rightarrow (\sigma, k) \models P$,
- $(\sigma, i) \models \Diamond P$ *si et seulement si* $\exists k \cdot i \leq k \leq |\sigma| \wedge (\sigma, k) \models P$,
- $(\sigma, i) \models P_1 \,\mathcal{W}\, P_2$ *si et seulement si* $(\sigma, i) \models \Box P_1 \vee (\sigma, i) \models P_1 \,\mathcal{U}\, P_2$.

Parce qu'elles sont couramment rencontrées dans la pratique, nous avons défini explicitement la sémantique des formules $\Box P$, $\Diamond P$ et $P_1 \,\mathcal{W}\, P_2$, même si celle-ci peut se déduire de celle des formules $P_1 \,\mathcal{U}\, P_2$, $P_1 \vee P_2$ et $\neg P$.

Définition 12 (Satisfaction d'une formule PLTL).

Une formule P de la PLTL est satisfaite par un système de transitions $ST = \langle S_0, S, A, T, \mathcal{L} \rangle$, et l'on note $ST \models P$, si et seulement si :

$$\forall \sigma \cdot (\sigma \in \Sigma(ST) \Rightarrow (\sigma, 0) \models P).$$

Lorsqu'une formule P de la PLTL n'est pas satisfaite par un système de transitions ST, on dit que ST *viole* la propriété, et l'on note $\neg\,(ST \models P)$, ce qui se traduit par :

$$\exists \sigma \cdot (\sigma \in \Sigma(ST) \wedge (\sigma, 0) \models \neg\, P).$$

Remarque. Par abus de langage, on dit souvent « P est vraie sur ST » pour signifier que P est satisfaite par ST, et « P est fausse sur ST » pour signifier que P n'est pas satisfaite par ST. De plus, $(\sigma, 0) \models P$ se note plus simplement $\sigma \models P$.

3.2.3 Expression des contraintes dynamiques B en PLTL

L'invariant dynamique et les modalités dynamiques B de systèmes à états finis peuvent s'exprimer par des formules de la PLTL. Considérons deux propositions p et q. L'invariant dynamique $\mathcal{P}(X, X')$ s'exprime par une formule PLTL de la forme $\Box\, P$, où P est défini par la grammaire suivante :

$$P, P_1, P_2 ::= pa \mid \bigcirc pa \mid P_1 \vee P_2 \mid \neg\, P$$

où pa est une proposition atomique.

Les modalités

$$\texttt{Select}\ p\ \texttt{Leadsto}\ q\ \texttt{Invariant}\ J\ \texttt{Variant}\ Va\ \texttt{End}$$

et

$$\texttt{Select}\ p\ \texttt{Until}\ q\ \texttt{Invariant}\ J\ \texttt{Variant}\ Va\ \texttt{End}$$

s'expriment respectivement par les formules PLTL $\Box\,(p \Rightarrow \Diamond\, q)$ et $\Box\,(p \Rightarrow p\ \mathcal{U}\ q)$.

3.2.4 Modélisation des propriétés des exemples

Nous proposons dans cette section de ré-exprimer au moyen de formules de la PLTL l'ensemble des propriétés dynamiques des deux exemples présentés au chapitre 2. Dans ce chapitre, la spécification du robot avait donné lieu à l'expression des propriétés dynamiques P1 à P14, tandis que la spécification du BRP avait donné lieu à l'expression des propriétés dynamiques P15 à P19. Le chapitre 2 présentait également l'expression, par des modalités ou des invariants dynamiques B, de chacune de ces propriétés.

Propriétés dynamiques du BRP

La propriété P15 (voir page 55), ajoutée à la spécification abstraite du BRP est exprimée en PLTL comme indiqué dans la figure 3.9.

La spécification raffinée du protocole est quant à elle enrichie des propriétés dynamiques P16, P17, P18 et P19 (voir page 56). L'expression PLTL de chacune de ces propriétés est donnée respectivement par les figures 3.10, 3.11, 3.12 et 3.13.

$$\Box\,(\mathtt{sa}_0 = \mathtt{act} \wedge \mathtt{ra}_0 = \mathtt{act}) \Rightarrow \Diamond\,((\mathtt{sa}_0 = \mathtt{inact} \wedge \mathtt{ra}_0 = \mathtt{inact}) \wedge ((\mathtt{st}_0 = \mathtt{complet} \wedge \mathtt{rt}_0 = \mathtt{complet} \wedge \mathtt{rf}_0 = \mathtt{sf}_0) \vee (\mathtt{st}_0 = \mathtt{partiel} \wedge \mathtt{rt}_0 = \mathtt{complet} \wedge \mathtt{rf}_0 = \mathtt{sf}_0) \vee (\mathtt{st}_0 = \mathtt{partiel} \wedge \mathtt{rt}_0 = \mathtt{partiel} \wedge \mathtt{rf}_0 \subset \mathtt{sf}_0)))$$

FIG. 3.9 – Expression PLTL de la propriété P15 de la spécification abstraite du BRP

$$\bigwedge_{\mathtt{A}\in\mathtt{Fichier}_1} \cdot \bigwedge_{\mathtt{B}\in\mathtt{Fichier}_1} \cdot \bigwedge_{\mathtt{d}\in\{[1],[2]\}} \cdot\ \Box\,(\mathtt{sf}_1 = \mathtt{d}{\rightarrow}\mathtt{A} \wedge \mathtt{rf}_1 = \mathtt{B} \wedge \bigcirc\,(\mathtt{rf}_1 = \mathtt{B}{\leftarrow}\mathtt{d}) \Rightarrow \bigcirc\,(\mathtt{sf}_1 = \mathtt{A}))$$

FIG. 3.10 – Expression PLTL de la propriété P16 de la spécification raffinée du BRP

$$\Box \Diamond\,(\mathtt{src}_1 = 0 \vee \mathtt{src}_1 = \mathtt{MAX} + 1)$$

FIG. 3.11 – Expression PLTL de la propriété P17 de la spécification raffinée du BRP

$$\bigwedge_{\mathtt{A}\in\mathtt{Fichier}_1} \cdot \bigwedge_{\mathtt{B}\in\mathtt{Fichier}_1} \cdot \bigwedge_{\mathtt{d}\in\{[1],[2]\}} \cdot\ \Box\,((\mathtt{sa}_1 = \mathtt{act} \wedge \mathtt{sf}_1 = \mathtt{d}{\rightarrow}\mathtt{A} \wedge \mathtt{rf}_1 = \mathtt{B} \Rightarrow \Diamond\,(\mathtt{rf}_1 = \mathtt{B}{\leftarrow}\mathtt{d} \vee \mathtt{src}_1 = \mathtt{MAX} + 1))$$

FIG. 3.12 – Expression PLTL de la propriété P18 de la spécification raffinée du BRP

$$\bigwedge_{\mathtt{A}\in\mathtt{Fichier}_1} \cdot \bigwedge_{\mathtt{B}\in\mathtt{Fichier}_1} \cdot \bigwedge_{\mathtt{d}\in\{[1],[2]\}} \cdot\ \Box\,((\mathtt{sa}_1 = \mathtt{act} \wedge \mathtt{ra}_1 = \mathtt{act} \wedge \mathtt{sf}_1 = \mathtt{d}{\rightarrow}\mathtt{A} \wedge \mathtt{rf}_1 = \mathtt{B}) \Rightarrow (\mathtt{sa}_1 = \mathtt{act} \wedge \mathtt{ra}_1 = \mathtt{act} \wedge \mathtt{sf}_1 = \mathtt{d}{\rightarrow}\mathtt{A} \wedge \mathtt{rf}_1 = \mathtt{B})\ \mathcal{U}\ (\mathtt{rf}_1 = \mathtt{A}{\leftarrow}\mathtt{d} \vee \mathtt{src}_1 = \mathtt{MAX} + 1))$$

FIG. 3.13 – Expression PLTL de la propriété P19 de la spécification raffinée du BRP

Propriétés dynamiques du robot

Les propriétés P1 et P2 (voir page 42), intégrées à la spécification abstraite du robot, sont exprimées par les formules PLTL données par les figures 3.14 et 3.15.

Les propriétés P3, P4 et P5 (voir page 43) du premier raffinement du robot sont exprimées en PLTL par les formules données respectivement dans les figures 3.16, 3.17 et 3.18.

Les propriétés P6, P7 et P8 (voir page 44) du deuxième raffinement du robot sont exprimées en PLTL par les formules données respectivement dans les figures 3.19, 3.20 et 3.21.

Les propriétés P9, P10, P11, P12, P13 et P14 (voir page 45) du troisième raffinement du robot sont exprimées en PLTL par les formules données respectivement dans les figures 3.22, 3.23, 3.24, 3.25, 3.26 et 3.27.

$$\Box\,(\mathtt{dt}_0 = \mathtt{occ} \Rightarrow \Diamond\,\mathtt{dt}_0 = \mathtt{vid})$$

FIG. 3.14 – Expression PLTL de la propriété P1 de la spécification abstraite du robot

$$\Box\,(\mathtt{dt}_0 = \mathtt{vid} \Rightarrow \Diamond\,\mathtt{dt}_0 = \mathtt{occ})$$

FIG. 3.15 – Expression PLTL de la propriété P2 de la spécification abstraite du robot

$$\Box\,((\mathtt{da}_1 = \mathtt{occ} \wedge \bigcirc\,\mathtt{da}_1 = \mathtt{vid}) \Rightarrow \mathtt{dt}_1 = \mathtt{vid})$$

FIG. 3.16 – Expression PLTL de la propriété P3 du premier raffinement du robot

3.3 Les autres logiques temporelles

La distinction entre les logiques du temps linéaire et les logiques du temps arborescent a été introduite par Leslie LAMPORT [Lam80]. Il remarque en effet que la formule $\Diamond\, p$ (« la proposition p sera inévitablement vraie dans le futur ») peut être interprétée de deux manières différentes dans le cas de programmes non-déterministes.

On représente l'exécution d'un programme non-déterministe par un arbre des calculs possibles.

Dans le cas de la LTL, $\Diamond\, p$ est équivalent d'un point de vue sémantique à $\neg\,\Box\,\neg\, p$, ce qui s'interprète par le fait que p sera vraie pour un état au moins dans le futur. Si l'on ne dispose pas d'opérateurs permettant de quantifier les chemins possibles, alors on ne peut pas

$$\Box\,(\mathtt{da}_1 = \mathtt{occ} \Rightarrow \mathtt{da}_1 = \mathtt{occ}\ \mathcal{U}\ \mathtt{dt}_1 = \mathtt{vid})$$

FIG. 3.17 – Expression PLTL de la propriété P4 du premier raffinement du robot

$$\Box\,((\mathtt{dt}_1 = \mathtt{occ} \wedge \mathtt{da}_1 = \mathtt{vid} \wedge \bigcirc\, \mathtt{dt}_1 = \mathtt{vid}) \Rightarrow \bigcirc\, \mathtt{da}_1 = \mathtt{vid})$$

FIG. 3.18 – Expression PLTL de la propriété P5 du premier raffinement du robot

$$\Box\,((\mathtt{dt}_2 = \mathtt{occ} \wedge \bigcirc\, \mathtt{dt}_2 = \mathtt{vid}) \Rightarrow \mathtt{de}_2 = \mathtt{vid})$$

FIG. 3.19 – Expression PLTL de la propriété P6 du deuxième raffinement du robot

$$\Box\,(\mathtt{dt}_2 = \mathtt{occ} \Rightarrow \mathtt{dt}_2 = \mathtt{occ}\ \mathcal{U}\ \mathtt{de}_2 = \mathtt{vid})$$

FIG. 3.20 – Expression PLTL de la propriété P7 du deuxième raffinement du robot

$$\Box\,(\mathtt{de}_2 = \mathtt{occ} \Rightarrow \Diamond\, \mathtt{de}_2 = \mathtt{vid})$$

FIG. 3.21 – Expression PLTL de la propriété P8 du deuxième raffinement du robot

$$\Box\,(\mathtt{posdt}_3 = \mathtt{haut} \wedge \mathtt{dt}_3 = \mathtt{vid} \Rightarrow \Diamond\, \mathtt{posdt}_3 = \mathtt{bas})$$

FIG. 3.22 – Expression PLTL de la propriété P9 du troisième raffinement du robot

$$\Box\,(\mathtt{posdt}_3 = \mathtt{bas} \wedge \mathtt{dt}_3 = \mathtt{occ} \Rightarrow \Diamond\, \mathtt{posdt}_3 = \mathtt{haut})$$

FIG. 3.23 – Expression PLTL de la propriété P10 du troisième raffinement du robot

$$\Box\,((\mathtt{posdt}_3 = \mathtt{haut} \wedge \bigcirc\, \mathtt{posdt}_3 = \mathtt{bas}) \Rightarrow \mathtt{dt}_3 = \mathtt{vid})$$

FIG. 3.24 – Expression PLTL de la propriété P11 du troisième raffinement du robot

$$\Box\,((\mathtt{posdt}_3 = \mathtt{haut} \wedge \mathtt{dt}_3 = \mathtt{occ}) \Rightarrow \mathtt{posdt}_3 = \mathtt{haut} \wedge \mathtt{dt}_3 = \mathtt{occ}\ \mathcal{U}\ \mathtt{dt}_3 = \mathtt{vid})$$

FIG. 3.25 – Expression PLTL de la propriété P12 du troisième raffinement du robot

$$\Box\,((\mathtt{posdt}_3 = \mathtt{bas} \wedge \bigcirc\, \mathtt{posdt}_3 = \mathtt{haut}) \Rightarrow \mathtt{dt}_3 = \mathtt{occ})$$

FIG. 3.26 – Expression PLTL de la propriété P13 du troisième raffinement du robot

$$\Box\,((\mathtt{posdt}_3 = \mathtt{bas} \wedge \mathtt{dt}_3 = \mathtt{vid}) \Rightarrow \mathtt{posdt}_3 = \mathtt{bas} \wedge \mathtt{dt}_3 = \mathtt{vid}\ \mathcal{U}\ \mathtt{dt}_3 = \mathtt{occ})$$

FIG. 3.27 – Expression PLTL de la propriété P14 du troisième raffinement du robot

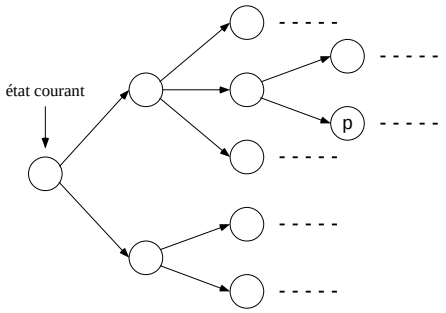

FIG. 3.28 – p est vraie dans un état du futur

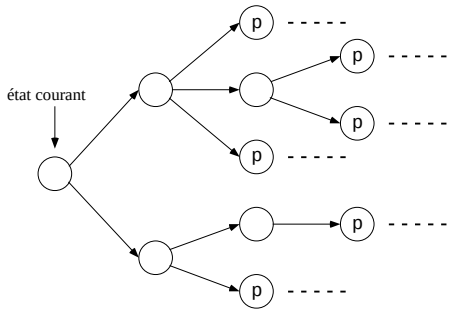

FIG. 3.29 – p est vraie dans tous les futurs possibles

distinguer le cas où p sera vraie dans un état du futur (voir la figure 3.28) de celui où p sera vraie dans tous les futurs possibles (voir la figure 3.29).

La logique du temps arborescent utilise les quantificateurs de chemin existentiel (noté E) et universel (noté A) pour distinguer ces deux interprétations différentes. Dans le cas de la CTL, ces quantificateurs s'appliquent directement aux opérateurs de la logique temporelle. Le STE de la figure 3.28 vérifie la formule $\mathsf{E}\Diamond p$, mais ne vérifie pas la formule $\mathsf{A}\Diamond p$. Le STE de la figure 3.29 vérifie à la fois les formules $\mathsf{E}\Diamond p$ et $\mathsf{A}\Diamond p$.

Intuitivement, les opérateurs précédés du quantificateur A expriment des propriétés vérifiées sur toutes les branches de l'arbre, tandis que ceux précédés de E expriment des propriétés vérifiées sur au moins un chemin de l'arbre.

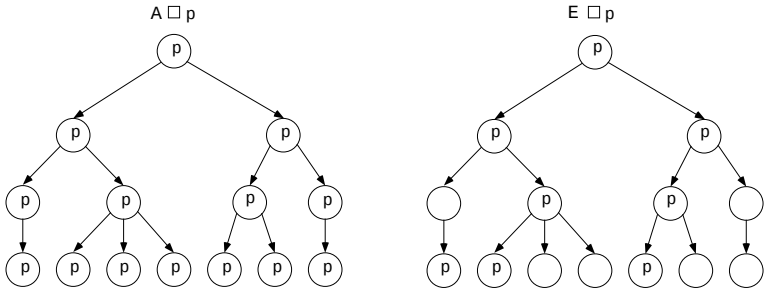

FIG. 3.30 – Combinaison des quantificateurs de chemins et de l'opérateur $\Box$

L'opérateur $\Box$ et le quantificateur A ne doivent pas être confondus. Le risque de confusion provient du fait qu'ils sont tous les deux des quantificateurs sur le futur. Le premier porte sur l'enchaînement des propriétés des états le long des séquences d'exécution alors que le deuxième définit les chemins considérés dans l'arbre. La figure 3.30 illustre cette différence. La propriété $\mathsf{A}\Box p$ énonce que toutes les séquences d'état atteignables vérifient la proposition $\Box p$, tandis que $\mathsf{E}\Box p$ énonce qu'il existe au moins un chemin qui vérifie $\Box p$.

3.3.1 La CTL*

La logique CTL*, permet, tout comme la PLTL présentée dans la section précédente, d'utiliser les opérateurs temporels. De plus, elle permet d'utiliser les quantificateurs de chemin existentiel et universel. Cette logique possède un plus grand pouvoir d'expression, qui recouvre celui de la PLTL.

La définition 13 présente les formules du futur de la CTL*.

Définition 13 (Formule du futur de la CTL*).

Soit pa, une proposition atomique. Une formule du futur de la CTL est définie par la grammaire suivante (où P, P_1 et P_2 sont des formules de la CTL*) :*

$$P ::= pa \mid \neg P \mid P_1 \vee P_2 \mid \bigcirc P \mid P_1 \,\mathcal{U}\, P_2 \mid \mathsf{A}\, P$$

Les formules du futur de la CTL* sont constituées des formules de la PLTL et de la formule $\mathsf{A}\,P$. Le quantificateur de chemin existentiel est défini à partir du quantificateur de chemin universel par $\mathsf{E}\,P \equiv \neg\,\mathsf{A}\,\neg\,P$.

La sémantique de la formule $\mathsf{A}\,P$ est la suivante : $(\sigma, i) \models \mathsf{A}\,P$ si et seulement si pour tout chemin σ' tel que $\sigma'_0 \ldots \sigma'_i = \sigma_0 \ldots \sigma_i$, on a $(\sigma', i) \models P$.

3.3.2 La CTL

La CTL est une logique du temps arborescent, qui exige que chaque opérateur temporel soit sous la portée immédiate d'un quantificateur A ou E. Ainsi, les STEs des figures 3.28 et 3.29 peuvent être distingués. Cependant, contrairement à la CTL*, cette logique impose de faire précéder chaque opérateur temporel d'un quantificateur, ce qui a pour effet de limiter son pouvoir d'expression tout en rendant parfois contraignante l'écriture des formules.

Définition 14 (Formule du futur de la CTL).

Soit pa, une proposition atomique. Une formule du futur de la CTL est définie par la grammaire suivante (où P, P_1 et P_2 sont des formules de la CTL) :

$$P ::= pa \mid \neg\,P \mid P_1 \vee P_2 \mid \mathsf{E}\bigcirc P \mid \mathsf{A}\bigcirc P \mid \mathsf{E}\,P_1\,\mathcal{U}\,P_2 \mid \mathsf{A}\,P_1\,\mathcal{U}\,P_2$$

A titre d'exemple, nous exprimons dans le formalisme de la CTL certaines des propriétés dynamiques du robot. Les expressions CTL des propriétés P1, P3, P4 et P5 sont les suivantes :

P1 : $\mathsf{A}\Box\,(dt_0 = \texttt{occ} \Rightarrow \mathsf{A}\Diamond\,dt_0 = \texttt{vid})$
P3 : $\mathsf{A}\Box\,((da_1 = \texttt{occ} \wedge \mathsf{E}\bigcirc\ da_1 = \texttt{vid}) \Rightarrow dt_1 = \texttt{vid})$
P4 : $\mathsf{A}\Box\,(da_1 = \texttt{occ} \Rightarrow \mathsf{A}(da_1 = \texttt{occ}\,\mathcal{U}\,dt_1 = \texttt{vid}))$
P5 : $\mathsf{A}\Box\,((dt_1 = \texttt{occ} \wedge da_1 = \texttt{vid} \wedge \mathsf{E}\bigcirc\ dt_1 = \texttt{vid}) \Rightarrow \mathsf{A}\bigcirc\ da_1 = \texttt{vid})$

3.3.3 Choix d'une logique temporelle

Le pouvoir d'expression de la logique CTL* recouvre ceux des logiques LTL et CTL (voir la figure 3.31).

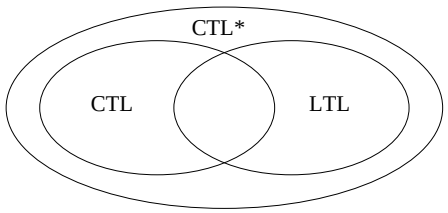

FIG. 3.31 – Pouvoirs d'expression comparé des logiques LTL, CTL et CTL*

Les logiques CTL et PLTL constituent deux restrictions de la CTL*. La première permet d'exprimer un sous-ensemble des propriétés CTL* sur des arbres d'exécution. La seconde permet d'exprimer des propriétés sur des séquences d'états.

Dans ce travail, nous avons choisi d'utiliser la PLTL afin d'exprimer des propriétés dynamiques dans le cadre des spécifications B. Celle-ci possède en effet un pouvoir d'expression suffisant pour pouvoir exprimer dans un autre formalisme les contraintes dynamiques proposées par Jean-Raymond ABRIAL dans le cadre de la méthode B (voir la définition 3, section 1.2.1), mais aussi un grand nombre des propriétés dynamiques que nous rencontrons dans la pratique, et qui ne sont pas forcément exprimables sous forme de contraintes dynamiques « à la B ».

La raison principale qui nous amène à choisir la PLTL est sa compatibilité avec le raffinement B. En effet, toute propriété PLTL qui n'utilise pas l'opérateur $\bigcirc$ est préservée[2] par le raffinement B car celui-ci n'introduit ni *livelock* ni *deadlock*, et que les nouveaux événements raffinent tous `skip`. Au contraire, une propriété CTL telle que $\mathsf{E}\Diamond p$ n'est pas forcément préservée à cause de la disparition du non-déterminisme par raffinement.

3.4 Conclusion

Nous avons présenté dans ce chapitre les systèmes de transitions étiquetées, que nous utilisons pour définir la sémantique des systèmes d'événements B présentés au chapitre 1.

Les logiques temporelles permettent de spécifier des comportements dynamiques, et leur sémantique se définit par rapport à des chemins d'exécution de systèmes de transitions étiquetées.

Nous proposons de spécifier les comportements dynamiques accompagnant les systèmes d'événements B en utilisant la logique PLTL, en raison de sa compatibilité avec le raffinement B. La vérification des propriétés ainsi spécifiées pourra être effectuée par la technique du model-checking PLTL, que nous présentons au chapitre suivant.

[2] Nous précisons qu'une propriété PLTL de la forme $\Box(p \Rightarrow \bigcirc q)$ est préservée sous la forme $\Box(p \Rightarrow p\,\mathcal{U}\,q)$.

Chapitre 4

Les automates et le model-checking PLTL

Le model-checking (que l'on traduit littéralement par « vérification de modèles ») est une technique de vérification permettant de prouver la cohérence de deux modèles : celui du système et celui de ses propriétés. Le model-checking est réalisé par des algorithmes qui permettent de savoir si un système de transitions donné vérifie une formule de logique temporelle.

L'algorithme de model-checking pour la PLTL, qui est dû essentiellement à Lichtenstein, Pnueli, Vardi et Wolper [LP85, VW86], effectue la vérification d'une formule temporelle en utilisant une représentation de celle-ci par un automate d'une classe particulière appelé automate de Büchi.

L'idée maîtresse du model-checking PLTL est de faire évoluer conjointement, en faisant leur produit synchrone, le système de transitions modélisant le système et l'automate modélisant la négation d'une propriété à vérifier, afin de prouver qu'aucun chemin d'exécution du système de transitions n'est accepté par l'automate de Büchi de la négation de la propriété. Si c'est le cas, alors le modèle du système ne satisfait pas la négation de la propriété, et donc il satisfait la propriété elle-même.

Les algorithmes naïfs de model-checking énumèrent de manière exhaustive les états et les transitions d'un système, afin de s'assurer qu'aucun chemin ne viole les propriétés spécifiées. Il en ressort que cette technique ne peut être appliquée qu'à des systèmes dont il est possible de donner une énumération exhaustive des états, c'est à dire des systèmes dits à *états finis*. Des techniques d'abstraction permettent toutefois de lever cette limite.

Le principal avantage lié à l'utilisation du model-checking est que cette technique de vérification est à la fois simple à mettre en œuvre et entièrement automatisable. Si le modèle du système est de taille exploitable, alors il existe des algorithmes implantés dans des *model-checkers* capables de répondre à la question : « le modèle du système satisfait-il la propriété φ ? », sans assistance de la part de l'utilisateur. De plus, lorsque la réponse est non, la plupart des model-checkers exhibent un chemin d'exécution violant la propriété, ce qui facilite la correction de l'erreur.

Cependant, outre le fait que cette technique ne permet d'adresser que des modèles dont le nombre d'états est fini, le principal problème pour l'utilisation du model-checking est connu sous le nom d'*explosion combinatoire du nombre d'états*. La modélisation de systèmes réels (qui peuvent être complexes) conduit généralement à la création d'un nombre d'états tel qu'il devient impossible de tous les conserver dans la mémoire d'un ordinateur, ce qui empêche en pratique l'exploration exhaustive de l'espace d'états. Des techniques permettant de combattre ce phénomène d'explosion combinatoire doivent alors être appliquées pour rendre possible le model-checking de systèmes de grande taille.

4.1 Théorie des langages et automates

Dans cette section, nous rappelons quelques concepts fondamentaux de la théorie des langages.

Ces notions sont utiles à notre propos dans le sens où le model-checking PLTL peut être vu comme la vérification que tous les « mots » définis par les chemins d'exécution d'un système de transitions sont des mots appartenant au « langage » que définit un propriété PLTL. Plus précisément, le model-checking PLTL s'assure qu'aucun des mots définis par un chemin d'exécution du système de transitions n'est un mot du langage défini par la négation d'une propriété.

Nous avons présenté au chapitre 3 le modèle du système étudié comme étant un système de transitions étiquetées. Une exécution d'un tel système est décrite par une séquence (potentiellement infinie) d'états de son système de transitions. Grâce à la fonction $\mathcal{L}$ (voir la définition 5), une proposition booléenne est associée à chaque état , de telle sorte qu'un chemin d'exécution du système de transitions peut être vu comme un mot dont chaque lettre est une proposition d'état, c'est à dire comme un mot formé sur l'alphabet défini par l'ensemble des propositions d'état (noté $\mathbb{P}_X$, et défini au chapitre 3).

Les propriétés que doivent respecter le système sont décrites par un ensemble de formules PLTL. Il a été établi dans [SPH84] qu'à partir de toute formule PLTL on peut construire un automate fini reconnaissant les mots satisfaisant la formule. Ainsi, en réalisant le produit synchrone de l'automate qui reconnaît les exécutions satisfaisant la négation d'une propriété avec le système de transitions qui modélise le système, on calcule l'intersection des deux langages. Si celle ci définit un langage vide, alors cela prouve que le système de transitions satisfait la propriété.

4.1.1 Éléments de la théorie des langages

Nous ne donnons pas ici de définition formelle mais nous nous contentons de la présentation intuitive des notions de mots finis et infinis.

Considérons un alphabet non vide A. Un *mot fini* sur A est un élément de A^*, c'est à dire une suite finie $a_0, \cdots, a_n$ de lettres de A. Un *mot infini* sur A est un élément de A^ω, c'est à dire une suite infinie $a_0, a_1, \cdots$ de lettres de A.

Parmi l'ensemble des mots qu'il est possible de construire à partir d'un alphabet, on ne s'intéresse souvent qu'à un sous-ensemble de mots « bien formés », c'est à dire respectant une règle de construction donnée.

Pour définir des langages *réguliers*, composés de mots finis, on utilise une *expression régulière*. Cette notion est étendue aux mots infinis par le biais des *expressions ω-régulières*, qui définissent des langages *ω-réguliers*.

Les expressions régulières permettent de décrire des ensembles de mots. Elles sont exprimées à l'aide des symboles « $+$ » pour exprimer un choix et « $*$ » en exposant pour exprimer la répétition d'un motif un nombre arbitraire mais fini de fois.

Par exemple, l'ensemble des mots définis sur l'alphabet $A = \{\mathtt{a}, \mathtt{b}\}$ commençant par la lettre `b` et tels que toute lettre `b` ultérieure est précédée au moins d'une lettre `a` est défini par l'expression régulière suivante : $b.(a + ab)^*$.

Les expressions ω-régulières permettent d'étendre la notion d'expression régulière aux mots infinis en autorisant la répétition de motifs un nombre infini de fois. En plus des deux symboles « $+$ » et « $*$ » déjà utilisés par les expressions régulières, les expressions ω-régulières utilisent le symbole « ω » (en exposant) pour indiquer une répétition infinie.

Par exemple, l'ensemble des mots définis sur l'alphabet $A = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ commençant par une suite finie de lettres `a` et `b` et se terminant par une suite infinie de lettres `c` est définie par l'expression ω-régulière suivante : $(a + b)^*.c^\omega$.

4.1.2 Automates et automates de Büchi

On utilise des automates pour *reconnaître* des langages réguliers. Par rapport aux notations que nous utilisons dans ce document, un automate fini est un système de transitions étiquetées fini (voir la définition 5) dépourvu de la fonction $\mathcal{L}$ d'interprétation des états, mais muni d'un ensemble d'états noté $\mathcal{F}$, et appelé l'ensembles des *états terminaux* de l'automate. Les étiquettes des transitions sont des lettres de l'alphabet sur lequel est défini le langage régulier à reconnaître.

Notons que si les automates permettent de reconnaître des langages réguliers, ils permettent aussi de les définir. La correspondance entre les notions d'expression régulière et d'automate est donnée par le théorème de Kleeene [Kle56] et ses extensions : *un langage est définissable par une expression régulière si et seulement si c'est un langage reconnaissable par un automate fini.*

On dit d'un mot appartenant au langage défini par un automate que c'est un mot *accepté* par l'automate.

Les automates utilisés pour reconnaître les langages ω-réguliers sont appelés des *automates de Büchi* [Büc62]. Dans le cas des automates de Büchi, les états de l'ensemble $\mathcal{F}$ sont appelés *états d'acceptation.* Il n'y a pas de différence entre un automate et un automate de Büchi, c'est la définition de l'acceptation d'un mot qui change selon que le mot est fini ou non.

Par abus de langage, nous parlerons toujours dans ce document d'automates de Büchi, que ce soit pour reconnaître un langage régulier ou un langage ω-régulier. Les automates de Büchi sont définis par la définition 15. En revanche, les définitions 16 et 17 d' acceptation de mots respectivement finis et infinis sont différentes

Définition 15 (Automate de Büchi).

Un automate (de Büchi) est un quintuplet $\mathcal{B} = \langle b_{init}, B, A, T_{\mathcal{B}}, \mathcal{F} \rangle$ *où :*

- b_{init} *est l'état initial,*
- B *est l'ensemble fini des états, incluant l'état* b_{init}*,*
- A *est un alphabet permettant d'étiqueter les transitions,*
- $T_{\mathcal{B}}$ *est l'ensemble fini des transitions étiquetées :* $T_{\mathcal{B}} \subseteq B \times A \times B$*,*
- $\mathcal{F}$ *est l'ensemble fini des états d'acceptation de l'automate :* $\mathcal{F} \subseteq B$*.*

Notation. On note $\Sigma(\mathcal{B})$ l'ensemble des séquences d'états d'un automate de Büchi $\mathcal{B} = \langle b_{init}, B, A, T_{\mathcal{B}}, \mathcal{F} \rangle$. On a que $\tau = b_0 b_1 \ldots$ est une séquence d'états de $\mathcal{B}$ (c'est à dire $\tau \in \Sigma(\mathcal{B})$) si et seulement si $b_0 = b_{init}$ et $\forall i \cdot i \geq 0 \Rightarrow b_i \xrightarrow{a_i} b_{i+1} \in T_{\mathcal{B}}$.

Remarque. Les automates de Büchi utilisés pour reconnaître les langages définis par les propriétés PLTL utilisent comme alphabet l'ensemble des propositions d'état $\mathbb{P}_X$.

Représentation graphique des automates

On représente graphiquement un automate en associant un cercle à chaque état et une flèche à chaque transition. L'état initial est repéré par une flèche incidente qui ne provient pas d'un état source, et les états d'acceptation sont repérés par des doubles cercles. On numérote généralement les états de 0 à n ($n+1$ est le nombre d'états de l'automate) et les flèches étiquetées représentent les transitions.

Les automates présentés dans les figures 4.1 et 4.2 permettent respectivement de reconnaître les langages $b.(a+ab)^*$ (langage régulier) et $(a+b)^*.c^\omega$ (langage ω-régulier) vus précédemment.

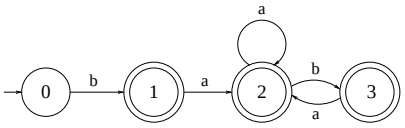

FIG. 4.1 – Un automate reconnaissant le langage régulier $b.(a+ab)^*$

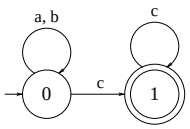

FIG. 4.2 – Un automate de Büchi reconnaissant le langage ω-régulier $(a+b)^*.c^\omega$

Remarque. Dans la figure 4.2, on remarque qu'on représente deux transitions (l'une portant l'étiquette `a` et l'autre l'étiquette `b`) par une seule flèche étiquetée par les deux lettres `a` et `b` séparées par une virgule.

Acceptation des mots d'un langage

On dit souvent qu'un automate permet de réaliser la « lecture » d'un mot (voir par exemple [Var96, CGP99]. Un automate accepte un mot en le lisant lettre par lettre, de gauche à droite. Partant de l'état initial, l'automate évolue vers un état successeur si l'une des transitions issues de l'état initial a pour étiquette la première lettre du mot. Le processus est réitéré à partir de cet état successeur avec la deuxième lettre du mot, et ainsi de suite...

La définition 16 établit qu'un mot fini est accepté par un automate si celui-ci ne s'est pas bloqué au cours de la lecture et qu'il a terminé dans un état d'acceptation.

Définition 16 (Acceptation d'un mot fini).

Une séquence $\tau = b_0 \cdots b_{n+1}$ *d'un automate* $\mathcal{B} = \langle b_{init}, B, A, T_{\mathcal{B}}, \mathcal{F} \rangle$ *accepte un mot* $\sigma = a_0 \cdots a_n$ *de l'alphabet* A *si et seulement si :*

1. $b_0 = b_{init} \wedge \forall i \cdot (i \in [0 \ldots n] \Rightarrow b_i \xrightarrow{a_i} b_{i+1} \in T_{\mathcal{B}})$
2. $b_{n+1} \in \mathcal{F}$.

Les mots infinis donnent lieu à des lectures infinies, aussi on ne peut pas utiliser la notion d'état terminal dans lequel s'arrêterait l'automate de Büchi à la fin de la lecture d'un mot pour définir son acceptation. On dit qu'un mot infini est accepté par un automate de Büchi s'il existe un état d'acceptation par lequel la lecture passe infiniment souvent. C'est ce qu'exprime la définition 17.

Définition 17 (Acceptation d'un mot infini).

Une séquence $\tau = b_0 b_1 \cdots$ *d'un automate de Büchi* $\mathcal{B} = \langle b_{init}, B, A, T_{\mathcal{B}}, \mathcal{F} \rangle$ *accepte un mot* $\sigma = a_0 a_1 \cdots$ *de l'alphabet* A *si et seulement si :*

1. $b_0 = b_{init} \wedge \forall i \cdot (i \geq 0 \Rightarrow b_i \xrightarrow{a_i} b_{i+1} \in T_{\mathcal{B}})$
2. *il existe un état d'acceptation apparaissant une infinité de fois dans* τ.

4.2 Model-checking PLTL

Puisque chaque chemin d'exécution d'un système de transitions peut être vu comme un mot dont chaque lettre est une proposition d'état, alors l'ensemble des chemins d'exécution d'un système de transitions définit un langage formé sur l'alphabet défini par l'ensemble des propositions d'état. Nous appelons ce langage le *langage du système de transitions.* Nous avons vu également qu'on associait à chaque formule P de la PLTL un langage ω-régulier, décrit par une expression ω-régulière décrivant la forme que doit respecter un chemin d'exécution pour satisfaire P. Nous appelons ce langage le *langage de la propriété.*

Le principe du model-checking est de vérifier que le langage du système de transitions est inclus dans celui de la propriété. Ceci est équivalent à vérifier que l'intersection du langage du système de transitions avec le langage de la négation de la propriété est un langage vide. En pratique, on calcule l'intersection de deux langages en réalisant le produit synchrone des automates les décrivant.

Nous précisons dans cette section quels sont les automates permettant de reconnaître des exécutions satisfaisant des propriétés PLTL (ou leurs négations, qui restent des propriétés PLTL), et nous définissons le produit synchrone d'un système de transitions avec un automate de Büchi. Puis, nous donnons intuitivement le principe de l'algorithme de model-checking PLTL.

4.2.1 Automates de reconnaissance de propriétés PLTL

On considère un système de transitions étiquetées dont l'ensemble des variables est X. Les formules du futur de la PLTL exprimant des propriétés par rapport à ce système se construisent à partir de l'ensemble $\mathbb{P}_X$ défini dans la section 3.1.1, et des opérateurs du futur (voir la définition 10 au chapitre 3). La définition 11 définit quant à elle la sémantique de la PLTL par rapport à un chemin d'exécution maximal d'un système de transitions, que nous appelons désormais une exécution.

Une exécution peut ainsi être vue comme un mot (fini ou infini) sur l'alphabet $\mathbb{P}_X$, et le théorème 1 [MSS88, Var94, VW94] établit qu'on peut toujours construire un automate de Büchi reconnaissant l'ensemble des exécutions satisfaisant une propriété PLTL donnée.

Théorème 1. *Étant donnée une formule P de la PLTL, on peut construire un automate de Büchi $\mathcal{B} = \langle b_{init}, B, \mathbb{P}_X, T_\mathcal{B}, \mathcal{F} \rangle$, tel que le langage défini par $\mathcal{B}$ est exactement l'ensemble des exécutions satisfaisant la formule P.*

Remarque. Par abus de langage, nous parlerons d'un automate de Büchi *reconnaissant* une propriété PLTL, ou encore d'un automate de Büchi *codant* une propriété PLTL, afin de signifier un automate de Büchi reconnaissant les *exécutions* satisfaisant une formule PLTL.

Il existe de nombreux algorithmes permettant de construire automatiquement un automate de Büchi à partir d'une propriété PLTL [WVS83, VW94, GPVW95, Cou99, EH00, GO01].

Nous donnons trois exemples d'automates de Büchi non-déterministes permettant de reconnaître des propriétés PLTL. Dans ces exemples, p, q et r représentent des propositions d'état. La figure 4.3 présente un automate permettant de reconnaître toutes les exécutions satisfaisant la formule $\Box\,(p \Rightarrow \bigcirc\, q)$. La figure 4.4 présente un automate reconnaissant la propriété $\neg\,\Box\,(p \Rightarrow \Diamond\, q)$ et la figure 4.5 présente un automate reconnaissant la propriété $\neg\,\Box\,(p \Rightarrow q\,\mathcal{W}\,r)$.

4.2.2 Produit synchrone d'automates

Nous avons présenté au chapitre 1 la sémantique des systèmes d'événements permettant de spécifier un système par un système de transitions étiquetées. Ces STE peuvent être considérés

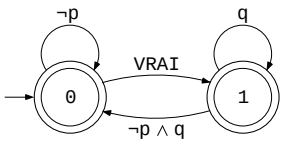

FIG. 4.3 – Un automate de Büchi reconnaissant $\Box\,(p \Rightarrow \bigcirc\, q)$

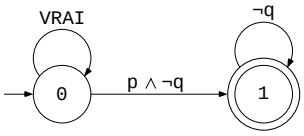

FIG. 4.4 – Un automate de Büchi reconnaissant $\neg\,\Box\,(p \Rightarrow \Diamond\, q)$

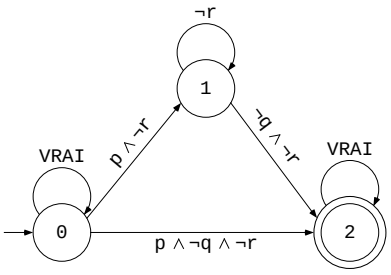

FIG. 4.5 – Un automate de Büchi reconnaissant $\neg\,\Box\,(p \Rightarrow q\,\mathcal{W}\,r)$

comme étant des automates d'un type particulier : ils ne possèdent pas d'état d'acceptation et ils sont munis d'une fonction de valuation $\mathcal{L}$ permettant d'associer un décor (c'est à dire une proposition d'état) à chaque état.

Chaque chemin de ce STE d'ensemble de variables X définit un mot (représentant une exécution) sur l'alphabet constitué de l'ensemble des décors $\mathbb{D}_X$ défini par $\mathcal{L}$. L'ensemble des mots « maximaux » (c'est à dire l'ensemble des mots infinis, ou finis qu'on ne peut pas prolonger), qui correspondent à l'ensemble des chemins d'exécution maximaux du STE, modélise l'ensemble des exécutions possibles du système.

En synchronisant le STE avec l'automate de Büchi modélisant une propriété PLTL à vérifier, on réalise la lecture de toutes les exécutions possibles du système, afin de s'assurer qu'aucune d'entre elles ne constitue une violation de la propriété.

Nous reprenons, dans cette section, la définition du produit synchrone d'un STE et d'un automate de Büchi telle qu'elle est présenté dans [CGP99], en l'adaptant à nos propres notations.

Le produit synchrone d'un STE et d'un automate de Büchi est réalisé en transformant au préalable le STE en un automate qui reconnaît le langage de ce STE.

Un système de transitions étiquetées $ST = \langle S_0, S, A, T, \mathcal{L}\rangle$, dont l'ensemble de variables est X, est transformé en un automate $\mathcal{A} = \langle \imath, S \cup \{\imath\}, \mathbb{D}_X, T_\mathcal{A}, S \cup \{\imath\}\rangle$ avec $s \xrightarrow{\mathcal{L}(s')} s' \in T_\mathcal{A}$ si et seulement si $s \xrightarrow{e} s' \in T$, et avec $\imath \xrightarrow{\mathcal{L}(s)} s \in T_\mathcal{A}$ si et seulement si $s \in S_0$.

L'automate $\mathcal{A}$ est un automate dont l'ensemble d'états est celui de ST auquel on ajoute un état initial noté $\imath$. L'ensemble des transitions est constitué des transitions de T dont les étiquettes sont remplacées par le décor de leur état cible, augmenté des transitions de $\imath$ vers chaque état de S_0, et étiquetées par le décor de l'état de S_0 auquel elles aboutissent.

Le produit synchrone d'un STE transformé en automate et d'un automate de Büchi est alors obtenu comme indiqué dans la définition 18.

Définition 18 (Produit synchrone d'une STE transformé et d'un automate de Büchi).

Le produit synchrone d'un automate $\mathcal{A} = \langle \imath, S \cup \{\imath\}, \mathbb{D}_X, T_\mathcal{A}, S \cup \{\imath\}\rangle$ obtenu par transformation d'un STE, et d'un automate $\mathcal{B} = \langle b_{init}, B, \mathbb{P}_X, T_\mathcal{B}, \mathcal{F}\rangle$ est obtenu en calculant l'intersection de $\mathcal{A}$ et $\mathcal{B}$ de la manière suivante :

$$\mathcal{A} \cap \mathcal{B} = \langle \imath \times b_{init}, (S \cup \{\imath\}) \times B, \mathbb{P}_X, T', (S \cup \{\imath\}) \times \mathcal{F}\rangle$$

avec $(s_1, b_1) \xrightarrow{p} (s_2, b_2) \in T'$ si et seulement si $s_1 \xrightarrow{p'} s_2 \in T_\mathcal{A} \wedge b_1 \xrightarrow{p} b_2 \in T_\mathcal{B} \wedge p' \Rightarrow p$.

Premier exemple

A titre d'exemple, considérons le STE ST, représenté dans la figure 4.6. Ce STE n'a qu'un seul chemin d'exécution, correspondant à la séquence d'états $\sigma = s_0 s_1 s_2 s_3$ telle que

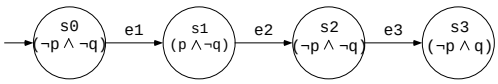

FIG. 4.6 – Un STE ST

$\mathcal{L}(s_0) = \neg\, p \wedge \neg\, q$, $\mathcal{L}(s_1) = p \wedge \neg\, q$, $\mathcal{L}(s_2) = \neg\, p \wedge \neg\, q$ et $\mathcal{L}(s_3) = \neg\, p \wedge q$. Sur la figure 4.6, ces décors d'état sont indiqués entre parenthèses à l'intérieur de chaque état. L'automate $\mathcal{A}$ obtenu en transformant ST comme indiqué précédemment est représenté dans la figure 4.7.

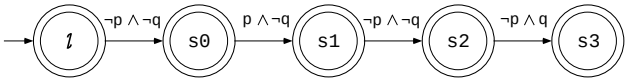

FIG. 4.7 – L'automate $\mathcal{A}$ obtenu par transformation de ST

La figure 4.9 représente l'automate obtenu en faisant le produit synchrone de $\mathcal{A}$ et d'un automate de Büchi $\mathcal{B}$ reconnaissant la formule PLTL $\Box(p \Rightarrow \Diamond q)$, représenté dans la figure 4.8.

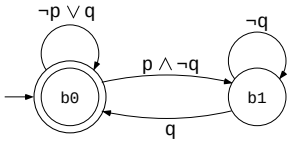

FIG. 4.8 – Un automate de Büchi $\mathcal{B}$ (déterministe) reconnaissant la propriété $\varphi = \Box(p \Rightarrow \Diamond q)$

L'automate produit représenté dans la figure 4.9 ne définit pas un langage vide, et donc le langage de ST n'est pas inclus dans celui de la négation de la propriété $\Box\,(p \Rightarrow \Diamond\, q)$. Comme $\Box\,(p \Rightarrow \Diamond\, q) \equiv \Diamond\,(p \wedge \Box \neg\, q)$, alors on conclut que ST viole la propriété $\Diamond\,(p \wedge \Box \neg\, q)$.

Deuxième exemple

Considérons maintenant le STE ST' représenté dans la figure 4.10. Là encore, ce STE n'a qu'un seul chemin d'exécution, correspondant à la séquence d'états $\sigma' = s'_0 s'_1 s'_2 s'_3$ avec $\mathcal{L}(s'_0) = p \wedge \neg\, q$, $\mathcal{L}(s'_1) = \neg\, p \wedge \neg\, q$, $\mathcal{L}(s'_2) = \neg\, p \wedge \neg\, q$ et $\mathcal{L}(s'_3) = \neg\, p \wedge \neg\, q$.

L'automate $\mathcal{A}'$ obtenu en transformant ST' est représenté dans la figure 4.11.

Le produit synchrone de l'automate $\mathcal{A}'$ obtenu par transformation de ST' et de l'automate de Büchi $\mathcal{B}$ reconnaissant la propriété $\Box\,(p \Rightarrow \Diamond\, q)$ (voir la figure 4.8), est représenté dans la figure 4.12. Comme cet automate définit un langage vide, on en déduit que le langage de ST' est inclus dans celui de la négation de la propriété $\Box\,(p \Rightarrow \Diamond\, q)$. Ce qui signifie que ST' satisfait la propriété $\neg\,\Box\,(p \Rightarrow \Diamond\, q) \equiv \Diamond\, p \wedge \Box \neg\, q$.

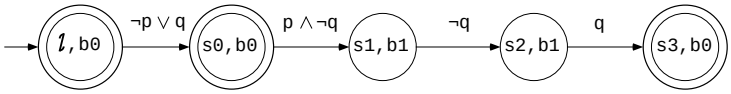

FIG. 4.9 – Le produit synchrone de ST transformé en automate et de $\mathcal{B}$

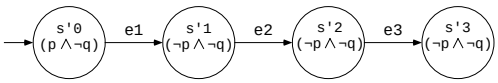

FIG. 4.10 – Un STE ST'

4.2.3 Principe de l'algorithme de model-checking PLTL

L'algorithme de model-checking pour la PLTL réalise le produit synchrone entre le STE du système et un automate de Büchi représentant la négation de la propriété P.

Ensuite, dans le cas d'un système de transitions ST ne comportant que des chemins infinis, le problème du model-checking « $ST \models P$? » est résolu par la détection de composantes fortement connexes sur l'automate produit. Si l'une de ces composantes fortement connexes contient au moins un état d'acceptation, alors le model-checker conclut à l'existence d'une violation de la propriété.

Pour détecter les composantes fortement connexes, il est possible d'utiliser l'algorithme de Tarjan [Tar83], qui est linéaire en temps d'exécution avec la taille du STE ($O(|T|+|S|)$). D'autres algorithmes peuvent être employés selon les besoins. Par exemple, lorsqu'on réalise la construction du STE et le produit synchrone simultanément, il peut être intéressant de ne pas utiliser l'algorithme classique mais une version adaptée. C'est le cas dans le model-checker SPIN [Hol91, Hol97a].

4.3 Limites à la mise en œuvre du model-checking PLTL

4.3.1 La finitude de l'espace d'états

Nous avons vu que le model-checking PLTL effectue l'exploration exhaustive du système de transitions représentant la spécification. Par conséquent, le model-checking ne peut s'appliquer qu'à des systèmes dont le nombre d'états est *fini*.

Il faut cependant préciser que certaines erreurs peuvent tout de même être détectées par model-checking dans des systèmes dont l'espace d'états est infini, si le ou les états conduisant

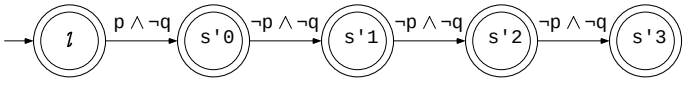

FIG. 4.11 – L'automate $\mathcal{A}'$ obtenu par transformation de ST'

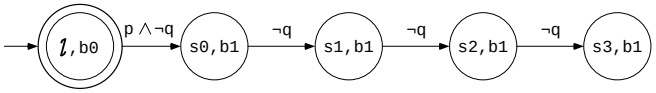

FIG. 4.12 – Le produit synchrone de ST' transformé en automate et de $\mathcal{B}$

à une erreur sont présents dans la mémoire de l'ordinateur au moment de la vérification. Par contre, l'absence d'erreur sur la partie finie contenue en mémoire ne permet en aucun de cas de conclure à une absence globale d'erreur pour un système infini. Le model-checking peut alors être utilisé comme une technique facile à mettre en œuvre permettant *éventuellement* de découvrir certaines erreurs, mais il devient alors une méthode de *validation*, et n'est plus une méthode de *vérification*.

Si l'on souhaite vérifier des systèmes dont l'espace d'états est infini par model-checking, il est nécessaire de transformer le STE infini en un STE fini en ayant par exemple recours à une méthode d'abstraction. Cela passe forcément par une perte d'information par rapport au STE initial, mais peut permettre de vérifier certaines propriétés. Par exemple, en fusionnant des états par suppression des variables qui ne font pas l'objet de la propriété, on préserve les propriétés de sûreté, mais pas les propriétés de vivacité.

4.3.2 La complexité du model-checking PLTL

Dans [SBB+99], Philippe SCHNOEBELEN définit la complexité en temps du model-checking PLTL de la manière suivante :

Pour ST, un STE et φ, une formule PLTL , le model-checking de « est-ce que ST satisfait φ ? » peut être résolu en temps $O((|S| + |T|) \times 2^{|\varphi|})$, où $|S|$ et $|T|$ désignent respectivement le nombre d'états et de transitions de ST, et où $|\varphi|$ désigne le nombre d'opérateurs de la formule φ.

Ainsi, le model-checking PLTL est linéaire selon la taille du STE, mais exponentiel selon la taille de la formule PLTL.

Il faut remarquer que les formules PLTL sont le plus souvent de « petite taille », ce qui fait que la complexité exponentielle de la vérification par model-checking selon la taille de la formule ne pose en général pas de problèmes. En effet, dans la pratique, on ne rencontre pas souvent de propriétés dont l'automate de Büchi a plus de 5 ou 6 états.

4.3.3 Le problème de l'explosion combinatoire

Le principal problème de l'application du model-checking à des systèmes réels est connu sous le nom d'*explosion combinatoire* du nombre d'états. Nous présentons deux causes (non exclusives) à ce problème.

Les systèmes concurrents

Lors de la spécification de systèmes complexes, il est courant de spécifier séparément les différentes unités composant le système. Lorsque ces unités peuvent évoluer de manière parallèle, on parle de systèmes concurrents. On représente alors le comportement global du système, c'est à dire l'ensemble des exécutions possibles, par l'*entrelacement* des exécutions propres à chacune des unités.

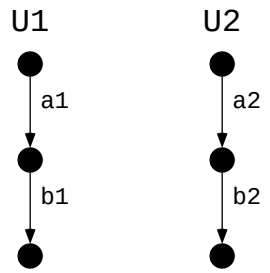

FIG. 4.13 – Deux unités U_1 et U_2 évoluant en parallèle

Considérons par exemple le système représenté dans la figure 4.13 où deux unités U_1 et U_2 évoluent de manière parallèle. Chacune de ces unités effectue deux « actions » : a_1 puis b_1 pour U_1, et a_2 puis b_2 pour U_2.

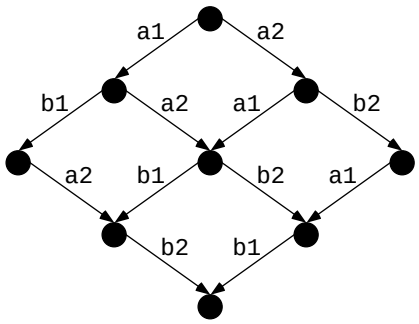

FIG. 4.14 – Entrelacement des actions de U_1 et U_2

Le STE obtenu par entrelacement des actions de U_1 et U_2 est représenté dans la figure 4.14. Il comporte 9 états et 12 transitions (pour 6 états et 4 transitions en faisant la somme respectivement du nombre d'états et du nombre de transitions des modèles U_1 et U_2).

De manière générale, un STE ST construit par entrelacement de n STEs $ST_1, \ldots, ST_n$ aura une taille de l'ordre de $|ST_1| \times \cdots \times |ST_n|$.

La modélisation avec variables d'états

Le problème de l'explosion combinatoire intervient également lorsque les états du système modélisent la conjonction de variables d'états prenant leurs valeurs dans un domaine fini : l'espace d'états croît alors de manière exponentielle avec le nombre de variables.

En effet, considérons un ensemble de $\{x_1, \ldots, x_n\}$ de variables d'états utiles à la description d'un système. On note $|Domaine(x_i)|$ le nombre de valeurs que peut prendre la variable x_i dans son domaine (fini). Le nombre d'états *potentiels* est alors de $\prod_{i=1}^{n} |Domaine(x_i)|$. Si tous les domaines ont le même nombre $|Domaine|$ de valeurs, alors le nombre d'états potentiels est $|Domaine|^n$. Il faut toutefois noter qu'un bon nombre de ces états potentiels ne sont pas des états atteignables. Le nombre d'états potentiels est donc une limite supérieure.

Toujours est-il que puisque l'algorithme de model-checking nécessite la construction explicite du STE à vérifier, le nombre d'états croît souvent de manière telle que le STE devient rapidement impossible à conserver dans la mémoire d'un ordinateur. Des techniques de réduction ayant pour but de maîtriser l'explosion combinatoire ont été conçues pour pouvoir effectuer la vérification de propriétés PLTL par model-checking sur des STE de grande taille.

4.4 Les techniques de réduction pour le model-checking

Nous présentons dans cette section quelques techniques pour le model-checking PLTL permettant de répondre au problème de l'explosion combinatoire du nombre d'états, en représentant l'espace d'états de manière différente. Cette présentation n'est pas exhaustive. Les méthodes présentées ici ont en commun de s'appliquer facilement au model-checking PLTL, contrairement à d'autres techniques de réduction telles que le model-checking symbolique ou les BDD (*Binary Decision Diagrams*) qui sont le plus souvent utilisées dans le cadre du model-checking CTL.

L'application de ces techniques passe souvent par une perte d'information au niveau du STE, et le choix d'une technique particulière est guidé par le type de propriétés que l'on souhaite vérifier. En effet, chaque technique permet de conserver certains types de propriétés mais pas d'autres.

4.4.1 Les techniques d'abstraction

Le terme *abstraction* désigne un ensemble de techniques ayant pour but de pouvoir appliquer le model-checking à des systèmes de grande taille en travaillant non pas sur le STE obtenu directement à partir de la spécification (que l'on appelle le STE concret), mais sur un STE de taille réduite (que l'on appelle le STE abstrait) représentant une abstraction des comportements initiaux.

Toutes ces techniques ont en commun de faire perdre de l'information à la spécification de départ, avec pour conséquence qu'une propriété vérifiable sur le modèle concret ne l'est pas

forcément sur le modèle abstrait. Cependant, l'utilisation de ces techniques est justifiée par le fait que toute l'information n'est pas nécessairement utile à la vérification d'une propriété donnée. Ainsi, le choix d'une technique particulière et surtout la construction d'une abstraction adaptée à la propriété permet en pratique sa vérification, à condition de pouvoir assurer que si une propriété est vérifiée par le système abstrait alors elle l'est également par le système concret.

L'abstraction par restriction

L'abstraction par restriction est une technique simple à mettre en œuvre et qui consiste à supprimer purement et simplement certains états ou certaines transitions du STE. On obtient alors un STE ne modélisant qu'une partie des comportements du système. En d'autres termes, cela revient à interdire certains comportements.

On peut par exemple supprimer un événement, ce qui se traduit par la suppression de toutes les transitions portant son étiquette, et éventuellement de leurs états cibles.

Puisqu'elle supprime certaines exécutions, cette méthode ne permet pas en pratique de vérifier un grand nombre de propriétés, à l'exception des propriétés d'atteignabilité. Par contre, si une erreur est détectée dans le système abstrait, alors on peut garantir que l'erreur est présente aussi dans le système concret. Ainsi, cette technique, par la simplicité de sa mise en œuvre, peut être utilisée pour détecter rapidement des erreurs, mais l'absence d'erreur dans le système abstrait ne saurait garantir que le modèle concret du système satisfait bien les propriétés.

Utilisation de l'interprétation abstraite

L'interprétation abstraite pour définir des abstractions a été introduite dans [CC77]. Elle est applicable à la vérification de propriétés. L'idée générale est de trouver, à partir d'une spécification (concrète) et d'une propriété à vérifier, une spécification abstraite, plus simple, telle que la satisfaction de la propriété par la spécification abstraite implique la satisfaction de cette propriété par la spécification concrète.

On réalise une abstraction grâce à la construction de deux fonctions α et γ telle que la paire (α, γ) forme une connexion de Galois. La fonction α, appelée *fonction d'abstraction*, associe à tout ensemble d'états du STE concret un ensemble d'états du STE abstrait. Inversement, la fonction γ, appelée *fonction de concrétisation*, associe à tout ensemble d'états du STE abstrait un ensemble d'états du STE concret.

En pratique, on construit une interprétation abstraite en définissant une ou plusieurs variables dont le domaine constitue une redéfinition de celui des variables concrètes. Les valeurs de ces variables permettent alors de recouvrir les valeurs associées à plusieurs états du STE concret.

Dans [BC00], Didier BERT et Francis CAVE proposent une méthode de construction d'abstractions intitulée *décomposition disjonctive*, applicable à des systèmes modélisés par des systèmes d'événements B. Son principe est de trouver n prédicats $\phi_1,\ldots,\phi_n$ tels que

$$I \Rightarrow \bigvee_{j=1}^{n} \phi_j$$

où I est l'invariant d'une spécification B. Cela permet la construction d'une connexion de Galois (α, γ).

Nous reprenons l'exemple du deuxième raffinement du robot, dont la spécification est donnée dans la figure 2.4 page 40. Le STE correspondant à cette spécification est représenté par la figure 3.5 de la page 66. Pour construire une abstraction de cette spécification, on définit une variable booléenne *syst_vide* qui vaut `VRAI` si aucun dispositif n'est occupé, et qui vaut `FAUX` sinon : $syst\_vide = (dt_2 = \texttt{vid} \wedge da_2 = \texttt{vid} \wedge de_2 = \texttt{vid})$. Ceci revient à définir deux prédicats ϕ_1 et ϕ_2 tels que

$$\phi_1 \mathrel{\widehat{=}} (dt_2 = \texttt{vid} \wedge da_2 = \texttt{vid} \wedge de_2 = \texttt{vid})$$

$$\phi_2 \mathrel{\widehat{=}} (dt_2 = \texttt{occ} \vee da_2 = \texttt{occ} \vee de_2 = \texttt{occ})$$

On a bien $I \Rightarrow \phi_1 \vee \phi_2 : dt_0 \in \{\texttt{vid}, \texttt{occ}\} \wedge dt_1 = dt_0 \wedge dt_2 = dt_1 \wedge da_1 \in \{\texttt{vid}, \texttt{occ}\} \wedge da_2 = da_1 \wedge de_2 \in \{\texttt{vid}, \texttt{occ}\} \Rightarrow \phi_1 \vee \phi_2$.

Le STE représentant cette abstraction est donné par la figure 4.15.

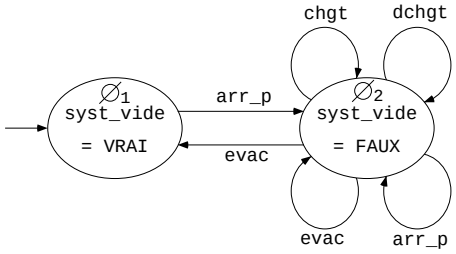

FIG. 4.15 – Une abstraction du STE du deuxième raffinement du robot

Dans cet exemple, la fonction α associe à l'ensemble constitué de l'état initial du STE concret l'ensemble constitué de l'état initial du STE abstrait. Elle associe à tout ensemble non vide du STE concret ne contenant pas l'état initial l'ensemble constitué du deuxième état du STE abstrait.

Inversement, la fonction γ associe à l'ensemble constitué de l'état initial du STE abstrait l'ensemble constitué de l'état initial du STE concret. Elle associe au deuxième état du STE abstrait l'ensemble constitué de tous les états non initiaux du STE concret.

L'abstraction que l'on réalise doit être choisie en fonction des propriétés à vérifier si l'on souhaite garantir qu'une propriété satisfaite par le modèle abstrait l'est aussi par le modèle

concret. Une étude systématique des propriétés préservées par abstraction a été menée dans [LGS+95]. Les auteurs y définissent un sous-langage du μ-calcul arborescent qui est préservé par les abstractions qui sont reliées par une connexion de Galois (α, γ).

L'abstraction que l'on réalise par interprétation abstraite garantit que si une propriété est satisfaite sur le modèle abstrait, alors sa concrétisation est satisfaite sur le modèle concret. Par contre, une erreur détectée sur le modèle abstrait est une erreur potentielle. En effet, réaliser une abstraction par interprétation abstraite a pour effet d'ajouter des chemins d'exécution dans le modèle abstrait. Dans ce cas, une violation d'une propriété détectée sur l'un des chemins abstraits ne signifie pas forcément qu'il existe un chemin concret violant lui aussi la propriété. Lorsque la réponse fournie par la vérification sur le modèle abstrait est qu'une erreur est potentielle, cela signifie que l'analyse a été réalisée sur un modèle trop abstrait pour pouvoir conclure. Par exemple, la détection d'une violation des propriétés P6, P7 ou P8 sur le STE de l'interprétation abstraite donné dans la figure 4.15 conduirait à déclarer qu'une erreur est potentielle dans le modèle concret. Par contre, la réussite (ou l'échec) de la vérification sur le STE de la figure 4.15 de la propriété $\Box\,(syst\_vide \Rightarrow \Diamond \neg\, syst\_vide)$, qui abstrait la propriété $\Box\,(da_2 = \texttt{vid} \wedge dt_2 = \texttt{vid} \wedge de_2 = \texttt{vid} \Rightarrow \Diamond\,(da_2 = \texttt{occ} \vee dt_2 = \texttt{occ} \vee de_2 = \texttt{occ}))$, permettrait d'affirmer que cette propriété est vérifiée (ou non) sur le modèle concret.

4.4.2 L'utilisation de contraintes ensemblistes

Le model-checking PLTL avec résolution de contraintes, tel qu'il a été conçu dans [Py00, Par00], est mené sur des modèles dont les états sont décorés par des contraintes logico-ensemblistes. Cette intégration des contraintes permet de représenter des ensembles d'états d'un STE, non pas par énumération des valeurs possibles de chacune des variables, mais par des contraintes sur le domaine d'appartenance des variables. Imaginons par exemple une variable x entière susceptible de prendre les valeurs 1, 2 et 3. Une énumération classique donnera naissance à trois états différents, chaque état instanciant une valeur particulière de x (on parle alors d'états *valués*). Ces trois états pourraient être représentés par un seul état *contraint*, c'est à dire par un état décoré par la contrainte $x \in \{1, 2, 3\}$. L'utilisation de contraintes ensemblistes permet de réaliser une forme d'interprétation abstraite.

L'idée principale de cette technique est de représenter par un seul état contraint un ensemble d'états valués ayant les mêmes valeurs de vérité vis à vis des propriétés d'états qui composent la propriété PLTL que l'on cherche à vérifier. Cette méthode permet donc de réduire l'explosion combinatoire en agrégeant plusieurs états en un seul. Elle ouvre également des perspectives quant à la vérification par model-checking des systèmes paramétrés ou infinis.

Les systèmes paramétrés sont des systèmes dont le modèle varie en fonction d'un certain nombre de paramètres (la taille d'une file d'attente par exemple, ou le nombre de processus, ...). Pour pouvoir appliquer le model-checking à un tel système, il faudrait instancier (quand c'est possible) toutes les valeurs possibles des paramètres, ce qui amènerait à créer autant de modèles qu'il y a de valeurs. L'exemple du BRP présenté au chapitre 2 est typiquement un système paramétré, dépendant d'au moins deux paramètres : le nombre de paquets par fichier et le nombre maximum de retransmissions autorisées d'un même paquet. Pour pouvoir

appliquer le model-checking à un tel système, nous avons fixé arbitrairement, au chapitre 3, la valeur de ces deux paramètres à 2. Il est évident que le model-checking sur un tel modèle ne peut prouver la satisfaction des propriétés que pour deux paquets et deux retransmissions maximum.

On distingue les STEs contraints qui sont des STEs dont le décor est un ensemble de contraintes, des STEs classiques que nous avons introduits au chapitre 3, dont le décor est une valuation définie par une proposition de $\mathbb{D}_X$. Il a été prouvé dans [Par00] que l'algorithme de model-checking par résolution de contraintes est consistant et complet dans le cas des systèmes à états finis. C'est à dire qu'un cycle est détecté dans l'automate du produit synchrone du STE contraint et de l'automate de la négation de la propriété, si et seulement si c'est un cycle qui existe dans l'automate du produit synchrone du STE valué et de l'automate de la négation de la propriété. Cette méthode permet de préserver toute les propriétés de la PLTL sur des modèles finis. Par contre, la méthode n'est plus décidable dans le cas de systèmes infinis.

4.4.3 Les techniques d'ordre partiel

Les techniques d'ordre partiel [Ove81, KP88, WG93] sont applicables à la modélisation de systèmes concurrents, décrits par un ensemble de processus évoluant de manière parallèle.

Ces techniques visent à réduire l'espace d'états qu'il est nécessaire d'explorer pour vérifier des propriétés sur le modèle du système en s'attaquant à l'une des sources potentielles de l'explosion combinatoire : la représentation du comportement global du système, c'est à dire les différentes exécutions possibles, par entrelacement des actions propres à chacun des processus. L'entrelacement est obtenu en faisant le produit des STEs décrivant les processus concurrents du système. Les états du STE ainsi obtenu sont appelés les états *globaux* du système.

Dans ce qui suit, l'*action* e_i désigne le fait de passer d'un état s à un état s' grâce à une transition $s \xrightarrow{e_i} s'$.

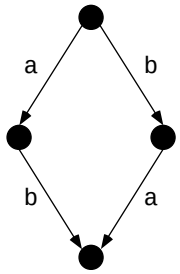

FIG. 4.16 – Entrelacement de deux actions a et b indépendantes

L'idée principale des techniques d'ordre partiel repose sur le fait que le STE modélisant le comportement global du système décrit souvent de manière redondante plusieurs exécutions équivalentes. Imaginons en effet deux actions a et b appartenant à deux processus différents mais qui n'interagissent pas. Alors le fait d'exécuter d'abord a et ensuite b ou d'abord b et ensuite a aboutit au même résultat, mais est représenté par deux chemins différents dans le

STE modélisant le comportement global du système (voir la figure 4.16, montrant le « diamant » caractéristique). Pour vérifier une propriété ne reposant pas sur l'ordre dans lequel s'enchaînent les actions, il suffit de parcourir l'un des chemins d'exécution.

En d'autres termes, nous dirons que les techniques d'ordre partiel permettent d'éviter la représentation des entrelacements superflus.

Ces techniques reposent sur le fait qu'il peut exister des actions indépendantes l'une de l'autre dans le parcours du STE modélisant le système global, et qu'alors l'ordre dans lequel ces actions sont exécutées n'a pas d'importance quant à l'état global atteint.

La définition de l'indépendance de deux actions repose sur deux conditions, la première stipulant que deux actions indépendantes ne peuvent se rendre mutuellement exécutables ou inexécutables, et la seconde indiquant que deux actions e_1 et e_2 indépendantes sont commutatives, c'est à dire qu'exécuter d'abord e_1 puis e_2 mène au même état global que d'exécuter d'abord e_2 et ensuite e_1.

En pratique, un calcul des actions indépendantes basé directement sur l'application de la définition de cette relation d'indépendance serait trop coûteux pour être exploitable, aussi on utilise des conditions suffisantes pour déterminer les ensembles d'actions indépendantes. Par exemple, deux actions sont indépendantes si les ensembles de variables intervenant dans les transitions qui leur correspondent sont disjoints.

Suite aux travaux de Antoni MAZURKIEWICZ [Maz86], on utilise la notion d'actions indépendantes pour définir une relation d'équivalence entre séquences d'actions : deux séquences d'actions sont équivalentes si elles peuvent êtres obtenues l'une de l'autre par permutation d'actions adjacentes. Deux séquences d'actions qui appartiennent à la même classe d'équivalence mènent au même état, et on n'explore alors qu'une seule séquence de la classe pour vérifier une propriété.

Les algorithmes mettant en œuvre des techniques d'ordre partiel reposent tous sur le même principe : ils effectuent un parcours de graphe classique, à cette différence près qu'à chaque état rencontré, un sous-ensemble de toutes les actions exécutables dans cet état est calculé, et que seules les actions appartenant à ce sous-ensemble sont explorées.

Principalement, deux techniques ont été proposées pour calculer de tels sous-ensembles :

- la technique des « ensembles persistants » (*persistent sets* en anglais), ayant donné lieu à toute une famille d'algorithmes [Val88, Val91, GW91],
- la technique des « ensembles en sommeil » (*sleep sets* en anglais) [God90, GW91].

Ces deux techniques sont implantées dans le model-checker SPIN afin de réduire l'espace d'états qu'il est nécessaire d'explorer pour vérifier des propriétés PLTL. Lors du premier *SPIN workshop*, Gerard J. HOLZMANN et Doron PELED ont exposé que les techniques d'ordre partiel permettent de conserver toutes les propriétés PLTL qui ne font pas intervenir l'opérateur temporel *suivant* $\bigcirc$ [HP95] (voir aussi [HP94]).

Les *persistent sets*

On cherche à ne pas explorer les entrelacements superflus. Pour cela, lors de l'exploration du STE du comportement global, on construit à chaque état s rencontré un ensemble *persistant* d'actions issues de s, et on n'exécute que les actions de cet ensemble.

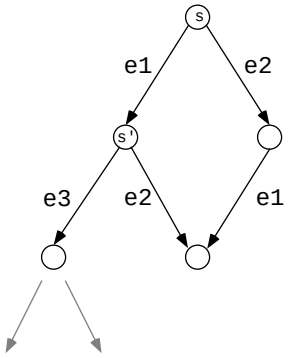

FIG. 4.17 – L'action e_3 n'est pas indépendante de e_2

Nous avons vu que si deux actions e_1 et e_2 sont indépendantes, alors les séquences d'actions e_1e_2 et e_2e_1 sont équivalentes. Cependant, choisir de n'explorer qu'une seule action parmi celles, indépendantes, qui sont issues d'un état n'est pas suffisant, comme le montre la figure 4.17, où e_1 et e_2 sont des actions indépendantes. Si l'on choisit de n'exécuter que l'action e_2 à partir de s, alors on n'explore pas la partie du STE issue de s' par exécution de e_3.

La construction d'ensembles persistants permet d'éviter l'oubli de telles exécutions. Intuitivement, un ensemble T_p d'actions est persistant dans un état s si et seulement si tout chemin d'exécution partant de s et ne faisant intervenir que des actions n'appartenant pas à T_p n'est composé que d'actions indépendantes de celles de T_p. En d'autres termes, les séquences d'actions n'appartenant pas à T_P issues de s ont une séquence d'actions appartenant à T_p issue de s équivalente.

quoique l'on fasse depuis s en restant en dehors de T_p n'a pas d'influence sur les comportements observés grâce à T_p.

Dans l'exemple de la figure 4.17, on peut construire deux ensembles persistants en s : $\{e_1\}$ et $\{e_1, e_2\}$. L'ensemble $\{e_2\}$ n'est pas persistant en s.

Remarque. L'ensemble de toutes les actions exécutables depuis un état s est un ensemble persistant en s puisque, de manière triviale, aucun état n'est accessible depuis s sans exécuter d'actions de cet ensemble.

En pratique, on utilise des heuristiques visant à construire depuis chaque état l'ensemble persistant le plus petit possible en espérant que cela minimisera le nombre de transitions à explorer globalement. Les différents algorithmes proposés permettant de calculer des ensembles persistants [Ove81, Val91, GW91] diffèrent par le compromis qu'ils essaient de trouver entre la taille de l'ensemble persistant et la complexité du calcul permettant de les obtenir. On trouve dans [HGP92] une comparaison de ces divers algorithmes.

Les *sleep sets*

La technique des *sleep sets* (que l'on peut traduire par « ensembles en sommeil ») [GW91] ne cherche pas à réduire le nombre d'états à explorer mais uniquement le nombre de transitions. Cette technique est en général utilisée conjointement à celle des ensembles persistants, notamment afin d'éviter d'exécuter plusieurs fois une action issue de l'entrelacement d'actions indépendantes que cette dernière technique n'a pas suffi à éliminer.

Si l'idée des ensembles persistants était d'anticiper les entrelacements superflus, celle des *sleep sets* est plutôt d'éviter leur redondance en conservant une trace des actions déjà exécutées. Pour cela, lors du parcours du STE modélisant le comportement global du système, on construit à chaque état rencontré un *sleep set* contenant des actions exécutables depuis cet état mais qui ne seront pas exécutées. La construction d'un *sleep set* est effectuée de la manière suivante :

- le *sleep set* associé à l'état initial est l'ensemble vide,
- le *sleep set* d'un état s' atteint depuis un état s par une action e est constitué du *sleep set* de s, auquel sont ajoutées les actions issues de s déjà exécutées lors du parcours, et duquel sont retirées les actions qui ne sont pas indépendantes de e.

Les actions appartenant au *sleep set* d'un état ne sont pas exécutées lors du parcours du STE.

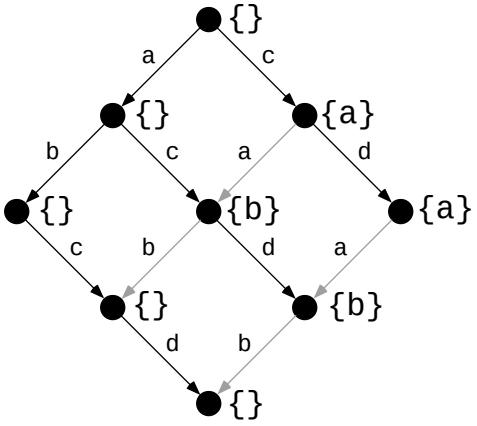

FIG. 4.18 – Exemples de *sleep sets*

La figure 4.18 montre les *sleep sets* associés à chaque état d'un STE obtenu en entrelaçant les actions a et b d'un processus et les actions c et d d'un autre processus, les actions de ces processus étant indépendantes deux à deux. Les *sleep sets* sont représentés entre accolades à droite des états, lorsque l'on effectue un parcours du STE en profondeur d'abord. Les actions non exécutées lors de ce parcours sont représentées en clair.

4.5 Conclusion

Nous avons vu dans ce chapitre comment le model-checking permettait de vérifier automatiquement (c'est à dire sans interaction avec l'utilisateur) des propriétés PLTL sur des systèmes à états finis modélisés par des STE.

Le principal obstacle à l'utilisation de cette technique est que les systèmes complexes ont souvent un trop grand nombre d'états, et dépassent la capacité des outils. On doit alors avoir recours à des techniques permettant de réduire ce phénomène d'explosion combinatoire afin de rendre possible la vérification par model-checking de systèmes de grande taille. Nous avons présenté quelques unes de ces techniques.

Dans la deuxième partie de ce document, nous présentons notre contribution à ce problème en proposant une technique permettant de vérifier sur des STE obtenus à partir de spécifications B des propriétés PLTL de manière modulaire. Cette méthode consiste à découper l'espace d'états, à partir de la relation de raffinement B entre deux spécifications, en un ensemble de modules indépendants et à vérifier les propriétés sur chacun des modules. Elle est applicable à une classe de propriétés PLTL, caractérisée par une condition suffisante que doit vérifier l'automate de Büchi qui code leur négation. Les techniques de réduction que nous avons présentées restent applicables à chacun de ces modules.

Deuxième partie

Contributions

Chapitre 5

La vérification modulaire : une réponse au problème de l'explosion combinatoire

Dans ce chapitre, nous présentons une méthode de vérification de propriétés dynamiques permettant de faire face au problème de l'explosion combinatoire. Commençons par re-situer le contexte d'application de cette méthode. Nous souhaitons vérifier formellement que des systèmes spécifiés en B satisfont des propriétés dynamiques exprimées en PLTL. En construisant un STE à partir de la spécification B, on peut utiliser la technique du model-checking pour assurer la vérification des propriétés. Le principal obstacle à l'utilisation de cette technique est l'explosion combinatoire du nombre d'états modélisant le système. Il s'agit donc de réduire la taille (c'est à dire le nombre d'états) du STE à vérifier par model-checking lorsque celui-ci devient trop gros pour tenir intégralement dans la mémoire d'un ordinateur.

5.1 Principe de la vérification modulaire

5.1.1 Idée

La technique de vérification modulaire que nous proposons repose sur une idée simple : si un STE est trop gros pour tenir intégralement dans la mémoire d'un ordinateur, on va le découper en un ensemble de « sous-STE[1] » de plus petite taille afin que chacun d'eux puisse être stocké en mémoire, et on va mener la vérification des propriétés sur chacun de ces sous-STE pris séparément. Nous appelons ces sous-STE des *modules*. Le STE initial est quant à lui appelé le STE global.

Le résultat que nous attendons de l'application de cette méthode est de pouvoir conclure, quand une propriété est vraie sur chacun des modules, qu'elle est vraie sur le STE initial.

[1] La notion de sous-STE est identique à celle de sous-graphe.

5.1.2 Le découpage en modules

Afin de perdre le moins d'information possible lors du découpage en modules, il semble raisonnable de considérer que chaque état du STE global doit appartenir à un module au moins. De cette manière, on s'assure de pouvoir vérifier de manière modulaire les propriétés invariantes du système puisque chaque état du STE global sera évalué au moins une fois au cours d'une analyse successive des modules.

Si chaque transition du STE global est présente dans au moins un des modules, alors on s'assure de pouvoir vérifier aussi de manière modulaire les invariants dynamiques. En effet, ceux-ci peuvent être vérifiés par model-checking en évaluant chaque état et son successeur. Si chaque transition du STE global est présente dans un module au moins, alors toute succession de deux états sera évaluée.

Nous proposons donc que le découpage modulaire constitue une partition des transitions du STE global, et un recouvrement des états de ce dernier. Les états peuvent appartenir à plusieurs modules puisqu'ils peuvent être à la fois l'état cible d'une transition d'un module, et l'état source d'une transition d'un autre module. Dans la figure 5.1, nous avons représenté un état s appartenant à deux modules M_1 et M_2. Chaque module est représenté entouré d'un ovale épais.

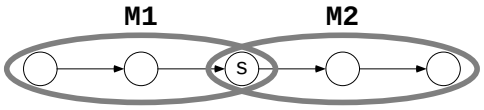

FIG. 5.1 – Un état appartenant à deux modules

Dans la définition 19, nous définissons un module issu d'un STE.

Soit $ST = \langle S_0, S, A, T, \mathcal{L} \rangle$ un système de transitions étiquetées. Considérons une partition de l'ensemble S des états. Chaque élément de cette partition peut être considéré comme une classe d'équivalence $\mathcal{C}_i$, et l'intersection des $\mathcal{C}_i$ deux à deux est vide. Nous appelons *transition de sortie* par rapport à $\mathcal{C}_i$ une transition $s \xrightarrow{e} s' \in T$ telle que $s \in \mathcal{C}_i$ et $s' \notin \mathcal{C}_i$. Nous définissons un module comme étant un système de transitions dont :

- l'ensemble d'états est $\mathcal{C}_i$ augmenté des états cibles des transitions de sortie,
- l'ensemble des états initiaux est l'ensemble des états de $\mathcal{C}_i$ qui n'ont pas de prédécesseur dans $\mathcal{C}_i$,
- l'ensemble des transitions est l'ensemble des transitions de ST dont l'état source appartient à $\mathcal{C}_i$,
- l'ensemble des étiquettes est l'ensemble des étiquettes de ST étiquetant des transitions du module,
- la fonction de valuation est celle de ST restreinte aux états du module.

Définition 19 (Module).

Soit $ST = \langle S_0, S, A, T, \mathcal{L} \rangle$ un système de transitions étiquetées. Soit $\mathcal{C}$ une classe d'équi-

valence définie par une partition de S. Un module issu de $\mathcal{C}$ est un système de transitions étiquetées $M = \langle S_{M_0}, S_M, A_M, T_M, \mathcal{L}_M \rangle$ défini de la manière suivante :

- $S_M = \mathcal{C} \cup \{s' \in S \ / \ \exists s \xrightarrow{e} s' \cdot s \xrightarrow{e} s' \in T \wedge s \in \mathcal{C} \wedge s' \notin \mathcal{C}\}$
- $S_{M_0} = \{s' \in \mathcal{C} \ / \ s' \in S_0 \vee \forall s \cdot s \xrightarrow{e} s' \in T \Rightarrow s \notin \mathcal{C}\}$
- $T_M = \{s \xrightarrow{e} s' \in T \ / \ s \in \mathcal{C}\}$
- $A_M = \{e \in A \ / \ s \xrightarrow{e} s' \in T_M\}$
- $\mathcal{L}_M$ *est la restriction de* $\mathcal{L}$ *sur* S_M

Nous appelons *états internes* d'un module les états qui appartiennent à la classe d'équivalence, et nous appelons *états de sortie* d'un module les états qui n'appartiennent pas à la classe d'équivalence mais qui sont les états cibles des transitions de sortie.

Avec la définition 19, l'ensemble des modules d'un STE définit bien une partition de l'ensemble des transitions du STE, et un recouvrement de l'ensemble des états du STE.

5.2 Définition de la vérifiabilité modulaire

Nous rappelons que nous proposons de vérifier les propriétés d'un système en découpant le STE modélisant ce système en un ensemble de modules. La *vérification modulaire* d'une propriété consiste alors à vérifier cette propriété sur chaque module de manière séparée et indépendante, et à conclure, lorsque la propriété est vraie sur chacun des modules, qu'elle est vraie globalement (c'est à dire sur le STE global).

Nous avons justifié intuitivement le fait que les propriétés invariantes et les invariants dynamiques pouvaient être vérifiés de cette manière. Qu'en est-il des autres propriétés dynamiques ?

Il faut tout d'abord noter que le fait qu'une propriété soit vraie sur tous les modules ne suffit pas à conclure qu'elle est globalement vraie. En effet, les propriétés dynamiques modélisent des comportements observables sur des successions de plusieurs états, c'est à dire sur des chemins d'exécution. Le découpage en modules a pour effet « d'interrompre » les chemins d'exécution et lorsqu'une propriété est vraie sur un chemin interrompu, rien ne permet *a priori* de garantir que la propriété serait restée vraie si l'on avait poursuivi l'exploration du chemin. Nous verrons plus loin dans cette section l'exemple de la propriété $\Box\,(p \Rightarrow \Box\, q)$, qui peut être vraie sur tous les modules d'un STE mais fausse globalement.

Cependant, un certain nombre de propriétés PLTL ont pour particularité d'être vraies globalement lorsqu'elles sont vraies sur tous les modules d'un STE. C'est notamment le cas des trois schémas de contrainte dynamique introduits par Jean-Raymond Abrial et Louis Mussat dans [AM98], et dont nous avons vu qu'ils pouvaient être modélisés par des formules PLTL.

Nous disons d'une propriété PLTL P qu'elle est *vérifiable modulairement* si et seulement si on a :

$$P \text{ est vraie sur chaque module} \Rightarrow P \text{ est vraie globalement.}$$

Ainsi, il faut s'assurer qu'une propriété est vérifiable modulairement avant de la vérifier de manière modulaire. Nous donnerons dans la section 5.4 une condition suffisante permettant d'affirmer qu'une propriété est vérifiable modulairement.

Afin de vérifier qu'une propriété est vérifiable modulairement, on suppose qu'elle est fausse globalement. En effet, on remarque que

$$P \text{ est vraie sur chaque module} \Rightarrow P \text{ est vraie globalement}$$

est équivalent à

$$P \text{ est fausse globalement} \Rightarrow P \text{ est fausse sur au moins un module..}$$

Si l'on réussit à prouver que lorsqu'une propriété est globalement fausse alors elle est également fausse sur au moins un module, alors on sait que la propriété est vérifiable modulairement.

Dans la définition 20, nous formalisons la notion de propriété PLTL vérifiable modulairement. Nous rappelons que la notation $\Sigma(ST)$ désigne l'ensemble des chemins d'exécution maximaux d'un système de transitions ST.

Définition 20 (Propriété vérifiable modulairement).

Soit P une propriété PLTL. Soit ST, un système de transitions étiquetées découpé en un ensemble $\mathcal{M}$ de modules. La propriété P est vérifiable modulairement si et seulement si on a :

$$(\forall M \cdot M \in \mathcal{M} \wedge M \models P) \Rightarrow ST \models P$$

ce qui est équivalent à :

$$\neg\,(ST \models P) \Rightarrow \exists M \cdot \exists \sigma \cdot M \in \mathcal{M} \wedge \sigma \in \Sigma(M) \wedge \sigma \models \neg\, P.$$

5.3 Deux exemples de propriétés face à la vérifiabilité modulaire

A titre d'exemple, nous allons observer le comportement de deux schémas de propriété lors d'un découpage modulaire. Ces deux schémas sont $\Box\,(p \Rightarrow \Diamond\, q)$ et $\Box\,(p \Rightarrow \Box\, q)$, où p et q sont des propositions.

Le premier de ces schémas est vérifiable modulairement, comme il sera établi dans la section 5.4, tandis que le second ne l'est pas. Dans chacun des deux cas, on considère que la propriété est fausse globalement, c'est à dire qu'il existe un chemin d'exécution d'un système de transitions étiquetées global ne satisfaisant pas la propriété, et on exhibe un tel chemin d'exécution. Lors du découpage modulaire, le chemin peut être interrompu, et on regarde si un tel chemin interrompu continue à violer la propriété.

5.3.1 Un exemple de propriété vérifiable modulairement : $\Box\,(p \Rightarrow \Diamond\, q)$

Supposons une propriété PLTL $P \mathrel{\widehat{=}} \Box\,(p \Rightarrow \Diamond\, q)$, globalement fausse. Cela signifie qu'il existe un chemin d'exécution d'un système de transitions étiquetées satisfaisant la négation de la propriété. On remarque que $\neg\, P$ est sémantiquement équivalent à $\Diamond\,(p \wedge \Box \neg\, q)$. Ainsi, il existe un chemin d'exécution contenant un état vérifiant $p \wedge \neg\, q$ et tel que tous les successeurs de cet état vérifient $\neg\, q$.

La figure 5.2 présente une tel chemin, ainsi qu'un découpage en deux modules qui a pour effet de couper en deux ce chemin.

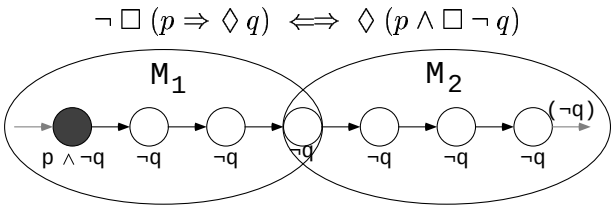

FIG. 5.2 – Un chemin satisfaisant $\neg\, \Box\,(p \Rightarrow \Diamond\, q)$, découpé en deux

Si l'on suppose qu'aucun état du module M_2 ne vérifie p, alors P est trivialement vraie sur ce module. Par contre, dans le module M_1, il existe un état satisfaisant $p \wedge \neg\, q$ et dont tous les successeurs satisfont $\neg\, q$. En conséquence, P est fausse sur le module M_1, ce qui est le résultat qu'on attendait d'une propriété vérifiable modulairement : puisqu'elle est fausse globalement, il doit exister un module dans lequel la propriété est également fausse, quel que soit le découpage modulaire.

5.3.2 Un exemple de propriété non vérifiable modulairement : $\Box\,(p \Rightarrow \Box\, q)$

Supposons une propriété PLTL $P \mathrel{\widehat{=}} \Box\,(p \Rightarrow \Box\, q)$, globalement fausse. Cela signifie qu'il existe un chemin d'exécution d'un système de transitions étiquetées satisfaisant la négation de la propriété. On remarque que $\neg\, P$ est sémantiquement équivalent à $\Diamond\,(p \wedge \Diamond \neg\, q)$. Ainsi, il existe un chemin d'exécution contenant un état vérifiant p et tel que l'un des successeurs de cet état vérifie $\neg\, q$.

Nous avons représenté un exemple d'un tel chemin dans la figure 5.3, ainsi qu'un découpage en deux modules qui a pour effet de couper en deux ce chemin.

Dans le module M_2, il se peut qu'aucun état ne vérifie p et P est alors trivialement vraie sur ce module. Dans le module M_1, l'état vérifiant p est maintenant suivi sur le chemin d'exécution coupé d'états qui vérifient tous q, de telle sorte que ce chemin (coupé) ne constitue plus une violation de la propriété $\Box\,(p \Rightarrow \Box\, q)$. Ainsi, si l'on suppose que les autres chemins du module M_1 satisfont aussi la propriété P, alors on a que P est vraie sur tous les modules (M_1 et M_2), alors qu'elle est globalement fausse. Ce contre-exemple suffit à prouver que $\Box\,(p \Rightarrow \Box\, q)$ n'est pas une propriété vérifiable modulairement.

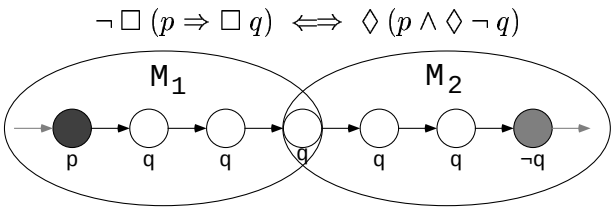

FIG. 5.3 – Un chemin satisfaisant $\neg \Box (p \Rightarrow \Box q)$, découpé en deux

5.3.3 Différence entre les deux types de propriétés

Le fait qu'une propriété de la forme $\Box (p \Rightarrow \Diamond q)$ soit vérifiable modulairement peut s'expliquer de la manière suivante : s'il existe un chemin global qui viole la propriété, alors ce chemin contient un état vérifiant p suivi d'états vérifiant tous $\neg q$. Lorsque l'on coupe ce chemin, l'état vérifiant p reste quand même suivi d'états vérifiant tous $\neg q$, de telle sorte que le chemin coupé viole lui aussi la propriété $\Box (p \Rightarrow \Diamond q)$.

Le cas d'une propriété de la forme $\Box (p \Rightarrow \Box q)$ est différent. En effet, pour qu'un chemin global viole cette propriété, il suffit que parmi les successeurs d'un état vérifiant p se trouve *un seul* état vérifiant $\neg q$. Si, après l'état vérifiant p, la coupure du chemin se produit alors qu'on n'a pas encore atteint un état vérifiant $\neg q$, alors le chemin coupé ne viole plus la propriété, comme illustré par la figure 5.3.

5.4 Preuve de la vérifiabilité modulaire pour une classe de propriétés PLTL

Dans cette section, nous prouvons qu'un certain nombre de propriétés PLTL sont des propriétés vérifiables modulairement. Nous donnons pour cela une condition, et nous montrons que cette condition est suffisante pour affirmer qu'une propriété est vérifiable modulairement [MMJ00, JMM01].

Pour donner cette condition suffisante, nous nous intéressons non pas à la propriété PLTL elle-même, mais à un automate de Büchi qui permet de coder sa négation (voir la section 4.2.1). Si un tel automate a une forme particulière (d'ordre syntaxique), décrite par la définition 21, alors la propriété dont il code la négation est une propriété vérifiable modulairement.

5.4.1 L'idée d'une classe d'automates de Büchi

Une manière de prouver qu'une propriété est vérifiable modulairement est de s'assurer que si cette propriété est fausse globalement, alors elle est nécessairement fausse sur l'un au moins des modules, quelle que soit la décomposition en modules.

La condition suffisante que nous donnons pour la vérifiabilité modulaire d'une propriété repose sur l'idée suivante : si, sur un chemin d'exécution, la violation d'une propriété devient

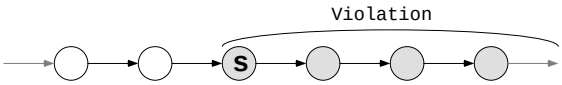

FIG. 5.4 – La violation permanente d'une propriété à partir d'un état s

permanente à partir d'un état s (voir la figure 5.4) alors ce chemin, s'il est coupé après l'état s, reste un chemin qui viole la propriété. C'est le cas par exemple lors de la violation d'une propriété de la forme $\Box\,(p \Rightarrow \Diamond\, q)$ (voir la figure 5.2 page 107).

La même chose peut être dite lorsqu'une seule transition suffit à provoquer une violation irrémédiable (et donc permanente) de la propriété. Par exemple, un chemin comportant un état s vérifiant une proposition p suivi d'un état s' vérifiant une proposition $\neg\, q$ viole la propriété $\Box\,(p \Rightarrow \bigcirc\, q)$ (voir la figure 5.5), même s'il est coupé après la transition $s \rightarrow s'$.

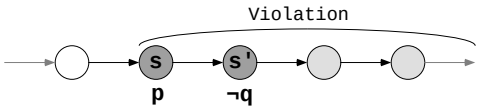

FIG. 5.5 – Une violation de la propriété $\Box\,(p \Rightarrow \bigcirc\, q)$

Comme on sait que tout état, mais aussi toute transition d'un STE global appartient nécessairement à l'un de ses modules, alors la violation de la propriété sera détectée au niveau de ce module.

Nous cherchons donc à reconnaître des propriétés qui, lorsqu'elles sont violées par un chemin d'exécution, le sont de manière permanente à partir de la succession de deux états au plus de ce chemin.

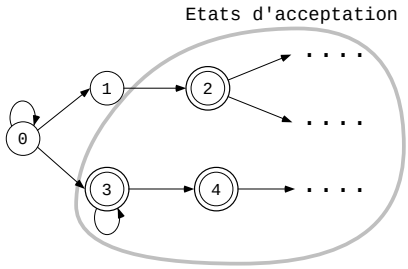

FIG. 5.6 – Un automate de Büchi reconnaissant la violation d'une propriété vérifiable modulairement

Un automate de Büchi capable de reconnaître une violation permanente à partir de la lecture de deux états consécutifs d'un chemin d'exécution est un automate de Büchi qui, après avoir commencé à reconnaître une violation, c'est à dire après avoir quitté son état initial, rejoint un état d'acceptation en passant par au plus un état intermédiaire, et qui ne

peut plus alors évoluer que sur des états d'acceptation. La figure 5.6 schématise un exemple d'un tel automate de Büchi.

Remarque. Le cas où l'automate rejoint directement les états d'acceptation après avoir quitté l'état initial correspond au cas où la violation de la propriété devient permanente à partir d'un seul état, tandis que le cas où l'automate passe auparavant par un état intermédiaire qui n'est pas un état d'acceptation correspond au cas où c'est la succession de deux états qui provoque la violation permanente.

Nous appelons $\mathcal{C}_{mod}$ la classe des automates de Büchi ayant cette propriété et cette classe est définie par la définition 21.

Définition 21 (La classe $\mathcal{C}_{mod}$ d'automates de Büchi).

Soit $\mathcal{B} = \langle b_{init}, B, P_{\mathcal{B}}, T_{\mathcal{B}}, \mathcal{F} \rangle$ un automate de Büchi. $\mathcal{B} \in \mathcal{C}_{mod}$ si et seulement si :

$$\forall \tau \cdot \tau \in \Sigma(\mathcal{B}) \wedge \tau = b_0 b_1 \ldots \Rightarrow \exists k \cdot k > 0 \wedge \forall i \cdot ((i < k \Rightarrow b_i = b_{init}) \wedge (i > k \Rightarrow b_i \in \mathcal{F})).$$

Il nous reste maintenant à établir la preuve que les automates de $\mathcal{C}_{mod}$ sont des automates qui reconnaissent des négations de propriétés vérifiables modulairement.

5.4.2 Preuve que les automates de $\mathcal{C}_{mod}$ reconnaissent la négation de propriétés vérifiables modulairement

La démonstration s'organise de la manière suivante : après avoir établi deux résultats préliminaires (les lemmes 2 et 3), on suppose qu'une propriété P de la PLTL est violée par un chemin d'exécution maximal d'un STE global, et on suppose qu'un automate $\mathcal{B} \in \mathcal{C}_{mod}$ accepte les exécutions qui violent P. On établit alors qu'une fois le STE découpé en modules, il existe un chemin maximal de l'un des modules qui est tel qu'il est lui aussi accepté par $\mathcal{B}$. Comme $\mathcal{B}$ accepte toutes les exécutions qui violent la propriété P, on en déduit que P est fausse sur le module auquel appartient ce chemin. C'est ce qu'établit le théorème 4 qui dit que si une propriété dont on peut coder la négation par un automate $\mathcal{B} \in \mathcal{C}_{mod}$ est fausse globalement, alors elle est fausse aussi sur un module.

Lemme 2. *Soit ST un système de transitions découpé en modules. Toute transition de ST appartient à un module.*

Démonstration. Ce résultat est trivial car, par construction, l'ensemble des ensembles de transitions de chaque module constitue une partition de l'ensemble des transitions de ST (voir la définition 19 d'un module d'un STE). □

Lemme 3. *Un chemin d'exécution d'un module qui passe par un état de sortie de ce module est un chemin d'exécution maximal de ce module, qui se termine par cet état de sortie.*

Démonstration. Soit $M = \langle S_{M_0}, S_M, A_M, T_M, \mathcal{L}_M \rangle$ un module d'un système de transitions $ST = \langle S_0, S, A, T, \mathcal{L} \rangle$. Soit $\sigma = s_0 s_1 \cdots s_c \cdots$ un chemin d'exécution de M tel que s_c est un état de sortie de M. Soit $\mathcal{C}$ la classe d'équivalence dont est issu M : $s_c \notin \mathcal{C}$ car s_c est un état de sortie de M.

Puisque $T_M = \{s \xrightarrow{e} s' \in T \ / \ s \in \mathcal{C}\}$ alors on a :

$$\forall s \cdot s \notin \mathcal{C} \Rightarrow \forall s' \cdot \forall e \cdot s \xrightarrow{e} s' \notin T_M.$$

Donc, en particulier, puisque $s_c \notin \mathcal{C}$ alors il n'existe pas de transition de T_M ayant s_c pour état source. Exprimé autrement, s_c n'a pas de successeur dans M et donc $\sigma = s_0 s_1 \cdots s_c \cdots$, que l'on peut écrire plus simplement $\sigma = s_0 s_1 \cdots s_c$, est un chemin maximal de M. □

Théorème 4. *Soit P une propriété PLTL et soit $\mathcal{B}$, un automate de Büchi permettant de coder $\neg P$. Si $\mathcal{B} \in \mathcal{C}_{mod}$ alors P est une propriété vérifiable modulairement.*

Démonstration. Soit P une propriété de la PLTL. Soit $ST = \langle S_0, S, A, T, \mathcal{L} \rangle$, un système de transitions violant la propriété : $\neg (ST \models P)$. Soit $\mathcal{M}$, un découpage de ST en modules.

On veut montrer que s'il existe $\mathcal{B} \in \mathcal{C}_{mod}$ tel que $\mathcal{B}$ reconnaît $\neg P$ alors il existe un module $M \in \mathcal{M}$ dans lequel la propriété est violée, c'est à dire qu'on veut montrer que :

$$\exists M \cdot \exists \sigma' \cdot M \in \mathcal{M} \wedge \sigma' \in \Sigma(M) \wedge \sigma' \models \neg P.$$

Soit donc $\mathcal{B} = \langle b_{init}, B, P_{\mathcal{B}}, T_{\mathcal{B}}, \mathcal{F} \rangle$, un automate de Büchi reconnaissant $\neg P$ dont on suppose qu'il appartient à $\mathcal{C}_{mod}$: $\mathcal{B} \in \mathcal{C}_{mod}$.

Puisque $\neg (ST \models P)$, alors on a :

$$\exists \sigma \cdot \sigma \in \Sigma(ST) \wedge \sigma \models \neg P.$$

Puisque $\mathcal{B}$ reconnaît $\neg P$ alors il existe une séquence $\tau = b_0 b_1 \ldots$ d'états de $\mathcal{B}$ acceptant $\sigma = s_0 s_1 \ldots s_{k-1} s_k \ldots$, et comme $\mathcal{B} \in \mathcal{C}_{mod}$ alors on a pour τ :

$$\exists k \cdot k > 0 \wedge \forall i \cdot ((i < k \Rightarrow b_i = b_{init}) \wedge (i > k \Rightarrow b_i \in \mathcal{F})).$$

Avec un tel k, considérons s_{k-1} et s_k, respectivement le $(k-1)$-ème et k-ème élément (état) du chemin σ. La transition $s_{k-1} \rightarrow s_k$ appartient nécessairement à un module (d'après le lemme 2). On nomme M ce module, $M = \langle S_{M_0}, S_M, A_M, T_M, \mathcal{L}_m \rangle$ et on a : $s_{k-1} \rightarrow s_k \in T_M$.

On considère σ_{k-1}, le chemin maximal de ST défini par :

$$\sigma_{k-1} \in \Sigma(ST) \wedge \sigma_{k-1} = s_{k-1} s_k \ldots$$

Par rapport à M, deux cas doivent être envisagés : soit σ_{k-1} est « coupé » dans M au niveau d'un état de sortie s_c de M, soit σ_{k-1} est intégralement contenu dans M. Dans les deux cas, on veut prouver que la partie de σ_{k-1} contenue dans M constitue un chemin d'exécution maximal de M qui viole la propriété P.

Premier cas. Le chemin $\sigma_{k-1} = s_{k-1} s_k \cdots$ maximal de ST est coupé dans M sur un état de sortie s_c de M. Ce premier cas s'écrit :

$$\exists c \cdot c \geq k \wedge \sigma_{k-1} = s_{k-1} s_k \cdots s_c s_{c+1} \cdots \wedge s_c \in S_M \wedge s_{c+1} \notin S_M.$$

On considère σ', le chemin σ_{k-1} coupé dans M : $\sigma' = s_{k-1} s_k \cdots s_c$ (avec $c \geq k$). σ' est un chemin maximal de M car s_c est un état de sortie de M (lemme 3). De plus, σ' est accepté par $\mathcal{B}$ pour les deux raisons suivantes :

1. puisque τ accepte σ alors on a :

$$\forall i \cdot i \geq 0 \Rightarrow \exists b_i \xrightarrow{p_i} b_{i+1} \cdot b_i \xrightarrow{p_i} b_{i+1} \in T_{\mathcal{B}} \wedge \mathcal{L}(s_i) \Rightarrow p_i$$

et donc en particulier on a :

$$\forall i \cdot i \in [k-1 \cdots c] \Rightarrow \exists b_i \xrightarrow{p_i} b_{i+1} \cdot b_i \xrightarrow{p_i} b_{i+1} \in T_{\mathcal{B}} \wedge \mathcal{L}(s_i) \Rightarrow p_i,$$

2. $\mathcal{B} \in \mathcal{C}_{mod}$ donc on a :

$$b_{k-1} = b_{init} \wedge b_{c+1} \in \mathcal{F}.$$

Ainsi, on a bien $\sigma' \models \neg P$.

Deuxième cas. Le chemin σ_{k-1} n'est pas coupé dans M, ce que l'on écrit :

$$\forall i \cdot i \geq k \wedge \sigma_{k-1} = s_{k-1}s_k \cdots \Rightarrow s_i \in S_M.$$

Puisque σ_{k-1} est un chemin maximal de ST, et que tous les états de σ_{k-1} sont des états de M alors σ_{k-1} est un chemin maximal de M. De plus, σ_{k-1} est accepté par $\mathcal{B}$ car :

1. puisque τ accepte σ alors on a :

$$\forall i \cdot i \geq 0 \Rightarrow \exists b_i \xrightarrow{p_i} b_{i+1} \cdot b_i \xrightarrow{p_i} b_{i+1} \in T_{\mathcal{B}} \wedge \mathcal{L}(s_i) \Rightarrow p_i$$

et donc en particulier on a :

$$\forall i \cdot i \geq k-1 \Rightarrow \exists b_i \xrightarrow{p_i} b_{i+1} \cdot b_i \xrightarrow{p_i} b_{i+1} \in T_{\mathcal{B}} \wedge \mathcal{L}(s_i) \Rightarrow p_i,$$

2. $\mathcal{B} \in \mathcal{C}_{mod}$ donc on a :

$$b_{k-1} = b_{init} \wedge \forall i \cdot i > k \Rightarrow b_i \in \mathcal{F}.$$

On en déduit que dans le deuxième cas, avec $\sigma' = \sigma_{k-1}$, on a aussi $\sigma' \models \neg P$. □

5.5 Application à des schémas de propriétés dynamiques

La classe $\mathcal{C}_{mod}$ d'automates de Büchi permet de définir une classe de propriétés PLTL vérifiables modulairement. Nous présentons dans cette section plusieurs schémas de propriétés qui appartiennent à cette classe.

5.5.1 Les contraintes dynamiques B sont vérifiables modulairement

Nous avons vu dans la section 3.2.3 que les contraintes dynamiques B définies par Jean-Raymond Abrial et Louis Mussat pouvaient être modélisées par des schémas de propriétés PLTL. L'observation des automates de Büchi qui codent les négations de chacun de ces schémas montre, d'après le théorème 4, que ces contraintes dynamiques sont vérifiables modulairement.

L'invariant dynamique

L'invariant dynamique $\mathcal{P}(X, X')$ s'exprime par une formule PLTL de la forme $\Box\, P$, où P est une formule PLTL définie par la grammaire suivante (voir la section 3.2.3 au chapitre 3) :

$$P ::= pa \mid \bigcirc\, pa \mid \neg\, P \mid P_1 \vee P_2$$

où pa est une proposition atomique.

Ces formules se mettent sous une forme normale conjonctive dont la négation $\neg\, P$ est une forme normale disjonctive, dont l'automate de Büchi appartient à $\mathcal{C}_{mod}$.

Par exemple, la figure 5.7 présente un automate de Büchi qui code la négation de $\Box\,(p \Rightarrow \bigcirc\, q)$, c'est à dire un automate de Büchi qui code $\Diamond\, p \wedge \bigcirc \neg\, q$. Cet automate appartient à $\mathcal{C}_{mod}$ et donc $\Box\,(p \Rightarrow \bigcirc\, q)$ est une propriété vérifiable modulairement.

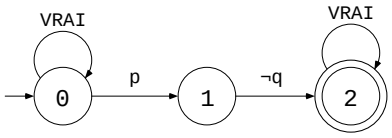

FIG. 5.7 – Un automate de Büchi reconnaissant $\neg\, \Box\,(p \Rightarrow \bigcirc\, q)$

La modalité `Leadsto`

La modalité **`Select`** p **`Leadsto`** q **`Invariant`** J **`Variant`** *Va* **`End`** s'exprime par une propriété PLTL de la forme $\Box\,(p \Rightarrow \Diamond\, q)$, où p et q sont des propositions. Un automate de Büchi qui code la négation de $\Box\,(p \Rightarrow \Diamond\, q)$ est présenté dans la figure 5.8 (la négation de $\Box\,(p \Rightarrow \Diamond\, q)$ est équivalente à $\Diamond\,(p \wedge \Box \neg\, q)$). Cet automate appartient à $\mathcal{C}_{mod}$ et donc $\Box\,(p \Rightarrow \Diamond\, q)$ est une propriété vérifiable modulairement.

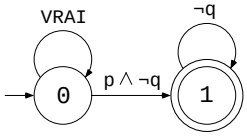

FIG. 5.8 – Un automate de Büchi reconnaissant $\neg\, \Box\,(p \Rightarrow \Diamond\, q)$

La modalité `Until`

La modalité **`Select`** p **`Until`** q **`Invariant`** J **`Variant`** *Va* **`End`** s'exprime par une propriété PLTL de la forme $\Box\,(p \Rightarrow p\, \mathcal{U}\, q)$, où p et q sont des propositions. Un automate de Büchi qui code la négation de $\Box\,(p \Rightarrow p\, \mathcal{U}\, q)$ est présenté dans la figure 5.9 (la négation de $\Box\,(p \Rightarrow p\, \mathcal{U}\, q)$ est équivalente à $\Diamond\,(p \wedge \neg\,(p\, \mathcal{U}\, q))$). Cet automate appartient à $\mathcal{C}_{mod}$ et donc $\Box\,(p \Rightarrow p\, \mathcal{U}\, q)$ est une propriété vérifiable modulairement.

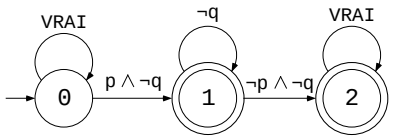

FIG. 5.9 – Un automate de Büchi reconnaissant $\neg \Box (p \Rightarrow p \,\mathcal{U}\, q)$

5.5.2 D'autres propriétés PLTL sont vérifiables modulairement

Il n'y a pas que les contraintes dynamiques B qui soient vérifiables modulairement. Nous examinons ici trois schémas de propriétés PLTL qui ne sont pas exprimables en B mais qui sont des propriétés vérifiables modulairement.

Le schéma de propriété $\Box (p \Rightarrow q \,\mathcal{U}\, r)$

Le schéma de propriété PLTL $\Box (p \Rightarrow q \,\mathcal{U}\, r)$, où p, q et r sont des propositions, n'est pas exprimable directement sous forme d'une contrainte dynamique B, mais est néanmoins un schéma de propriété très couramment utilisé dans la pratique. Nous avons représenté dans la figure 5.10 un automate de Büchi qui code la négation de cette propriété, et nous en déduisons que $\Box (p \Rightarrow q \,\mathcal{U}\, r)$ est une propriété vérifiable modulairement.

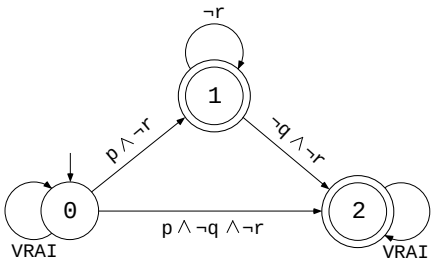

FIG. 5.10 – Un automate de Büchi reconnaissant $\neg \Box (p \Rightarrow q \,\mathcal{U}\, r)$

Le schéma de propriété $\Box (p \Rightarrow (\Diamond q) \,\mathcal{U}\, r)$

Le schéma de propriété PLTL $\Box (p \Rightarrow (\Diamond q) \,\mathcal{U}\, r)$ (où p, q et r sont des propositions) est moins couramment rencontré dans la pratique, mais il peut cependant être utilisé pour spécifier des besoins spécifiques.

Sa signification intuitive est la suivante : sur tous les chemins d'exécution, un état vérifiant p a parmi ses successeurs un état vérifiant r. Entre ces deux états, on a toujours la possibilité de rencontrer un état vérifiant q dans le futur. Par exemple, un chemin d'exécution maximal d'un STE dont les états satisfont successivement $p, \neg q, q, \neg q, q, r$ est un chemin d'exécution qui satisfait la propriété. En revanche, un chemin d'exécution maximal d'un STE dont les états

vérifient successivement $p, \neg q, q, \neg q, r \wedge \neg q, \neg q, \neg q, \ldots, \neg q, \ldots$ est un chemin d'exécution qui viole la propriété.

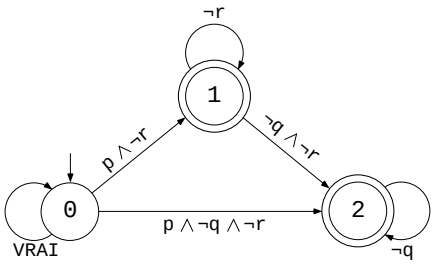

FIG. 5.11 – Un automate de Büchi reconnaissant $\neg \Box (p \Rightarrow (\Diamond q) \mathcal{U} r)$

Nous représentons dans la figure 5.11 un automate de Büchi codant la négation de $\Box (p \Rightarrow (\Diamond q) \mathcal{U} r)$. Cet automate appartient à $\mathcal{C}_{mod}$, et on en déduit que $\Box (p \Rightarrow (\Diamond q) \mathcal{U} r)$ est une propriété vérifiable modulairement.

Le schéma de propriété $\Box ((p \wedge \bigcirc t) \Rightarrow q \mathcal{U} (r \wedge \Diamond z))$

Nous nous intéressons maintenant au schéma de propriété PLTL suivant : $\Box ((p \wedge \bigcirc t) \Rightarrow q \mathcal{U} (r \wedge \Diamond z))$, où p, q, r, z et t sont des propositions. Ce schéma de propriété est relativement complexe, et il est difficile de lui attacher une signification intuitive réaliste. Il ne fait donc pas partie *a priori* des schémas que l'on rencontre en pratique. Autrement dit, il est peu probable qu'un spécifieur ait besoin d'avoir recours à un tel schéma pour spécifier les comportements attendus d'un système. Si nous le présentons, c'est pour montrer qu'il est néanmoins facile de s'assurer qu'il est vérifiable modulairement.

La figure 5.12 représente un automate de Büchi qui code la propriété $\neg \Box ((p \wedge \bigcirc t) \Rightarrow q \mathcal{U} (r \wedge \Diamond z))$. On voit que cet automate appartient à $\mathcal{C}_{mod}$, aussi on en déduit que $\Box ((p \wedge \bigcirc t) \Rightarrow q \mathcal{U} (r \wedge \Diamond z))$ est une propriété vérifiable modulairement.

5.6 Conclusion

Afin de répondre au problème de l'explosion combinatoire auquel se heurte la technique du model-checking, nous proposons de découper les STEs à vérifier en un ensemble de modules de petite taille, chacun de ces modules pouvant être vérifié par model-checking de manière indépendante. L'ensemble des transitions de chaque module doit constituer au moins une partition de l'ensemble des transitions du STE global.

Les propriétés PLTL pour lesquelles nous pouvons conclure qu'elles sont vraies globalement lorsqu'elles sont vraies sur chacun des modules sont appelées les propriétés vérifiables modulairement.

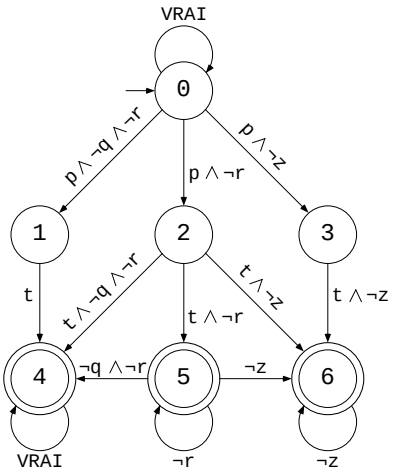

FIG. 5.12 – Un automate de Büchi reconnaissant $\neg \Box ((p \wedge \bigcirc t) \Rightarrow q \,\mathcal{U}\, (r \wedge \Diamond z))$

Nous avons défini une classe $\mathcal{C}_{mod}$ d'automates de Büchi codant des négations de propriétés dynamiques, et nous avons prouvé que le fait que la négation d'une propriété PLTL puisse être codée par un automate de la classe $\mathcal{C}_{mod}$ était une condition suffisante pour que cette propriété soit vérifiable modulairement.

Dans le chapitre suivant, nous proposons une méthode de construction des modules guidée par le raffinement B, et nous définissons les grandes lignes d'un algorithme de model-checking modulaire.

Chapitre 6

Mise en œuvre de la vérification modulaire

6.1 Le problème du « bon » découpage modulaire

Nous avons vu au chapitre précédent que certaines propriétés, dites vérifiables modulairement, avaient pour particularité de pouvoir être vérifiées de manière séparée sur chacun des modules issus d'un STE de la manière suivante : lorsque la propriété est vraie sur tous les modules, on en conclut qu'elle est vraie globalement. Le problème est qu'on a besoin que la propriété soit vraie sur *tous* les modules pour pouvoir conclure. En effet, lorsque la propriété est fausse sur l'un au moins des modules, on ne peut rien conclure quant à sa validité sur le STE global.

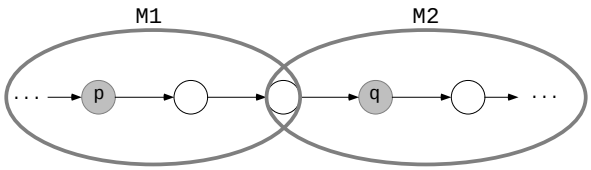

FIG. 6.1 – Une propriété vraie globalement peut devenir fausse sur un module

Considérons l'exemple de la propriété $\Box\,(p \Rightarrow \Diamond\, q)$ (où p et q sont des propositions), et supposons qu'elle soit vraie sur un STE global : dans tous les chemins d'exécution maximaux de ce STE, tout état vérifiant p compte parmi ses successeurs un état vérifiant q (ou l'état vérifiant p vérifie q lui même). Lors du découpage du STE global en modules, il est tout à fait possible qu'un chemin soit coupé entre un état vérifiant p et un état (successeur) vérifiant q, comme représenté dans la figure 6.1. Dans le module M_1 de cette figure, l'état vérifiant p n'a plus d'état vérifiant q parmi ses successeurs, de telle sorte que $\Box\,(p \Rightarrow \Diamond\, q)$ est violée sur ce module, bien que la propriété soit globalement vraie.

L'exemple de cette propriété peut être vu comme la modélisation de l'inévitabilité d'une

réponse (sous la forme d'un état vérifiant q) après une situation donnée (sous la forme d'un état vérifiant p). Si le découpage modulaire provoque une coupure avant que la réponse ne soit arrivée, rien ne permet de dire si la réponse serait arrivée ou non en l'absence de découpage.

Un autre découpage que celui présenté dans la figure 6.1 aurait peut-être permis de « voir » que la propriété était vraie au niveau de chaque module, et on aurait pu alors en conclure qu'elle était vraie globalement, puisque $\Box(p \Rightarrow \Diamond q)$ est modulaire. Le problème consiste donc à trouver un « bon » découpage modulaire, c'est à dire un découpage modulaire tel qu'une propriété vraie globalement le soit aussi au niveau des modules.

Ce problème ne doit pas être confondu avec celui évoqué au chapitre précédent et qui consistait à trouver un certain *type de propriété* : les propriétés qui sont vérifiables modulairement, quel que soit le découpage. Ici, nous sommes à la recherche d'un certain *type de découpage* : un découpage qui rende la vérification modulaire possible en pratique.

On peut exprimer cette différence autrement en disant que l'information que nous apporte la vérifiabilité modulaire d'une propriété est que si elle est vraie sur tous les modules, alors elle est vraie globalement. Par contre, si elle est vraie globalement, cela ne signifie pas qu'elle est vraie sur tous les modules.

Trouver un découpage modulaire tel qu'une propriété vraie globalement le soit aussi sur chaque module revient à trouver un découpage qui « enferme » les comportements que l'on cherche à observer dans les modules. Par exemple, lorsque l'on cherche à observer l'arrivée d'une réponse après un état satisfaisant une proposition donnée, on souhaite qu'un module contenant un état vérifiant cette proposition contienne aussi un état modélisant l'arrivée de la réponse. Dans l'exemple de la figure 6.1, cela signifie que l'on souhaite avoir un module M'_1 contenant à la fois l'état vérifiant p et l'état vérifiant q, comme c'est par exemple le cas dans la figure 6.2.

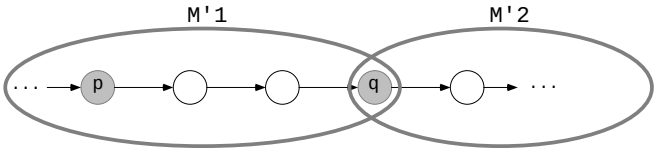

FIG. 6.2 – Un bon découpage modulaire pour la vérification de la propriété $\Box(p \Rightarrow \Diamond q)$

Ainsi, on cherche à ce que les comportements dynamiques du système ne soient pas interrompus par le découpage modulaire, ce qui revient à dire que l'on cherche à produire un découpage *sémantique* d'un STE en modules, relativement aux propriétés que l'on cherche à vérifier.

Un découpage modulaire guidé par le raffinement B [JMM99] nous permet de nous approcher de cet idéal.

6.2 Le découpage modulaire guidé par le raffinement B

6.2.1 Conséquences du raffinement B sur les systèmes de transitions exprimant leur sémantique

Nous avons vu au chapitre 1, dans la section 1.3 que la notion de raffinement B était construite autour de l'idée de raffinement temporel : raffiner peut être vu comme consistant à décrire ce que l'on voit lorsqu'on observe un système plus souvent. De cette manière, des événements qui étaient « cachés » au niveau abstrait (ils valaient `skip`) deviennent observables, et ils sont définis dans la spécification raffinée. Les événements qui étaient déjà présents dans la spécification abstraite se retrouvent sous le même nom mais avec une nouvelle définition dans la spécification raffinée. Les événements qui sont définis pour la première fois dans la spécification raffinée sont appelés les *nouveaux* événements, tandis que ceux qui sont redéfinis sont appelés les *anciens* événements.

Le raffinement conduit donc à l'observation de suites d'occurrences de nouveaux événements qui viennent s'intercaler entre les occurrences des anciens événements.

Au niveau des systèmes de transitions étiquetées qui donnent la sémantique des systèmes d'événement, le raffinement d'un événement se traduit par le raffinement des transitions qui modélisent l'occurrence de cet événement (et qui portent la même étiquette que lui). A l'état source d'une transition abstraite correspond un ensemble d'états du système de transitions raffiné qui sont tous en relation de collage avec cet état source abstrait. Ces états du système de transitions raffiné peuvent être reliés entre eux par des transitions qui correspondent à l'occurrence de *nouveaux* événements. A l'état cible d'une transition abstraite correspond un ensemble d'états du système de transitions raffiné qui sont tous en relation de collage avec cet état cible abstrait. Le lien entre ces deux ensembles d'états est assuré par des transitions correspondant à l'occurrence de l'ancien événement.

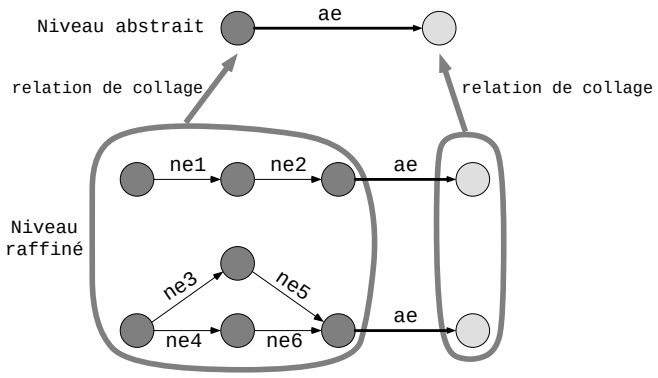

FIG. 6.3 – Le raffinement d'une transition

La figure 6.3 donne l'exemple du raffinement d'une transition correspondant à l'occurrence d'un événement *ae* (pour « ancien événement »). Dans cette figure, les transitions d'étiquette

ne_i (pour i valant 1, 2, 3, 4, 5 ou 6) correspondent à l'occurrence des nouveaux événements (notons que les ne_i ne sont pas nécessairement distincts).

De manière plus générale, la figure 6.3 illustre que le raffinement d'une transition par une famille de chemins respecte les conditions que nous exprimons informellement de la manière suivante :

- toute transition étiquetée par le nom d'un ancien événement est raffinée par un ensemble de chemins formés d'une suite de transitions étiquetées par des noms de nouveaux événements et terminés par une transition étiquetée par le nom de l'ancien événement,
- l'état source de l'ancienne transition satisfait la relation de l'invariant de collage avec l'état initial et tous les états intermédiaires de chaque chemin,
- l'état cible de l'ancienne transition satisfait la relation de l'invariant de collage avec l'état final de chacun des chemins.

Ajoutons également que, en conséquence de la définition 4 du raffinement d'un système d'événements (voir page 34), on sait que les chemins correspondant au raffinement d'une transition ne contiennent ni cycle ni blocage.

Nous illustrons le concept de raffinement de transitions en considérant deux transitions abstraites du BRP, et en exhibant dans les deux cas les chemins qui raffinent ces transitions. Ces deux exemples font référence aux représentations que nous avons données au chapitre 3 des systèmes de transitions abstrait et raffiné du BRP. La représentation du système de transitions abstrait avait été donnée par la figure 3.1 de la page 63 et la représentation du système de transitions raffiné avait été donnée par la figure 3.2 de la page 65.

Exemple 1

Nous considérons dans cet exemple le raffinement de la transition qui relie les états s_1 et s_4 de la figure 3.1. Cette transition porte l'étiquette `Abort_s`. Le chemin qui raffine cette transition se termine par une transition de même étiquette qui est précédée par deux transitions correspondant à deux occurrences du nouvel événement `Resend_s`, comme représenté dans la figure 6.4.

Exemple 2

Nous considérons dans cet exemple le raffinement de la transition d'étiquette `End_r` qui relie les états s_0 et s_1 de la figure 3.1. Les chemins qui raffinent cette transition sont tous terminés par une transition d'étiquette `End_r` précédée de transitions correspondant à l'occurrence des nouveaux événements `Receive_r` et `Resend_s`. La transition abstraite et les chemins qui la raffinent sont représentés dans la figure 6.5.

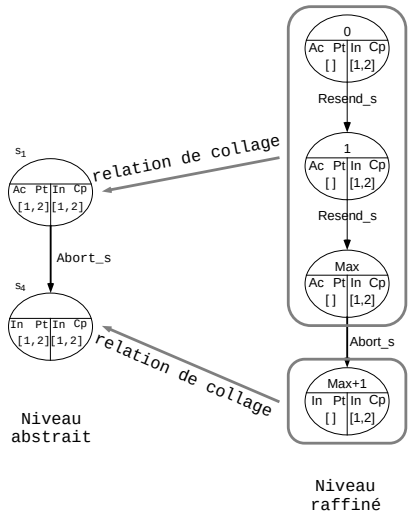

FIG. 6.4 – Le raffinement d'une transition d'étiquette `Abort_s` du BRP

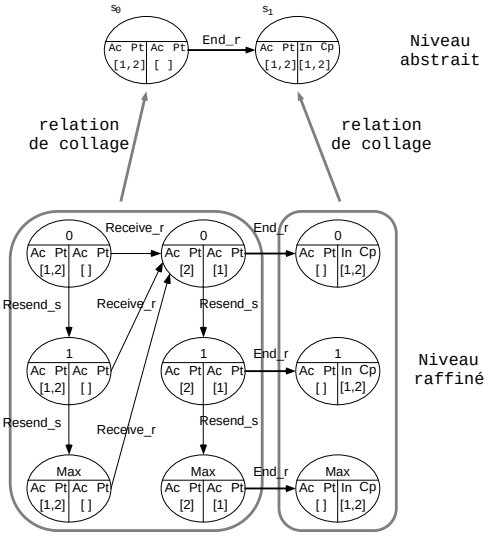

FIG. 6.5 – Le raffinement d'une transition d'étiquette `End_r` du BRP

6.2.2 Les propriétés dynamiques face au processus de raffinement

Distinction entre nouvelles et anciennes propriétés

Nous avons déjà noté au chapitre 2 (dans la section 2.1.6) que la démarche de raffinement B permettait d'introduire les propriétés dynamiques que doit respecter le système pas à pas, c'est à dire que des *nouvelles* propriétés dynamiques sont introduites à chaque étape du raffinement. Ces nouvelles propriétés modélisent des comportements devenus observables grâce à l'introduction des nouveaux événements (et des nouvelles variables), et qui sont donc des comportements qui n'auraient pas pu être observés dans la spécification abstraite.

Par exemple, lors du troisième raffinement du robot, nous avons exprimé la propriété dynamique P11 suivante : « la pince ne peut pas descendre chargée ». Avant l'introduction de l'événement `desc`, qui modélise la descente de la pince, cette propriété n'aurait pas pu être exprimée. La propriété P11 est donc une *nouvelle* propriété du troisième raffinement du robot. Par opposition, toutes les propriétés exprimées lors d'une phase antérieure de spécification (c'est à dire lors de la spécification abstraite, du premier ou du deuxième raffinement) sont appelées des *anciennes* propriétés.

La conservation de la PLTL par le raffinement B

Les propriétés que nous cherchons à vérifier par une méthode de model-checking modulaire sont les nouvelles propriétés. En effet, les anciennes propriétés, à l'exception de celles utilisant l'opérateur $\bigcirc$, sont conservées par le raffinement B, comme il a été établi dans [BDJK00, BDJK01]. Intuitivement, ceci est justifié par le fait que le raffinement B n'introduit ni livelock ni deadlock, de telle sorte qu'on ne peut pas empêcher un chemin qui menait d'un état vérifiant p à un état vérifiant q (pour une propriété de la forme $\Box(p \Rightarrow \Diamond q)$ par exemple), de continuer à faire de même lorsqu'on ajoute des transitions entre ces deux états. De plus, si un état du STE abstrait vérifie une proposition p, alors l'ensemble des états du STE raffiné qui lui sont reliés par collage vérifient tous la même proposition p en raison du « bégaiement » du raffinement : tous les nouveaux événements raffinent `skip`.

La PLTL est conservée par le raffinement B parce qu'elle ne modélise pas en général la longueur des chemins susceptibles de vérifier une propriété donnée. Par exemple, une propriété de la forme $\Box(p \Rightarrow \Diamond q)$ ne dit pas en combien de transitions on doit atteindre un état vérifiant q après celui vérifiant p. Il y a toutefois une exception avec l'opérateur $\bigcirc$ (*suivant*), et c'est pourquoi les propriétés qui l'utilisent ne sont pas conservées par le raffinement. En effet, cet opérateur spécifie que l'on doit atteindre un état donné en *une* transition exactement. Si l'on « allonge » les chemins lors du processus de raffinement, cette notion d'unicité de la transition reliant deux états est perdue.

On peut malgré tout conserver partiellement les propriétés utilisant l'opérateur $\bigcirc$ sous une forme moins restrictive lors du processus de raffinement. Par exemple, une propriété de la forme $\Box(p \Rightarrow \bigcirc q)$ peut être ré-exprimée par une propriété de la forme $\Box(p \Rightarrow \Diamond q)$, ou par une propriété de la forme $\Box(p \Rightarrow p \,\mathcal{U}\, q)$. On a alors qu'un état vérifiant p continue après

raffinement à mener à un état vérifiant q (*modulo* la relation de collage), mais on ne conserve pas l'idée que les chemins reliant ces états sont de longueur 1.

Ainsi, puisque les anciennes propriétés sont conservées par le raffinement B (éventuellement en les ré-exprimant), on n'a pas besoin de re-vérifier une propriété établie à un niveau antérieur de spécification lors des étapes suivantes du raffinement. A chaque étape, on ne vérifie que les nouvelles propriétés.

Nous cherchons donc à produire un découpage modulaire respectant la sémantique des *nouvelles* propriétés.

Puisque les nouvelles propriétés sont observables grâce à l'introduction des nouveaux événements, alors nous espérons pouvoir observer les comportements modélisés par ces propriétés sur des suites d'occurrences de nouveaux événements, éventuellement terminées par l'occurrence d'un ancien événement. En effet, intuitivement, si le comportement modélisé par une propriété était observable sur une suite comportant plusieurs occurrences d'anciens événements, alors cette propriété modéliserait un comportement qui était observable à un niveau antérieur de spécification, et on aurait donc pu (et dû) exprimer cette propriété comme une ancienne propriété à ce moment là.

6.2.3 Construction des modules par raffinement

Puisque les suites d'occurrences de nouveaux événements terminées par l'occurrence d'un ancien événement correspondent au concept que nous avons présenté de raffinement de transition, alors nous proposons de construire les modules de la manière suivante : un module contiendra l'ensemble des chemins raffinant les transitions abstraites issues d'un *même* état du STE abstrait. Il y aura donc autant de modules dans le STE raffiné qu'il y avait d'états dans le STE abstrait.

Remarque. Certains états du STE abstrait peuvent être des états terminaux, c'est à dire des états dont aucune transition n'est issue. Un tel état donne naissance à un module constitué des états en relation de collage avec l'état abstrait, et dont l'ensemble des transitions est vide.

Afin de formaliser cette notion de module issu du raffinement B, nous commençons, dans la définition 22, par définir μ, la relation de collage qui relie tout état du STE raffiné à un état du STE abstrait.

Définition 22 (Relation de collage μ).

Soient $\mathcal{S}_1 = \langle X_1, I_1, F_1, Init_1, A_1, E_{A_1} \rangle$, *une spécification abstraite de sémantique* $ST_1 = \langle S_{0_1}, S_1, A_1, T_1, \mathcal{L}_1 \rangle$ *et* $\mathcal{S}_2 = \langle X_2, I_2, F_2, Init_2, A_2, E_{A_2} \rangle$, *une spécification raffinée de sémantique* $ST_2 = \langle S_{0_2}, S_2, A_2, T_2, \mathcal{L}_2 \rangle$ *telles que* $\mathcal{S}_1 \sqsubseteq \mathcal{S}_2$. *Nous définissons* μ, *une relation de collage sur* $S_2 \times S_1$ *de la manière suivante :*

$$\forall s_1 \cdot \forall s_2 \cdot s_1 \in S_1 \wedge s_2 \in S_2 \wedge s_2 \; \mu \; s_1 \iff \mathcal{L}_2(s_2) \wedge I_2 \Rightarrow \mathcal{L}_1(s_1).$$

Cette relation de collage nous permet de définir (définition 23) une relation $\equiv$ qui relie deux états du STE raffiné lorsqu'ils sont en relation de collage avec le même état du STE abstrait.

La relation $\equiv$ est une relation d'équivalence, comme établi par la proposition 5.

Définition 23 (Relation d'équivalence $\equiv$).

Soient deux spécifications $\mathcal{S}_1$ et $\mathcal{S}_2$ telles que $\mathcal{S}_1 \sqsubseteq \mathcal{S}_2$, et de sémantiques respectives $ST_1 = \langle S_{0_1}, S_1, A_1, T_1, \mathcal{L}_1 \rangle$ et $ST_2 = \langle S_{0_2}, S_2, A_2, T_2, \mathcal{L}_2 \rangle$. Nous définissons $\equiv$, une relation définie sur $S_2 \times S_2$ de la manière suivante :

$$\forall s_2 \cdot \forall s_2' \cdot s_2 \in S_2 \wedge s_2' \in S_2' \wedge s_2 \equiv s_2' \iff \exists s_1 \cdot s_1 \in S_1 \wedge s_2 \,\mu\, s_1 \wedge s_2' \,\mu\, s_1 \wedge (\forall s_1' \cdot s_1' \in S_1 \wedge s_2 \,\mu\, s_1' \wedge s_2' \,\mu\, s_1' \Rightarrow s_1' = s_1).$$

Proposition 5. *La relation $\equiv$ est une relation d'équivalence.*

Démonstration. La relation $\equiv$ est trivialement réflexive, transitive et symétrique. □

Grâce à la relation d'équivalence $\equiv$, nous pouvons partitionner l'ensemble des états d'un STE raffiné en classes d'équivalences de la relation $\equiv$. La définition 24 décrit ce qu'est un système de transitions étiquetées issu d'une telle classe d'équivalence.

Définition 24 (STE issu d'une classe d'équivalence de la relation $\equiv$).

Soit un système de transitions étiquetées $ST_2 = \langle S_{0_2}, S_2, A_2, T_2, \mathcal{L}_2 \rangle$. Soit $S_\mathcal{C} \subseteq S_2$, une classe d'équivalence de la relation $\equiv$ sur $S_2 \times S_2$. Le système de transitions étiquetées $ST_\mathcal{C} = \langle S_{0_\mathcal{C}}, S_\mathcal{C}, A_\mathcal{C}, T_\mathcal{C}, \mathcal{L}_\mathcal{C} \rangle$ issu de cette classe d'équivalence est défini de la manière suivante :

- *$S_\mathcal{C}$ est l'ensemble des états,*
- *$T_\mathcal{C} = \{ s \xrightarrow{e} s' \;/\; s \xrightarrow{e} s' \in T_2 \wedge s \in S_\mathcal{C} \wedge s' \in S_\mathcal{C} \}$,*
- *$S_{0_\mathcal{C}}$ est l'ensemble des états initiaux de $S_\mathcal{C}$ par rapport à $T_\mathcal{C}$,*
- *$A_\mathcal{C}$ est la restriction de A_2 aux étiquettes de $T_\mathcal{C}$,*
- *$\mathcal{L}_\mathcal{C}$ est la restriction de $\mathcal{L}_2$ sur $S_\mathcal{C}$.*

La figure 6.6 représente l'ensemble des classes d'équivalence définies par la relation $\equiv$ pour le STE associé à la spécification raffinée du BRP. Cette figure est à mettre en relation avec la figure 3.1 de la page 63, qui représentait le STE associé à la spécification abstraite du BRP : chaque état des ensembles d'états notés s_0', s_1', s_2', s_3', s_4', s_5' et s_6' dans la figure 6.6 est respectivement en relation de collage avec les états notés s_0, s_1, s_2, s_3, s_4, s_5 et s_6 dans la figure 3.1.

Relativement à un STE issu d'une classe d'équivalence de la relation $\equiv$, nous décrivons maintenant dans la définition 25 ce que sont les états de sortie et les transitions de sortie d'un tel STE.

Définition 25 (États et transitions de sortie).

Soit un système de transitions étiquetées $ST_2 = \langle S_{0_2}, S_2, A_2, T_2, \mathcal{L}_2 \rangle$ et soit une partition de S_2 par la relation $\equiv$. Soit $ST_\mathcal{C} = \langle S_{0_\mathcal{C}}, S_\mathcal{C}, A_\mathcal{C}, T_\mathcal{C}, \mathcal{L}_\mathcal{C} \rangle$, un système de transitions étiquetées issu de l'une des classes d'équivalence de la relation $\equiv$.

L'ensemble $S_{sort}(ST_\mathcal{C}, ST_2)$ des états de sortie de $ST_\mathcal{C}$ dans ST_2 est défini de la manière suivante :

$$S_{sort}(ST_\mathcal{C}, ST_2) = \{ s \in S_2 \setminus S_\mathcal{C} \;/\; \exists s_i \xrightarrow{e} s \cdot s_i \xrightarrow{e} s \in T_2 \wedge s_i \in S_\mathcal{C} \}.$$

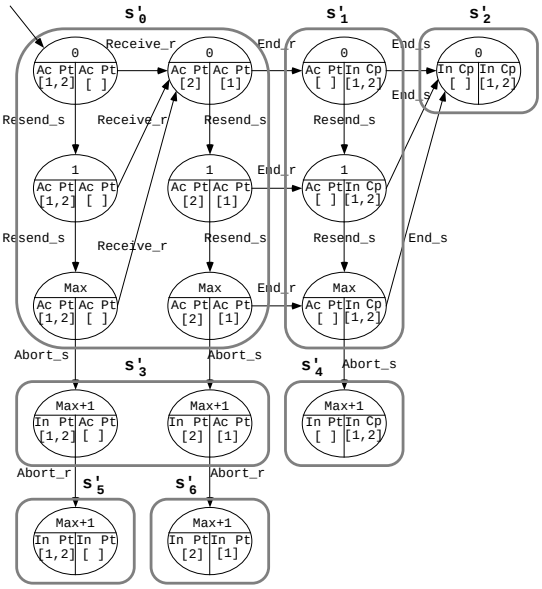

FIG. 6.6 – La partition des états du STE raffiné du BRP par la relation $\equiv$

L'ensemble $T_{sort}(ST_{\mathcal{C}}, ST_2)$ des transitions de sortie de $ST_{\mathcal{C}}$ dans ST_2 est défini de la manière suivante :

$$T_{sort}(ST_{\mathcal{C}}, ST_2) = \{s_i \xrightarrow{e} s \in T_2 \ / \ s_i \in S_{\mathcal{C}} \wedge s \notin S_{\mathcal{C}} \wedge s \in S_2\}$$

Nous pouvons alors définir ce qu'est un module obtenu par raffinement : c'est un STE issu d'une classe d'équivalence de la relation $\equiv$ et augmenté de ses états et transitions de sortie. C'est ce qu'exprime la définition 26.

Définition 26 (Module obtenu par raffinement).

Soit un système de transitions étiquetées $ST_2 = \langle S_{0_2}, S_2, A_2, T_2, \mathcal{L}_2 \rangle$ et soit une partition de S_2 par la relation $\equiv$. Soit $ST_{\mathcal{C}} = \langle S_{0_{\mathcal{C}}}, S_{\mathcal{C}}, A_{\mathcal{C}}, T_{\mathcal{C}}, \mathcal{L}_{\mathcal{C}} \rangle$, un système de transitions étiquetées issu de l'une des classes d'équivalence de la relation $\equiv$.

Le module correspondant à $ST_{\mathcal{C}}$ est un système de transitions étiquetées $M = \langle S_{0_M}, S_M, A_M, T_M, \mathcal{L}_M \rangle$ défini de la manière suivante :

- $S_{0_M} = S_{0_{\mathcal{C}}}$,
- $S_M = S_{\mathcal{C}} \cup S_{sort}(ST_{\mathcal{C}}, ST_2)$,
- A_M *est la restriction de* A_2 *aux étiquettes de* $T_{\mathcal{C}} \cup T_{sort}(ST_{\mathcal{C}}, ST_2)$,
- $T_M = T_{\mathcal{C}} \cup T_{sort}(ST_{\mathcal{C}}, ST_2)$,
- $\mathcal{L}_M$ *est la restriction de* $\mathcal{L}_2$ *sur* $S_{\mathcal{C}} \cup S_{sort}(ST_{\mathcal{C}}, ST_2)$.

6.2.4 Application aux exemples

Découpage modulaire du BRP

Le STE abstrait du BRP comporte 6 états, qui donnent naissance à 6 classes d'équivalence de la relation $\equiv$ dans le STE raffiné, c'est à dire à 6 modules.

La numérotation des modules du BRP fait référence aux numéros que nous avons attribué aux états du BRP abstrait (voir la figure 3.1, page 63) : les états d'un module M_i (avec $i \in \{1, 2, 3, 4, 5, 6\}$) sont en relation de collage avec l'état s_i du STE abstrait du BRP.

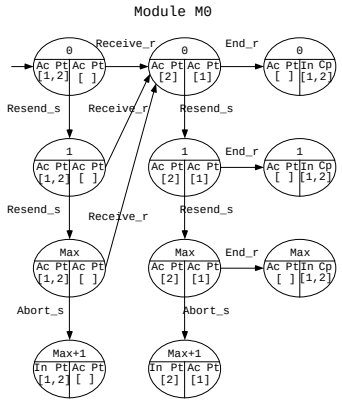

FIG. 6.7 – Le module M_0 du BRP raffiné

La figure 6.7 présente le module M_0 du STE raffiné du BRP.

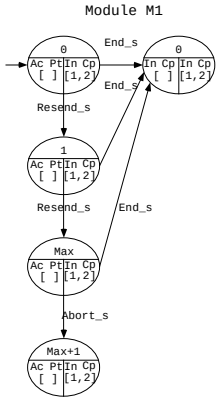

FIG. 6.8 – Le module M_1 du BRP raffiné

La figure 6.8 présente le module M_1 du STE raffiné du BRP.

La figure 6.9 présente le module M_3 du STE raffiné du BRP.

Les modules M_2, M_4, M_5 et M_6 du STE raffiné du BRP sont des modules constitués

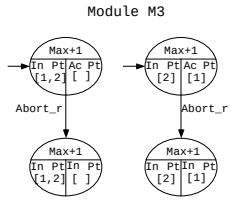

FIG. 6.9 – Le module M_3 du BRP raffiné

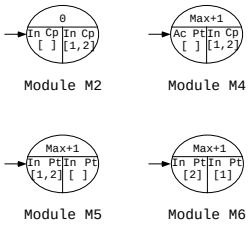

FIG. 6.10 – Les modules M_2, M_4, M_5 et M_6 du BRP raffiné

chacun d'un seul état. Ce sont des modules issus des états abstraits terminaux du BRP. Ces 4 modules sont représentés dans la figure 6.10.

Découpage modulaire du robot

Le premier raffinement du robot est découpé en deux modules $M1_0$ et $M1_1$, puisque le graphe abstrait contenait deux états. Ces deux modules sont représentés dans la figure 6.11.

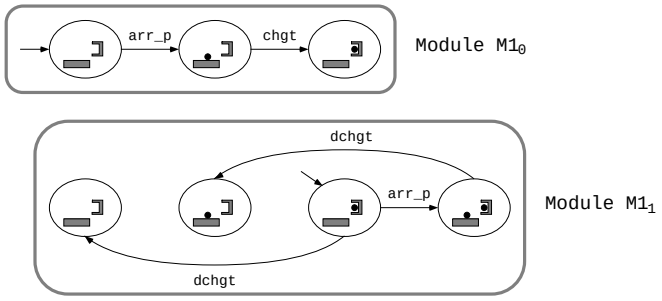

FIG. 6.11 – Les 2 modules du premier raffinement du robot

Le STE associé au deuxième raffinement du robot est découpé en 4 modules $M2_0$, $M2_1$, $M2_3$ et $M2_4$ que nous représentons dans la figure 6.12.

Le STE associé au troisième raffinement est à son tour découpé en 8 modules. Nous avons choisi de ne pas les représenter pour ne pas alourdir la lecture de ce document.

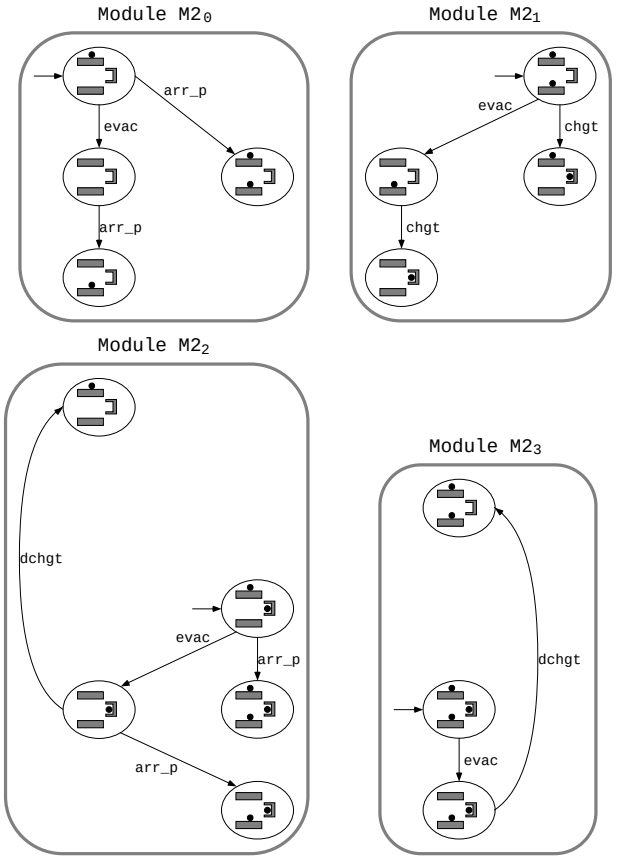

FIG. 6.12 – Les 4 modules du deuxième raffinement du robot

6.3 Principe d'un algorithme de vérification modulaire

Nous avons déjà noté qu'à chaque étape de la spécification, c'est à dire à chaque nouveau raffinement, on n'avait besoin de vérifier que les nouvelles propriétés, puisque les anciennes sont conservées par le raffinement B.

On peut supposer que dans les premières étapes de la spécification, les propriétés peuvent être vérifiées par une technique de model-checking classique (sans découpage modulaire) car à haut niveau d'abstraction, le nombre d'états servant à modéliser le système est en général faible. Par contre, plus le niveau de raffinement est détaillé, plus le risque d'explosion combinatoire du nombre d'états grandit. La technique de vérification modulaire pourra alors être utilisée pour répondre à ce problème.

Les grandes lignes d'un algorithme mettant en œuvre la vérification modulaire d'une propriété P de la PLTL sont les suivantes :

1. s'assurer que P est une propriété vérifiable modulairement,
2. calculer l'ensemble des modules, c'est à dire la partition du STE raffiné,
3. vérifier par model-checking la propriété P sur chacun des modules pris séparément,
4. conclure que P est satisfaite globalement si elle est vraie sur chaque module ; sinon, informer l'utilisateur que la vérification de P a échoué.

La vérification du fait qu'une propriété est vérifiable modulairement peut être automatisée en construisant un automate de Büchi codant la négation de cette propriété, et en s'assurant que celui-ci appartient à $\mathcal{C}_{mod}$, ce qui revient à vérifier que la condition suffisante pour la vérifiabilité modulaire d'une propriété que nous avons exhibée dans la section 5.4 est vérifiée. De nombreux algorithmes permettent de construire un automate de Büchi codant une propriété PLTL, et notamment celui proposé par Kousha ETESSAMI[1] et Gerard J. HOLZMANN dans [EH00]. Cet algorithme permet d'obtenir des automates de Büchi optimisés.

Le calcul des modules peut être obtenu par le raffinement des transitions issues de chaque état du STE abstrait, comme indiqué dans la section 6.2.3.

La vérification par model-checking d'une propriété PLTL sur un module peut être obtenue par n'importe quel model-checker capable de traiter la PLTL. On peut par exemple utiliser SPIN, mais on devra auparavant modéliser les STEs à vérifier en *Promela*, qui est le langage de description des automates utilisé par SPIN.

Le fait que la vérification modulaire d'une propriété a échoué peut être interprété de trois manières différentes :

– soit la propriété est réellement fausse,

[1] Un outil de construction d'automates de Büchi à partir de formules PLTL est disponible en ligne et téléchargeable sur le site des *Bell Laboratories*, à l'adresse internet « `http://lics.cs.bell-labs.com:8080/dok.html` ». L'ensemble des automates de Büchi codant des propriétés PLTL présentés dans ce document ont été vérifiés grâce à cet outil.

- soit la propriété, exprimée comme une nouvelle propriété, correspond en fait à un ancien comportement, et cette propriété aurait dû être exprimée sous une autre forme à un niveau antérieur de la spécification,
- soit le découpage modulaire n'a pas réussi à « enfermer » le comportement modélisé par la propriété.

6.4 Combiner preuve et model-checking pour optimiser la vérification

Lors de la vérification modulaire, à chaque étape du raffinement, on découpe le STE raffiné en autant de modules qu'il y avait d'états dans le STE abstrait précédent. Or le nombre d'états des STEs à tendance à croître de manière exponentielle au fur et à mesure des raffinements successifs du modèle initial (ce phénomène est connu sous le nom d'explosion combinatoire du nombre d'états).

En conséquence, on risque d'assister lors du processus de raffinement à une explosion combinatoire du nombre de modules à vérifier par model-checking. Chacun de ces modules étant en principe de taille « raisonnable », la vérification par model-checking sera possible pour chacun d'eux en termes d'espace mémoire. Mais l'accroissement du nombre de modules à vérifier peut avoir pour effet de remplacer le problème initial de croissance exponentielle de l'espace mémoire occupé par un problème de croissance exponentielle du temps de traitement.

Si l'on souhaite répondre à ce problème, des techniques de réduction du nombre de modules à vérifier par model-checking doivent alors être mises en œuvre.

6.4.1 Une méthode d'élimination de modules

La méthode que nous proposons dans cette section repose sur une combinaison des techniques de preuve et de model-checking pour la vérification des nouvelles propriétés sur les modules. Elle a été présentée dans [MBJM00] et propose « d'éliminer » certains modules avant la vérification par model-checking.

La technique d'élimination des modules peut être utilisée pour la vérification de propriétés PLTL de la forme $\Box\,(p \Rightarrow Q)$ où p est une proposition et où Q est une formule quelconque de la PLTL, comme par exemple q, $\Diamond\, q$, $q\,\mathcal{U}\,r$, ...Ce type de propriétés est très couramment rencontré dans la pratique.

Certains modules sont susceptibles d'échapper à la vérification par model-checking en faisant la constatation suivante :

$$\Box\,(p \Rightarrow Q) \iff (\Box \neg p) \vee (\Diamond\, p \wedge \Box\,(p \Rightarrow Q))$$

Intuitivement, l'équivalence présentée ci-dessus a la signification suivante, au niveau d'un STE : soit aucun état du STE ne vérifie p, auquel cas $\Box\,(p \Rightarrow Q)$ est trivialement vraie sur

ce STE ; soit il existe au moins un état qui vérifie p, et alors on est obligé de vérifier que sur tous les chemins, si un état vérifie p, alors la propriété Q est vérifiée à partir de cet état.

On peut donc, avant de mener la vérification par model-checking sur chaque module, séparer ceux-ci en deux catégories :

1. ceux vérifiant $\Box(\neg p)$,
2. ceux contenant au moins un état vérifiant p.

Seuls les modules appartenant à la deuxième catégorie devront être vérifiés par model-checking, puisqu'on sait que $\Box(p \Rightarrow Q)$ est vraie sur les modules de la première catégorie.

La séparation des modules en deux catégories est effectuée par une technique de preuve automatique. En effet, $\Box(\neg p)$ étant une propriété invariante, on n'a pas à exhiber de variant et d'invariant pour en assurer la vérification.

Faire la séparation des modules en deux catégories ajoute une étape dans les grandes lignes de l'algorithme de vérification modulaire que nous avons présentées dans la section 6.3.

Les grandes lignes d'un algorithme mettant en œuvre la vérification modulaire de propriétés PLTL sur des STEs en combinant les techniques de preuve et de model-checking sont alors les suivantes :

1. s'assurer que P est une propriété vérifiable modulairement et écrite sous la forme $\Box(p \Rightarrow Q)$,
2. calculer l'ensemble des modules,
3. tenter sur chaque système d'événements associé à chaque module de vérifier par preuve $\Box(\neg p)$,
4. vérifier par model-checking la propriété P sur chacun des modules pour lesquels la vérification automatique par preuve de $\Box(\neg p)$ a échoué,
5. conclure que P est satisfaite globalement si elle est vraie sur chaque module vérifié par model-checking ; sinon, informer l'utilisateur que la vérification de P a échoué.

6.4.2 Obligations de preuve pour la démonstration de $\Box \neg p$

Nous donnons ici les obligations de preuve qui devront être vérifiées (automatiquement) si l'on souhaite prouver qu'un module satisfait $\Box \neg p$ avec le démonstrateur de théorèmes de B. Ces obligations de preuve ont pour but de montrer que tous les états initiaux du module satisfont $\Box \neg p$ et que tous les événements appliqués dans le module maintiennent la condition invariante $\Box \neg p$.

Afin de mener cette démonstration, on doit extraire de chaque module M les deux ensembles suivants :

- S_{M_0}, l'ensemble des états initiaux du module,
- A_M, l'ensemble des noms des événements appliqués dans le module.

Les obligations de preuve sont générées par rapport à un système d'événements B restreint aux événements appliqués dans le module, et dont l'invariant est I. On suppose que I a été vérifié.

Chaque état $s_{i_0} \in S_{M_0}$ est traduit en une initialisation $Init_i$ du système d'événements en affectant les valeurs correspondant à s_{i_0} aux variables d'état du système. L'obligation de preuve à vérifier pour chaque $Init_i$ ainsi définie est la suivante :

$$[Init_i] \neg p.$$

Ceci prouve que les états initiaux du module satisfont tous $\neg p$.

Pour tout événement $e_i \in A_M$ et défini par $e_i \mathrel{\widehat{=}} sub_i$, on devra vérifier l'obligation de preuve suivante :

$$I \wedge \neg p \Rightarrow [sub_i] \neg p.$$

Ceci prouve que les événements appliqués dans le module maintiennent $\neg p$.

6.4.3 Quelques exemples d'application de cette technique

Le premier raffinement du BRP

Considérons les propriétés P16, P18 et P19 (la propriété P17 n'est pas de la forme $\Box(p \Rightarrow Q)$), exprimées par rapport à la spécification raffinée du BRP au chapitre 2.

La propriété P16 modélisait le fait qu'un même paquet, même s'il est retransmis par l'émetteur, ne peut jamais être enregistré deux fois sur le récepteur. Son expression PLTL était la suivante :

$$\bigwedge_{\mathtt{A} \in \mathtt{Fichier}_1} \cdot \bigwedge_{\mathtt{B} \in \mathtt{Fichier}_1} \cdot \bigwedge_{\mathtt{D} \in \{[1],[2]\}} \cdot \Box\,(\mathtt{sf}_1 = \mathtt{d} \rightarrow \mathtt{A} \wedge \mathtt{rf}_1 = \mathtt{B} \wedge \bigcirc\,(\mathtt{rf}_1 = \mathtt{B} \leftarrow \mathtt{d}) \Rightarrow \bigcirc\,(\mathtt{sf}_1 = \mathtt{A}))$$

Cette propriété ne sera à vérifier par model-checking que sur le module M_0 du BRP raffiné car les autres modules vérifient tous $\Box \neg\,(\mathtt{sf}_1 = \mathtt{d} \rightarrow \mathtt{A} \wedge \mathtt{rf}_1 = \mathtt{B} \wedge \bigcirc\,(\mathtt{rf}_1 = \mathtt{B} \leftarrow \mathtt{d}))$.

La propriété P18 modélisait le fait que si l'émetteur peut envoyer un paquet, alors ce paquet sera inévitablement reçu par le récepteur ou réexpédié MAX fois, et son expression PLTL est la suivante :

$$\begin{aligned}\bigwedge_{\mathtt{A} \in \mathtt{Fichier}_1} \cdot \bigwedge_{\mathtt{B} \in \mathtt{Fichier}_1} \cdot \bigwedge_{\mathtt{D} \in \{[1],[2]\}} \cdot \Box\,((\mathtt{sa}_1 = \mathtt{act} \wedge \mathtt{sf}_1 = \mathtt{d} \rightarrow \mathtt{A} \wedge \mathtt{rf}_1 = \mathtt{B}) \\ \Rightarrow \Diamond\,(\mathtt{rf}_1 = \mathtt{B} \leftarrow \mathtt{d} \vee \mathtt{src}_1 = \mathtt{MAX} + 1))\end{aligned}$$

Cette propriété n'est à vérifier par model-checking que sur le module M_0 du BRP raffiné car les autres modules satisfont $\Box\,(\mathtt{sa}_1 = \mathtt{inact} \vee \mathtt{sf}_1 = [\,])$.

La propriété P19 modélisait le fait que si les deux sites sont actifs, alors un paquet situé sur le site émetteur y reste jusqu'à ce qu'inévitablement, soit l'émetteur abandonne, soit le paquet est reçu par le récepteur. Son expression PLTL est la suivante :

$$\bigwedge_{A \in Fichier_1} \cdot \bigwedge_{B \in Fichier_1} \cdot \bigwedge_{D \in \{[1],[2]\}} \cdot$$
$$\Box\,((\mathtt{sf}_1 = \mathtt{d}{\rightarrow}\mathtt{B} \wedge \mathtt{rf}_1 = \mathtt{A} \wedge \mathtt{sa}_1 = \mathtt{act} \wedge \mathtt{ra}_1 = \mathtt{act})$$
$$\Rightarrow ((\mathtt{sf}_1 = \mathtt{d}{\rightarrow}\mathtt{B} \wedge \mathtt{rf}_1 = \mathtt{A} \wedge \mathtt{sa}_1 = \mathtt{act} \wedge \mathtt{ra}_1 = \mathtt{act})\ \mathcal{U}\ (\mathtt{rf}_1 = \mathtt{A}{\leftarrow}\mathtt{d} \vee \mathtt{src}_1 = \mathtt{MAX} + 1)))$$

Cette propriété n'est à vérifier par model-checking que sur le module M_0 du BRP raffiné car les autres modules satisfont $\Box\,(\mathtt{sa}_1 = \mathtt{inact} \vee \mathtt{ra}_1 = \mathtt{inact})$.

Le premier raffinement du robot

Les trois propriétés dynamiques P3, P4 et P5 exprimées lors du premier raffinement du robot (voir la section 3.2.4 page 73 pour leur expression PLTL) doivent être vérifiées par model-checking sur les deux modules décrits dans la figure 6.12 issus de ce raffinement car on ne peut établir la propriété générique $\Box\,(\neg\, p)$ sur aucun d'eux.

Le deuxième raffinement du robot

Nous considérons les propriétés dynamiques établies lors du deuxième raffinement du robot.

La propriété P6 exprimait le fait que la pince ne peut décharger que sur un dispositif d'évacuation vide. Son expression PLTL est la suivante :

$$\Box\,((\mathtt{dt}_2 = \mathtt{occ} \wedge \bigcirc\, \mathtt{dt}_2 = \mathtt{vid}) \Rightarrow \mathtt{de}_2 = \mathtt{vid})$$

ce qui est équivalent à

$$\Box\,((\mathtt{dt}_2 = \mathtt{occ} \wedge \mathtt{de}_2 = \mathtt{occ}) \Rightarrow \bigcirc\,(\mathtt{dt}_2 = \mathtt{occ}))$$

Cette propriété n'est à vérifier par model-checking que sur les modules $M2_1$, $M2_2$ et $M2_3$ du deuxième raffinement du robot (le module $M2_0$ est « éliminé ») car le module $M2_0$ satisfait $\Box\,(\mathtt{dt}_2 = \mathtt{vid} \vee \mathtt{de}_2 = \mathtt{vid})$.

La propriété P7 exprimait le fait que la pince occupée le reste jusqu'à ce qu'inévitablement le dispositif d'évacuation se libère. Son expression PLTL est la suivante :

$$\Box\,(\mathtt{dt}_2 = \mathtt{occ} \Rightarrow \mathtt{dt}_2 = \mathtt{occ}\ \mathcal{U}\ \mathtt{de}_2 = \mathtt{vid})$$

Comme pour la propriété P6, la propriété P7 n'est à vérifier par model-checking que sur les modules $M2_1$, $M2_2$ et $M2_3$ du deuxième raffinement du robot. En effet, le module $M2_0$ satisfait $\Box\,(\mathtt{dt}_2 = \mathtt{vid})$.

La propriété P8 exprimait le fait que si une pièce est placée sur le dispositif d'évacuation alors elle sera évacuée. Son expression PLTL est la suivante :

$$\Box\,(\mathtt{de}_2 = \mathtt{occ} \Rightarrow \Diamond\, \mathtt{de}_2 = \mathtt{vid})$$

Cette propriété doit être vérifiée par model-checking sur tous les modules du deuxième raffinement du robot car sur aucun d'eux on ne peut établir $\Box\,(\mathtt{de}_2 = \mathtt{vid})$.

6.5 Conclusion

Pour vérifier modulairement une propriété, on a besoin que celle-ci soit vraie sur tous les modules, ce qui pose le problème de trouver un découpage modulaire pour lequel les propriétés ont de grandes chances d'être vraies. Nous avons proposé dans ce chapitre de guider le découpage par le raffinement B : les nouvelles propriétés sont vérifiées sur des modules contenant les suites d'occurrence de nouveaux événements. Le découpage guidé par le raffinement B est un découpage sémantique.

Nous avons également présenté une méthode de vérification modulaire combinant preuve et model-checking : lorsqu'on sait prouver automatiquement qu'un module satisfait une propriété, ce module n'a pas à être vérifié par model-checking.

Au chapitre suivant, nous menons la vérification par model-checking modulaire des deux exemples présentés au chapitre 2, ce qui permet de relever quelques limites pratiques à l'applicabilité de la méthode.

Chapitre 7

Limites de la technique de vérification modulaire

7.1 Vérification modulaire des propriétés des exemples

Nous présentons dans cette section les résultats de l'application de la méthode de vérification modulaire aux exemples du BRP et du robot. À chaque raffinement, les résultats sont représentés sous la forme d'un tableau indiquant pour chaque propriété et pour chaque module le résultat de la vérification de la propriété sur le module (ce résultat peut être soit `VRAI`, soit `FAUX`).

Nous avons de plus indiqué quels étaient les modules dont la vérification par model-checking pouvait être évitée, pour une propriété particulière, en appliquant la méthode de vérification modulaire avec élimination de modules (voir la section 6.4.1). Cette indication est donnée par le fait que le résultat de la vérification (qui est forcément `VRAI`) est accompagné de la mention « $(\Box \neg p)$ ».

7.1.1 Vérification modulaire des propriétés du BRP

Présentation des résultats

Les propriétés P16 à P19 avaient été exprimées lors du raffinement du robot. Nous redonnons leur expression PLTL, en rappelant que $Fichier_1$ est défini par $\{[\,], [\mathtt{1}], [\mathtt{2}], [\mathtt{1}, \mathtt{2}]\}$ et que nous avions attribué à MAX la valeur 2 (voir la section 2.2 au chapitre 2) :

P16	$\bigwedge_{\mathtt{A}\in\mathtt{Fichier}_1} \cdot \bigwedge_{\mathtt{B}\in\mathtt{Fichier}_1} \cdot \bigwedge_{\mathtt{D}\in\{[1],[2]\}} \cdot$ $\Box\,((\mathtt{sf}_1 = \mathtt{d}{\rightarrow}\mathtt{A} \wedge \mathtt{rf}_1 = \mathtt{B} \wedge \bigcirc\,(\mathtt{rf}_1 = \mathtt{B}{\leftarrow}\mathtt{d})) \Rightarrow \bigcirc\,(\mathtt{sf}_1 = \mathtt{A}))$
P17	$\Box\,\Diamond\,(\mathtt{src}_1 = 0 \vee \mathtt{src}_1 = \mathtt{MAX}+1)$
P18	$\bigwedge_{\mathtt{A}\in\mathtt{Fichier}_1} \cdot \bigwedge_{\mathtt{B}\in\mathtt{Fichier}_1} \cdot \bigwedge_{\mathtt{D}\in\{[1],[2]\}} \cdot \Box\,((\mathtt{sa}_1 = \mathtt{act} \wedge \mathtt{sf}_1 = \mathtt{d}{\rightarrow}\mathtt{A} \wedge \mathtt{rf}_1 = \mathtt{B})$ $\Rightarrow \Diamond\,(\mathtt{rf}_1 = \mathtt{B}{\leftarrow}\mathtt{d} \vee \mathtt{src}_1 = \mathtt{MAX}+1))$
P19	$\bigwedge_{\mathtt{A}\in\mathtt{Fichier}_1} \cdot \bigwedge_{\mathtt{B}\in\mathtt{Fichier}_1} \cdot \bigwedge_{\mathtt{D}\in\{[1],[2]\}} \cdot$ $\Box\,((\mathtt{sf}_1 = \mathtt{d}{\rightarrow}\mathtt{B} \wedge \mathtt{rf}_1 = \mathtt{A} \wedge \mathtt{sa}_1 = \mathtt{act} \wedge \mathtt{ra}_1 = \mathtt{act})$ $\Rightarrow ((\mathtt{sf}_1 = \mathtt{d}{\rightarrow}\mathtt{B} \wedge \mathtt{rf}_1 = \mathtt{A} \wedge \mathtt{sa}_1 = \mathtt{act} \wedge \mathtt{ra}_1 = \mathtt{act})$ $\mathcal{U}\,(\mathtt{rf}_1 = \mathtt{A}{\leftarrow}\mathtt{d} \vee \mathtt{src}_1 = \mathtt{MAX}+1)))$

Les résultats de la vérification modulaire de ces propriétés sont donnés dans le tableau 7.1. Les modules M_0 à M_6 sont représentés dans la section 6.2.4 (page 126).

		Modules						
		M_0	M_1	M_2	M_3	M_4	M_5	M_6
Propriétés	P16	VRAI	VRAI $(\Box\neg p)$	VRAI $(\Box\neg p)$	VRAI $(\Box\neg p)$	VRAI $(\Box\neg p)$	VRAI $(\Box\neg p)$	VRAI $(\Box\neg p)$
	P17	FAUX	VRAI	VRAI	VRAI	VRAI	VRAI	VRAI
	P18	VRAI	VRAI $(\Box\neg p)$	VRAI $(\Box\neg p)$	VRAI $(\Box\neg p)$	VRAI	VRAI $(\Box\neg p)$	VRAI $(\Box\neg p)$
	P19	VRAI	VRAI $(\Box\neg p)$	VRAI $(\Box\neg p)$	VRAI $(\Box\neg p)$	VRAI $(\Box\neg p)$	VRAI $(\Box\neg p)$	VRAI $(\Box\neg p)$

TAB. 7.1 – Résultats de la vérification modulaire du BRP raffiné

Analyse des résultats de la vérification modulaire du BRP

Les propriétés P16, P18 et P19 sont vraies sur chacun des modules issus de la spécification raffinée du BRP. Comme ce sont des propriétés vérifiables modulairement, on en déduit qu'elles sont satisfaites par la spécification raffinée complète.

La vérification de la propriété P17 (vérifiable modulairement) échoue sur le module M_0. La vérification modulaire ne permet donc pas de conclure que la propriété est respectée ou non par la spécification. Cependant, le model-checking sur le STE complet correspondant à cette spécification montre que la propriété P17 est vraie.

Ici, l'échec de la vérification de cette propriété provient du fait qu'elle n'est pas vraiment une *nouvelle* propriété. Rappelons l'expression de la propriété P17 :

$$\Box\,\Diamond\,(src_1 = 0 \vee src_1 = MAX + 1)$$

Elle signifie qu'il est toujours le cas que le compteur de retransmissions prendra inévitablement la valeur 0 ou $MAX+1$. Or, la valeur 0 pour src_1 à la fin de l'exécution du protocole signifie que le transfert du fichier a été complet, tandis que la valeur $MAX+1$ signifie que le transfert a échoué, et qu'il est donc partiel.

Ce que P17 exprime est donc le fait que le protocole terminera et que le transfert sera inévitablement complet ou partiel. Ceci aurait pu être exprimé dès la spécification abstraite du BRP sous la forme d'une *ancienne* propriété de la manière suivante :

$$\textit{(P17')}\ \Box\,(sa_0 = \texttt{act} \vee ra_0 = \texttt{act} \Rightarrow \Diamond\,(sa_0 = \texttt{inact} \wedge ra_0 = \texttt{inact} \wedge (st_0 = \texttt{complet} \vee st_0 = \texttt{partiel})).$$

La propriété P17' était vraie sur le BRP abstrait, et comme elle est conservée par le raffinement, alors elle reste vraie sur le BRP raffiné.

7.1.2 Vérification modulaire des propriétés du robot

Présentation des résultats pour le premier raffinement du robot

Le premier raffinement du robot était accompagné de la spécification de trois propriétés dynamiques, les propriétés P3, P4 et P5, dont nous rappelons l'expression PLTL :

P3	$\Box\,((\texttt{da}_1 = \texttt{occ} \wedge \bigcirc\,\texttt{da}_1 = \texttt{vid}) \Rightarrow \texttt{dt}_1 = \texttt{vid})$
P4	$\Box\,(\texttt{da}_1 = \texttt{occ} \Rightarrow \texttt{da}_1 = \texttt{occ}\ \mathcal{U}\ \texttt{dt}_1 = \texttt{vid})$
P5	$\Box\,((\texttt{dt}_1 = \texttt{occ} \wedge \texttt{da}_1 = \texttt{vid} \wedge \bigcirc\,\texttt{dt}_1 = \texttt{vid}) \Rightarrow \bigcirc\,\texttt{da}_1 = \texttt{vid})$

Les deux modules $M1_0$ et $M1_1$ issus du premier raffinement du robot sont représentés dans la figure 6.11 de la page 127.

		Modules	
		$M1_0$	$M1_1$
Propriétés	P3	VRAI	VRAI
	P4	VRAI	VRAI
	P5	VRAI ($\Box\,\neg\,p$)	VRAI

TAB. 7.2 – Résultats de la vérification modulaire du premier raffinement du robot

Le tableau 7.2 présente les résultats de la vérification modulaire de ces trois propriétés (vérifiables modulairement) sur les deux modules.

Présentation des résultats pour le deuxième raffinement du robot

Les propriétés dynamiques P6, P7 et P8 avaient été exprimées lors du deuxième raffinement du robot. Leur expression PLTL était la suivante :

P6	$\Box\,((\texttt{dt}_2 = \texttt{occ} \wedge \bigcirc\,\texttt{dt}_2 = \texttt{vid}) \Rightarrow \texttt{de}_2 = \texttt{vid})$
P7	$\Box\,(\texttt{dt}_2 = \texttt{occ} \Rightarrow \texttt{dt}_2 = \texttt{occ}\ \mathcal{U}\ \texttt{de}_2 = \texttt{vid})$
P8	$\Box\,(\texttt{de}_2 = \texttt{occ} \Rightarrow \Diamond\,\texttt{de}_2 = \texttt{vid})$

		Modules			
		$M2_0$	$M2_1$	$M2_2$	$M2_3$
Propriétés	P6	VRAI ($\Box \neg p$)	VRAI	VRAI	VRAI
	P7	VRAI ($\Box \neg p$)	FAUX	FAUX	VRAI
	P8	FAUX	FAUX	FAUX	FAUX

TAB. 7.3 – Résultats de la vérification modulaire du deuxième raffinement du robot

Les quatre modules $M2_0$, $M2_1$, $M2_2$ et $M2_3$, issus du deuxième raffinement du robot, ont été représentés dans la figure 6.12 de la page 128.

Le tableau 7.3 présente les résultats de la vérification modulaire des propriétés P6, P7 et P8 (vérifiables modulairement) sur les modules $M2_0$, $M2_1$, $M2_2$ et $M2_3$.

Présentation des résultats pour le troisième raffinement du robot

Lors du troisième raffinement du robot, nous avons exprimé les propriétés dynamiques P9 à P14, dont nous redonnons l'expression PLTL :

P9	$\Box\,(\mathtt{posdt_3} = \mathtt{haut} \wedge \mathtt{dt_3} = \mathtt{vid} \Rightarrow \Diamond\, \mathtt{posdt_3} = \mathtt{bas})$
P10	$\Box\,(\mathtt{posdt_3} = \mathtt{bas} \wedge \mathtt{dt_3} = \mathtt{occ} \Rightarrow \Diamond\, \mathtt{posdt_3} = \mathtt{haut})$
P11	$\Box\,((\mathtt{posdt_3} = \mathtt{haut} \wedge \bigcirc\, \mathtt{posdt_3} = \mathtt{bas}) \Rightarrow \mathtt{dt_3} = \mathtt{vid})$
P12	$\Box\,((\mathtt{posdt_3} = \mathtt{haut} \wedge \mathtt{dt_3} = \mathtt{occ}) \Rightarrow \mathtt{posdt_3} = \mathtt{haut} \wedge \mathtt{dt_3} = \mathtt{occ}\; \mathcal{U}\; \mathtt{dt_3} = \mathtt{vid})$
P13	$\Box\,((\mathtt{posdt_3} = \mathtt{bas} \wedge \bigcirc\, \mathtt{posdt_3} = \mathtt{haut}) \Rightarrow \mathtt{dt_3} = \mathtt{occ})$
P14	$\Box\,((\mathtt{posdt_3} = \mathtt{bas} \wedge \mathtt{dt_3} = \mathtt{vid}) \Rightarrow \mathtt{posdt_3} = \mathtt{bas} \wedge \mathtt{dt_3} = \mathtt{vid}\; \mathcal{U}\; \mathtt{dt_3} = \mathtt{occ})$

Dans la section 6.2.4, nous n'avions pas représenté les 8 modules issus du troisième raffinement du robot. Ceux-ci, numérotés de $M3_0$ à $M3_7$ sont représentés dans la figure 7.1.

Le tableau 7.4 présente les résultats de la vérification modulaire des propriétés P9 à P14 sur les 8 modules issus du troisième raffinement du robot. Les propriétés P9 à P14 sont toutes vérifiables modulairement.

Analyse des résultats de la vérification modulaire du robot

Si la vérification modulaire avait donné un bon taux de succès pour la vérification des propriétés du BRP, il n'en a va pas de même pour les propriétés du robot. On constate en effet que seul le premier raffinement du robot (voir le tableau 7.2) a pu être vérifié entièrement de cette manière.

Lors du deuxième raffinement du robot, seule la propriété P6 a pu être prouvée de manière modulaire. Les propriétés P7 et P8 n'étant pas vraies sur tous les modules, on ne peut pas conclure qu'elles sont respectées par le système. Elles sont pourtant vraies sur le STE complet.

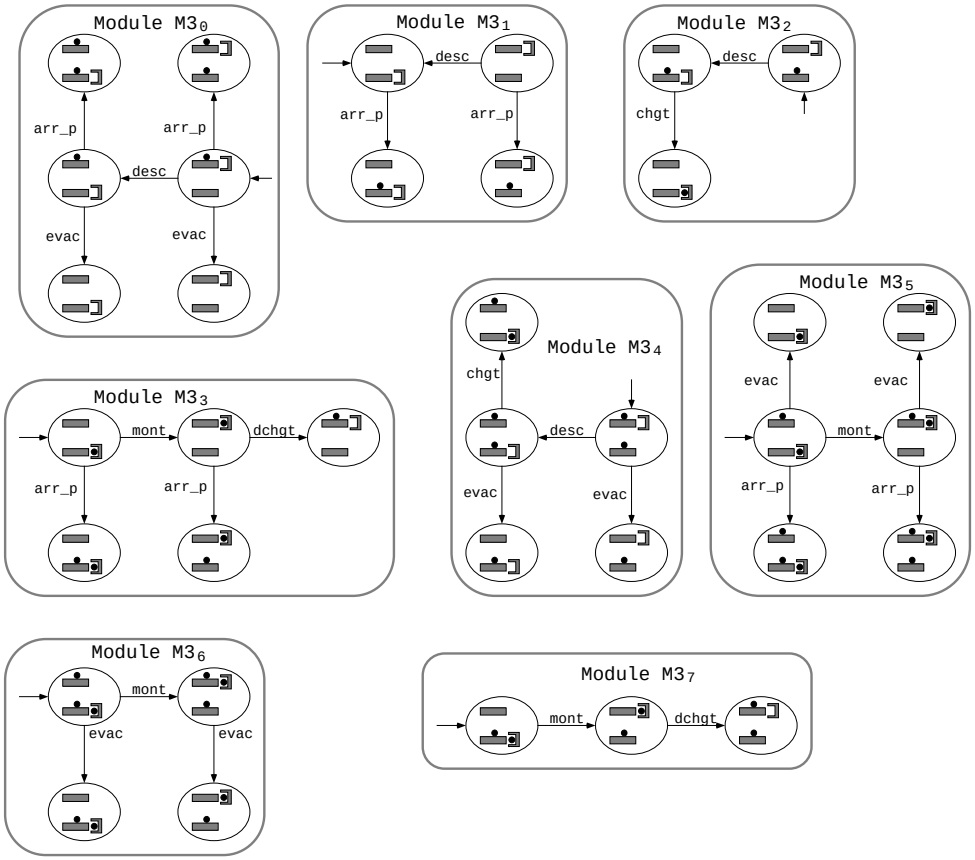

FIG. 7.1 – Les 8 modules du troisième raffinement du robot

		Modules							
		$M3_0$	$M3_1$	$M3_2$	$M3_3$	$M3_4$	$M3_5$	$M3_6$	$M3_7$
Propriétés	P9	FAUX	FAUX	VRAI	FAUX	FAUX	VRAI ($\Box\neg p$)	VRAI ($\Box\neg p$)	FAUX
	P10	VRAI ($\Box\neg p$)	VRAI ($\Box\neg p$)	FAUX	FAUX	FAUX	FAUX	FAUX	VRAI
	P11	VRAI	VRAI	VRAI	VRAI ($\Box\neg p$)	VRAI	VRAI ($\Box\neg p$)	VRAI ($\Box\neg p$)	VRAI ($\Box\neg p$)
	P12	VRAI ($\Box\neg p$)	VRAI ($\Box\neg p$)	VRAI ($\Box\neg p$)	FAUX	VRAI ($\Box\neg p$)	FAUX	FAUX	VRAI
	P13	VRAI ($\Box\neg p$)	VRAI ($\Box\neg p$)	VRAI ($\Box\neg p$)	VRAI	VRAI ($\Box\neg p$)	VRAI	VRAI ($\Box\neg p$)	VRAI
	P14	FAUX	FAUX	VRAI	VRAI ($\Box\neg p$)	FAUX	VRAI ($\Box\neg p$)	VRAI ($\Box\neg p$)	VRAI ($\Box\neg p$)

TAB. 7.4 – Résultats de la vérification modulaire du troisième raffinement du robot

Contrairement à la propriété P17 du BRP, il ne semble pas que les propriétés P7 et P8 soient en fait des anciennes propriétés exprimées trop tardivement. Il ne reste alors qu'une cause possible à l'échec de la vérification de ces propriétés : le découpage modulaire n'a pas permis d'observer les comportements qu'elles décrivent.

Lors du troisième raffinement du robot, seules deux propriétés (P11 et P13) ont pu être vérifiées de manière modulaire. Pour les autres propriétés, la vérification modulaire échoue puisque les propriétés ne sont pas vraies sur tous les modules. Là encore, on s'assure facilement que toutes les propriétés sont vraies globalement.

Le découpage modulaire que nous avons proposé, s'il permet la vérification modulaire des propriétés du BRP, n'est donc pas adapté au problème du robot. En d'autres termes, il ne constitue pas un « bon » découpage modulaire dans le cas du robot. La section suivante présente quelques pistes permettant d'expliquer cette situation, afin d'y remédier.

7.2 Vers un autre découpage modulaire

Commençons par noter que les propriétés du robot pour lesquelles la vérification modulaire permet de conclure sont des propriétés n'utilisant que l'opérateur $\bigcirc$ comme opérateur temporel. Ces propriétés sont considérées en langage B comme des invariants dynamiques. En PLTL, on parle de *propriétés de sûreté*.

On ne s'étonne pas que les propriétés utilisant l'opérateur $\bigcirc$ soient vérifiées de manière modulaire car elles expriment des comportements observables sur des successions de deux états. Sachant que toute succession de deux états appartient nécessairement à un module, on sait que les comportements que ces propriétés décrivent sont soit observables dans les modules, soit non définis.

Les propriétés dynamiques du robot pour lesquelles la vérification modulaire échoue sont des propriétés utilisant les opérateurs $\Diamond$ et $\mathcal{U}$ de la PLTL, et leur satisfaction au niveau des modules dépend donc du découpage modulaire appliqué au STE global.

7.2.1 Le problème de l'asynchronisme des événements

Examinons le cas de l'échec de la vérification d'une propriété du robot pour tenter de comprendre ce qui provoque cet échec.

Considérons par exemple la propriété P8, exprimée lors du deuxième raffinement du robot. Cette propriété exprime que « si une pièce est placée sur le dispositif d'évacuation, alors elle sera inévitablement évacuée ». Elle s'exprime en PLTL par

$$\Box\,(de_2 = \texttt{occ} \Rightarrow \Diamond\, de_2 = \texttt{vid}).$$

Cette propriété est bien liée à l'introduction de l'événement *evac* lors du deuxième raffinement du robot. Considérons le module $M2_0$, que nous représentons une nouvelle fois (en numérotant ses états) dans la figure 7.2.

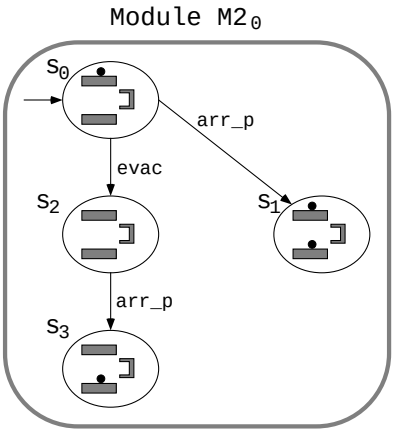

FIG. 7.2 – Le module $M2_0$ issu du deuxième raffinement du robot

La propriété P8 n'est pas vraie sur le module $M2_0$ car le chemin maximal $s_0 s_1$ ne la satisfait pas (le chemin maximal s_1 ne la satisfait pas non plus). On passe de s_0 à s_1 en appliquant directement l'ancien événement arr_p, sans même avoir appliqué le nouvel événement *evac*, c'est à dire sans avoir eu le temps d'observer un *nouveau* comportement. Ceci provient du fait que l'arrivée d'une pièce sur le dispositif d'arrivée peut se produire de manière totalement *asynchrone* avec l'évacuation d'une pièce qui se trouve sur le dispositif d'évacuation. Exprimé autrement, l'arrivée de la pièce ne se produit pas forcément *après* une opération d'évacuation. Or, nous avons défini les modules de telle sorte qu'ils permettent l'observation uniquement de ce qui se produit *avant* l'occurrence d'un ancien événement.

Dans le cas du BRP, cela ne pose pas de problème car les nouveaux événements que l'on introduit lors du raffinement de la spécification abstraite sont bien synchronisés par rapport aux anciens événements. Dans la spécification abstraite, on se contente d'observer qu'un transfert de fichier (complet ou partiel) a lieu entre le site émetteur et le site récepteur, et les événements modélisent directement la fin de ce transfert, sans qu'on puisse observer comment celui-ci s'est déroulé. Lors du raffinement de cette spécification, on précise comment le transfert a lieu, c'est à dire paquet par paquet. Les comportements devenus observables grâce à ce raffinement sont donc bien « contenus » dans les modules.

Dans le cas du robot en général, et donc en particulier dans le module $M2_0$, les événements arr_p et *evac* étant asynchrones, l'évacuation d'une pièce peut se produire soit avant l'arrivée d'une nouvelle pièce (comportement modélisé par le chemin $s_0 s_2 s_3$), soit après (comportement non modélisé dans le module $M2_0$). Ainsi, le comportement indiquant qu'une opération d'évacuation aura inévitablement lieu n'est pas nécessairement observable sur tous les chemins du module, ce qui explique que la vérification de la propriété PLTL modélisant ce comportement échoue au niveau des modules.

Le découpage modulaire que nous avons proposé n'est donc pas adapté à l'observation des comportements asynchrones.

7.2.2 Proposition d'un découpage modulaire adapté aux événements asynchrones

Afin de répondre à ce problème, nous proposons pour le robot un nouveau découpage modulaire des deuxième et troisième raffinements du robot. Ce découpage a pour but de forcer l'observation des nouveaux comportements dans les modules, en dépit de l'asynchronisme des événements. Il ne repose pas sur une justification théorique approfondie, et il doit à ce titre être considéré comme une perspective à la recherche d'un découpage modulaire tenant compte du possible asynchronisme des événements.

L'idée du nouveau découpage

Intuitivement, l'idée principale de ce nouveau découpage consiste à laisser les nouveaux événements se produire, même si leur occurrence a lieu après l'occurrence d'un ancien événement. Ainsi, sur l'exemple du module $M2_0$ précédemment cité, on pourra observer l'évacuation, même si celle-ci se produit après que l'arrivée d'une nouvelle pièce a eu lieu.

On risque alors d'observer des redondances entre les différents modules, puisqu'une même transition pourra appartenir à plusieurs modules distincts.

Notons que la condition suffisante de vérifiabilité modulaire que nous avons présentée dans la section 5.4 reste applicable avec ce type de découpage, puisque celle-ci repose sur l'idée que toute transition du STE global appartient nécessairement à l'un des modules. Le plus petit découpage (en terme de nombres de transitions) répondant à cette exigence est un découpage proposant une partition des transitions du STE global, ce qui est le cas du découpage modulaire que nous avons proposé dans la section 6.2.3. Le découpage modulaire que nous proposons maintenant ne repose plus sur une partition des transitions du STE global, mais sur un recouvrement de celles-ci, et la condition de vérifiabilité modulaire reste valide pour ce découpage.

Afin de limiter la redondance des transitions entre les différents modules, nous proposons de plus de n'appliquer qu'un seul ancien événement par module, ce qui n'était pas le cas avec le découpage précédent (voir par exemple le module $M2_1$ du deuxième raffinement du robot dans la figure 6.12 page 128, dans lequel deux anciens événements sont appliqués : les événements *dchgt* et *arr_p*). Ceci aura pour effet de produire plus de modules qu'auparavant, mais en limitant la taille de chacun d'eux.

Construction des modules par ce nouveau découpage

La méthode de construction des « nouveaux » modules est alors la suivante : soit s_1 un état du STE abstrait,

1. construire l'ensemble des états du STE raffiné qui sont en relation de collage avec s_1,
2. choisir un ancien événement parmi ceux correspondant à une transition issue de s_1,

3. relier entre eux ces états du STE raffiné en appliquant soit les nouveaux événements, soit l'ancien événement choisi.

Ainsi, si d'un état abstrait partent plusieurs transitions, alors on construira autant de modules qu'il y a de transitions issues de cet état. Le nombre total de modules que produit cette méthode de découpage est égal au nombre de transitions du STE abstrait.

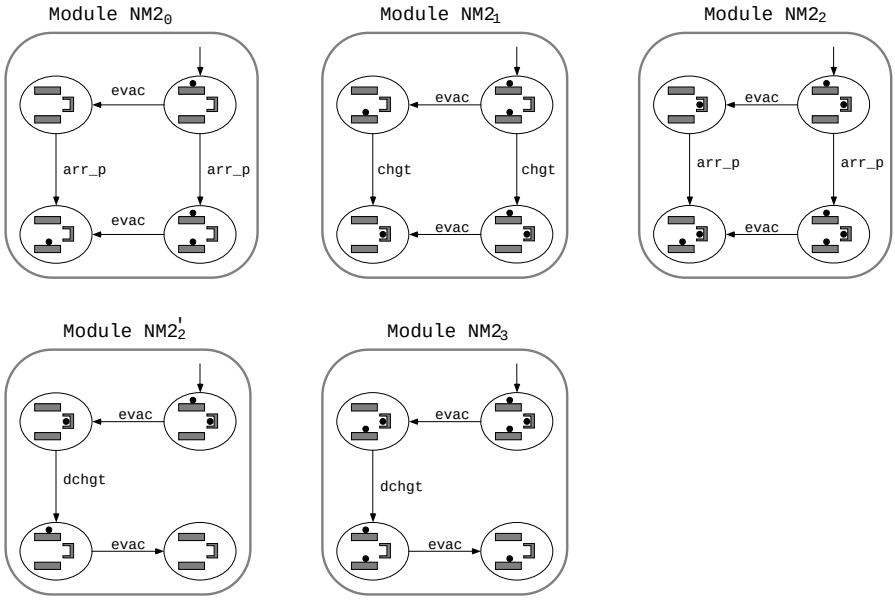

FIG. 7.3 – Un autre découpage modulaire du deuxième raffinement du robot

Nous représentons dans la figure 7.3 l'ensemble des modules (au nombre de 5) qu'il est possible de construire de cette manière lors du deuxième raffinement du robot.

La figure 7.4 présente quant à elle l'ensemble des modules (au nombre de 12) obtenus par l'application de cette technique lors du troisième raffinement du robot.

Remarque. La numérotation des nouveaux modules est basée sur celle des anciens. Tous les modules s'appellent maintenant *NM* (pour *nouveau module*), suivi du numéro du raffinement, avec en indice le numéro du module qui leur correspondait dans l'autre découpage. Les noms de modules portant un prime correspondent au cas où deux anciens événements étaient issus de l'état abstrait.

7.2.3 Vérification modulaire des deuxième et troisième raffinements du robot avec le nouveau découpage

Avec ce découpage, mieux adapté aux systèmes asynchrones, nous menons de nouveau la vérification modulaire des propriétés du robot pour les deuxième et troisième raffinements.

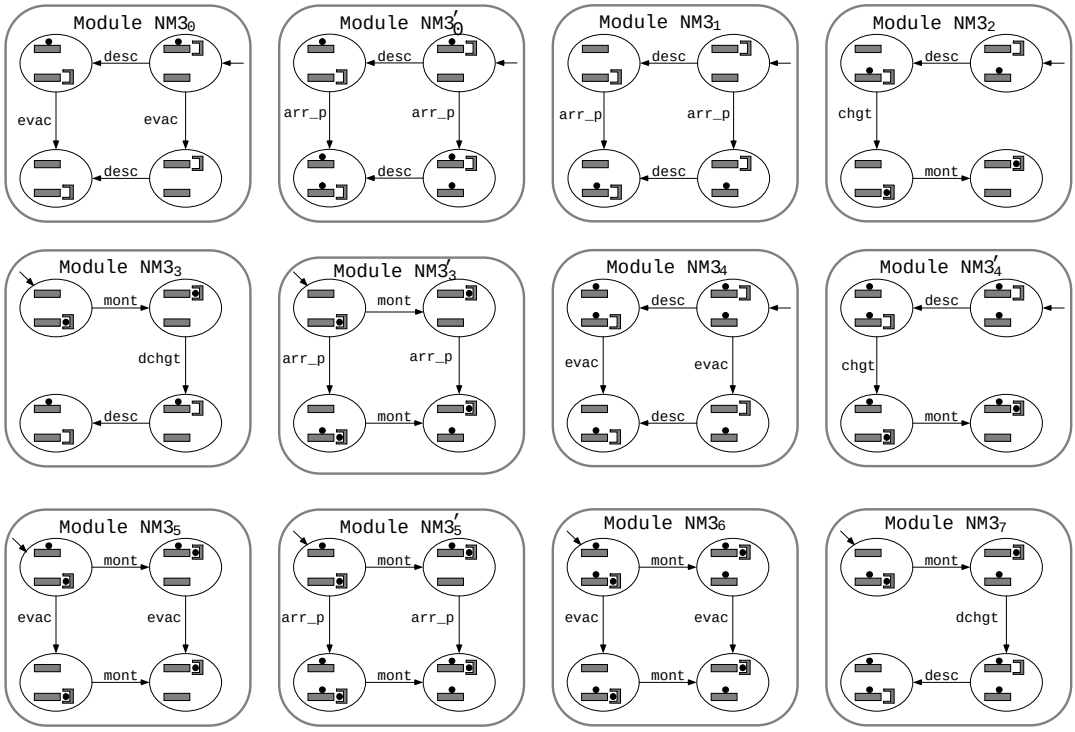

FIG. 7.4 – Un autre découpage modulaire du troisième raffinement du robot

Présentation des résultats

Les résultats de la vérification modulaire des propriétés P6, P7 et P8 sur les « nouveaux » modules issus du deuxième raffinement du robot (appelés $NM2_0$, $NM2_1$, $NM2_2$, $NM2'_2$ et $NM2_3$) sont présentés dans le tableau 7.5.

		Modules				
		$NM2_0$	$NM2_1$	$NM2_2$	$NM2'_2$	$NM2_3$
Propriétés	P6	VRAI	VRAI ($\Box \neg p$)	VRAI	VRAI	VRAI
	P7	VRAI ($\Box \neg p$)	VRAI	VRAI	VRAI	VRAI
	P8	VRAI	VRAI	VRAI	VRAI	VRAI

TAB. 7.5 – Résultats de la vérification modulaire du deuxième raffinement du robot avec le nouveau découpage

Les résultats de la vérification modulaire des propriétés P9 à P14 sur les nouveaux modules issus du troisième raffinement du robot sont présentés dans le tableau 7.6. Ces modules sont appelés $NM3_0$, $NM3'_0$, $NM3_1$, $NM3_2$, $NM3_3$, $NM3'_3$, $NM3_4$, $NM3'_4$, $NM3_5$, $NM3'_5$, $NM3_6$ et $NM3_7$.

Analyse des résultats

On constate dans le tableau 7.5 que le nouveau découpage a atteint ces objectifs en ce qui concerne la vérification des propriétés du deuxième raffinement du robot puisque les trois propriétés exprimées à ce niveau de raffinement ont maintenant pu être vérifiées de manière modulaire, et notamment les propriétés P7 et P8, dont la vérification modulaire avait échoué avec le découpage précédent.

Dans le tableau 7.6, on constate que les propriétés P9 et P10 ont maintenant pu être vérifiées de manière modulaire, ce qui n'était pas le cas auparavant. Par contre, les propriétés P12 et P14 n'ont toujours pas pu être vérifiées de manière modulaire, malgré ce nouveau découpage.

Ceci provient du fait que ces deux propriétés ne sont pas intégralement nouvelles. En effet, considérons l'exemple de la propriété P12, qui dit que la pince en position haute et chargée reste en haut jusqu'à ce qu'inévitablement elle se libère. Son expression PLTL est la suivante :

$$\Box\,((posdt_3 = \texttt{haut} \wedge dt_3 = \texttt{occ}) \Rightarrow posdt_3 = \texttt{haut} \wedge dt_3 = \texttt{occ}\;\mathcal{U}\; dt_3 = \texttt{vid}).$$

Ce qui pose problème dans la vérification modulaire de cette propriété, c'est que l'inévitabilité de la libération de la pince n'est pas nécessairement observée dans les modules. Or, on a déjà établi que la pince occupée se libérera inévitablement au moyen de la propriété P1 au niveau de la spécification abstraite du robot (voir page 42). L'inévitabilité de la libération de la pince est donc un ancien comportement, que l'on n'a pas besoin de revérifier lors du troisième raffinement puisque cette inévitabilité est conservée par le raffinement.

Propriétés	Modules											
	$NM3_0$	$NM3'_0$	$NM3_1$	$NM3_2$	$NM3_3$	$NM3'_3$	$NM3_4$	$NM3'_4$	$NM3_5$	$NM3'_5$	$NM3_6$	$NM3_7$
P9	VRAI	VRAI	VRAI	VRAI	VRAI	VRAI $(\Box\neg p)$	VRAI	VRAI	VRAI $(\Box\neg p)$	VRAI $(\Box\neg p)$	VRAI $(\Box\neg p)$	VRAI
P10	VRAI $(\Box\neg p)$	VRAI $(\Box\neg p)$	VRAI $(\Box\neg p)$	VRAI	VRAI	VRAI	VRAI $(\Box\neg p)$	VRAI	VRAI	VRAI	VRAI	VRAI
P11	VRAI $(\Box\neg p)$	VRAI $(\Box\neg p)$	VRAI $(\Box\neg p)$	VRAI	VRAI	VRAI	VRAI $(\Box\neg p)$	VRAI	VRAI	VRAI	VRAI	VRAI
P12	VRAI $(\Box\neg p)$	VRAI $(\Box\neg p)$	VRAI $(\Box\neg p)$	FAUX	VRAI	FAUX	VRAI $(\Box\neg p)$	FAUX	FAUX	FAUX	FAUX	VRAI
P13	VRAI	VRAI	VRAI	VRAI	VRAI	VRAI $(\Box\neg p)$	VRAI	VRAI	VRAI $(\Box\neg p)$	VRAI $(\Box\neg p)$	VRAI $(\Box\neg p)$	VRAI
P14	FAUX	FAUX	FAUX	VRAI	FAUX	VRAI $(\Box\neg p)$	FAUX	VRAI	VRAI $(\Box\neg p)$	VRAI $(\Box\neg p)$	VRAI $(\Box\neg p)$	FAUX

TAB. 7.6 – Résultats de la vérification modulaire du troisième raffinement du robot avec le nouveau découpage

Il reste néanmoins qu'on veut prouver que la pince ne pourra pas redescendre avant que sa libération ait eu lieu, ce qui est un nouveau comportement. Or ce comportement a pu être prouvé de manière modulaire lors du troisième raffinement puisqu'il correspond à la propriété P11, dont l'expression PLTL est la suivante :

$$\Box\,((posdt_3 = \texttt{haut} \wedge \bigcirc\, posdt_3 = \texttt{bas}) \Rightarrow dt_3 = \texttt{vid}),$$

et dont la signification est que la descente de la pince ne peut se produire que lorsque celle-ci est vide.

Ainsi, la propriété P12 mélange l'expression d'un ancien comportement (l'inévitabilité de la libération de la pince) avec celle d'un nouveau comportement (l'impossibilité de descendre chargée). Il en va de même pour la propriété P14 qui est duale : on sait que la pince vide se chargera inévitablement (propriété P2), et que la montée ne peut se produire que si la pince est occupée (propriété P13).

La vérification modulaire échoue pour ce type de propriétés car elle ne peut observer que les comportements réellement nouveaux. Notons qu'en écrivant la propriété P12 sous une forme plus faible utilisant l'opérateur temporel $\mathcal{W}$ (à moins que) :

$$\textit{(P12')}\ \Box\,((posdt_3 = \texttt{haut} \wedge dt_3 = \texttt{occ}) \Rightarrow posdt_3 = \texttt{haut} \wedge dt_3 = \texttt{occ}\ \mathcal{W}\ dt_3 = \texttt{vid})$$

on s'affranchit de l'obligation de voir la pince se libérer, puisque la propriété P12' se lit : « la pince chargée et en position haute le reste *sauf si* elle se libère ». Le problème est que, écrite sous cette forme, la propriété n'est plus vérifiable modulairement car le schéma $\Box\,(p \Rightarrow q\,\mathcal{W}\,r)$ n'est pas un schéma de propriété vérifiable modulairement.

Ainsi, si l'on souhaite rester dans le cadre des propriétés vérifiables modulairement, il sera à la charge de l'utilisateur de comprendre, face à l'échec de la vérification modulaire des propriétés P12 et P14, que ces propriétés sont en fait redondantes par rapport à celles qui ont déjà été exprimées (et vérifiées), et qu'on n'a donc pas besoin d'essayer de les vérifier.

7.3 Conclusion

Nous avons, au cours de ce chapitre, appliqué la technique de vérification modulaire à chacune des propriétés dynamiques exprimées au cours des spécifications des exemples du BRP et du robot.

Nous avons vu que le découpage modulaire proposé au chapitre 6 permettait de vérifier les propriétés du BRP, mais n'était pas adapté au problème du robot, en expliquant que ce découpage modulaire supposait une synchronisation des événements, ce qui n'est pas le cas dans la spécification du robot, où les événements sont en général asynchrones les uns par rapport aux autres. Nous avons alors proposé un autre découpage permettant de traiter de manière modulaire les systèmes asynchrones, et nous avons montré que cet autre découpage permettait la vérification modulaire de la plupart des propriétés du robot. Ce découpage est moins efficace que le précédent en termes de nombre de modules à vérifier, car le premier

découpage modulaire créait autant de modules qu'il y avait d'états dans le graphe abstrait, tandis que le second crée autant de modules qu'il y avait de transitions dans le graphe abstrait.

Nous avons aussi vu qu'indépendamment du problème du découpage modulaire, la vérification modulaire échoue pour des propriétés qui ne sont pas, au sens strict, des nouvelles propriétés. Ce problème soulève la question suivante, d'ordre méthodologique : le spécifieur doit-il être libre d'exprimer les propriétés dynamiques quand bon lui semble, ou doit-il réfléchir à la signification de celles-ci afin de les exprimer au bon moment lors du processus de raffinement ? La méthode de vérification modulaire suppose que le spécifieur est un expert capable, lors de l'échec de la vérification d'une propriété, de décider si cette propriété est vraiment nouvelle ou pas.

Précisons toutefois qu'une thèse en cours[1] au Laboratoire d'Informatique de l'Université de Franche-Comté (LIFC) étudie, entre autres, le problème des propriétés qui ne sont que partiellement nouvelles. La technique développée dans ce travail porte le nom de *reformulation* [BDJK00, BDJK01] : à chaque niveau du raffinement, en plus de redéfinir les anciens événements, on peut aussi enrichir les anciennes propriétés en les *reformulant*. Une « partie » d'une propriété reformulée ayant déjà été prouvée au niveau de spécification antérieur, on ne vérifie (par une technique de preuve sur les nouveaux événements) que la partie de la propriété correspondant à un nouveau comportement. L'utilisation de cette technique permettrait de vérifier les propriétés P17 (du BRP) et P12 et P14 (du robot), dont la vérification modulaire a échoué, mais qui sont des reformulations de propriétés antérieures.

[1] Thèse menée par Christophe Darlot sous la direction de Françoise Bellegarde et de Jacques Julliand.

Conclusion et perspectives

1 Synthèse et bilan des travaux présentés

1.1 Synthèse

Le travail présenté dans cette thèse prend pour cadre la spécification de systèmes réactifs sous forme de systèmes d'événements B. La méthode B permet de spécifier les systèmes au travers de raffinements successifs d'un modèle initial, ce qui permet de n'introduire les détails de fonctionnement du système que petit à petit, avec pour objectif de mieux maîtriser le processus de développement des logiciels.

La spécification des comportements dynamiques souhaités permet de s'assurer que le système développé réalise bien ce qu'on attend de lui. Cependant, l'expression de tels comportements en B nécessite l'écriture de variants et d'invariants de boucle qui ne sont pas directement liés au cahier des charges. D'autre part, la vérification par preuve de ces contraintes dynamiques est rarement obtenue de manière automatique.

Nous avons voulu libérer l'utilisateur de la spécification de ces variants et de ces invariants de boucle en proposant de décrire les comportements dynamiques par des formules de la PLTL. De plus, l'utilisation du model-checking pour vérifier ces propriétés à l'avantage d'être entièrement automatique.

Cependant, outre le fait que le model-checking ne peut adresser que des problèmes à nombre d'états finis, une limite connue à l'utilisation de cette technique est l'explosion combinatoire du nombre d'états.

Nous avons proposé de répondre à cette limite en ne menant jamais la vérification par model-checking sur l'intégralité du modèle du système, mais en découpant celui-ci en un ensemble de modules de plus petite taille et en menant la vérification de manière séparée sur chacun des modules.

Nous avons défini une classe de propriétés PLTL pour lesquelles une telle vérification est possible. Ces propriétés sont dites *vérifiables modulairement*.

Pour mettre en œuvre la vérification modulaire, il faut disposer d'un découpage modulaire tel que les propriétés à vérifier soient vraies sur chacun des modules. Pour obtenir un découpage se rapprochant de cet idéal, nous avons proposé de guider le découpage par le raffinement

B, en constatant que les nouvelles propriétés, celles que l'on cherche à vérifier par model-checking modulaire, sont observables grâce à l'introduction, lors du processus de raffinement, de nouveaux événements. Comme les modules obtenus grâce au raffinement B contiennent les suites d'occurrences de ces nouveaux événements, ce découpage laisse supposer que les nouveaux comportements seront bien observables à l'intérieur des modules.

Nous avons mené la vérification modulaire des propriétés de deux exemples, et nous avons constaté que si la méthode marchait bien dans un cas, elle échouait sur un certain nombre de propriétés dans l'autre cas, bien que ces propriétés soient vraies globalement.

En constatant que cet échec provenait de l'asynchronisme des événements permettant de faire évoluer le système, nous avons proposé un autre découpage, moins efficace que le premier en termes de nombre d'états et de transitions à explorer, mais qui permet de tenir compte de l'asynchronisme des événements.

Nous avons également proposé d'augmenter l'efficacité de la vérification modulaire en utilisant une technique combinant preuve et model-checking, et qui consiste à éviter la vérification par model-checking de certains modules lorsque ceux-ci peuvent être vérifiés par preuve de manière automatique.

1.2 Bilan

Notre travail a permis de montrer qu'il était possible, pour toute une classe de propriétés de la PLTL incluant des propriétés de sûreté et de vivacité, de mener une vérification par model-checking en découpant un modèle initial de grande taille en un ensemble de modules de petite taille, et en faisant la vérification des propriétés de manière séparée et indépendante sur chaque module. Ceci permet de répondre au problème de l'explosion combinatoire puisque le modèle global n'aura jamais à être présent intégralement en mémoire.

Pour que ce résultat théorique puisse être utilisé en pratique, il faut savoir produire un découpage modulaire grâce auquel les propriétés à vérifier, lorsqu'elles sont vérifiées par le système, sont aussi vraies sur chacun des modules. En se basant sur le raffinement, et pour des propriétés qui sont nouvelles par rapport à un niveau de raffinement donné, on produit un découpage modulaire qui est bien adapté aux systèmes dont les événements sont synchronisés les uns par rapport aux autres. De plus, comme il est basé sur une partition du modèle initial, ce découpage permet de limiter au maximum la redondance des informations d'un module à l'autre.

Ce découpage n'est cependant pas adapté à la vérification de systèmes dont les événements sont asynchrones, et le découpage que nous avons proposé pour la vérification modulaire de tels systèmes est moins efficace en termes de redondances des informations d'un module à l'autre. Il reste néanmoins, comme le premier, un découpage sémantique du système à vérifier.

Enfin, la mise en œuvre de la vérification modulaire suppose que le spécifieur a un niveau d'expertise suffisant pour savoir à quel moment, lors du processus de raffinement, il doit exprimer les propriétés de telle sorte qu'elles soient réellement des nouvelles propriétés.

2 Travaux connexes à la vérification modulaire

Il existe d'autres approches mettant en œuvre une vérification de propriétés de logique temporelle par model-checking modulaire.

Notamment, de nombreux travaux s'intéressent à la vérification séparée des composants d'un système composé de plusieurs processus évoluant en parallèle. Ces méthodes sont regroupées sous le nom général de *vérification compositionnelle* [CLM89]. La notion de vérification compositionnelle est orthogonale à celle de vérification modulaire que nous avons présentée.

Dans ces méthodes, la stratégie adoptée est la vérification individuelle de chacun des composants du système, et la mise en commun de tous ces résultats individuels permet de s'assurer que le système satisfait globalement les propriétés spécifiées. Ceci repose sur le fait que la compositionalité d'un opérateur de mise en parallèle garantit qu'une propriété d'un processus est préservée sur la totalité du système [KL93, Lam94, AdAHM99].

Généralement, ces méthodes permettent de vérifier des propriétés de sûreté étant donné qu'un composant isolé de son environnement est une abstraction du système parallèle qui préserve uniquement les propriétés de sûreté. Par exemple, il a été prouvé dans [KL93] la correction de la règle suivante :

$$\frac{C_1 \vdash I_2 \Rightarrow I_1, C_2 \vdash I_1 \Rightarrow I_2}{C_1 \parallel C_2 \vdash I_1 \wedge I_2}$$

qui indique que si le composant C_1 satisfait I_1 sous l'hypothèse d'environnement I_2 et que C_2 satisfait I_2 sous l'hypothèse I_1, alors $C_1 \parallel C_2$ satisfait $I_1 \wedge I_2$.

Ces méthodes reposent sur le principe appelé *assume guarantee paradigm* [Pnu85]. Ce principe dit qu'une formule de la forme $\langle\varphi\rangle P \langle\psi\rangle$ (où φ et ψ sont des formules temporelles et P est un processus) est vraie si on garantit que P satisfait ψ dans toute exécution où l'environnement satisfait φ.

Quelques unes, et notamment [CGK97], effectuent également une vérification compositionnelle de propriétés de vivacité.

Dans notre approche, un module n'est pas un composant d'un système parallèle. Pour nous, un module est obtenu par partition du modèle du système. Ainsi, du point de vue de la complexité, notre méthode est additive tandis que la compositionalité est multiplicative, c'est à dire que si la complexité de la vérification d'un système de transitions d'espace d'états S et d'espace de transitions T est on $O(|S| + |T|)$, la complexité moyenne pour un module de la vérification modulaire d'un système composé de n modules est en $O(\frac{|S|+|T|}{n})$, et la vérification compositionnelle est en $O(\sqrt[n]{|S| + |T|})$.

La modularité telle que nous l'avons présentée peut être combinée avec la compositionalité. Dans [BJK00], il a été montré que, dans certaines conditions, la composition parallèle est compatible avec le raffinement, c'est à dire que si C_1 est raffiné par C_2, alors $C_1 \parallel C_3$ est raffiné par $C_2 \parallel C_3$.

Une méthode de vérification modulaire de propriétés CTL a été proposée par K. LASTER et O. GRUMBERG dans [LG98]. Cette méthode repose sur un découpage syntaxique d'une spécification qui utilise des structures de contrôle telles que `While`, `If`, ..., ce qui rend nécessaire la circulation des informations de marquage entre les modules, par le biais de « fonctions de supposition » (*assumption functions*). En reposant sur un découpage sémantique, notre méthode permet de vérifier les propriétés de chaque module de manière réellement indépendante. Par contre, la méthode de LASTER et GRUMBERG est définie pour toute la CTL, alors que notre méthode n'est définie que pour une classe de propriétés PLTL.

3 Perspectives

Le travail présenté dans cette thèse appelle des travaux futurs qui permettront de compléter certains des résultats que nous avons obtenus, ou qui donneront des réponses à certaines questions laissées ouvertes.

3.1 Une condition de vérifiabilité modulaire exploitant le découpage par raffinement

Telle que nous l'avons définie dans la section 5.4 du chapitre 5, la condition suffisante de vérifiabilité modulaire est valide quel que soit le découpage modulaire adopté, à condition que celui-ci repose au moins sur une partition des transitions du modèle global. Le découpage modulaire guidé par le raffinement B est un découpage valide pour cette condition suffisante, mais nous n'avons pas pris en compte la spécificité de ce découpage par rapport à un autre. En tenant compte de ses particularités, on pourrait élargir la classe des propriétés vérifiables modulairement.

En effet, lorsque le découpage est tel que nous l'avons défini dans la section 6.2.3 du chapitre 6, on sait que la sortie d'un module ne peut se produire que par le franchissement d'une transition correspondant à l'occurrence d'un ancien événement.

Pour la condition suffisante, l'automate de Büchi codant la négation d'une propriété vérifiable modulairement n'admet pas de boucle sur un état intermédiaire entre l'état initial et un état d'acceptation. Ainsi, le schéma de propriété PLTL $\Box\,(p \Rightarrow q\,\mathcal{W}\,r)$ ne tombe pas dans le cas de la condition suffisante de vérifiabilité modulaire, à cause de la boucle sur l'état intermédiaire portant le numéro 1 dans l'automate de Büchi codant sa négation (voir la figure 7.5). Si la coupure du chemin d'exécution provoquée par la sortie du module se produit alors qu'on est en train de « boucler » sur l'état 1, alors on n'a pas reconnu que ce chemin constituait une violation de la propriété $\Box\,(p \Rightarrow q\,\mathcal{W}\,r)$.

Cependant, si l'on réussit à prouver que le déclenchement d'une transition appliquant un ancien événement est incompatible avec la condition booléenne $\neg\, r \wedge q$, alors on est sûr que la coupure ne pourra pas se produire pendant une boucle sur l'état 1. Cette information peut être exploitée pour établir que si, dans le module, on a quitté l'état initial de l'automate

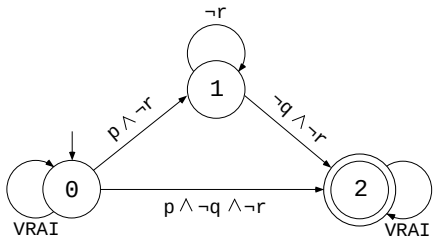

FIG. 7.5 – Un automate de Büchi codant $\neg \Box (p \Rightarrow q \mathcal{W} r)$

de Büchi, alors en sortie de module on aura reconnu une violation de la propriété puisqu'on quittera le module sur un état d'acceptation.

La propriété $\Box (p \Rightarrow q \mathcal{W} r)$ pourrait donc être vérifiable modulairement dans le contexte précis d'un système et de son raffinement.

Plus généralement, on peut se poser la question de savoir si un découpage particulier permet d'établir une condition de vérifiabilité modulaire dépendant du contexte de ce découpage.

3.2 Méthodologie du raffinement

Notre découpage modulaire étant basé sur le processus de raffinement B, et plusieurs raffinements étant possibles pour spécifier un système, on peut se demander si le choix d'un raffinement plutôt qu'un autre a une influence sur le succès de la méthode de vérification modulaire. En d'autres termes, l'ordre et la manière d'introduire les détails de la spécification permettent-t-ils d'accroître le taux de succès de la méthode ?

Par exemple, dans le cas de systèmes partiellement asynchrones, on peut choisir de spécifier en une seule étape tous les comportements asynchrones, de manière à ce que les raffinements suivants permettent bien l'observation des nouveaux comportements sur les modules.

Ceci est envisageable car au début du processus de raffinement, le nombre d'états du modèle est en général suffisamment faible pour qu'on puisse appliquer le model-checking directement (sans découpage modulaire) pour vérifier les propriétés.

Cette idée aurait pour conséquence d'imposer au spécifieur une *méthodologie du raffinement* basée sur les propriétés que l'on cherche à vérifier de manière modulaire.

3.3 Outils implantant la méthode

Une estimation réaliste du taux de succès de la méthode ainsi que de son impact sur la maîtrise de l'explosion combinatoire passe par le test de la méthode sur de nombreux problèmes concrets. Il est donc nécessaire de disposer d'un outil implantant la méthode pour pouvoir la valider en pratique.

L'une des composantes de cet outil devra être en mesure de décider si une propriété est vérifiable modulairement. Cette procédure de décision pourra être basée sur l'analyse des cycles de l'automate de Büchi codant la négation de la propriété, afin de prouver que cet automate appartient bien à la classe $\mathcal{C}_{mod}$ que nous avons définie au chapitre 5 dans la section 5.4.1.

D'autre part, un outil implantant la méthode devra être capable de produire les modules directement à partir des spécifications B, pour ne pas avoir à construire l'intégralité du modèle avant de le découper. La question de cet algorithme reste ouverte.

Une fois les modules construits, on devra utiliser un model-checker capable de traiter la PLTL (c'est le cas de SPIN par exemple) pour vérifier les propriétés sur les modules.

3.4 Extension de la méthode à d'autres logiques temporelles

La méthode de vérification modulaire que nous avons présentée utilise la logique temporelle PLTL pour formaliser les propriétés à vérifier. On peut se demander si la notion de propriété vérifiable modulairement est applicable à des propriétés modélisées dans une autre logique temporelle, comme CTL.

On sait déjà que certaines propriétés de la CTL sont vérifiables modulairement puisque les 3 schémas de contrainte dynamique de ABRIAL et MUSSAT appartiennent à la partie commune des logiques PLTL et CTL (voir la figure 3.31 page 77). Ces 3 schémas, exprimés par $\Box\,(p \Rightarrow \bigcirc\, q)$, $\Box\,(p \Rightarrow \Diamond\, q)$ et $\Box\,(p \Rightarrow p\,\mathcal{U}\,q)$ en PLTL s'expriment respectivement en CTL par $\mathsf{A\,G}\,(p \Rightarrow \mathsf{A\,X}\,q)$, $\mathsf{A\,G}\,(p \Rightarrow \mathsf{A\,F}\,q)$ et $\mathsf{A\,G}\,(p \Rightarrow \mathsf{A}\,(p\,\mathcal{U}\,q))$.

L'une des difficultés à établir le même type de résultat de vérifiabilité modulaire pour la CTL provient du fait que notre démonstration de la vérifiabilité modulaire de propriétés PLTL repose sur les automates de Büchi qu'il est possible de leur associer. Ceci n'est pas applicable dans le cas de la CTL qui est une logique arborescente, non modélisable par des automates de Büchi reconnaissant des langages ω-réguliers.

La démonstration de la vérifiabilité modulaire de la classe de propriétés PLTL que nous avons exhibée aurait probablement pu être obtenue sans avoir recours aux automates de Büchi, mais ceux-ci constituent un outil grâce auquel on peut facilement caractériser les propriétés de cette classe.

3.5 Caractérisation complète des propriétés vérifiables modulairement

Pour terminer, rappelons que la condition de vérifiabilité modulaire que nous avons donnée pour une propriété PLTL est une condition suffisante. Nous ne savons pas si elle est également nécessaire.

Il nous semblerait particulièrement intéressant de pouvoir *caractériser* les propriétés vérifiables modulairement, en exhibant une condition qui soit à la fois nécessaire et suffisante.

Bibliographie

[Abr96a] Jean-Raymond Abrial. *The B Book : Assigning Programs to Meanings*. Cambridge University Press, 1996.

[Abr96b] Jean-Raymond Abrial. Extending B without changing it (for developing distributed systems). In Henri Habrias, éditeur, *Proc. of the 1-st Conference on the B method*, Putting into Practice Methods and Tools for Information System Design, pages 169–190, Nantes, France, novembre 1996. IRIN Institut de recherche en informatique de Nantes.

[Abr97] Jean-Raymond Abrial. Constructions d'automatismes industriels avec B. In *Congrès AFADL*, Toulouse, France, mai 1997. ONERA-CERT.

[AdAHM99] R. Alur, L. de Alfaro, T. A. Henzinger, et F. Y. C. Mang. Automating modular verification. In *Proc. of the 10-th Conf. on Concurrency Theory*, volume 1664 de *LNCS*, pages 82–97. Springer-Verlag, juillet 1999.

[AM97] J.-R. Abrial et L. Mussat. Specification and design of a transmission protocol by successive refinements using B. In *Ecole d'été de Marketoberdorf*, LNCS, 1997.

[AM98] J.-R. Abrial et L. Mussat. Introducing dynamic constraints in B. In *Proc. of the 2-nd Conference on the B method*, volume 1393 de *LNCS*, pages 83–128. Springer-Verlag, avril 1998.

[Büc62] J. R. Büchi. On a decision method in restricted second-order arithmetic. In *Proc. of the Int. Congress on Logic, Methodology and Philosophy of Science*, pages 1–12. Stanford Univ. Press, 1962.

[BC00] D. Bert et F. Cave. Construction of finite labelled transition systems from B abstract systems. In *Proc. of the 2-nd Int. Conf. on Integrated Formal Methods (IFM 2000)*, volume 1945 de *LNCS*, pages 235–254, Dagstuhl, Germany, novembre 2000. Springer-Verlag.

[BCMD90] J. R. Burch, E. M. Clarke, K. L. McMillan, et L. J. Dill, D. L.and Hwang. Symbolic model checking : 10^{20} states and beyond. In *Proc. of the 5-th Annual IEEE Symposium on Logic in Computer Science*, pages 428–439, Philadelphia, Pensylvania, USA, juin 1990. IEEE Computer Society Press.

[BDJK00] F. Bellegarde, C. Darlot, J. Julliand, et O. Kouchnarenko. Reformulate dynamic properties during B refinement and forget variants and loop invariants. In *Proc. of the Int. Conf. ZB'2000*, volume 1878 de *LNCS*, août 2000.

[BDJK01] F. Bellegarde, C. Darlot, J. Julliand, et O. Kouchnarenko. Reformulation : a way to combine dynamic properties and B refinement. In *Proc. of the Int. Symp. Formal Methods Europe (FME 2001)*, volume 2021 de *LNCS*, pages 2–19, Berlin, Germany, mars 2001.

[BJK00] F. Bellegarde, J. Julliand, et O. Kouchnarenko. Ready-simulation is not ready to express a modular refinement relation. In *Proc. of the Int. Conf. on Fondamental Aspects of Software Engineering, (FASE'2000)*, volume 1783 de *LNCS*, pages 266–283. Springer-Verlag, avril 2000.

[But99] Mickael Butler. CSP2B : a practical approach to combining CSP and B. In *Proc. of the World Congress on Formal Methods (FM'99)*, volume 1708 de *LNCS*, pages 490–508. Springer-Verlag, 1999.

[CC77] P. Cousot et R. Cousot. Abstract Interpretation : A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *4-th ACM Symp. on Principles of Programming Languages (POPL'77)*, pages 238–252, Los Angeles, California, USA, 1977. ACM Press.

[CE81] E. M. Clarke et E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on Logics of Programs*, volume 131 de *LNCS*, pages 52–71, New York, NY, USA, mai 1981. Springer Verlag.

[CES86] E. M. Clarke, E. A. Emerson, et P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specification. In *ACM transactions on Programming Languages and Systems*, volume 2, pages 244–263, 1986.

[CGK97] S.C. Cheung, D. Giannakopoulou, et J. Kramer. Verification of liveness properties using compositional reachability analysis. In *ESEC/SIGSOFT FSE, 6-th European Software Engineering Conference*, volume 1301 de *LNCS*, pages 227–243. Springer-Verlag, novembre 1997.

[CGP99] E. M. Jr Clarke, O Grumberg, et D. A. Peled. *Model Checking*. MIT Press, 1999.

[CLM89] E. M. Clarke, D. E. Long, et K. L. McMillan. Compositional Model Checking. In *Proc. of the 4-th Annual Symposium on Logic in Computer Science*, pages 353–362, Pacific Grove, California, USA, 1989. IEEE Computer Society Press.

[Cou99] Jean-Michel Couvreur. On-the-fly verification of linear temporal logic. In *Proc. of FM'99*, volume 1708 de *LNCS*, pages 253–271, Toulouse, France, septembre 1999. Springer-Verlag.

[CWB94] Cuéllar, I. Wildgruber, et D. Barnard. Combining the design of industrial systems with effective verification techniques. In *Formal Methods Europe, (FME'94)*, volume 873 de *LNCS*, pages 639–658. Springer-Verlag, 1994.

[DG97] D. Dams et R Gerth. The bounded retransmission protocol revisited. In *Proc. of the 2-nd Int. Workshop on Verification of Infinite State Systems (Infinity'97)*, volume 9 de *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 1997. Extended abstract.

[dKRT96] P.R. d'Argenio, J.-P. Katoen, T. Ruys, et G.J. Tretmans. Modeling and verifying a bounded retransmission protocol. In *Proc. of the 1-st Workshop on Applied*

Formal Methods in System Design, pages 114–128. University of Maribor Press, 1996.

[EH82] E. A. Emerson et J. Y. Halpern. Decision procedures and expressiveness in the temporal logic of branching time. In *14-th ACM symp. on Theory of Computing (STOC'82)*, pages 169–180, San Fransisco, USA, mai 1982.

[EH00] K. Etessami et G. J. Holzmann. Optimizing büchi automata. In *Proc. of the 11-th int. conf. on Concurrency Theory (CONCUR'2000)*, volume 1877 de *LNCS*, pages 153–167, University Park, PA, USA, août 2000. Springer-Verlag.

[GO01] P. Gastin et D. Oddoux. Fast LTL to Büchi Automata Translation. In G. Berry, H. Comon, et A. Finkel, éditeurs, *Proc. of the 13-th Conf. on Computer Aided Verification (CAV'01)*, volume 2102 de *LNCS*, pages 53–65. Springer-Verlag, 2001.

[God90] Patrice Godefroid. Using partial orders to improve automatic verification methods. In *Proc. of the 2-nd workshop on Computer Aided Verification (CAV'90)*, volume 531 de *LNCS*, pages 176–185, Rutgers, juin 1990.

[God96] Patrice Godefroid. *Partial-order methods for the verification of concurrent systems : an approach to the state-explosion problem*, volume 1032 de *LNCS*. Springer-Verlag Inc., New York, NY, USA, 1996.

[GPVW95] R. Gerth, D. Peled, M. Y. Vardi, et P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Protocol Specification Testing and Verification (PSTV'95)*, pages 3–18, Warsaw, Poland, juin 1995. Chapman & Hall.

[GV96] J.F. Groote et J.C. Van de Pol. A bounded retransmission protocol for large data packets : a case study in computer checked verification. In *Proceedings of AMAST'96*, volume 1101 de *LNCS*, pages 536–550, Munich, Germany, 1996. Springer-Verlag.

[GW91] P. Godefroid et P. Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. In *Proc. of the 3-rd workshop on Computer Aided Verification (CAV'91)*, volume 575 de *LNCS*, pages 332–342, Aalborg, juillet 1991. Springer-Verlag.

[HGP92] G. J. Holzmann, P. Godefroid, et D. Pirottin. Coverage preserving reduction strategies for reachability analysis. In *Proc. of the 12-th Int. Symp. on Protocol Specification, Testing and Verification*, Lake Buena Vista, Florida, USA, juin 1992.

[Hol91] Gerard J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, Englewood Cliffs, New Jersey, USA, 1991.

[Hol97a] Gerard J. Holzmann. The model checker SPIN. *IEEE Trans. on Software Engineering*, 23(5) :279–295, mai 1997. Special issue on Formal Methods in Software Practice.

[Hol97b] Gerard J. Holzmann. State compression in *spin*. In *3-rd SPIN Workshop*, Twente University, avril 1997.

[HP94] G. J. Holzmann et D. Peled. An improvement in formal verification. In *Proc. of Formal Description Techniques, FORTE'94*, pages 197–211, Berne, Switzerland, octobre 1994. Chapman & Hall.

[HP95] G. J. Holzmann et D. Peled. Partial order reduction of the state space. In *1-st SPIN Workshop*, Montréal, Quebec, 1995. Position paper.

[HS96] K. Havelund et N. Shankar. Experiments in theorem proving and model checking for protocol verification. In *Proc. of FME'96*, volume 1051 de *LNCS*, Oxford, UK, 1996. Springer-Verlag.

[JMM99] J. Julliand, P.-A. Masson, et H. Mountassir. Modular verification of dynamic properties for reactive systems. In *Proc. of the 1-st Int. Workshop on Integrated Formal Methods (IFM'99)*, pages 89–108, York, UK, juin 1999. Springer-Verlag.

[JMM01] J. Julliand, P.-A. Masson, et H. Mountassir. Vérification par model-checking modulaire des propriétés dynamiques introduites en B. *Technique et Science Informatiques*, 20(7) :927–957, septembre 2001.

[KL93] R. P. Kurshan et L. Lamport. Verification of a multiplier : 64-bits and beyond. In *Proc. of the 5-th Int. Conf. on Computer Aided Verification (CAV'93)*, volume 697 de *LNCS*, pages 166–179, Elounda, Greece, 1993. Springer-Verlag.

[Kle56] S. C. Kleene. Representation of events in nerve nets and finite automata. In C. E. Shannon et J. McCarthy, éditeurs, *Automata Studies*, pages 3–41. Princeton University Press, Princeton, New Jersey, USA, 1956.

[KP88] S. Katz et D. A. Peled. An efficient verification method for parallel and distributed programs. In *Workshop on Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, volume 354 de *LNCS*, pages 489–507. Springer-Verlag, 1988.

[Lam80] Leslie Lamport. "sometime" is sometime "not never". In *ACM symposium on Principles of Programming Languages*, pages 174–185, 1980.

[Lam94] Leslie Lamport. A temporal logic of actions. *ACM Transactions of on Programming Languages and Sytems (TOPLAS)*, 16(3) :872–923, mai 1994.

[LG98] K. Laster et O. Grumberg. Modular model-checking of software. In *proc. of the Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'98*, volume 1384 de *LNCS*, pages 20–35, Lisbonne, Portugal, mars 1998. Springer-Verlag.

[LGS+95] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, et S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6 :1–35, 1995.

[LP85] O. Lichtenstein et A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proc. of the 12-th ACM Symp. on Principles of Programming Languages (POPL'85)*, pages 97–107, New Orleans, USA, 1985.

[Maz86] Antoni Mazurkiewicz. Trace theory. In *Petri Nets : Applications and Relationships to Other Models of Concurrency, Advances in Petri Nets 1986, Part*

II, Proceedings of an Advanced Course, volume 255 de *LNCS*, pages 279–324. Springer-Verlag, 1986.

[MBJM00] H. Mountassir, F. Bellegarde, J. Julliand, et P.-A. Masson. Coopération entre preuve et model-checking pour vérifier des propriétés LTL. In *actes du congrès AFADL'2000*, Grenoble, France, décembre 2000.

[MMJ00] P.-A. Masson, H. Mountassir, et J. Julliand. Modular verification for a class of PLTL properties. In *Proc. of the 2-nd Int. Conf. on Integrated Formal Methods (IFM 2000)*, volume 1945 de *LNCS*, pages 398–419, Dagstuhl Castle, Germany, novembre 2000. Springer-Verlag.

[MSS88] D. E. Muller, A. Saoudi, et P. E. Schupp. Weak alternating automata give a simple explanation of why most temporal and dynamic logics are decidable in exponential time. In *Proc. of the 3-rd IEEE Symposium on Logic in Computer Science*, pages 422–427, Edinburgh, Scotland, UK, juillet 1988.

[Ove81] W. T. Overman. *Verification of concurrent systems : function and timing.* Thèse de Doctorat, University of California, Los Angeles, USA, 1981.

[Par00] Benoît Parreaux. *Vérification de systèmes d'événements B par model-checking PLTL : Contribution à la réduction de l'explosion combinatoire en utilisant de la résolution de containtes ensemblistes.* Thèse de Doctorat, Université de Franche-Comté, LIFC, Besançon, France, décembre 2000.

[Pel94] D. A. Peled. Combining partial order reduction with on-the-fly model-checking. In *Proc. of the 6-th Int. Conf. on Computer Aided Verification (CAV'94)*, volume 818 de *LNCS*, pages 377–390, Stanford, California, USA, juin 1994. Springer-Verlag.

[Pnu81] Amir Pnueli. The temporal semantics of concurrent programs. *Theoretical Computer Science*, 13 :45–60, 1981.

[Pnu85] Amir Pnueli. In transition from global to modular temporal reasoning about programs. *Logics and Models of Concurrent Systems*, pages 123–144, 1985.

[Py00] Laurent Py. *Evaluation de spécifications formelles B en programmation logique avec contraintes ensemblistes : Application à l'animation et au model-checking.* Thèse de Doctorat, Université de Franche-Comté, LIFC, Besançon, France, janvier 2000.

[QS82] J.-P. Queille et J. Sifakis. Speicification and verification of concurrent systems in CESAR. In *Proc. of the Int. Symp. on Programming*, volume 137 de *LNCS*, pages 337–351, Turin, Italy, avril 1982. Springer-Verlag.

[RJB98] J. Rumbaugh, I. Jacobson, et G. Booch. *The Unified Modeling Language - Reference Manual.* Addisson Wesley, 1998.

[SBB+99] P. Schnoebelen, B. Bérard, M. Bidoit, F. Laroussinie, et A. Petit. *Vérification de logiciels : Techniques et outils du model-checking.* Vuibert Informatique, 1999.

[SPH84] R. Sherman, A. Pnueli, et D. Harel. Is the interesting part of process logic unintersting : a translation from PL to PDL. *SIAM Journal on Computing*, 13(4) :825–839, 1984.

[Spi88] J. M. Spivey. *Understanding Z : A Specification Language and its Formal Semantics*, volume 3 de *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1988.

[Tar83] R. E. Tarjan. *Data structures and network algorithms*. SIAM, Philadelphia, USA, 1983.

[Val88] Anti Valmari. Error detection by reduced reachability graph generation. In *Proc. of Application and Theory of Petri Nets*, juin 1988.

[Val91] Anti Valmari. Stubborn sets for reduced state space generation. In *Advances in Petri Nets 1990*, volume 483 de *LNCS*, pages 491–515. Springer-Verlag, 1991.

[Var94] Moshe Y. Vardi. Nontraditional applications of automata theory. In M. Hagiya et J. C. Mitchell, éditeurs, *Theoretical Aspects of Computer Software*, pages 575–597. Springer Verlag, Berlin, Heidelberg, Germany, 1994.

[Var96] Moshe Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Logics for concurrency : structure versus automata*, volume 1043 de *LNCS*, pages 238–266. Springer-Verlag, 1996.

[VW86] M. Y. Vardi et P. Wolper. An automata theoretic approach to automatic program verification. In *Proc. of the 1-st IEEE Symp. on Logic in Computer Science (LICS'86)*, pages 332–344, Cambridge, USA, juin 1986.

[VW94] M. Y. Vardi et P. Wolper. Reasonning about infinite computations. *Information and Computation*, 115(1) :1–37, 1994.

[WG93] P. Wolper et P. Godefroid. Partial-order methods for temporal verification. In *Proc. of the Int. Conf. on Concurrency Theory (CONCUR'93)*, volume 715 de *LNCS*, pages 233–246, Hildesheim, août 1993. Springer-Verlag.

[WVS83] P. Wolper, M. Y. Vardi, et A. P. Sistla. Reasoning about infinite computation paths. In *Proc. of the 24-th IEEE Symp. on Foundation of Computer Science*, pages 185–194, Tuscan, 1983.

Résumé. Le travail présenté dans cette thèse prend pour cadre la spécification de systèmes réactifs sous forme d'événements B. À une spécification B événementielle, on intègre l'expression de propriétés dynamiques décrites en termes de formules de logique temporelle linéaire propositionnelle (PLTL). La vérification de ces propriétés par une technique de preuve n'est pas automatisable, aussi nous proposons d'effectuer leur vérification par model- checking.

Afin de combattre le problème de l'explosion combinatoire lié à l'utilisation du model-checking, nous proposons de découper par partition le graphe d'accessibilité issu de la spécification en un ensemble de modules de petite taille, et de mener la vérification sur chacun des modules de manière séparée. Cette méthode est appelée *vérification modulaire*. Nous voulons être en mesure, à partir de la vérification séparée d'une propriété sur chacun des modules, de décider si la propriété est vérifiée globalement.

Toutes les propriétés ne se prêtent pas à une telle vérification car certaines d'entre elles peuvent être fausses globalement bien que vraies sur chacun des modules. Nous définissons alors les propriétés qui se prêtent à une telle vérification de la manière suivante (ces propriétés sont dites vérifiables modulairement) : une propriété est vérifiable modulairement si et seulement si le fait qu'elle est vraie sur chaque module implique qu'elle est vraie globalement.

Il faut noter également que tous les découpages n'auront pas la même efficacité relativement à la vérification des propriétés. En effet, la propriété a besoin d'être vraie sur chaque module pour que nous puissions conclure. Or, un découpage aléatoire aurait de fortes chances de voir échouer la vérification d'une propriété dans un ou plusieurs modules, du fait que certains chemins qui rendent la propriété vraie pourraient être coupés. Le problème est donc de produire un découpage modulaire en accord avec les propriétés que l'on cherche à vérifier.

En résumé, nous tentons de répondre aux questions suivantes :

- Quelles sont les propriétés vérifiables modulairement ?
- Comment produire un « bon » découpage modulaire ?

En réponse à la première question, nous prouvons formellement que les propriétés PLTL issues des 3 schémas de contraintes dynamiques introduits par J.-R. Abrial et L. Mussat sont vérifiables de manière modulaire. Nous prouvons également que tout une classe de propriétés PLTL, incluant les 3 schémas précités, est vérifiable de manière modulaire. Nous exhibons une condition suffisante de vérifiabilité modulaire d'une propriété. Cette condition repose sur l'analyse syntaxique de l'automate de Büchi qui code la négation d'une propriété PLTL. Elle est donc facilement automatisable.

Afin de répondre à la deuxième question, nous proposons de guider la décomposition modulaire par le raffinement B, ce qui offre l'avantage de produire un découpage sémantique du graphe d'accessibilité à chaque niveau du raffinement. Nous faisons alors la distinction entre les nouvelles propriétés, introduites à un niveau donné de raffinement, et les anciennes propriétés, établies aux niveaux précédents de raffinement et conservées par celui-ci. La vérification modulaire porte sur les nouvelles propriétés.

Mots-clés. Vérification formelle, model-checking, systèmes d'événements B, logique temporelle, automates de Büchi, vérification modulaire.