REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE

وزارة التعـليـم العـالــي و البـحـث العـلـمي
MINISTERE DE L'ENSEIGNEMENT SUPERIEUR ET DE LA RECHERCHE SCIENTIFIQUE

جـامعـة عمّــار ثـلـيــجــي بالأغـــواط
UNIVERSITE AMAR TELIDJI LAGHOUAT

كـلـيـــة العـلـــوم
FACULTE DES SCIENCES

قســـم الرياضيات و الإعلام الآلي
DEPARTEMENT DE MATHEMATIQUE ET INFORMATIQUE

Mémoire de MASTER

Domaine : Mathématique informatique
Filière : Informatique
Option : Réseaux, Systèmes et Applications Reparties

Par :
Naouri Nadia

THEME

La validation formelle de protocole de communication

Soutenu publiquement devant le jury composé de :

Président : Ben douma Taher
Examinateur : Belabassi Amel
Examinateur : Kachna Lakhdar
Encadreur : M.Djoud

Année Universitaire 2011 - 2012

Dédicace

A mes parents.

A ma très chère famille

A tous ceux qui me sont chers.

Remerciements

Au terme de ce travail, je tiens à exprimer mes sincères remerciements à tous ceux qui ont contribué à son accomplissement.

J'adresse, donc, mes sincères remerciements encadreur mémoire M. Djoudi qui m'a orienté, encouragé et soutenu.

Je remercie les membres du jury d'avoir mon travail en le jugeant.

Je tiens aussi à remercier tout ceux qui ont contribué à mon formation .

Enfin, je remercie tous ceux qui m'ont toujours soutenu et réconforté.

RÉSUMÉ

L'objectif de ce mémoire est d'étudier les techniques de validation formelle pour évaluer la qualité des protocoles. Dans ce mémoire, on a commencé par la présentation de principes, l'intérêt, les avantages etles inconvénients des techniques de la vérification.Par la suite, ona étudié les techniques de descriptions formelles en détails.A la fin, on a essayé d'étudie le protocole *Stop &Wait*, sa description formelle avec Promela est sa validation avec Spin.

Mots clés

VERIFICATION FORMELLE, MODEL CHECKING, PROUVE FORMELLE, MODELE CHEKING,SPIN, PROMELA.

Abstract

The objective of this thesis is to study the formal validation techniques to evaluate the quality of protocols. So, we began with the presentation of the principles, the goals, the advantages and disadvantages of the techniques of verification. Subsequently, we studied the techniques of formal descriptions in detail. At the end, we tried to study *Stop & Wait* protocol, its formal description is Promela and its validation with Spin.

Keywords

FORMAL VERIFICATION, MODEL CHECKING, FORMAL PROOF, SPIN, PROMELA .

Table des matières

Liste des figures

Liste des tableaux

GLOSSAIRE

ACL2	**A** Computational **L**ogic for Applicative **C**ommon Lisp
BDD	**D**iagrammes de **D**écisions**B**inaires
CCS	**C**alculus **o**f **C**ommunicating **S**ystems
HOL	**H**igher **O**rder **L**ogic
Cop	**Coq**uand
Estelle	**E**xtended **S**tate **T**ransition **L**anguage
ISO	**O**pen **S**ystems **I**nterconnection
FSA	**F**inite **S**tate **A**utomaton
FSM	**F**inite **S**tate **M**achine
CPN	**C**olor **P**etri **N**et
LOTOS	**L**anguage **o**f **T**emporal **O**rdering **S**pecification
LTL	**L**ogiques de **T**emporelle **l**inéaire
ML	***M****éta-****L****angagefonctionnel*
PLTL	**P**ropositional **L**inear ***T****emporal* ***L****ogic*
Promela	**P**rocess **M**eta **L**anguage
RdP	**R**éseaux**D**e **P**étri
UPPAAL	***Upp****sa****la****enSuède*

INTRODUCTION GENERALE

Avec le besoin croissant pour des applications réparties ainsi que pour des réseaux plus fiables, il y a une demande croissante pour des protocoles de communication qui doivent être utilisés d'une manière fiable pour l'échange de données entre les applications et les éléments du réseau. A part la spécification et la normalisation de ces protocoles, le test de conformité des implantations des protocoles par rapport aux spécifications constitue un aspect très important de la garantie de la qualité. Ainsi, la vérification formelle et les approches de test ont été développées en se basant sur les techniques de description formelle.

Objectif du mémoire

L'objectif de ce mémoire est d'étudier les techniques de validation formelle pour évaluer la qualité des protocoles.

Organisation de la thèse

Le mémoire est organisé comme suit :

Le premier chapitre présente le principe, l'intérêt, les avantages et les inconvénients des techniques de la vérification formelle.

Le deuxième chapitre présente les techniques de descriptions formelles.

Le dernier chapitre présente le protocole Stop &Wait, sa description formelle avec Promela est développée et son validation avec Spin.

Chapitre 1

Les techniques de la vérification formelle

1.1.Introduction

De point de vue historique, les racines des méthodes formelles, surtout pour ce qui concerne la vérification software, datent des années 60 [10]. En effet, à cette époque-là un intérêt particulier a été accordé à cette nouvelle approche. Mais, au départ, le démarrage n'était pas très bien réussi. Plusieurs facteurs ont joué contre l'expansion de ces méthodes.à savoir : cependant

1. Les notations n'étaient pas claires et simples à manipuler donc demandaient un effort supplémentaire lors de l'étude ce qui risque de résoudre le problème trop complexe.
2. Les techniques de vérification étaient loin d'être capable de traiter les cas réels qui concernent des systèmes de tailles très importantes.
3. Les outils utilisés n'étaient pas adaptés aux configurations étudiées ou bien encore trop complexes pour être utilisés.
4. Seules quelques études triviales ont étaient réalisées, ce qui demande à l'utilisateur un effort supplémentaire pour étudier chaque cas.
5. Un nombre très limité de personnes ont maitrisaient ce sujet.

Mais, malgré tous ces points négatifs la vérification formelle était très prometteuse, par exemple pour le domaine de la *vérification hardware* [29]. Cela découlait en effet de plusieurs facteurs dont on cite :

1. La structure des composants hardware est hiérarchique et régulière. C'est que toute entité est composée d'un ensemble de sous-unités.
2. La réutilisation de la même architecture ou d'une architecture similaire est très courante dans le domaine hardware. Donc, il est possible de conserver les

mêmes preuves pour des nouveaux systèmes en appliquant quelques modifications minimes.

3. La spécification hardware est plus structurée comparée à la spécification software.
4. Les primitives utilisées sont simples. En effet, représenter une porte *NAND* est beaucoup plus simple que s'attaquer à une boucle *while*, par exemple.
5. Le prix des erreurs dans le design pour ce qui concerne les composants hardware, et particulièrement atteignent les microprocesseurs, peut cause des retards pouvant atteindre 6 mois et des pertes dépassant les millions de dollars surtout une fois les masques de production réalisés.

Ces facteurs ont joué en faveur de la vérification formelle et lui permis de gagner petit à petit une place plus importante. C'est pourquoi actuellement ,à cette approche est associée une image prodigieuse. En effet, son succès dans le traitement de plusieurs cas, tel que la *vérification de protocoles*, lui a permis de s'imposer comme l'une des plus importantes et plus prometteuses méthode actuellement [5] .

L'un des facteurs qui fait preuve de l'importance de l'approche formelle est le fait qu'elle est actuellement petit-à-petit intégrée dans le domaine industriel, montrant la preuve de sa capacité de s'attaquer à des systèmes réels. D'autre part, l'intérêt que lui portent des groupes de vérification à l'échelle mondiale, tel que IBM, Motorola, Intel etc., y présente un très important renfort.

1.2. Vérification formelle

Définition1. La vérification consiste à établir l'exactitude d'une spécification et/ou implantation d'un système. En d'autres mots, l'objectif de la vérification est de répondre à la question suivante: "*Est-ce que le système est exact?*". Les techniques de vérification sont applicables si la structure interne de l'implantation est connue.

Définition2. La vérification formelle consiste à examiner que certaines propriétés souhaitées sont respectées sur un système donné. Cette vérification se fait par la confrontation d'une représentation concise et non ambiguë des propriétés et d'un modèle mathématique, basé sur un langage formel, représentant le système. Le mécanisme de vérification ayant pour but d'examiner que les propriétés souhaitées sont incluses dans les comportements du système.

Définition3. Les méthodes de vérification formelles [17] sont des techniques permettant de raisonner rigoureusement, à l'aide de logiques mathématiques, sur des programmes informatiques ou des matériels électroniques, afin de démontrer leur validité par rapport à une certaine spécification. Ces méthodes permettent d'obtenir une très forte assurance de l'absence de bug dans les logiciels, c'est-à-dire d'acquérir des niveaux d'évaluation d'assurance élevés .Elles sont basées sur les sémantiques des programmes, c'est-à-dire sur des descriptions mathématiques formelles du sens d'un programme donné par son code source (ou parfois, son code objet). Cependant, elles sont généralement couteuses en ressources (humaines et matérielles) et actuellement réservées aux logiciels les plus critiques. Leur amélioration et l'élargissement de leurs champs d'application pratique sont la motivation de nombreuses recherches scientifiques en informatique.

Ces points de vue partent d'une correcte énumération des caractéristiques des méthodes formelles. Les avantages des méthodes de vérification formelle sont nombreux. En effet, ces méthodes :

1. Considèrent toutes les entrées possibles au système.
2. Vérifient la validité des propriétés du système mathématiquement.
3. Ne nécessitent pas une spécification des sorties du système prévues.
4. Permettent, pour certains outils, d'identifier les traces des erreurs s'il y a lieu.
5. Traitent, pour certains outils, tous les cas possibles.

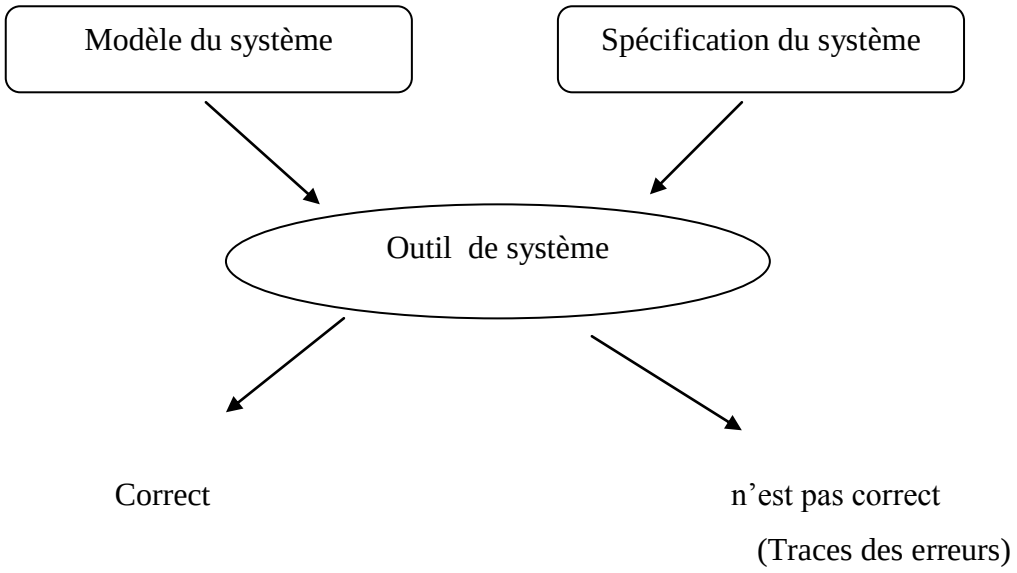


Figure 1. Modèle de la vérification formelle

La vérification formelle possède, elle aussi, certains inconvénients [10]. En effet, elle demande un effort supplémentaire pour parvenir à une description complète et simple du système à vérifier. C'est qu'il est nécessaire de définir une spécification, d'une part, tenant en considération tous les détails du système, et d'autre part, assez simple à manipuler dans la phase de vérification.

Il existe deux principales approches de vérification formelle : La preuve formelle, le Model Checking.

1.3. Les approches de la vérification formelle

1.3.1. La preuve formelle

Introduite par Hoare [4], cette approche est basée sur une preuve mathématique. Elle consiste à décrire le système et ses propriétés dans un modèle de sémantique axiomatisable et à démontrer que les propriétés peuvent être obtenues à partir du système en utilisant les règles d'inférence. Cette approche est utilisée dans des outils tels que l'atelier B [2], Coq [15]et PVS [19], etc.

Cette approche a l'avantage de pouvoir traiter des systèmes à nombre d'états infini. Elle ne repose pas sur une construction explicite d'un modèle de comportement de type états/transitions, très coûteux en mémoire, puisqu'elle est capable d'inférer des conclusions directement à partir d'une description d'événements ou d'opérations permettant de faire évoluer le système.

L'utilisation des techniques de preuve est cependant rendue difficile par le fait que l'utilisateur doit être capable de modéliser le système par un modèle mathématique sur lequel s'effectue la preuve et raffiner ce modèle mathématique jusqu'à obtenir le système à implanter [9].

Lorsqu'il s'agit d'un système conçu et implémenté, le seul moyen pour générer le modèle abstrait correspondant est l'utilisation des assertions logiques (précondition, postcondition).

Néanmoins, la principale contrainte à laquelle fait face la preuve formelle est la complexité qui restreint son utilisation à des systèmes de taille limitée.

1.3.2. Le Model Checking

Le Model Checking est une approche algorithmique qui repose sur une idée simple : si l'on énumère toutes les situations possibles dans lesquelles peut se trouver le système, on saura s'assurer qu'aucune de ces situations n'est en contradiction avec les comportements que l'on souhaités [9].

Du fait que le Model Checking procède par énumération exhaustive des états du système à vérifier, la représentation de tous les comportements possibles d'un système avec un nombre d'états infini, conduit rapidement à un dépassement des capacités de stockage. Ce phénomène est connu sous le nom d'explosion combinatoire, ce problème a fait l'objet de plusieurs recherches et a donné lieu à des techniques d'abstraction permettant d'aborder la vérification des systèmes infinis.

1.4. La preuve formelle

1.4.1. Présentation

La preuve de théorème [11,18,21] est une technique orientée événements. L'implantation et la spécification doivent être formalisées selon un calcul logique, i.e., l'effet des actions est spécifié à l'aide de formules logiques ou temporelles. La preuve de théorème montre que la formule logique de l'implantation est une conclusion de la formule logique de la spécification. "Est-ce que l'implantation I est conforme à la spécification S?" est formulée comme un théorème du système formel. La preuve de théorème consiste à offrir la preuve du théorème en question.

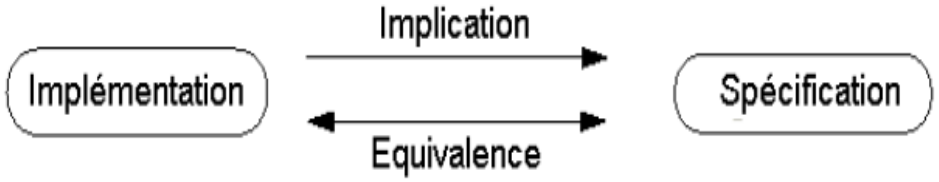


Figure 2. La vérification par démonstrateur de théorèmes.

1.4.2. Définitions

Preuve de théorèmes. Consiste à énoncer des propositions et à les démontrer dans un système de déduction de la logique mathématique. La preuve de théorèmes est une approche qui tend à être de plus en plus automatisée et assistée par ordinateur,c'est pour cela que le terme " preuve automatique de théorème" est souvent utilisé.

Théorème. Proposition qui peut être mathématiquement démontrée, c'est-à-dire une assertion (proposition mathématique admise) qui peut être établie comme vraie au travers d'un raisonnement logique construit à partir d'axiomes (vérité indémontrable qui doit être admise). Une fois le théorème démontré, il devient alors une hypothèse utilisable.

Preuve. Permet d'établir une proposition à partir de propositions initiales, ou précédemment démontrées à partir de propositions initiales, en s'appuyant sur un ensemble de règles de déduction. Une fois démontrée, la proposition peut ensuite être elle-même utilisée dans d'autres démonstrations. Dans ce cas, on la nomme généralement lemme. La preuve, bien que nécessaire à la classification de la proposition comme "théorème", n'est pas considérée comme faisant partie du théorème. La prevue comprend :

- Des axiomes ou des postulats : principes utilisés dans la construction d'un système déductif, mais qu'on ne démontre pas eux-mêmes.
- D'autres théorèmes déjà démontrés.

Chaque étape de la preuve est liée aux précédentes par des règles d'inférence logiques. Au sens large, toute assertion effectivement démontrée peut prendre le nom de théorème. Dans les ouvrages de mathématiques, il est cependant d'usage de réserver ce terme aux affirmations considérées comme particulièrement intéressantes ou importantes. Selon leur importance ou leur utilité, les autres assertions peuvent prendre des noms différents :

- Lemme : assertion servant d'intermédiaire pour démontrer un théorème plus important.

- Corollaire : résultat qui découle directement d'un théorème prouvé.
- Proposition : résultat relativement simple qui n'est pas associé avec un théorème particulier.
- Remarque : résultat intéressant ou conséquence qui peut faire partie de la preuve ou d'une autre affirmation.
- Conjecture : proposition mathématique dont on ignore la valeur de vérité. Une fois prouvée, une conjecture deviant un théorème.

Avec l'avènement des ordinateurs et des systèmes d'aide à la démonstration, des mathématiciens contemporains rédigent des démonstrations qui sont amenées à être vérifiées par des programmes dits assistants de preuve.

1.4.3. Assistants de preuve

Un assistant de preuve est un logiciel permettant l'écriture et la vérification de preuves mathématiques, soit sur des théorèmes au sens usuel des mathématiques, soit sur des assertions relatives à l'exécution de programmes informatiques. L'écriture de preuves entièrement formelles est une activité extrêmement fastidieuse ; de nombreuses étapes qui seraient sautées, car considérées comme évidentes pour le lecteur familier des mathématiques, doivent être décortiquées dans les plus grands détails. Cependant, l'assistant de preuve peut fournir plus ou moins d'automatisation pour limiter le travail de l'utilisateur humain.

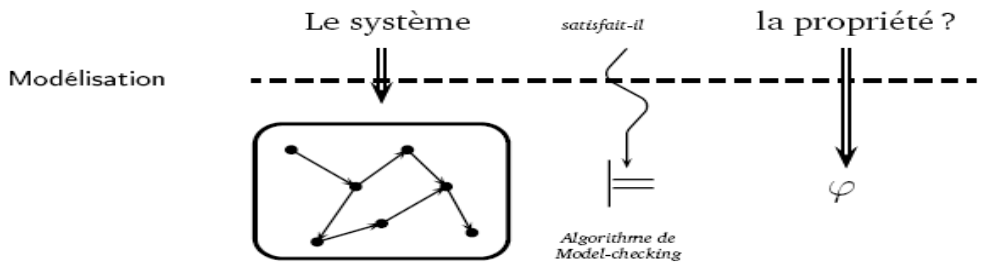
De nombreux assistants de preuves sont disponibles les plus connus: Coq , HOL , PVS , ACL2 , Isabelle , LEGO .

Une répulsion à l'usage des assistants de preuve est que, de toute façon, la sécurité des preuves obtenues repose sur le bon fonctionnement de l'assistant. En effet, les assistants de preuves sont de gros logiciels complexes, dont on peut soupçonner qu'ils soient eux-mêmes bugués. Certains assistants de preuve, comme Coq , produisent un terme de preuve dont la vérification peut être déléguée à un logiciel beaucoup plus simple qu'un assistant complet .

1.5. Le Model Checking

1.5.1. Le principe du model-checking

La vérification par model-checking[27] consiste à vérifier les propriétés souhaitées d'un système sur une traduction mathématique de celui-ci. Son principe est illustré dans la figure1.3



.

Figure 3. Le principe du model-checking

Afin de vérifier un système, le concepteur doit :

- réaliser un *modèle* du système qu'il souhaite vérifier. Cette tâche est délicate puisque, la plupart du temps, le système est décrit en langue naturelle ou sous forme de programme incompatible avec le formalisme des modèles et il est alors nécessaire de traduire, le plus souvent à la main, cette description.
- spécifier formellement, par exemple dans un langage logique, les *propriétés* attendues du système. Nous donnons quelques exemples de propriétés :
  - **Le système peut-il se bloquer ?**
  - **Après une panne, le signal d'alarme se déclenche-t-il ?**
  - **Après une pression sur le bouton d'appel, l'ascenseur arrive-t-il un jour ?**
  - **...**

Une fois ces deux étapes réalisées, un logiciel dénommé *model-checker*est utilisé afin de déterminer si le modèle du système vérifie les propriétés. Si la traduction du modèle et des propriétés est correcte, on s'attend à ce que le système réel ait les mêmes propriétés que le modèle.

L'avantage essentiel du model-checking sur les méthodes fondées sur le test et la simulation est l'*exhaustivité* : une propriété valide sur le modèle le sera dans absolument toutes les situations alors qu'un scénario réel de fonctionnement a pu être oublié dans une procédure de test. Par ailleurs, lorsqu'une propriété n'est pas vérifiée dans un modèle, les outils existant fournissent une *trace d'exécution* du modèle

démontrant que la propriété n'est pas satisfaite. Il est alors plus facile de déterminer l'origine de l'erreur et de la corriger.

En contrepartie, l'effectivité du model-checking est limitée par la taille des systèmes qu'on peut analyser en pratique. En effet, les modèles ont, même pour des systèmes relativement simples, des tailles importantes et on parle du problème de l'*explosion combinatoire*. L'analyse du modèle est alors impossible en raison de la complexité importante des calculs nécessaire. Dans de telles situations, l'utilisateur doit simplifier le modèle et proposer une *abstraction* qui préserve les propriétés à vérifier.

1.5.2. Les approches du model checking

Il existe deux principales approches de model checking qui diffèrent sur la manière dont le comportement souhaité du système est décrit :

1.5.2.1. Approche basée sur la logique (hétérogène) :

Dans cette approche issue des travaux de Quielle et Sifakis (1981) et Clarke et Emerson (1981), les comportements attendus du système sont décrits par un ensemble de propriétés exprimées dans une logique appropriée (temporelle[1] ou modale[2]), le système est généralement modélisé par un automate à états finis et l'algorithme de model checking consiste à vérifier si le modèle satisfait ces propriétés pour un ensemble donné d'états initiaux[21].

1.5.2.2. Approche basée sur le comportement (homogène) :

Comme son nom l'indique, dans l'approche homogène, les comportements attendus sont exprimés dans la même notation (exemple, les automates), la vérification repose sur la confrontation des comportements attendus et souhaités au moyen de relations d'équivalence ou de pré-ordre. Les relations d'équivalences expriment généralement la notion de « se comporter comme », tandis que les relations de pré-ordre représentent la notion de « se comporter au moins comme ».

---

[1]Logique qui permet de spécifier des propriétés qui change dans le temps.
[2]Logique exprimant certains concepts modaux tels que l'obligation, permission

Plusieurs relations d'équivalence et de pré-ordre ont été définies. Les plus connues sont la relation d'équivalence de bisimulation [3] et la relation de pré-ordre d'inclusion[4][21].

Les deux approches hétérogène et homogène sont conceptuellement différentes pourtant des relations entre elles peuvent être établies. En effet, selon la logique, deuxs ystèmes sont dits équivalents seulement s'ils satisfont les mêmes formules. D'autre part, il est établi que deux modèles bisimilaires satisfont les mêmes formules de la logique. Par conséquent, si à l'aide de l'approche basée sur la logique, il s'avère que deux modèles satisfont les mêmes propriétés alors on peut affirmer que ces deux modèles ont deux comportements équivalents. Inversement, et à fortiori, puisque en générall'approche basée le comportement est plus rapide que celle basée sur la logique, si deux modèles sont équivalents alors il est clair qu'ils satisfont les mêmes propriétés.

1.6. Les propriétés d'un système informatique

Les propriétés généralement vérifiées dans les systèmes informatiques ont été classées par *Lamport* en 1977 en deux grandes catégories :

1.6.1. Les propriétés de sûreté (safety) ou propriétés d'invariance

Elles énoncent des conditions qui doivent toujours être vérifiées c'est à dire, maintenues à vrai.[3] Cela équivaut à exprimer que quelque chose de mauvais ne se produira pas. Des exemples de telles propriétés sont : l'atteignabilité (d'un état), l'absence de blocage, l'exclusion mutuelle, etc.

1.6.2. Les propriétés de vivacité (liveness)

Elles expriment qu'une situation particulière se produira. Les propriétés de vivacité assurent que quelque chose de bon se produira. Des exemples de telles propriétés sont:la disponibilité, l'absence de famine, ou la garantie de service, etc.

---

[3] Fait qu'un automate puisse simuler tout comportement d'un autre automate et vice versa

[4]Un automate A est inclus dans un automate B si tous les mots acceptés par A sont acceptés par B

Cette classification, bien qu'informelle, est fort utile puisque, d'après le théorème de décomposition [3], toute propriété exprimable en logique temporelle peut se mettre sous la forme d'une conjonction de propriétés de sûreté et de vivacité.
Cette classification a été redéfinie et étendue par d'autres types de propriétés.
Dont ; Les propriétés d'équité (fairness) elles stipulent que toute action atomique inconditionnelle éligible sera exécutée un jour.

1.7. Les outils de la vérification

La vérification automatique des protocoles est intrinsèquement liée à des outils logiciels. Cette section présente brièvement quelques-uns des outils les plus utilisés de vérification formelle disponibles. Tous ces outils sont générales et peuvent être appliqués à un certain nombre d'applications différentes pour vérifier un large éventail de systèmes.

1.7.1. Le système HOL

Le prouveur de *théorème HOL* est un outil qui a été développé à l'université de Cambridge . C'est un système basé sur une logique expressive et générale qui permet d'avoir une formulation correcte et pratique. Ce système est basé sur le prouveur de théorèmes LCF et hérite plusieurs de ces concepts. A ce jour il existe trois versions du système HOL. La première *version (HOL88)* développée sous ML , la deuxième a été *(HOL90)* développée sous SML et la troisième version *(HOL98)* qu'on va utiliser tout au long de notre étude.

1.7.1.1. La logique HOL

La logique de HOL est une variété de la logique d'ordre supérieur basée sur une formulation de la théorie simple des types de Church. Elle se présente comme une extension de la logique des prédicats parce que :

1. Les variables peuvent être instanciées par des fonctions, et les arguments des fonctions peuvent être des fonctions.
2. Les fonctions peuvent être représentées par des l-abstractions.

3. Chaque terme possède un type qui peut être polymorphe(au sens du langage SML).

Les notations utilisées dans la logique du système HOL sont présentées dans le tableau suivant

Terme	**Notation HOL**	**Notation Standard**	**Description**
Vrai	**T**	**T**	**Vrai**
Faux	F	⊥	Faux
Négation	~P	¬P	Non P
Disjonction	P \/ Q	P ∨ Q	P ou Q
Conjonction	P /\ Q	P ∧ Q	P et Q
Implication	P ==>Q	P→ Q	P implique Q
Egalité	P = Q	P = Q	P égale Q
Quantification (")	!x. P	∀x. P	Pour tous x : P
Quantification ($)	?x. P	∃x. P	Il existe x : P
Terme (ε)	x. P	εx. P	Un x tel que P
Conditionnelle	P => Q | R	(P => Q, R)	Si P alors Q sinon R

Tableau 1. La logique du système HOL

1.7.1.2. Le prouveur HOL

Le système *HOL* est le résultat du codage de la logique décrite précédemment dans le langage fonctionnel *ML* . Les termes sont représentés comme des objets de *ML* de type *term*. Le langage *ML* permet de manipuler ces termes et de contrôler leur bonne formation au moyen d'un algorithme d'inférence de type. Un terme de la logique *HOL* doit être présenté au système entre guillemets. Si ML lui attribue le terme *term*, alors cela veut dire qu'il est bien formé. Les types de la logique *HOL* sont codés comme des objets de type type, en les faisant précéder de deux points verticales.

La fonction principale du système HOL est de montrer que certains termes de la logique HOL sont des théorèmes. Les théorèmes prouvés dans le système sont des

objets d'un autre type du langage ML appelé *thm*. Un théorème est représenté par un ensemble de termes qu'on appelle hypothèses et un terme qu'on appelle conclusion. Etant donné un ensemble d'hypothèses G et une conclusion t, on note le théorème correspondant par G |- t.Si G est vide alors le théorème était noté tout simplement |- t.

Les théorèmes sont introduits dans le système HOL soit en postulant comme des axiomes, soit en les déduisant des théorèmes existants par des règles d'inférences décrites dans le métalangage ML et qu'on appelle tactique. La preuve d'un théorème est une succession de règles d'inférences appliquées aux axiomes ou aux théorèmes déjà prouvés. Les règles d'inférences vérifient que la déduction de la conclusion d'un théorème de ses hypothèses est conforme aux règles logiques de HOL. Le noyau du système HOL est constitué de cinq axiomes et de huit règles d'inférences primitives à partir desquelles toutes les autres règles de la logique sont dérivées.

Le résultat d'une session HOL est un objet appelé théorie. Cet objet constitue un ensemble de types, de constantes, d'axiomes, de définitions et de théorèmes. Le système fournit des possibilités d'étendre des théories existantes et de former des hiérarchies de théories. Si des résultats d'autres théories doivent être utilisés dans la théorie en question, on doit alors déclarer ces théories comme parents de la théorie qu'on développe. Les théories permettent une structuration des faits.

Un autre concept très utile est celui de librairie. Une librairie est une collection de théories, de théorèmes, de tactiques et de fonctions ML. Elle n'est pas nécessairement chargée lors du lancement du système HOL, néanmoins elle peut être chargée dynamiquement lors d'une session avec HOL.

Par exemple : HOL utilisée pour vérifier le protocole AODV.

1.7.2. Le simulateur et vérificateur des systèmes concurrents SPIN

SPIN [16] est le nom d'un outil de validation formelle crée par les laboratoires Bell et maintenu par un groupe de développement depuis sa création en 1980. SPIN permet de modéliser et de valider des systèmes distribués et parallèles en utilisant une approche appelé " *on the fly* ", qui consiste principalement à générer un évaluateur automatisé en langage ANSI C. Cet évaluateur une fois compilé et exécuté, produit une représentation très compacte du système basée sur des automates de Boochi , sur laquelle il est possible d'effectuer un nombre de preuves classiques dans le domaine du model-checking mais aussi de vérifier la validité de formules logiques, exprimées par le développeur en logique temporelle linéaire. Ce qui rend le model compacte créé par SPIN différent, est qu'il est créé " sur le champ" (dont vient le nom "*on the fly*"), à la place de créer tout l'espace d'états qui peut être potentiellement gigantesque dans le modèles de taille conséquente.

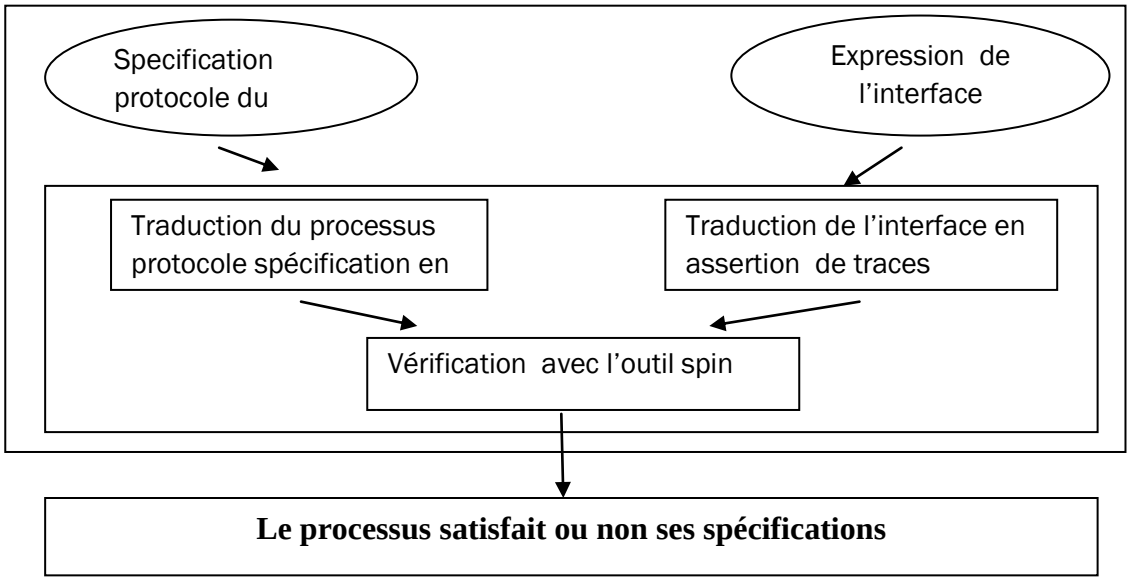


Figure 4 Les démarches de la vérification avec SPIN

Ceci a comme résultat Le langage accepté par SPIN s'appelle Promela, nom qui vient de l'acronyme anglais **Pro**cess **Me**ta **La**nguage. Un des aspects les plus intéressants de Promela /SPIN est aussi l'approche "ingénieur".

Le langage donne des constructions qui sont familières à ceux qui connaissent le langage C, sans pour autant perdre de vue qu'il s'agit d'un langage de spéciation formelle et pas un langage de programmation.

En *Promela* il existe la notion de variable, qui a une signification équivalente a celle de C: une variable est une valeur symbolique qui existe dans un certain contexte, qui contient une valeur et qui peut être manipulé avec les opérations appropriées. Pour les variables, le langage *Promela* dé nit un ensemble assez riche de types de données primitives.

Dans le tableau 1.2 sont listés les types primitifs de *Promela.* Le langage accepte aussi la définition de rangées d'une ou plusieurs dimensions, en utilisant une syntaxe analogue à celle du langage de programmation C. Par exemple l'expression byte array[20];déclare une variable appelée array qui représente une rangée de 20 octets.

Type	**équivalent en C**	**intervalle de valeurs**
Bit	champ de bits	0..1
Bool	champ de bits	0..1
Byte	char,byte	0..255
Short	Short,int	$-2^{15}-1..2^{15}-1$
Int	Int	$-2^{31}-1..2^{31}-1$

Tableau 2. Types de données primitifs dans Promela

Dans le langage mentionné il est possible de définir des valeurs symboliques. Pour cela on a la construction mtype. Avec mtype on peut déclarer les étiquettes qui sont habituelles dans les modèles de protocoles, par exemple,

mtype ={ ack, nak, err, next, accept }.

1.7.3. L'outil UPPAAL[22]

UPPAAL a été créé par l'université *Uppsala en Suède* et *l'université Aalborg* au *Danemark* .Les fondements théoriques ont été publiés en 1994 par *W.Yi, P.Pettersson* et *M.Daniels*. La première version a été distribuée en1995.UPPAAL est un model checker intégrant une interface graphique permettant la construction d'automates temporisés. Lors de cette construction nous pouvons spécifier pour un état ,en plus de ses invariants ,si il est *"urgent"* ou *"committed"*.

Urgent Un état de contrôle est dit «urgent "lorsqu'on ne peut attendre dans aucun états du système, lorsque le système atteint cet état. Le système est donc obligé de faire des transitions instantanées, d'un état du système à un autre, tant que le système est dans un état urgent.

CommittedUn état de contrôle est dit *"committed"* lorsqu'on ne peut pas attendre dans cet état *(=urgent)*et quand on est dans *untel*état, la prochaine transition doit faire sortir de l'état (ou d'un états) *committed*.

Nous pouvons également définir les horloges locales (propre à un automate) ou globales (communes à tous les automates). UPPAAL permet de vérifie les fomules suivantes:

- ∃ ◊: il existe un chemin le long du quel p est vrai un jour.
- ∃ □p: il existe un chemin le long du quel p est toujours vrai.
- ∀◊p: le long d et out chemin p est vrai un jour.
- ∀□p: le long d et out chemin p est toujours vrai.
- p→q: quand p est vrai, alors q sera forcément vrai un jour.
- Où p et q sont du type :
- p,q :=état d'automate| horloge˜ valeur| $p \wedge q \mid p \cup q \mid \neg p \mid p \Rightarrow q$ | deadlock.

1.7.4. Les réseaux de Petri colorés [9]

Développé Par *CPN Group, Université d'Aarhus, Denmark.* Il est un outil le plus utilisés dans son domaine, dédié à la simulation de *réseaux de Petri* de haut niveau. Il se compose de :

- CPN editor : permet d'éditer les réseaux colorés et de vérifier la syntaxe,
- CPN simulator : permet de simuler le comportement de réseau (les règles de franchissement, les transitions franchissables, etc),
- CPN CPN state *space tool* : supporte la vérification sur l'espace d'états.

CPN Tools combine la capacité des réseaux de Petri coloré et la capacité d'un langage fonctionnel *(le langage Standard ML).CPN Tools* fournit des primitives pour d'écrire les processus concourants, tandis que le langage fournit des primitives pour définir des types de données (ensemble de couleur) et des manipulations des données (expression d'arcs, gardes, etc.).

1.7.5. L'assistant de preuve Coq

Coq [6] est un assistant de preuve développé à *l'INRIA,* à l'Ecole polytechnique et à l'Université de Paris XI (et antérieurement à l'Ecole normale supérieure de Lyon) dans le cadre du projet *TypiCal.* Coq est fondé sur le calcul des constructions (introduit par Thierry *Coquand, CoC* abrégé en anglais, d' où un jeu de mots justifiant le nom du système), la théorie des types d'ordre supérieur et un langage de spécification sous forme de lambda-calcul typé. Le calcul des constructions utilisé dans Coq comprend directement les constructions inductives, d'où son nom de calcul des constructions inductives (CIC).

Plus particulièrement, Coq permet :

- de manipuler des assertions du calcul,
- de vérifier mécaniquement des preuves de ces assertions,
- d'aider à la recherche de preuves formelles,
- de synthétiser des programmes certifiés à partir de preuves constructives de leurs spécifications.

Parmi les grands succès de Coq, on peut citer la démonstration complètement mécanisée du théorème des quatre couleurs par *Georges Gonthier et Benjamin Werner. Coq* est un logiciel libre distribué selon les termes de la licence *GNU LGPL*, La documentation de la version courante du système (Version 8.21) se trouve en ligne sur [19].

1.8. Conclusion

Dans ce chapitre, nous avons fait un survol sur les différents types de techniques de la vérification formelle. Nous avons aussi expliqué la différence qui existe entre les approches du méthodes formelle (Modele checking , Théorème du preuve) . Par la suite nous représentons les diffèrent outillés de vérification formelle (*Spin , Cpn , Uppaala , Hol*).

Chapitre 2

Les techniques de la modélisation et de la description formelle

2.1. Introduction

Dans ce chapitre, nous allons présenter les méthodes de spécifications les plus utilisées ainsi que les techniques de description (ou spécification) formelle. L'écriture de la spécification permet une compréhension approfondie du système à développer. Une fois l'investissement d'écriture d'une spécification formelle effectué, nous disposons d'un texte exploitable par des outils de preuve ou d'évaluation symbolique. Nous pouvons alors valider la spécification en prouvant des propriétés souhaitées, ou en réfutant des propriétés correspondant à des situations interdites. Par ailleurs, nous pouvons aussi dériver une maquette, de la spécification formelle, qui en réalise partiellement les fonctionnalités spécifiées, et tester celles-ci très tôt dans le développement.

Dans ce qui suit, nous présentons le modèle de machine à états finis (*FSM*), les extensions de ce modèle ainsi que d'autres méthodes de spécification de systèmes.

2.2. Modélisation d'un système

Définition. La modélisation consiste en la conversion du système à étudier en un modèle exprimé par un formalisme accepté par l'outil de vérification de modèles. Cette étape est souvent la plus compliquée parce qu'elle exige une bonne connaissance du domaine d'application ainsi que du langage de modélisation et elle nécessite l'utilisation de l'abstraction pour éliminer les détails[8].

2.2.1. Structure de Kripke

2.2.1.1. Définition Les structures de kripke

Une structure de Kripke permet de modéliser un système en représentant tous ses états et les transitions qui permettent de passer d'un état à un autre.

Soit ***P*** un ensemble fini de propositions booléennes. Une structure de Kripke sur ***P***est un quintuplé ***M = (S,E,T,I ,L)***où :

- ***S*** est l'ensemble des états ;
- ***E*** est l'ensemble des étiquettes des transitions.
- $T \subseteq S \times E \times S$ est la relation de transition ; $\forall s \in S \bullet \exists s' \in S, e \in E, (s,e,s') \in T$;
- $I \subseteq S$ est l'ensemble des états initiaux ;
- $L : S \rightarrow 2^P$est l'application qui associe à tout état de ***S*** l'ensemble fini des propositions booléennes vérifiées dans cet état[5][30].

Chaque transition est accompagnée d'un ensemble d'actions qui permettront au système de changer d'état en modifiant les variables d'état du système.

Une transition peut être gardée par une condition sur les variables d'états. Le franchissement de la transition n'est possible que si la condition est vérifiée.

Notons que pour la description des systèmes de processus, on utilise l'appellation "*systèmes de transitions*" ou même "*systèmes de transitions étiquetées*" (STE), plus connus en théorie des langages sous le nom "automates".

Il est possible de représenter un automate en utilisant une représentation graphique.

Les états sont représentés par des ronds, l'état initial est distingué par une flèche arrivant sur cet état et sans origine, l'état terminal, s'il en existe est représenté par deux cercles concentriques, les transitions sont des arcs orientés dans le sens état de départ vers état d'arrivée, l'arc désignant la transition est annoté par l'étiquette.

Une exécution d'un automate est une suite d'états décrivant une évolution possible du système.

Un arbre d'exécution d'un automate est une représentation arborescente de toutes les exécutions du système qui est très souvent infinie. La racine de l'arbre est l'état initial

[5]L'état du système est décrit au moyen d'un ensemble de variables d'état.

de l'automate, elle a pour fils tous les états atteignables[6]à partir de la racine, ces nouveaux nœuds ont à leur tour des successeurs qui sont identifiés de la même manière. Un état est dit atteignable (accessible) dans un automate s'il apparaît dans l'arbre des exécutions de l'automate, autrement dit, s'il existe au moins une exécution dans laquelle il figure.

Le dépliage d'un automate consiste à produire un automate dans lequel toutes les transitions de l'automate sont présentes. Il est très souvent utilisé dans le cas de la présence de gardes et de variables d'états [30].

Un chemin dans un automate est une suite σ, finie ou infinie, de transitions **(*si,,ei,, si'*)** qui s'enchaînent, c'est-à-dire ***si' = si+1***pour tout *i*. Un chemin est souvent noté **s1→ e1 s2 → e2 s3 …**

La longueur d'un chemin **"σ"**est le nombre de transitions qu'il contient. Cette longueur peut être infinie. Dans la terminologie des systèmes de processus, un chemin est également désigné par le mot trace.

Une exécution partielle est un chemin partant de l'état initial.

Une exécution complète est une exécution (partielle) maximale, c'est-à-dire une exécution qui ne peut être prolongée. Elle est donc soit infinie, soit terminée dans un état s_n duquel n'est issue aucune transition de l'automate considéré (et dans ce cas on parle d'une exécution avec blocage).Lors de la modélisation de systèmes ou programmes réels on se rend très souvent compte de leur complexité qui rend non envisageable de les aborder dans leur totalité. Toutefois, ils font intervenir plusieurs sous-systèmes ou modules. Ainsi, pour construire la modélisation du système global, il est alors naturel de commencer par modéliser chacune des composantes du système. Ensuite, combiner les modélisations des différentes composantes (produit cartésien, produit synchronisé, etc….) afin d'obtenir la modélisation du système global.

La structure de Kripke représentant l'évolution du système via tous ces états accessibles est généralement obtenu à partir d'une description formelle du système (Algèbre de processus, automates, Réseaux de Petri, etc.).

[6]Les états qui peuvent être atteints au moyen d'une transition

2.2.2. Les diagrammes de décisions binaires (BDDs)

Les diagrammes de décisions binaires ou *binar decisions diagrams* ont été Diffusés par *Bryant* [24] à partir de 1986 pour remédier au problème de l'explosion des états très souvent rencontré dans le model checking. En effet, ils permettent de représenter de manière symbolique l'ensemble des états du système. Il s'agit de représenter le système par un vecteur de variables booléennes et de vérifier la satisfaisabilité de formules booléennes :

$V = \{v_1, \dots, v_n\}$

$f : \{0,1\}_n \rightarrow \{0,1\}$

$f(x_1, \dots, x_n) = 1$ l'affectation $(v_i \leftarrow x_i)$satisfait la formule f[24].

Ces formules sont représentées sous une forme canonique qui se rapproche des arbres de décisions binaires. En fait, un BDD est un Graphe Orienté Acyclique (DAG), les nœuds de ce graphe sont les variables représentant les états du système ou la valeur que prend la formule booléenne. Bryant [24] a démontré que si un ordre total est établi pour l'occurrence des variables du système, alors à toute formule booléenne correspondra un BDD appelé OBDD *(Ordered Binary Decision Diagram).*

La construction d'un OBDD se fait par la réduction de l'arbre de décision binaire ordonné correspondant à la formule. L'arbre de décision binaire ordonné est construit en ajoutant à chaque niveau de l'arbre une variable booléenne, et ceci dans l'ordre préétabli. Ainsi, la racine correspond à la première variable. Le nœud racine génère deux arcs gauche et droit représentant le cas où le nœud racine est égal respectivement à 0 ou 1. Les deux arcs droit et gauche marquent chacun un nœud qui représente la deuxième variable de l'ordre préétabli. Et ainsi de suite jusqu'à ce que toutes les variables soient explorées. Les dernières variables permettent de déterminer la valeur de la formule représentée par l'arbre dont les feuilles sont par conséquent 0 ou 1. Ainsi, la valeur de la formule représentée par l'arbre de décision binaire est obtenue en parcourant le graphe de la racine aux feuilles.

L'OBDD est construit à partir de l'arbre de décision binaire ordonné au moyen des deux règles de réduction suivantes :

1. Combiner les sous arbres isomorphes en un seul sous arbre.

2. Eliminer les nœuds dont le fils gauche et droit sont isomorphes [24].
La taille du graphe orienté acyclique obtenu est fortement liée à l'ordre des variables.

2.3. Les techniques de description formelle

Nous allons décrire les modèles sur lesquels reposent les techniques de description formelle : *les automates d'états finis, les réseaux de Pétri, etc*…. Ensuite, deux langages normalisés seront décrits : *Estelle et Lotos* ; Estelle utilisant les automates et Lotos l'ordonnancement temporel. Ces langages de description de protocoles ont été développés par l'ISO. Ils possèdent tous les deux une sémantique et une syntaxe formelle.

2.3.1. Machines à états finis

Un **automate fini** (on dit parfois **machine à états finis**), en anglais *finite state automata* ou *finite state machine* (FSA, FSM), est une machine abstraite utilisée en théorie de la calculabilité et dans l'étude des langages formels. Un automate est constitué d'*états* et de *transitions*. Son comportement est dirigé par un mot fourni en entrée : l'automate passe d'état en état, suivant les transitions, à la lecture de chaque lettre de l'entrée. Un automate fini possède un nombre fini d'états distincts : il ne dispose donc que d'une mémoire bornée.

Une machine à états peut être représentée graphiquement par un graphe étiqueté. Les états sont des cercles (ou des ellipses) et les transitions sont des flèches orientées de leur état de départ vers leur état de destination. Les transitions sont étiquetées par des expressions de la forme *ev & garde / action*. Les états contiennent leur nom et sont étiquetés par leurs actions d'entrée et de sortie. L'état initial est repéré par une flèche spéciale en forme de L [16].

Formellement,[16] un Automate Fini Déterministe (DFA) est un quintuple **: (*S*, Σ, *T*, *s*, *A*)**

• un alphabet **(Σ)**

• un ensemble d'états **(*S*)**

• une fonction de transition **(*T* : *S* × Σ → *S*).**

• un état de départ **(*s* ∈ *S*)**

• un ensemble d'états acceptant **(*A* ⊆ *S*)**

2.3.2. Les réseaux de Pétri

Les *réseaux de Pétri (RdP)* ont été introduits au début des années 60 par C.A. Pétri, puis développés au *MIT* autour de 1972. Ils permettent, en particulier, de modéliser et d'analyser des systèmes de processus concurrents. La spécification des protocoles de communication en *RdP* a mené à des recherches et des analyses très importantes.

Un *RdP* est composé de deux types d'objets : les places et les transitions. L'ensemble des places permet de représenter les états du système ; l'ensemble des transitions représente alors l'ensemble des événements dont l'occurrence provoque la modification de l'état du système. Les places jouent donc le rôle de variables d'états du système et sont à valeur entière. Ces valeurs entières sont représentées par autant de marques affectées à une place. L'état du système est alors associé à un marquage définissant pour toute place le nombre de marques qui lui sont affectées (ou bien qu'elle contient). À l'occurrence d'un événement correspond le franchissement d'une transition.[1]

Un réseau de Pétri est un quadruple ***R* = <*P*,*T* ; *Pré*, *Post*>**où :

- **P** est un ensemble fini de places et est représenté graphiquement par des cercles ;
- **T** est un ensemble fini de transitions et est représenté graphiquement par des barres ;
- ***Pré* : *PxT* → |*N*** est l'application d'incidence avant ;
- ***Post* : *PxT* → |*N*** est l'application d'incidence arrière.

Exemple

Soit l'exemple décrit dans la figure 5.

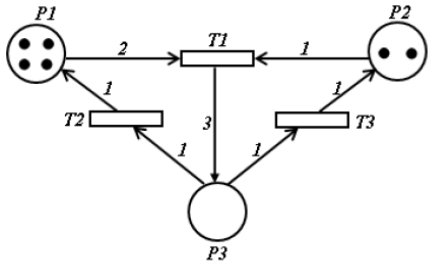


Figure 5 Exemple d'un réseau de Petri

- ***P = {P1, P2, P3, } et T = {t1, t2,t3}***

Un marquage[7] d'un réseau de Pétri est une application de **P** dans **N**. Si **M** est un marquage d'un RdP, **M(P)** est le marquage de la place **P** et est représenté par la présence de **M(P)** points ou marques à l'intérieur du cercle symbolisant la place **P**. Ces points sont aussi appelés jetons. Un réseau marqué est donc le couple :

- ***N = <R ; M>***
- où R est un *RdP* et M un ensemble fini non vide de marques.
- Un *RdP* peut être considéré comme un graphe orienté biparti dont les arcs sont values : le graphe d'un réseau est un quadruplet :
- ***G = <P, T ; ϒ, V>***où :
  - **P** est un ensemble fini de places,
  - **T** est un ensemble fini de transitions.
  - ϒ est défini tel que :
    - ***∀p ∈P,ϒ(p) = {t ∈T | Pré (p,t) > 0} ;***
    - ***∀t ∈T,ϒ(t) = {p ∈P | Post (p,t) > 0} ;***
  - **V** est la évaluation de tous les arcs tel que :
    - ***∀p ∈P, ∀t ∈T, V(p,t) = Pré(p,t) et***
    - ***V(t,p) = Post(p,t).***

Une transition t est dite déclenchable pour un marquage ***M*** si, et seulement si, pour tout p d'entrée : ***P(p) _ Pré(p,t),*** c'est-à-dire si toute place d'entrée de cette transition possède un nombre de marques au moins égal à la valeur de l'arc reliant chacune de ces places à la transition t.[1]

2.3.3. Le langage Estelle

Estelle (Extended State Transition Language) est une de deux techniques de description formelle dénies par l'ISO pour la description des protocoles de communication. La syntaxe et la sémantique du langage sont bien définis dans des documents proposés par l'ISO, mais il n'y a pas un outil logiciel standard associé avec le langage. Cependant il existe plusieurs outils développés par des tiers (EDT, PetDingo , ESTIM , XEC) qui acceptent le langage Estelle ou des dialectes.

Estelle permet de modéliser le système en utilisant une structure hiérarchique de modules, où chaque module peut être décomposé en sous-modules plus simples. Les modules communiquent à travers des points d'interaction et les messages sont véhiculés par des canaux de communication. [18].

Le comportement dynamique de chaque module, où qu'il soit dans la hiérarchie, peut être décrit par un automate étendu communicant. Les principaux éléments du langage Estelle sont:

Module: Les modules dans ce langage représentent explicitement des machines à états finis. La notation permet de définir l'ensemble d'états du module, les transitions et les messages échangés parmi les différents automates qui font la structure du système.

Point d'interaction: Un point d'interaction est le point d'accès d'un module. En Estelle le seul type de communication supporté correspond au type asynchrone. Donc chaque point d'interaction a une queue ordonnée associée où les messages d'entrée sont stockés jusqu'à sa récupération explicite.

Canal: Les canaux servent comme lien explicite entre deux, et seulement deux points d'interaction. Les canaux sont des entités déclarées explicitement dans la spécification. Elles n'ont pas de capacité de stockage de messages et sont indifférentes au contenu qu'elles portent. Sa fonction principale est de permettre de définir la structure du système à modéliser et de représenter les mécanismes d'échange de données [27].

Nous présentons un exemple de transition *Estelle*

```
from state0to state1
when data-in(p0,p1)
provided guard
begin
var:=p0;
if var = 10then
q0:=p1;q1:=p1;
else q0:=0; q1:=0;
output data-out(q0,q1)
end;
```

2.3.4.Le langage LOTOS

LOTOS [15] vient de l'acronyme anglais Language of Temporal Ordering Specification, il est standardisé par l'organisation internationale de standards dans la norme8807. Tout comme Estelle, LOTOS est un langage conçu pour modéliser des systèmes parallèles ou distribués, mais l'approche pour la modélisation et la manipulation formelle est différent.

En termes généraux, le langage LOTOS divise la modélisation du système en deux parties séparées mais qui ensemble créent un comportement cohérent:

- Modélisation des données: pour représenter les données et les opérations qui leur sont habituellement associées, LOTOS utilise une notation basée sur le langage de représentation de types de données abstraits ActOne. ActOne est un langage spécialement conçu pour représenter à un niveau fondamental les données, tant ses possibles valeurs que les opérations qui leur sont applicables.
- Modélisation [28] du processus d'exécution; LOTOS utilise une sémantique opérationnelle inspirée directement du langage d'algèbre de processus CCS. L'approche de CCS (et LOTOS) est d'exprimer le comportement du système en représentant la séquence des événements visibles par un observateur externe.

Exemple :

Il y a des types de données : Natural Number

• Il y a des ports de communication : port

• Il y a des processus : P1, P2

• Il y a des actions : port ! 0, port ? x : Nat , port ! succ(x).

```
specificationExemple : noexit
libraryNaturalNumberendlib
behavior
P1[port] | [port]| P2[port]
where
process P1 [port] : noexit :=
port ! 0 ; P2[port] (* P1 envoie 0 puis se comporte comme P2 *)
endproc (* P1 *)
process P2 [port] : noexit :=
port ? x : Nat ; port ! succ(x) ; P2[port] (* P2 reçoit x, envoie x+1, et recommence ... *)
endproc (* P2 *)
endspec (* Exemple *)
```

2.3.5.Le langage Promela[14]

. *Promela (PROcess Meta LAnguage)* est un langage impératif qui ressemble au langage C mais enrichi de quelques primitives de communication. Une spécification *Promela* peut contenir trois types d'objets [13] :

- Les processus, qui sont des objets globaux. Un processus peut en activer d'autres à l'aide du mot clé *run*. Un modèle *Promela* peut contenir plusieurs processus mais il doit en contenir au moins un. Dans l'exemple ci-dessous, on définit un processus qui permet l'affichage d'un message *(Hello world')* et on montre son initialisation au niveau de la partie *init*.

```
proctypeprocessus_declaration()

{ printf(UHello world! \n)

init{runprocessus_declaration();
```

• Les variables, qui peuvent être de cinq types de données *(bit, bool, short, byte et int)* globales ou locales. *Promela* fournit aussi la possibilité de créer des structures et des tableaux (d'une seule dimension). La définition ci-dessous déclare une structure contenant un champ entier et un autre booléen.

```
typedeftype_structure
        intvar_int;
        boolvar_bool;
```

• Les canaux de communication, qui sont utilisés pour le transfert de messages entre les processus. La communication via ces canaux est généralement asynchrone, mais on peut également définir un port rendez-vous pour la communication synchrone. Les champs d'un message ne peuvent pas être des tableaux. Une instruction d'envoi est exécutable si le canal n'est pas plein alors que celle de réception est exécutable si le canal n'est pas vide. Les messages sont insérés et retirés dans un ordre *FIFO*. Par exemple, la déclaration ci-dessous définit un canal pouvant contenir quatre éléments, entiers.

```
channom_channel [4] of tint};
```

Promela offre également la possibilité de définir des séquences d'exécution atomiques (qui ne peuvent pas être interrompues) qui peuvent être déterministes ou non, des structures de sélection avec des gardes et des boucles, etc. L'exemple ci-dessous définit un *processus_moniteur* qui, de façon répétitive, à partir de la valeur du premier

paramètre du message *(varl)*, décide s'il va effectuer une opération d'addition ou de multiplication *(sur résulta L, avec var2 le deuxième élément du message)* et cela en invoquant d'autres processus. Le timeout assure que si aucun autre processus n'est actif et ne peut envoyer un message à *processus_moniteur*, alors son exécution se terminera.

```
chan transfert [4] of {boo!, inti; intresultatresultat = 0 i

proctypeprocessus_moniteur() { do .. transfert? varl, var2 -> if
                                        .. varl == true -> run processus_add(var2); :: else
                                        -> run processus_mul(var2); fi ;
        .. timeout -> break i od

proctypeprocessus_add(intvar) { resultat = resultat + var; }

proctypeprocessus_mul(intvar) { resultat = resultat * var; }

}
```

2.3.6. Spécification des propriétés à vérifier

Avant de commencer la vérification proprement dite, il faut identifier ce qu'on souhaite vérifier. La littérature parle généralement de propriétés comme la sûreté, la vivacité, l'équité, etc, Pour spécifier les propriétés, on utilise un langage particulier (déclaratif) adapté au choix du formalisme de modélisation effectué dans première étape.

Pour énoncer formellement des propriétés comportementales des systèmes, on peut utiliser, entre autres, la logique modale ou la logique temporelle. En général, on utilisera une logique temporelle.

Les logiques temporelles font partie d'une famille de langages «spécialisés dans les énoncés et raisonnements faisant intervenir la notion d'ordonnancement dans le temps» [26] (le temps est discret dans ce cas). Elles offrent des «opérateurs calqués

sur des constructions linguistiques comme les adverbes, les temps de la conjugaison des verbes, etc., de sorte que les énoncés en langue naturelle et leur formalisation en logique temporelle soient assez proches» [25].

Il existe plusieurs logiques temporelles. «LTL *(Linear Temporal Logic)* et CTL *(Computation TreeLogic)* sont deux des logiques temporelles les plus utilisées dans les outils de vérification formelle» [25]. «Ces deux logiques sont interprétées sur des structures de Kripke» et sont donc basées sur les états (par opposition aux logiques temporelles basées sur les actions et interprétées sur des systèmes de transitions).

Bien que l'objet de notre mémoire porte principalement sur la logique LTL sur laquelle est basé Orchids, on présentera dans ce qui suit les deux logiques avec des exemples des propriétés qu'elles permettent d'exprimer.

2.3.6.1.La logique modale

Une logique modale est une logique qui supporte les concepts de possibilité, d'existence et de nécessité. Une telle logique est utilisée pour exprimer des propriétés qui sont locales par rapport à l'état courant. Les opérateurs de base d'une telle logique sont:

- La possibilité: <action> p énonce qu'il est possible d'exécuter action et ensuite d'atteindre un état qui satisfait p.
- La nécessité: [action] p énonce que quand action se produit, l'état dans lequel on arrive satisfait nécessairement p.

- On peut déduire d'autres concepts à partir de la négation de ceux cités en haut:
  - L'impossibilité: Non<> qui est l'opposé de la possibilité
  - La contingence: Non [] qui est l'opposé de la nécessité.

L'exemple ci-dessous (figure 6), montre un modèle (Modèle 1) où à partir de l'action « pièce », il est toujours possible de faire l'action « biscuits ». Par contre, cela n'est pas possible dans Je cas du Modèle 2, à cause du choix non déterministe effectué au début.

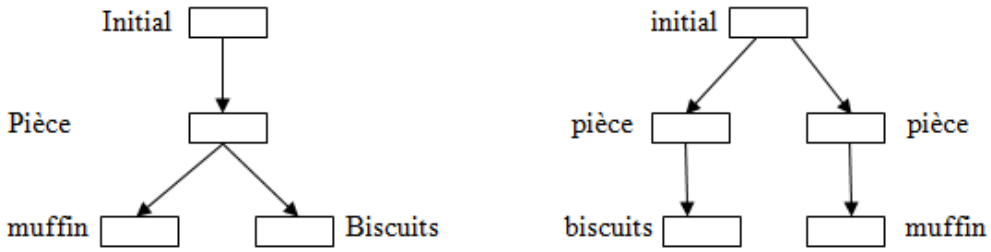


Figure 6 Exemple

En terme de traces (séquences d'actions), les deux modèles sont considérés comme ayant le même comportement puisque les deux génèrent l'ensemble suivant de traces:
{pièce; muffin, pièce; biscuits}. Or, ces deux modèles n'ont pas nécessairement les mêmes propriétés modales, par exemple: [pièce] <muffin>True est vraie à l'état initial du Modèle 1, alors qu'elle ne l'est pas à l'état initial du Modèle 2.
Donc, la logique modale s'avère intéressante si on a besoin d'exprimer des propriétés locales [24].

2.3.6.2. La logique temporelle

Le comportement souhaité d'un système est généralement formulé par un ensemble de propriétés exprimées en logique temporelle. La logique temporelle est une forme de logique spécialisée dans les énoncés et raisonnements faisant intervenir la notion d'ordonnancement dans le temps [25]. C'est dans, que *A. Pnueli* a proposé pour la première fois de l'utiliser pour la spécification formelle des propriétés comportementales des systèmes. En ajoutant aux connecteurs et aux opérateurs de la logique classique des opérateurs temporels, la logique temporelle permet d'énoncer formellement des propriétés sur les enchaînements d'états, c'est-à-dire sur les états qui composent les exécutions du système étudié.

Ces logiques permettent de diviser 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.

Plusieurs classifications ont été pro posées pour les logiques temporelles, nous avons choisi de présenter la plus répandue à savoir celle qui repo se sur la manière de représenter le temps.

2.3.6.2.1. Les logiques de temps linéaire (LTL, PLTL, ...) : elles permettent de spécifier des propriétés formulées sur des séquences d'états. Une telle logique n'est pas susceptible d'exprimer les exécutions alternatives qui sont générées aux instants où une sélection non déterministe est permise. Elle définit donc le comportement prévu du système en spécifiant l'unique futur possible (temps linéaire). Les *combinateurs temporel susuels* sont les suivants [25]:

- **X *p* : *p***est vraie à partir de l'état suivant *(Next).*
- ***FP* : *P***est vraie à partir d'un état futur ou de l'état actuel *(Final/y).*
- **G *p* : *p***est vraie dans tous les états suivants, incluant l'état actuel *(Global/y).*
- ***pl U p2* : *p2***est vraie à partir d'un état *q* (futur ou actuel) et *pl*est vraie pour tous les états qui précédent l'état *q (Until).*

On peut aussi combiner ces opérateurs, par exemple:

- ***GF p* : *p***est vérifiée une infinité de fois dans le futur.
- ***FGP* : *P***est toujours vérifiée à partir d'un certain état.

Exemple

Pour illustrer l'utilisation de la logique temporelle linéaire dans la formalisation des problèmes pratiques, nous donnons l'exemple du publiphone qui consiste à modéliser les actions qu'un utilisateur exécute pendant l'opération de tentative de téléphoner jusqu'à la fin de la communication. Donnons, d'abord, la procéduretéléphoner :

- L'utilisateur doit décrocher le téléphone.
- Le système affiche insérer une carte.
- L'utilisateur insère une carte.
- Le système affiche le nombre d'unités restantes.
- L'utilisateur compose son numéro de téléphone avec la possibilité de le corriger pour obtenir le bon numéro dans un délais de 2 secondes.
- L'utilisateur communique avec son correspondant, tant qu'il lui reste des unités.

- Si la carte est épuisée, l'utilisateur a 10 secondes pour la changer.
- Lorsque la communication est terminée, l'utilisateur raccroche le téléphone.
- Le système affiche retirer la carte.
- L'utilisateurretiresa carte.

Ensuite, formalisons les propositions atomiques :

- u_décrocher : l'utilisateur décroche le téléphone.
- sys_affiche(message) : le système affiche le message.
- u_insérer_carte : l'utilisateur insère une carte.
- u_compser_no : l'utilisateur compose le numéro.
- u_correction_no : l'utilisateur corrige le numéro.
- numéros_correct : le numéros est correct.
- u_communiquer : l'utilisateur communique avec son correspondant.
- u_changer_carte_vide : l'utilisateur change sa carte qui est épuisée.
- u_raccrocher : l'utilisateur raccroche le téléphone.
- u_retirer_carte : l'utilisateur retire sa carte.
- ltl_téléphoner : établit la clause de l'activité téléphoner.

Enfin, on obtient la formule [1] :

u_decrocher ^ ***X*** sys_affiche(*Insérer carte*) ^

XX [(u_insrer_carte ^ ***X*** sys_affiche(nb_units_restantes)) ^

(***X*** u_composer_no _ ***F*** (u_correction_no***U***numros_correct)) ^

((***X*** u_communiquer _ ***F*** (u_changer_carte_vide***U***unites(carte) > minimum))

U (u_raccrocher ^ u_retirer_carte))])ltl_telephoner

Pour téléphoner, l'utilisateur doit décrocher le téléphone, puis, dans l'état suivant ***(opérateur X)***, il doit insérer sa carte. Ensuite, il compose son numéros avec la possibilité ***(opérateur F)*** de le corriger. Si le numéros est correct, l'utilisateur pourra, alors communiquer avec la possibilité de changer sa carte. La communication dure jusqu'à ce qu'il raccroche le téléphone et retire sa carte.

On a pu exprimer la notion de boucle à l'aide de l'opérateur ***U***. Par exemple, on corrige le numéros de téléphone un nombre infini de fois jusqu'à obtenir le bon numéros. En plus, l'opérateur ***X*** précise le séquencement des évènements :

l'utilisateur compose son numéros juste après que le système affiche le nombre d'unités restantes.

2.3.6.2.2. La logique de temps arborescent CTL (Computation TreeLogic):

Elle permet de spécifier des propriétés d'états. Ainsi, elle considère plusieurs futurs possibles à partir d'un état du système. Les connecteurs temporels sont X, G, *F* et *U* de la logique LTL, et des quantificateurs de chemin (combinateurs exprimant le côté arborescent) : *A p,* qui énonce que toutes les exécutions partant de l'état courant satisfont la propriété *p; E p,* qui énonce qu'à partir de l'état courant, il existe une exécution satisfaisant *p.* Ces connecteurs s'utilisent par paire avec les autres, comme suit:

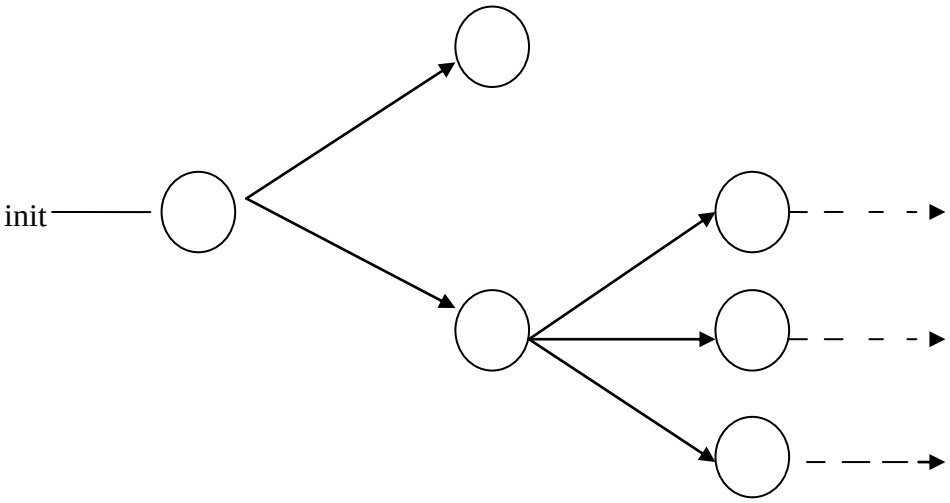


Figure 7 logique temporelle

o*EF P* énonce qu'il est possible d'atteindre un état où *p* est vérifiée.
o*AF P* dit que pour toute exécution, **il** existe un état où *p* est vérifiée.
o*AG P* dit que *p* est vraie partout (futur).
o*EG p* énonce qu'il existe une exécution où *p* est toujours vraie.
o*AX P* dit que tous les états immédiatement successeurs satisfont *p.*
o*EXp* énonce qu'il existe une exécution dont le prochain état satisfait *p.*

Le choix d'une logique dépend des spécificateurs de propriétés et des propriétés qu'on veut exprimer. Mais lorsqu'on utilise un outil de vérification de modèles, **il** faut respecter la logique qu'il utilise et spécifier les propriétés dans cette logique.

Exemple :

On reprend l'exemple du publiphone pour illustrer comment on modélise le problème dans la logique *CTL* .

La clause ltl_téléponer sera ctl_téléphoner. La formalisation en *CTL* donne :

u_decrocher ^ u_inserer_carte ^

AX [(u_composer_no ^ ***EF*** u_reprise_sur_erreur) ^

AX ((u_communiquer ^ ***EF***changer_carte_vide)***U***

(u_raccrocher ^ u_retirer_carte))]) ctl_telephoner

Pour téléphoner, l'utilisateur doit décrocher le téléphone et insérer une carte (l'ordre de ces deux actions n'intervient pas). Puis, dans tous les états possibles ***(opérateur AX)***, il composera le numéros, et pour terminer il raccrochera le téléphone et retirera sa carte.

Pour composer le numéros, l'utilisateur pourra ***(opérateur E)*** se ramener à le corriger ***(opérateur F)***. Dans tous les états possibles, l'utilisateur communique au téléphone jusqu'à ***(opérateur U)*** ce qu'il raccroche et retire sa carte avec une possibilité de l'échanger ***(opérateur EF)***. ***EF*** se traduit dans notre cas par éventuellement au lieu de finalement pour "eventually".

On remarque que cette version n'est pas aussi précise que la précédente. De plus, on donne des informations de plus en plus floues. Remarquons, par exemple, l'absence de l'indicateur de temps qui précise la durée pour changer une carte téléphonique.

2.4. Conclusion

Dans ce chapitre, on a présenté des différentes techniques utilisées pour la spécification des systèmes, notamment *FSM* .Ainsi que, les *réseaux de* Petri et des langages formels basés sur l'algèbre de processus comme Estelle, LOTOS et Promela qui constituent d'autres formalismes pour la spécification des systèmes concurrents ou distribués. Enfin on a discuté les différents types de logique utilisés pour la vérification de propriétés d'un système.

Chapitre3

Etude du protocole Stop and Wait

3.1. Introduction

Dans ce chapitre, nous allons proposer une description formelle pour le protocole **«** *Stop and Wait***»** avec P*romela,* ensuite nous allons vérifier des propriétés du modèle obtenu et présenter le résultat de la vérification avec l'outil *Spin* par ce que Les modèles sont décrits en Spin en Promela, le Process Meta Language. La syntaxe de ProMeLa s'inspire de celle du langage C : par exemple les expressions arithmétiques et logiques utilisent la même syntaxe qu'en C (et Java, du coup).

Nous allons décrire les tâches effectuées par le protocole en question, et présenter son fonctionnement.

3.2. Vérification d'un algorithme d'exclusion mutuelle

Nous prenons l'exclusion mutuelle de l'algorithme de *Peterson* comme un exemple. Dans le problème d'exclusion mutuelle, il existe une collection de processus asynchrones. Si deux ou plusieurs processus tentent d'exécuter le même code et accéder aux mêmes données, il existe un problème potentiel où ils peuvent se recouvrir. Le problème de l'exclusion mutuelle est le problème de la restriction de l'accès à une section critique dans le code à un seul processus à la fois, en supposant que l'indivisibilité de la lecture et d'écriture des instructions. (Le problème disparaît si l'on peut supposer un tout indivisible test-and-jeu d'instructions.) Une première solution a été d'abord publiée par Dijkstra.

Un algorithme d'exclusion mutuelle

```
Boolean array b(0;1) integer k, i, j,
comment process i, with i either 0 or 1 and j = 1-i;
C0:   b(i) := false;
C1:   if k != i then begin
C2:   if not (b(j) then go to C2;
      else k := i; go to C1 end;
      else critical section;
      b(i) := true;
      remainder of program;
      go to C0;
      end
```

Le modèle de la solution avecPromela, et la preuve de l'exactitude de l'algorithme.

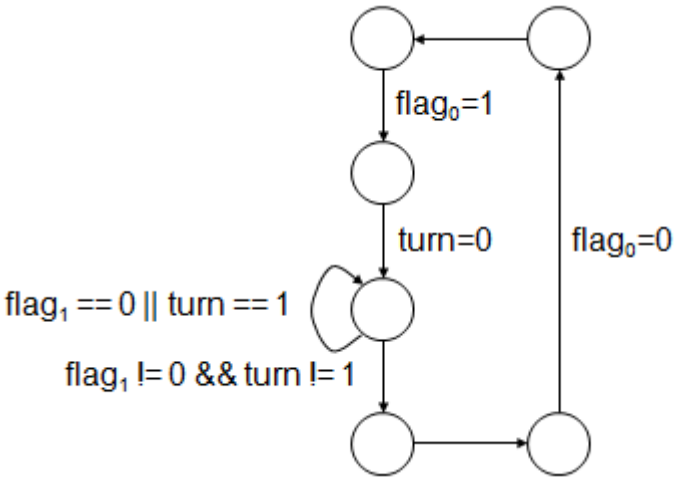


Figure 8 Modèle d'automate

Ce problème continue d'être populaire.Par exemple pour deux processus, une solution particulièrement élégante a été publiée par Peterson.

En Promela la solution peut être modélisée comme suit :

La variable prédéfinie_pid contient l'identifiant du processus en cours, donc ce variable contient la valeur 0 ou 1.

```
1 #define true
 2 #define false          0
 3 boolflag[2];
 4 bool turn;
 5 active [2] proctypeuser()
 6 {     flag[_pid] = true;
 7       turn = _pid;
 8       (flag[1-_pid] == false || turn == 1-_pid);
 9crit:skip;     /* critical section */
10       flag[_pid] = false
11 }
```

L'algorithme de Peterson en Promela

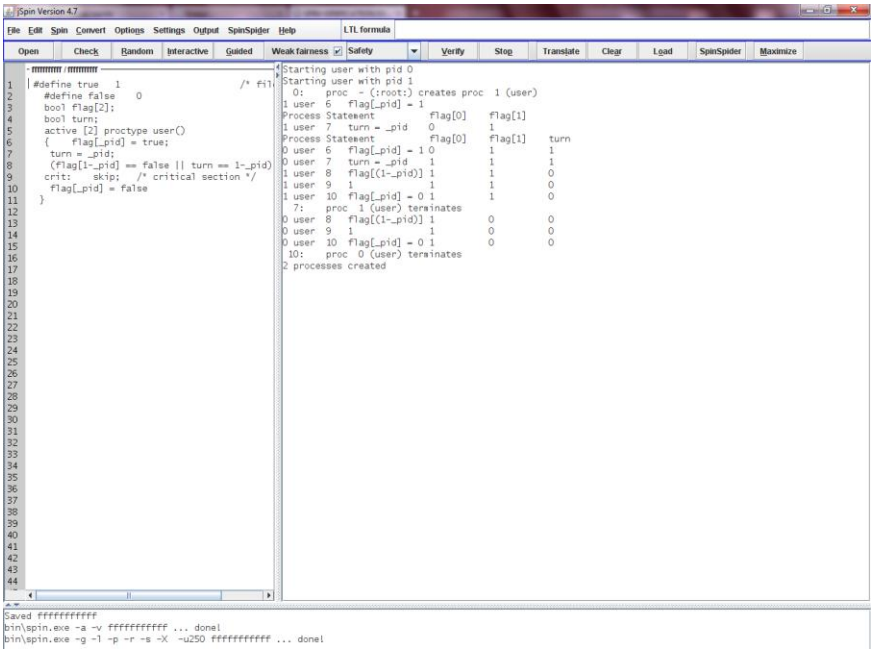


Figure 9 Résultat de validation

3.3. Le protocole Stop and Wait

Le but de la suite de ce travail est d'étudier et de vérifier des propriétés sur le protocole de communication Stop and Wait . Ce protocole dit de niveau "liaison de données" de la pile protocolaire du modelé OSI [23] (figure 10), assure quelle transfert de l'information (une suite de paquets de données) s'effectue dans l'ordre, sans perte de données et à un débit raisonnable. Plus précisément nous voulons étudier un protocole de communication entre une machine A (émetteur) qui veut transférer une certaine quantité d'informations à une machine B (récepteur) à travers un support physique (canal). Ce support physique transporte les données avec un certain taux d'erreur, c'est-à-dire qu'une donnée peut être mal transmise ou perdue de sorte que la machine B est incapable de la comprendre ou de la recevoir. La fonction élémentaire du protocole est donc d'assurer que, même en cas d'erreur (due au support physique), les données ne seront pas perdues.

Cette section est consacrée à la présentation du protocole "Stop and Wait", nous faisons une description du protocole et montrons son intérêt dans un réseau de communication.

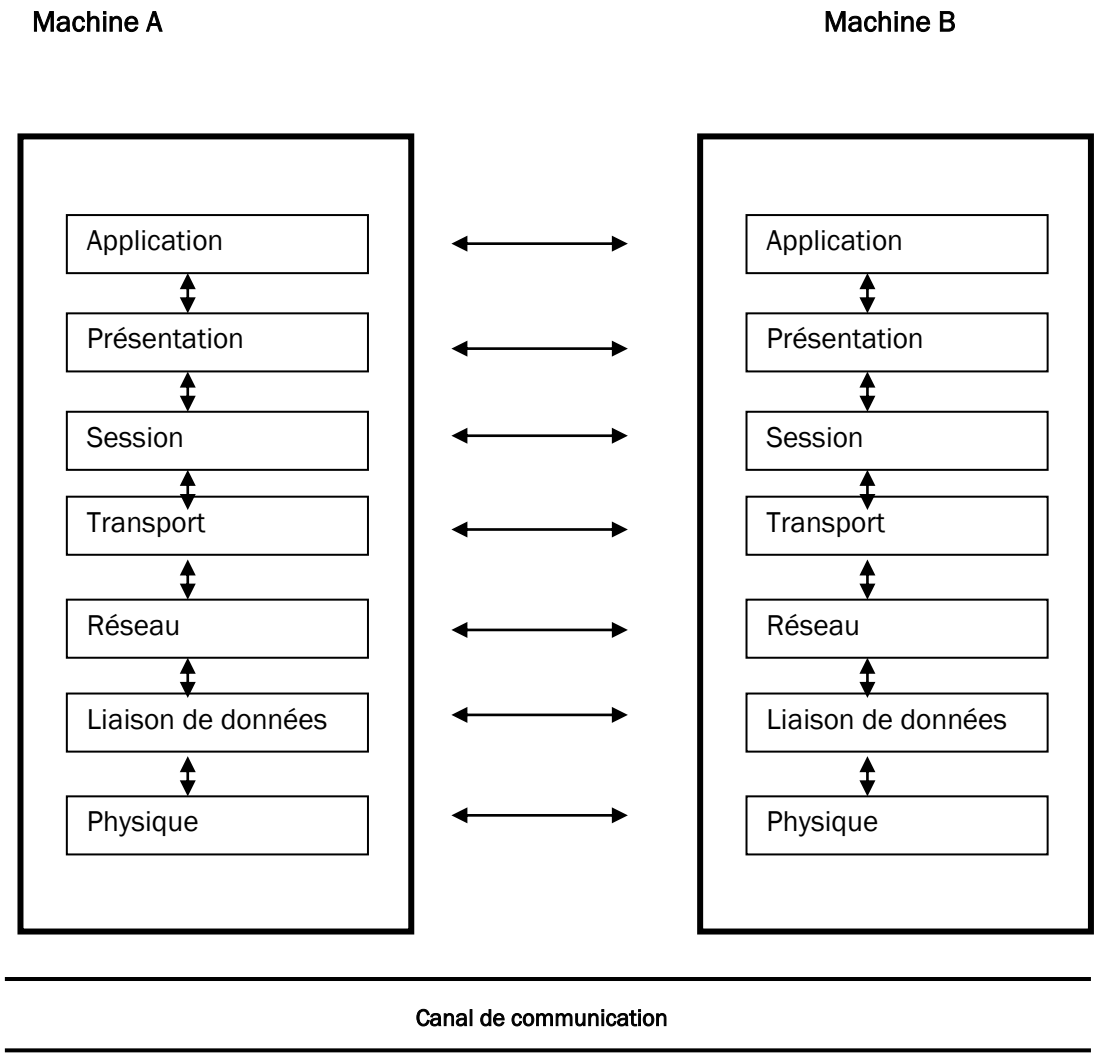


Figure 10 . Pile protocolaire du modèle OSI

3.3.1. Principes des protocoles de la couche Liaison de données

Le rôle de la couche liaison de données est de fournir à la couche réseau un canal de communication fiable et efficace comme si la machine source et la machine destination étaient reliées entre elles par un "lien" parfait [23]. Cependant, en réalité il y a des erreurs de communication entre les deux machines, un débit fini et un délai de propagation non nul. L'acheminement des données entre les couches réseaux d'une machine A et d'une machine B peut se faire suivant 3 types de services :

- mode datagramme (sans connexion et sans acquittement),

- mode sans connexion avec acquittements (offre plus d'efficacité et de sécurité),
- mode avec connexion et acquittements (offre encore plus de services grâce à la négociation d'options).

La couche liaison peut être représentée en 2 sous couches :

- LLC (Logical Link Control),
- MAC (Medium Access Control).

Dans un LAN (réseau local), en raison de l'environnement contrôle : câbles (de qualité) et équipements dans un environnement non "agressif" et des faibles distances qui entraînent de faible taux de perte, de l'homogénéité des machines (contrôle de flux, machines de puissance comparable) la sous couche LLC est minimale. La gestion des erreurs est sous la responsabilité de la couche transport (pas de duplication qui pourrait entraîner des problèmes de double demande de retransmission).

3.3.2.Description du Protocole "Stop and Wait"

Lorsqu'une machine A veut transférer une donnée (un fichier) à une autre machine B, la donnée est d'abord décomposée en plusieurs unités (appelée paquets) et le protocole de communication (sur les deux machines) contrôle si les paquets ont été bien transférés. La gestion du transfert des paquets s'effectue à l'aide d'accusés de réception (noté ACK), qui constituent une information de contrôle. Cette information est en général ajoutée à l'information brute (DATA), par exemple en tête de chaque paquet. Le récepteur doit donc envoyer à l'émetteur des messages de bonne réception ou de non-réception. Ces messages ne contiennent pas d'information brute, ce sont uniquement des messages de commande.

La communication s‘effectue donc dans les deux sens sur le canal. Le protocole, au moyen de la gestion des réponses (ACK du récepteur), doit garantir le séquencement (arrivée des paquets dans l'ordre) et la détection des erreurs (pertes de messages) [23], [23]. Nous allons présenter les grandes lignes de ce protocole à l'aide de deux procédures ; l'une est associée à l'émetteur et l'autre est associée au récepteur. Ensuite on donnera plus de détails sur la gestion des paquets et des acquittements.

Les deux algorithmes suivants résument d'une manière simple le fonctionnement du protocole "Stop and Wait"

Procédure associée à l'émetteur :

L'émetteur envoie un message de données au récepteur
Et se met en attente de recevoir un ACK
***Si** Un ACK est reçu avant une période déterminée appelée Time Out*
alors
L'émetteur vérifie si l'ACK reçu contient des erreurs
Si** Des erreurs sont détectées **alors
L'émetteur retransmet le même message de données
Et se met en attente d'un nouvel ACK
***Sinon** (l'ACK ne contient pas d'erreurs)*
L'émetteur transmet le prochain message de données dans sa file d'attente
***Sinon** (l'ACK n'a pas été reçu au bout Time Out unite de temps)*
L'émetteur retransmet le même message de données
et se met en attente d'un nouvel ACK

Procédure associée au récepteur :

Le récepteur se met en attente de recevoir un nouveau message de données
Si** Un nouveau message arrive **alors
Le récepteur vérifie si le message reçu contient des erreurs
Si** Des erreurs sont détectées **alors
Le message de données est ignore
et le récepteur continue a être en état d'attente
***Sinon** (le message reçu ne contient pas d'erreurs)*
Le récepteur initie la transmission d'un message ACK

Informations de contrôle

- Séquencement : les paquets reçus par le récepteur doivent être livrés dans le bon ordre.
- Fenêtre de transmission : le nombre k de tampons de stockage des paquets en attente est fixe (fini). La valeur de k est appelée fenêtre de transmission.
- Numérotation : la numérotation des paquets n'est pas infinie. Cette numérotation tient sur un nombre de bits fixe x, soit une numérotation des paquets modulo 2^x. La fenêtre de transmission k est dans tous les cas plus petite que 2^x.
- Numéro de séquence en émission (Ns) et numéro de séquence en réception
- (Nr) : Ns représente le numéro (modulo 2^x) du dernier paquet envoyé par l'émetteur et Nr représente le numéro du dernier ACK envoyé par le récepteur.
- Tf (Data frame transmission time) : le temps nécessaire pour la transmission d'un message de données.
- Ta (ACK transmission time) : le temps nécessaires pour la transmission d'un message d'acquittement.
- Tprop (Time propagation) : le temps de propagation d'un message dans le canal de communication.
- Tproc (Processing time) : le temps nécessaire pour la détection des erreurs dans le message reçu.

Tout (Time out) : le temps d'attente avant la retransmission d'un message.

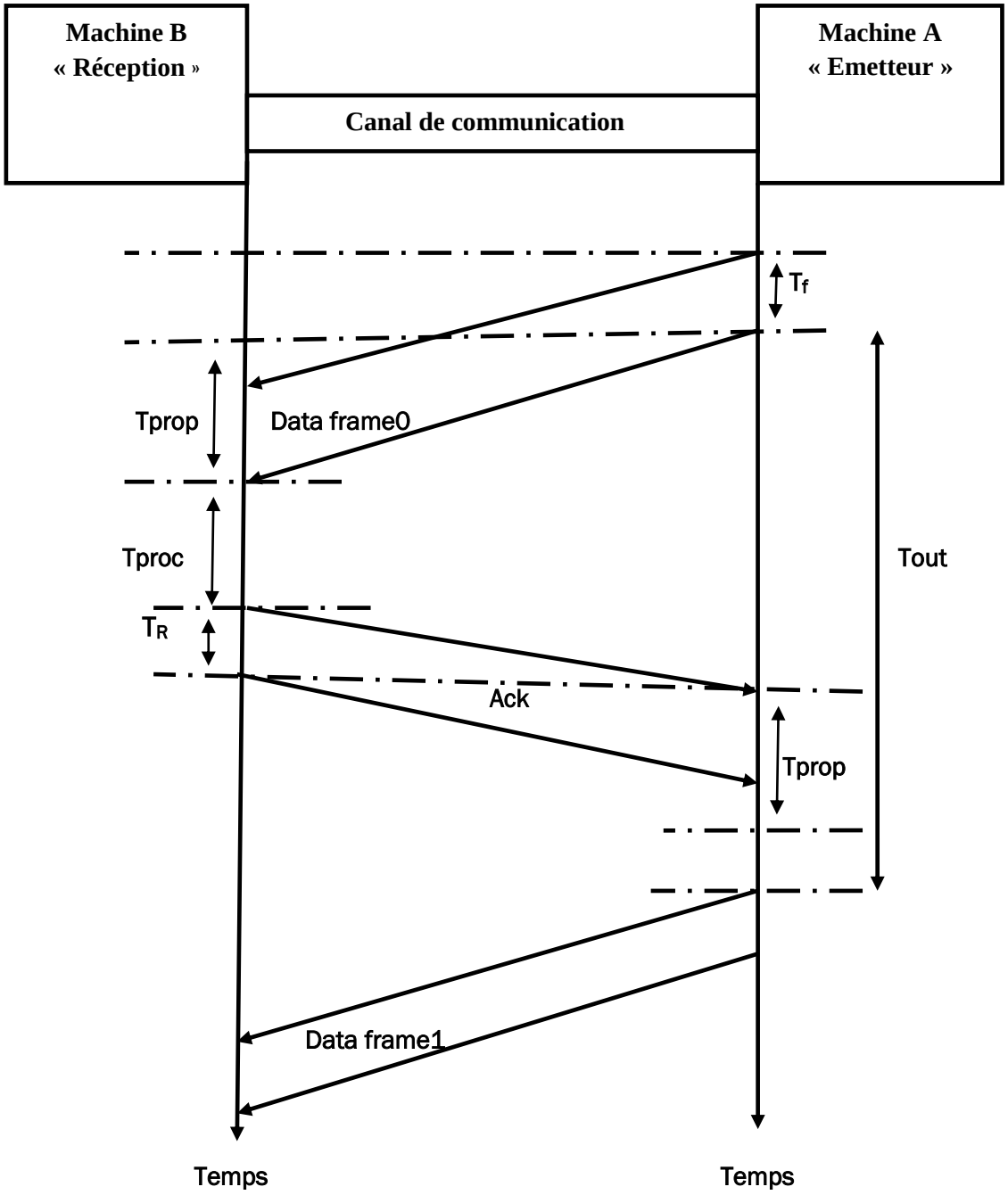


Figure 11 Informations de contrôle du protocole "Stop and Wait"

3.3.4. Modélisation et vérification du protocole

L'automate de modèle du protocole "**Stop and Wait**" est illustré dans la figure 12.

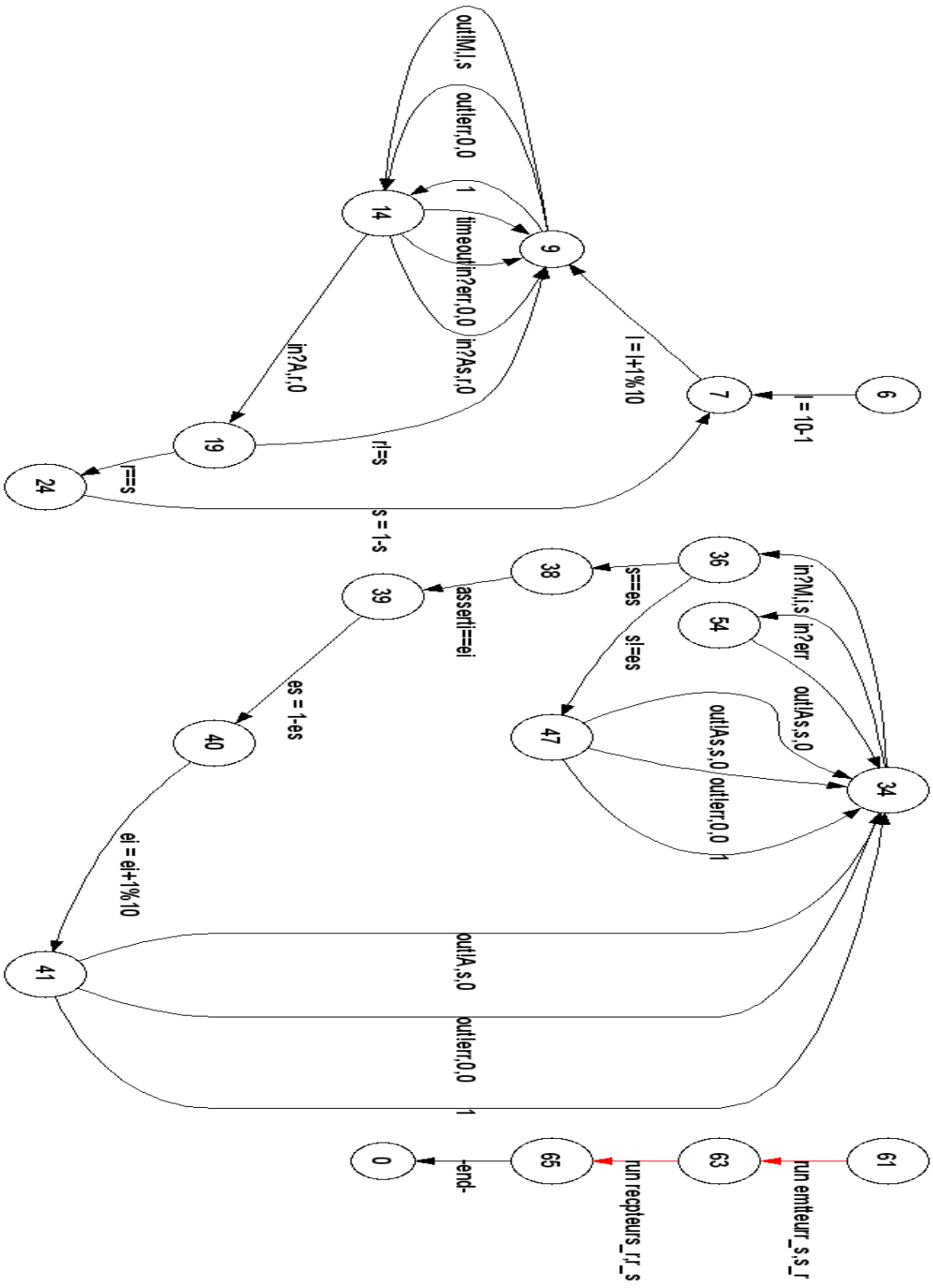


Figure 12 Automate généré par SpiderSpin

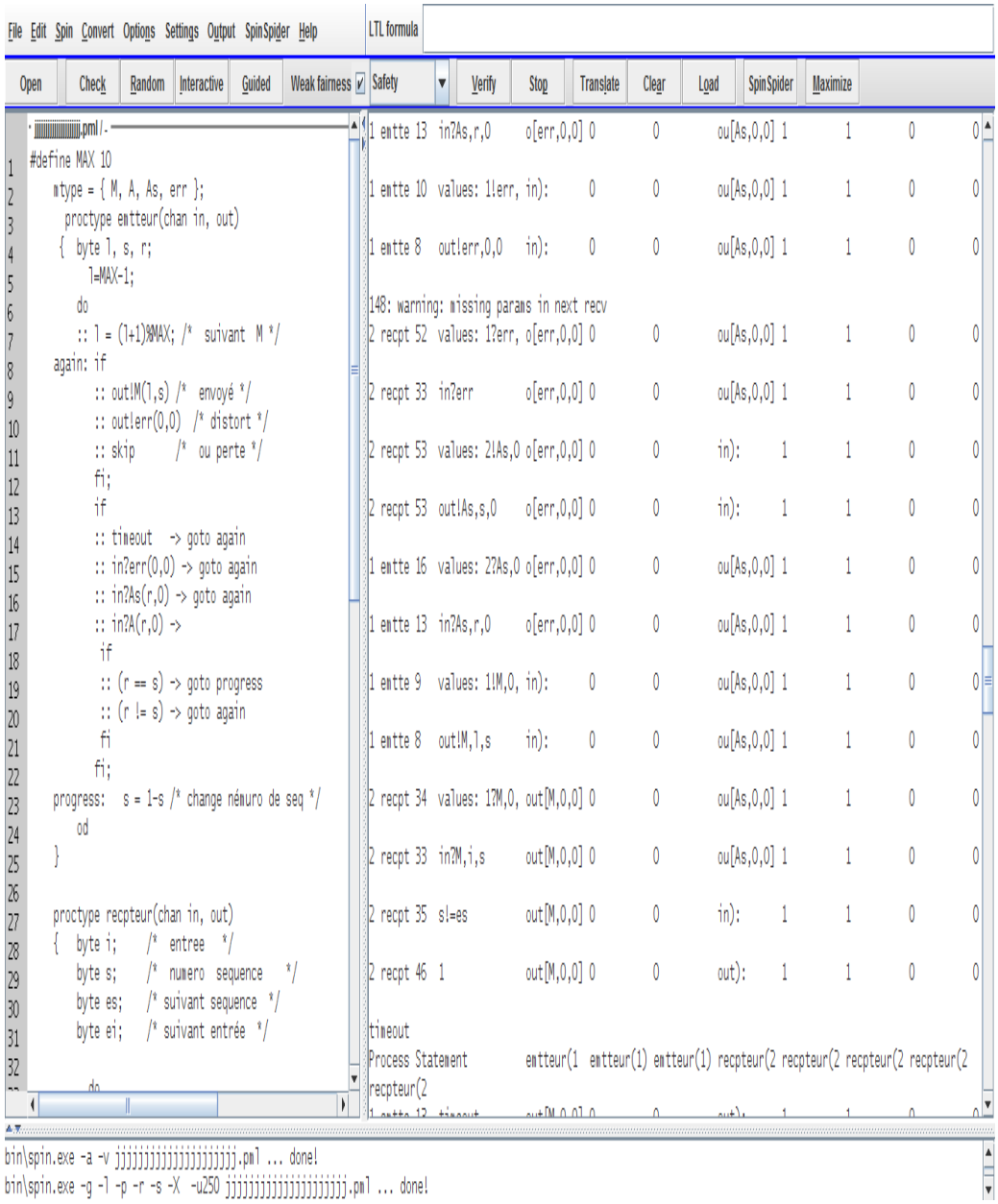
3.3.5. Résultat de validation

Nous avons montré que réseau propriétés protocoles peuvent être formellement vérifiée à l'aide SPIN model-checker en prouvant l'exactitude.

La Propriété qui peut être prouvée pour le stop & wait avec spin :

- Dans le cas d'un milieu avec possibilité d'erreur mais sans double chevauchement, il est possible de prouver que le *S &W* préserve l'ordre des données, c.-à-d.
    - La chaîne des messages reçus ***est dans le même ordre*** de la chaîne des messages envoyés
    - Moins évidemment les messages envoyés et pas encore arrivés

- Le protocole du stop & wait a aussi la propriété d'être capable de retourner au fonctionnement normal après n'importe quelle situation d'erreur
- Il pourra cependant avoir envoyé à l'usager les mauvaises données .



3.4. Conclusion

Dans ce chapitre en premier lieu, on a présenté le problème d'exclusion mutuelle et la solution de Peterson pour ce problème. Ensuite, on a étudié le protocole de la couche liaison Stop &Wait, on a discuté son principe de fonctionnement, les messages utilisés, et on a proposé une spécification avec Promela, on a utilisé l'outil Spin pour valider notre proposition, et SpinSpider pour tracer l'automate des états possibles.

Conclusion générale

Ce mémoire s'inscrit dans le cadre de la vérification formelle de protocoles de communication, Pour cela, nous avons d'abord, étudié les différentes techniques de la vérification formelle existantes : à savoir : la preuve formelle et le model checking. Une étude comparative de ces deux approches, prenant en considération le contexte de la problématique, nous a permis de dégager la technique de model checking comme étant la plus adéquate pour la vérification des systèmes répartis. Ensuite, nous nous sommes penchés sur le model checking pour présenter : les classes de propriétés qu'il permet de vérifier, les outils les plus utilisés dans la vérification formelle, en suite, nous avons étudié les différentes techniques utilisées pour la modélisation et description formelle des systèmes, à la fin on a proposé un modèle pour le protocole Stop &Wait, on a valider notre specification avec Spin.

Comme perspective, il serait intéressant d'étude des protocoles de la couche MAC de réseaux sans fils comme WiFi, ZigBee ou BlueTooth.

Annexe

Code source

```
#define MAX        10
mtype = { M, A, As, err };
proctypeemtteur(chan in, out)

    {   byte l, s, r;
        l=MAX-1;
        do
        :: l = (l+1)%MAX; /*  suivant  M */
again: if
::out!M(l,s) /*  envoyé */
::out!err(0,0)  /* distort */
:: skip       /*  ou perte */
fi;
if
:: timeout   ->goto again
::in?err(0,0) ->goto again
::in?As(r,0) ->goto again
::in?A(r,0) ->
                if
                :: (r == s) ->goto progress
```

```
                :: (r != s) ->goto again
                fi
fi;
progress:   s = 1-s       /* change némuro de seq */
        od
   }

proctyperecpteur(chan in, out)
   {   byte i;        /*  entree   */
       byte s;              /*  numero  sequence    */
       byte es;      /* suivant sequence  */
       byteei;       /* suivant entrée  */

do
::in?M(i, s) ->
if
        :: (s == es) ->
assert(i == ei);
progress:           es = 1 - es;
ei = (ei + 1)%MAX;
if
::out!A(s,0)          /* envoye   */
::out!err(0,0)        /* distortion */
:: skip                                 /* ou perte */
```

```
fi
     :: (s != es) ->
if
::out!As(s,0)                          /* envoye   */
::out!err(0,0)                        /* distortion */
:: skip                                 /* ou perte */
fi
fi
::in?err ->
out!As(s,0)
od
      }

init {
        chans_r = [1] of { mtype,byte,byte };
        chanr_s = [1] of { mtype,byte,byte };
        atomic {
                runemtteur(r_s, s_r);
                runrecpteur(s_r, r_s)
        }
   }
```

Bibliographie

[1] Adamou M., « Contribution à la modélisation en vue d'une conduite des systèmes flexibles d'assemblage à l'aide des réseaux de Petri orientés objets ». Thèse de Doctorat, Université de Franche-Comté, 1997.

[2] « Atelier B », http://www.atelierb.societe.com/

[3] B. Alpern, F.B. Schneider, «Defining liveness», Distributed Computing, vol 21, n°4, p.181-185, 1985.

[4] C. A. R. Hoare. *« An axiomatic basis for computer programming »*, Communications of the ACM, 12 : 576-580, 1969.

[5] C.H. Pygott. *«Verification of VIPERS's ALU. Technical report, Divisional Memo (Draft) »*, the Royal Signals and Radar Establishment, 1991.

[6] http ://www.coq.inria.fr

[7] D. Câmara, F.Loureiro, F.Filali ,«Methodology for Formal Verification of Routing Protocols for Ad Hoc Wireless Networks»,*2007.*

[8] D. Déharbe, *« Une introduction au model checking et à SMV »* Journée Initiation Les outils pour la vérification, 2002.

[9] E. Clarke, O. Grumberg et D. Peled. 1999. « Model Checking ». Cambridge, Mass.: MIT Press.

[10] E. M. Clarke and J. M. Wing. *»Formal Methods: State of the Art and Future Directions»* .ACM Computing Surveys, December

1996.

[11] Fred Kroeger, *«Temporal Logic of Programs » , Monographs on Theoretical Science. Springer-Verlag*.

[12] Gerard J. Holzmann. «The Model Checker SPIN ». Software Engineering, 23(5):279_295, 1997.

[13] G.J. Holzmann. 2000. « Using Spin ». Plan 9 Programmer Manual Documents, pages: 353-382, Vita Nuova Holdings Ltd, York England. 2nd edition.

[14] G. Tremblay. 2003. *«Une introduction à la vérification de modèles »*. Université du Québec à Montréal. Séminaire du doctorat en informatique cognitive.

[15] G. Huet, G. Kahn, C. Paulin-Mohring, *« The Coq Proof Assistant A Tutorial »* 1999, Coq project.

[16] Hedidhouibi ,*«utilisation des réseaux de Petri a intervalles pour la regulation d'une qualité : application a une manufacture de tabac »*, thèse doctorat .2005.

[17] Henri Habrias and Marc Frappier .« Software Specification Methods », ISTE Ltd, 2006.

[18] Jean- François Monin. *«Understanding Formal Method »*. *Springer*, 2003.

[19] J. Crow, S. Owre, J Rushby, N. Shankar, M. Srivas, « *A Tutorial Introduction to PVS »*,Workshop on Industrial-Strength Formal Specification Techniques,1995.

[20] K. L. McMillan, *« Symbolic Model Checking An approach to the state explosion problem »*, CMU-CS-92-131, 1992

[21] Manuel BACLET,*« Applications du Model-Checking à des*

Problèmes de Vérification de Systèmes sur Puce » thèse doctorat. 2005.

[22] M. Daniels W. Yi, P. Pettersson. *«* Automatic Verification of Real-Time Communicating Systems By Constraint-Solving*»*. In Dieter Hogrefe and Stefan Leue, editors, *Proc. of the* th*Int. Conf. on Formal Description Techniques*, pages 223–238. North–Holland, 1994.

[23] Naim Aber, « Verification d'un protocole Stop-And-Wait : Combiner Model-Checking et Preuve de théorémes », Rapport de Stage,2009.

[24] Ouvrage collectif -Coordination Philippe Schenoeblen. 1999. *« Vérification de logiciels, techniques et outils du model-checking »*. Paris: Vuibert.

[25] P. Sehnoebelen, B. Bérard: M. Bidoit, F. Laroussinie, and A. Petit. *«Vérification de logiciels: techniques et outils du model-checking»*. Vuibert, avril 1999.

[26] saknicharfeddine ,*«* la logique temporelle»,2004.

[27] S. Chouali, « *Contribution du raffinement à la vérification de systèmes sous hypothèses d'équité* » thèse de doctorat. 2003.

[28] T. Bolognesi et E. Brinksma, *« Introduction to the ISO Specification Language LOTOS » ,* Computer Networks and ISDN Systems, vol. 14, no. 1, pp.25-59, 1990.

[29] T. Kropf. *«Introduction to Formal Hardware Verification» .Springer Verlag*, 1999.

[30] Y. Aït Ameur, *« Notes de cours sémantique »* 2004.