République Algérienne Démocratique et Populaire
الجمهورية الجزائرية الديمقراطية الشعبية
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
وزارة التعليم العالي والبحث العلمي
Université Chadli Bendjedid
جامعة الشاذلي بن جديد
Faculté des Sciences et de la Technologie
كلية العلوم و التكنولوجيا
Département d'Informatique
قسم الاعلام الالي

MEMOIRE

Présentée par

SADOUD SABIL RACHED

Pour l'obtention de diplôme de

MASTER

Filière : Informatique

Spécialité : Systèmes Informatiques Intelligents

Thème

PRISM : un outil de vérification probabiliste des systèmes embarqués- Etude de cas-

Soutenue le : / / 2020 Devant le Jury composé de

Qualité	Nom et Prénom	Grade	Université
Président	GASMI Ibtissem	MCB	Chadli Bendjedid El-Tarf
Rapporteur	**FERROUM Assia**	**MCB**	**Chadli Bendjedid El-Tarf**
Examinateur	ZEKRI Meriem	MCB	Chadli Bendjedid El-Tarf

Année Universitaire : 2019/2020

Remerciements

Ce travail est l'aboutissement d'un dur labeur et de beaucoup de sacrifices ; nos remerciements vont d'abord au Créateur de l'univers qui nous a doté d'intelligence, et nous a maintenu en santé pour mener à bien cette année d'étude. Je tiens aussi à adresser mes remerciements à ma famille, et plus précisément à mon frère et ma sœur qui m'ont toujours soutenus et poussés à continuer mes études. Ce présent travail a pu voir le jour grâce à leur soutien.

Je souhaite également remercier les professeurs et tous mes amis , qui ont bien voulu répondre à mes différentes questions sur le domaine dans le but de permettre la facilité de l'élaboration de ce travail.

Un remerciement spécial au docteur ASSIA FERROUM qui m'a informé et mis sur le chemin de ce programme d'étude

Dédicace

Merci ALLAH (mon dieu) de m'avoir donné la capacité d'écrire et de réfléchir, la force d'y croire, la patience d'aller jusqu'au bout de rêve.

Je dédie ce travail

A ma chère mère

A mon cher père

Pour tous leurs sacrifices, leur amour, leur tendresse, leur soutien et leurs prières tout au long de mes études.

A Mes frères

Pour leurs encouragements permanents et leurs soutiens moral.

A mes chères ami (e)s :

A toute ma famille

A toute mes autres ami(e)s

A tous ceux que j'aime et ceux qui m'aiment.

RACHED

Table des matières

Remerciements 2

Dédicace 3

Liste des figures 7

Liste des tableaux 8

Liste des acronymes 9

1 Problématique 11

2 Motivation : 11

3 Objective : 11

4 Contenu de mémoire : 11

Chapitre 1 : systèmes embarqués 13

1 Introduction 13

2 Définition des systèmes embarqués : 13

3 Historique : 15

4 Spécificités d'un système embarqué : 15

4.1 Caractéristiques générales : 15

4.2 Différences avec un ordinateur de bureau : 15

4.3 Architecture d'un système embarqué : 15

5 Caractéristiques des systèmes embarqués 16

6 Complexité des systèmes embarqués 16

7 Classification des systèmes embarqués 17

7.1 Système Transformationnel : 18

7.2 Système Interactif : 18

7.3 Système Réactif ou Temps Réel : 18

8 Domaines d'application des systèmes embarqués : 18

9 CONCLUSION : 19

1 Introduction 20
2 Formalismes de modélisation 21
2.1 Chaînes de Markov à temps discret 21
2.2 Processus de décision de Markov 21
2.3 Chaînes de Markov à temps continu 22
2.4 Modèles de Markov Cachés 23
3 Formalismes de spécification 26
3.1 Quelques propriétés 26
3.2 La logique PCTL 27
3.2.1 La syntaxe de PCTL : 27
3.2.2 La sémantique de PCTL 28
3.3 La logique POCTL : 29
3.3.1 La syntaxe de POCTL 29
3.3.2 La sémantique de POCTL 30
4 Algorithmes de model-checking probabiliste 31
5 Conclusion 33
1 Introduction 35
2 Une brève histoire 36
3 Thèmes et tendances 36
4 Language de Prism : 36
4.1 Identifiants : 37
4.2 Expression : 38
4.3 Frais et récompenses (en englais Costs and rewards) : 39
4.3.1 Frais? ou récompenses? 39
4.3.2 Propriétés : 39
4.4 Spécification de la propriété : 40
5 Des exemples de modèles probabilistes en langage de Prism : 41

5.1 Exemple du modèle DTMC : 41
5.1 Exemple du modèle CTMC : 43
5.2 Exemple du modèle MDP : 44
5.2.1 Le problème d'alimentation des philosophes : 44
5.2.2 Une solution aléatoire 45
5.2.3 Vérification des propriétés avec Prism : 48
5.3 Exemple PTA: 49
6 Conclusion 51
1 Introduction 52
2 Le télescope spatial Hubble 52
2.1 Description technique 53
2.2 Missions d'entretien 53
3 Le Système réel 53
4 Modélisation avec le modèle HMM 53
5 Choix des propriétés à vérifier 58
6 Spécification des propriétés choisies en POCTL 59
7 Intégration du modèle HMM de HST dans PRISM : 60
7.1 Explication de code : 60
8 Intégration des propriétés choisies à vérifier dans PRISM 61
8.1 Une simulation de modèle en PRISM : 61
9 Vérification des propriétés 62
9.1 Vérification manuelle des propriétés : 62
9.2 Vérification automatique des propriétés par PRISM 63
10 Résultats et discussions 65
11 Conclusion 65

Liste des figures

Figure 1.1 : Système électronique embarqué 14
Figure 2.1 : Mise à jour des états de croyance……. 25
Figure 2.2 : La syntaxe de la logique PCTL 27
Figure 2.3: La sémantique de PCTL 28
Figure 2.4 : La syntaxe de la logique POCTL..... 30
Figure 2.5 : La sémantique de POCTL 31
Figure 3.1 : Les raisons de la récompense (rawards) 40
Figure 3.2 : Model dtmc de dés..... 41
Figure 3.3 : Code Prism de l'algorithme de dés..... 42
Figure 3.4: Vérification de propriété (P=? [F s=7 & d=5]). 43
Figure 3.5 : Serveur traiter fille dattente 43
Figure 3.6 : Table de philosophes 45
Figure 3.7 : Model mdp sur prism de table de philosophes 45
Figure 3.8 : Une simulation à valeur de temps de situation des philosophes..... 48
Figure 3.9 : Vérification de propriétés avec Prism..... 49
Figure 3.7 : Exemple d'Automate Temporisé Probabiliste (PTA)..... 50
Figure 4.1 : Le modèle HMM du télescope spatial Hubble (HST) 54
Figure 4.2 : La matrice L du HMM du télescope HST 56
Figure 4.3- La matrice P du HMM du HST 57
Figure 4.4 : La matrice μ du HMM correspondante 57
Figure 4.5 : Le vecteur π du HMM du télescope HST 58
Figure 4.6: HMM du télescope HST en langage de Prism..... 60
Figure 4.7 :Une simulation de modèle en Prism 61
Figure 4.8 : Vérification de propriété 01 «P<= 1 [Xe !(six)] ». 64
Figure 4.9 : Vérification de propriété 02 « P>= 0 (Xe accident) ». 64
Figure 4.10 : Vérification de propriété 03 « P<=0.008 (Xe accident) ». 65

Liste des tableaux

Tableau 1.1: Comparaison aux systémes informatiques standards 14
Tableau 4.1 : Propriété en syntaxe de logique POCTL et leur Traduction en langage de Prism 61

Liste des acronymes

ABS. *Anti blockier system*

ACTL. *Action Computation Tree Logic*

ADN. *Acide DésoxyriboNucléique*

ASIC. *Application-Specific Integrated Circuit*

BDDs. *Binary Decision Diagrams*

CSRL. *Continuous Stochastic Reward Logic*

CTMC. *Continuous-Time Markov Chains*

DSP. *Digital-Signal Processor*

DTMC. *Discrete-Time Markov Chains*

EM. *Expectation Maximization*

EPS. *Education Physique et Sportive*

ESA. *Agence Spatiale Européenne*

GPIO. *General Purpose Input/Output*

GPS. *Global Positioning System*

GSM. *Global System for Mobile*

HMM. *Hidden Markov Model*

HST. *Hubble Space Telescope*

IHM. *Interface Homme Machine*

IRM. *Imagerie par Résonance Magnétique*

MDP. *Markov Decision Process*

MEMS. *Micro Electro Mechanical System*

NASA. *National Aeronautics and Space Administration*

PAs. *Probabilistic Automata*

PCTL. *Probabilistic Computation Tree Logic*

PDA. *Personal Digital Assistant*

POCTL. *Probabilistic Observation branChing-time Temporal Logic*

PRCTL. *Probabilistic Reward Computation Tree Logic*

PTA. *Automates Temporisés Probabilistes*

RF. *Radiofréquence*

SD. *Secure Digital*

SOC. *System On a Chip*

TR. *Temps Réel*

Introduction générale

Les systèmes embarqués sont de plus en plus utilisés dans de nombreux domaines d'application : l'industrie des télécommunications, l'électronique grand public, l'automobile et l'aéronautique, ... Ils font partie de notre vie quotidienne via internet ou via toutes sortes de systèmes intégrés comme les appareils électroménagers, les smartphones, les téléviseurs intelligents, les montres, etc.

1 Problématique

La liste des catastrophes connues de tels systèmes est très longue. Un des exemples célèbres de systèmes critiques aux conséquences néfastes est la machine de radiothérapie.

2 Motivation :

Il est important de s'assurer de la sûreté de fonctionnement de tous ces systèmes avant de les mettre en œuvre, en particulier les applications mettant en danger la santé publique. Dans ces applications, outre leur dangerosité en cas de mauvais fonctionnement, elles impliquent un préjudice économique important, difficile à supporter parfois pour les personnes voire les états.

3 Objective :

Pour pallier à ces problèmes, une solution en vogue est la vérification formelle, pour établir la conformité avec les exigences requises. Cette conformité est fréquemment abordée à l'aide de techniques telle que la vérification de modèles ou model-checking. Un model-checker est un outil de vérification à l'aide de model-checking, par exemple PRISM.

L'objectif de ce travail est de faire une étude de cas d'un système embarqué probabiliste choisi avec PRISM.

4 Contenu de mémoire :

Nous avons structuré ce mémoire en quatre chapitres.

Le premier chapitre donne une présentation des systèmes embarqués.

Le chapitre 2 intitulé « Model-checking probabiliste» dresse un état de l'art relatif au model-checking et sa version probabiliste. Cette technique nous semble offrir, un très bon compromis entre : la couverture des différentes exécutions du système, la vérification proprement dite, le haut niveau de compétence requis pour son utilisation, et la simplicité de sa mise en œuvre. Nous avons présenté aussi dans ce chapitre le modèle HMM ainsi que la logique POCTL comme un formalisme de modélisation dans notre approche.

Le chapitre 3 présente notre objet d'étude qui est le model-cheker PRISM.

Le chapitre 4 intitulé « Étude de cas » présente des résultats expérimentaux obtenus à partir d'une mise en œuvre de Prism sur le système embarqué réel HST : la présentation du système HST, la construction de son modèle HMM, le choix des propriétés à vérifier, la spécification de ces propriétés choisies en logique POCTL, la vérification par PRISM.

Une conclusion achève ce mémoire, mettant en avant les principaux aspects de notre travail avec des perspectives de nos futurs travaux.

Chapitre 1 : systèmes embarqués

1 Introduction

L'humain besoin au début de l'informatique un system spécifique pour résoudre un problème ou une tâche spécifique alors il besoin de system embarqué. Dans ce chapitre en définir ce system et leur types et leur Domaines d'applications.

2 Définition des systèmes embarqués :

"Embedded system" : tout système conçu pour résoudre un problème ou une tâche spécifique mais n'est pas un ordinateur d'usage général. Utilisent généralement un microprocesseur combiné avec d'autres matériels et logiciels pour résoudre un problème de calcul spécifique. Système électronique et informatique autonome ne possédant pas des entrées-sorties standards. Le système matériel et l'application sont intimement liés et noyés dans le matériel et ne sont pas discernables comme dans un environnement de travail classique de type PC.[1]
Le système comprend une partie matérielle et une partie logicielle. Quelle que soit la nature et la complexité du système, on décompose un système embarqués en :

- le système contrôlé.
- le système de contrôle.

Le système contrôlé = environnement (procédé) équipé d'une instrumentation qui réalise l'interface avec le système de contrôle.
Le système de contrôle = éléments matériels (microprocesseurs…) et logiciels dont la mission est d'agir sur le procédé via les actionneurs en fonction de l'état de ce procédé indiqué par les capteurs de manière maintenir ou conduire le procédé dans un état donné

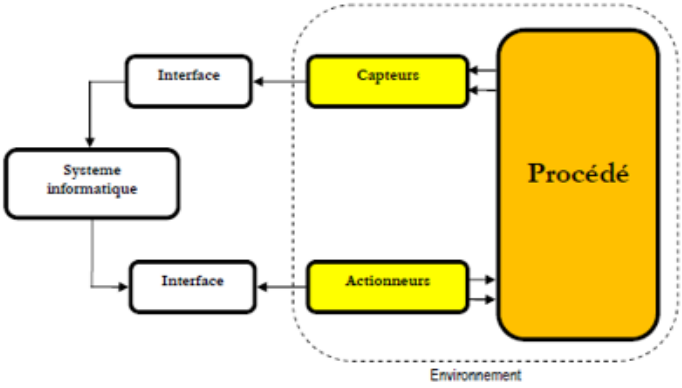

Figure 1.1 : Système électronique embarqué

Un système électronique embarqué ou enfoui est un élément constitutif d'un système plus complexe pour lequel il rend des services bien précis (contrôle, surveillance, communication…). Il est constitué de parties matérielles et logicielles qui sont conçues spécifiquement pour réaliser une fonction dédiée.

Système embarqué = Système électronique/informatique conçu pour réaliser une ou plusieurs tâches précises.

Les caractéristiques principales d'un système électronique embarqué sont :

 Autonomes : Une fois enfouis dans l'application ils ne sont (le plus souvent) plus accessibles

 Temps réel : Les temps de réponses de ces systèmes sont aussi importants que l'exactitude des résultats.

 Réactifs : Il doit réagir à l'arrivée d'informations extérieures non prévues.

Comparaison aux systèmes informatiques standards :	
Informatique :	**Embarqué :**

• Processeur standard
  ○ Multiples unités fonctionnelles (flottant)
  ○ Vitesse élevée (> GHz)
  ○ Consommation électrique élevée
  ○ Chaleur
  ○ Taille
• MMU (mémoire virtuelle)
• OS
• Cache
• Grand nombre de périphériques
	• Processeur dédié (contrôleur)
	  ○ Architecture adaptée
	  ○ Vitesse faible (~200 MHz)
	  ○ 8-32bits : mémoire limitée
	  ○ Basse consommation
	  ○ Petite taille, grand volume => faible coût
	• Processeur DSP (traitements)
	  ○ Très puissants
	• Quelques Mo de mémoire
	• RTOS

Tableau 1.1: comparaison aux systémes informatiques standards

3 Historique :

Le concept des systèmes embarqués a vu le jour suite à l'apparition Apollo Guidance computer en 1967 et en 1971. Ils ont connu encore beaucoup plus d'envergure avec l'invention d'Intel 4004 qui était considérée comme le premier microprocesseur. A la base ces systèmes contenaient qu'un seul processeur mais actuellement ils peuvent même contenir des centaines.

4 Spécificités d'un système embarqué :

4.1 Caractéristiques générales :

▪ Dédié à une application spécifique.

▪ Coût réduit, maximisation rapport performance/prix.

▪ Volume restreint (compact, pas modulaire).

▪ Capacité mémoire adaptée.

▪ Capacité de calcul appropriée à l'application.

▪ Exécution temps réel (souvent).

▪ Fiabilité et sécurité de fonctionnement.

▪ Consommation d'énergie maitrisée.

• Voir très faible en cas d'utilisation sur batterie.

4.2 Différences avec un ordinateur de bureau :

▪ L'interface IHM (Interface Homme machine) est adaptée selon l'application. Cela peut être de simples leds et boutons jusqu'à un écran tactile. Il doit être robuste (étanche, choc, ..).

▪ Généralement : pas de clavier, écran réduit.

▪ Le système embarqué dispose de périphériques et capteurs spécifiques pour son application.

• Carte SD, mémoire flash, …

• Wifi, Bluetooth, module GSM-GPS, …

• Température, pression, actuateurs (moteurs, …)

4.3 Architecture d'un système embarqué :

▪ Réalisé autour d'un microcontrôleur (UC) ;

• UC est un système à processeur dans un seul chip (SOC) ;

• Comprend : processeur, mémoire, GPIO (entrées/sorties simples configurables), contrôleur de bus, contrôleur d'interruption, contrôleur d'écran, USB, Ethernet ;
• Très bon rapport performance/prix et performance/consommation ;
▪ Ensemble compact (volume optimisé) ;
▪ Démarrage autonome du système (boot) ;
• Pas de disque dur, utilisation de mémoire flash, …
▪ Généralement pas d'extension possible ;
• Construction non modulaire.

5 Caractéristiques des systèmes embarqués

Les systèmes embarqués fonctionnent généralement en Temps Réel (TR) : les opérations de calcul sont alors faites en réponse à un événement extérieur (interruption matérielle). La validité et la pertinence d'un résultat dépendent du moment où il est délivré. Une échéance manquée induit une erreur de fonctionnement qui peut entraîner soit une panne du système (plantage), soit une dégradation non dramatique de ses performances.
Lorsque les systèmes embarqués sont utilisés dans les produits de grande consommation, ils sont fabriqués en grande série. Les exigences de coût se traduisent alors en contraintes sur les différentes composantes du système : utilisation de faibles capacités mémoires et de petits processeurs (4 bits ou 8 bits), mais en grand nombre. Ainsi, les systèmes embarqués sont particulièrement sensibles au coût de production. Il existe des applications dans lesquelles les contraintes de coût de production et de maintenance ont une importance de même niveau que les performances envisagées.

Dans les systèmes embarqués autonomes, la consommation d'énergie est un point critique pour le coût. En effet, une consommation excessive augmente le prix de revient du système embarqué, car il faut alors des batteries de forte capacité. [1]

6 Complexité des systèmes embarqués

Les système embarqués requièrent souvent un faible encombrement (faible poids) PDA (Personal Digital Assistant), Internet et téléphone mobiles, …). Leur technologie fait alors appel à une électronique et à des applications portables où l'on doit minimiser aussi bien l'encombrement que la consommation électrique. Par conséquent, la réalisation du packaging

afin de faire cohabiter sur une faible surface de l'électronique analogique, de l'électronique numérique, des composantes RF (Radiofréquence) sans interférences est une tâche difficile. En effet, les performances des systèmes sur carte deviennent obsolètes dans le contexte des besoins actuels. Dans les stratégies de conception actuelles, un système embarqué est généralement intégré sur un support silicium unique constituant ainsi un système complet intégré sur une puce SoC (System on a Chip).

Les systèmes sur puce contiennent généralement une grande variété de dispositifs programmables tels que des microcontrôleurs, des processeurs de traitement de signaux DSP (Digital-Signal Processor) et des ASIC qui sont développés pour des applications complexes nécessitant une production en grande série.

Les mémoires (ROM et RAM) y sont intégrés pour le stockage des données et des programmes. Ces composants digitaux cohabitent généralement sur le même support de silicium avec des composants analogiques et mixtes divers tels que des composantes radiofréquence (RF) comme moyen de communication, des composantes optiques pour le transfert de données à haut débit, des MEMS (Micro Electro Mechanical System) pour l'interfaçage avec le monde externe, des convertisseurs analogique/numérique et numérique/analogique requis pour le dialogue interne. L'objectif est d'obtenir une coopération harmonieuse entre composants embarqués afin de garantir des services globaux. Des contraintes d'implémentation physique sont liées à la consommation de ressources et au contexte de déploiement tel que le poids, la taille physique, la résistance aux vibrations, ou aux irradiations,…, etc.

7 Classification des systèmes embarqués

- Temps réel dur (‘hard real-time’) : le non-respect des contraintes temporelles entraîne la faute du système.

– **Exemple :** contrôle de trafic aérien, système de conduite de missile, ...

- Temps réel souple (‘soft real-time’) : le respect des échéances est important mais le non-respect des échéances n’a pas de graves conséquences

– **Exemple :** système d'acquisition de données pour affichage

- Temps réel ferme (‘firm real-time’) : temps réel souple, mais si l’échéance est dépassée le résultat obtenu n’a plus de valeur (et est donc écarté).

– **Exemple :** Projection vidéo

7.1 Système Transformationnel :

Activité de calcul, qui lit ses données et ses entrées lors de son démarrage, qui fournit ses sorties, puis meurt.

7.2 Système Interactif :

Système en interaction quasi permanente avec son environnement, y compris après l'initialisation du système ; la réaction du système est déterminée par les événements reçus et par l'état courant (fonction des événements et des réactions passés) ; le rythme de l'interaction est déterminé par le système et non par l'environnement.

7.3 Système Réactif ou Temps Réel :

Système en interaction permanente avec son environnement, y compris après l'initialisation du système ; la réaction du système est déterminée par les événements reçus et par l'état courant (fonction des événements et des réactions passées) ; mais le rythme de l'interaction est déterminé par l'environnement et non par le système.

8 Domaines d'application des systèmes embarqués :

- **Domaine grand public :**

Smart phone, console de jeux, appareil photos, lecteur audio, ...

- **Moyens de transport :**

Gestion moteur/entrainement, ordinateur de bord, ABS, GPS, système navigation, système d'aide (EPS, ..), …etc

=> automobiles, avions, trains, bateau, véhicule électrique, …

- **Equipement médicaux** (diagnostic, thérapeutique, vital) :

Imagerie (rayon X, ultra-sons, IRM), endoscopie, caméra, monitoring, perfusion, lasers, chirurgie, stimulateur cardiaque, …

- **Equipements de télécommunication :**

Station mobile, routeur, gateway, satellite, …

- **Equipement industriels :**

Commande, contrôle répartit, capteurs intelligents, ...

9 CONCLUSION :

La sécurité des systèmes et des données est un problème critique pour nos sociétés développées à cause des coûts de plus en plus important, le nombre d'attaques logicielles explose, de plus en plus de systèmes embarqués mobiles et communicants, menaces fortes des attaques matérielles. Et pour aborder la sécurité des systèmes et des données qui subissent des attaques sont soft ou hard impliquant des protections soft ou hard, il faut envisager des solutions à tous les niveaux, du système au transistor, définir des politiques de sécurité (sécurisé suffisante au bon moment), définir des méthodes de conception sécurisée assurant les performances des systèmes, ne rien négliger (aspect humain, légal, logicielle, matérielle, extérieur, intérieur). La sécurité sa concerne l'informatique et l'électronique (embarquée en particulier), les télécommunications, les individus, les entreprises, la société...

Chapitre 02 : Model-Cheking Probabiliste

1 Introduction

La vérification formelle basée sur la logique temporelle a été prolongée avec succès au domaine des modèles probabilistes. Dans ce chapitre, nous introduisons les notions de base associées au model-checking probabiliste.

Le model checking probabiliste est une technique de vérification formelle automatique pour l'analyse de systèmes qui présentent un comportement stochastique. La technique est similaire au model-checking classique. La différence majeure est que le modèle probabiliste contient des informations supplémentaires sur la probabilité ou le calendrier des transitions entre états, ou pour être plus précis, il peut modéliser le comportement stochastique.

Ce chapitre donne une introduction au model-checking probabiliste, il élabore des concepts de base pertinents pour cette thèse, à commencer par les principaux modèles probabilistes utilisés par la plupart des model chekers existants à savoir : les chaînes de Markov à temps discret (*Discrete-Time Markov Chains* ou DTMC), les processus de décision de Markov (*Markov Decision Processes* ou MDP) ainsi que les chaînes de Markov à temps continu (*Continuous-Time Markov Chains* ou CTMC), avec la logique PCTL (*Probabilistic Computation Tree Logic*) utilisée pour la spécification des propriétés pour les DTMC et les MDP.

Nous présentons aussi dans ce chapitre, le modèle probabiliste HMM que nous allons introduire pour la première fois au domaine de la vérification probabiliste dans ce travail, à travers la réalisation d'un model checker le supportant et la logique temporelle probabiliste correspondant POCTL (*Branching-time temporal logic Probabilistic Observation*) qui est une prolongation de la logique PCTL, et finalement une vue générale sur les algorithmes de model-checking probabiliste.

2 Formalismes de modélisation

Les types de modèles stochastiques les plus utilisés dans les model-checkers probabilistes les plus connus sont : (DTMC) [2, 3], (MDP) [3] et (CTMC) [3]. Nous présentons aussi dans cette section le type que nous allons adopter dans notre approche qui est le (HMM) [4].

2.1 Chaînes de Markov à temps discret

Les chaînes de Markov à temps discret [2, 3] ou DTMC (de l'anglais *Discrete-Time Markov Chain*) sont le modèle le plus simple représentant des systèmes probabilistes. Ce sont des systèmes de transitions où les transitions sont étiquetées par la probabilité de passer d'un état à un autre.

Définition : Soit *AP* (de l'anglais *Atomic Proposition*) un ensemble fixe de propositions atomiques. Une chaîne de Markov à temps-discret est un tuple $M = (S, s_0, P, L)$ tel que :

- S est un ensemble fini non vide d'états,
- $s_0 \subseteq S$ est l'état initial,
- $P : S \times S \rightarrow [0, 1]$ est une fonction qui assigne une probabilité à chaque transition de telle sorte que pour tout $s \in S$, on a :

$$\sum_{s' \in S} P(s, s') \in \{0,1\}$$

- $L : S \rightarrow 2^{AP}$, est une fonction qui assigne à chaque état $s \in S$, un ensemble de propositions atomiques $L(s)$ valides dans s.

2.2 Processus de décision de Markov

Les DTMC ne permettent pas de modéliser le non-déterminisme. Les processus de décision de Markov [3] sont une généralisation des DTMC qui permettent à la fois un choix probabiliste et non déterministe.

Comme pour les DTMC, le choix probabiliste sert à modéliser et quantifier les sorties possibles d'une action aléatoire tandis que le non-déterminisme permet de décrire par exemple des systèmes concurrents, i.e., des systèmes fonctionnant en parallèle.

Définition : Soit *AP* un ensemble fixe de propositions atomiques. Un processus de décision de Markov ou MDP (de l'anglais *Markov Decision Process*) est un tuple M = (S, l_{init}, *Act*, *P*, *L*) tel que :

- S est un ensemble fini non vide d'états,
- l_{init} : S → [0; 1] est la distribution initiale telle que :

$$\sum_{s \in S} l_{init}(S) = 1$$

- *Act* est un ensemble d'actions, $P : S \times Act \times S \rightarrow [0, 1]$ est la fonction des transitions de probabilité telle que pour tout état $s \in S$ et action $\alpha \in Act$:

$$\sum_{s' \in S} P(s, \alpha, s') \in \{0,1\}$$

- L : S → 2^{AP}, une fonction d'étiquetage.

2.3 Chaînes de Markov à temps continu

Les chaînes de Markov à temps continu (*Continuous- Time Markov Chains* ou CTMC) [3], étend aussi les DTMC mais d'une manière différente. Pour les DTMC, une transition correspond à une étape discrète d'un point de vue temporel. Dans un CTMC, les transitions peuvent être exécutées en temps réel. Chaque transition est étiquetée d'un coût, définissant le délai qui s'écoule avant qu'elle ne soit exécutée.

Définition : Soit *AP* un ensemble de propositions atomiques. Une *chaîne de Markov à temps continu* est un tuple $M = (S, s_0, R, L)$ où :

- *S* est un ensemble fini d'états,
- s_0 est l'état initial,
- $R : S \times S \rightarrow R_{\geq 0}$ est la matrice des taux de transition,
- $L : S \rightarrow 2^{AP}$ est une fonction d'étiquetage.

2.4 Modèles de Markov Cachés

Les modèles de Markov cachés (*Hidden Markov Model* ou HMM) [4] ont été développés vers la fin des années 60 et sont très importants pour beaucoup d'applications, particulièrement la reconnaissance de la parole [5], la reconnaissance de caractères [6], l'analyse biologique [7], et les problèmes de classification de protéines [8]. Récemment, les HMMs suscitent une attention importante dans le contexte de la modélisation des canaux de communication « *communication channel modelling* » [9] et des propriétés QoS dans les réseaux sans fil [10, 11].

Un HMM [4] peut être vu comme un automate fini dont les transitions sont pondérées par des probabilités, dites probabilités de transition. De plus, chaque état de l'automate peut "émettre" des symboles d'un alphabet selon une loi de probabilité donnée que l'on appelle loi de probabilité d'émission. L'adjectif "caché" provient du fait qu'une suite de symboles générée par un HMM ne permet pas de savoir par quelle suite d'états le HMM est passé pour la générer, nous disons par conséquent que les états du HMM sont "cachés".

Un HMM est un processus doublement stochastique avec un processus sous-jacent qui n'est pas observable. Ce dernier est uniquement observé à travers un autre ensemble de processus stochastiques qui produit la séquence des symboles [4]. En donnant une séquence d'observations, nous ne pouvons pas connaître exactement l'état occupé mais nous allons connaître la distribution de probabilité sur l'ensemble des états, cette information est captée par ce que nous appelons état ***de croyance*** [11].

Définition : Soit *AP* un ensemble de propositions atomiques. Un *modèle de Markov caché* est un sextuplet $H = (S, P, \theta, \mu, \pi, L)$ où :

- $S = \{s_1, s_2, \ldots s_n\}$ est un ensemble fini de n états. L'état courant du HMM à l'instant t est noté qt ($q_t \in S$) ;
- P est une matrice de probabilités de transitions entre les états de la chaîne :

  $\forall\, i,j \in [1 \ldots n]$, $\forall t \in [1 \ldots T] : a_{ij} = P(i, j) = p(q_{t+1} = s_j \mid q_t = s_i)$ avec : $a_{ij} \geq 0, \forall\, i, j$ et $\sum_{j=1}^{n} a_{ij} = 1$;
- $\theta = \{O_1, O_2, \ldots, O_M\}$ est L'ensemble des M symboles observables dans chaque état. Un élément V_t de θ désigne un symbole observé à l'instant t ;
- μ est la matrice de distribution des probabilités d'observations : $\mu_j(k)$ est la probabilité d'observer le symbole O_k quand le modèle se trouve dans l'état j, soit : $\mu_j(k) = P(V_t = O_k \mid q_t = s_j)$, $1 \leq j \leq n$, $1 \leq k \leq M$, $\mu(k) \geq 0$ $\forall j, k$ et $\sum_{k=1}^{M} \mu_j(k) = 1$;

- $\pi = \{\pi_i\}, i = 1, .. , n$; est le vecteur de distribution des probabilités de transitions initiales ; π_i représente la probabilité que l'état de départ du modèle est l'état i, soit : $\pi_i = P(q_1 = s_i)$, $1 \leq i \leq n$ avec : $\pi_i \geq 0 \ \forall i$ et $\sum_{i=1}^{n} \pi_i = 1$;
- $L : S \rightarrow 2^{AP}$ est une fonction d'étiquetage.

Les états de croyance : L'observation dépend stochastiquement et exclusivement de l'état courant, en général la même observation pourrait être émise par plusieurs états différents, par conséquence, nous sommes incertains au sujet de l'état courant mais nous pouvons rassembler l'historique des observations dans un état de croyance (ou état d'information) qui est une distribution sur s.

Un état de croyance n'est pas réellement un état de HMM, c'est une façon de décrire ce que nous savons au sujet de l'état donnant l'historique des observations. L'ensemble de tous les états de croyance possibles est appelé espace de croyance, et est dénoté par C.

Nous utilisons s^t avec $s^t \in S$ pour indiquer l'état au temps t, $O^t \in \theta$ pour indiquer l'observation au temps t et c^t pour indiquer l'état de croyance à l'instant t. Soit $O_i \in \theta$ où i = 1, 2,..., t. L'état de croyance c_t à l'instant t est la distribution de S à l'instant t donnant l'historique de l'observation $o_0, o_1,..., o_t$: $c^t(s) = p(s^t = s \mid O^0 = o_0 , O^1 = o_1 , \dots , O^t = o_t , H) , \forall s \in S$ [18].

Maintenant en donnant l'historique des observations $o_0, o_1, \dots, o_t$, la question est comment calculer l'état de croyance c_n : l'état de croyance au temps 0 dépend seulement de la distribution initiale et la première observation, l'état de croyance au temps t prise toute notre information sur le passé, comme résultat nous pouvons calculer l'état de croyance courant c_t en basant sur l'état de croyance précédent c_{t-1} et l'observation courante o_t ; comme illustre la figure 2.1.

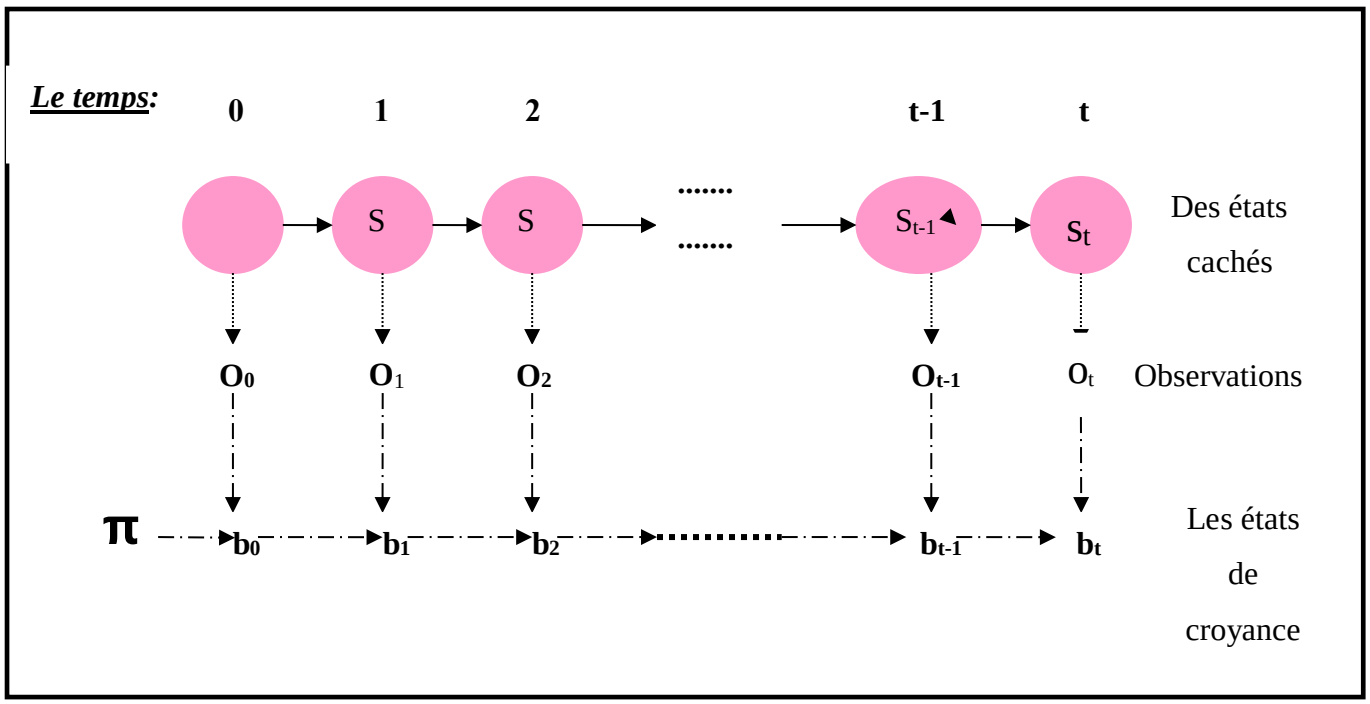

Figure 2.1 : Mise à jour des états de croyance [11]

- Les états reliés par des flèches continues entre eux représentent les états cachés d'une exécution.

- Les flèches pointillées entre les états et les observations indiquent que l'observation O_t est produite par l'état s_t accordé à la fonction d'observation μ, comme cas particulier C_0 est en fonction de O_0 et la distribution initiale π.

Les trois problèmes des HMMs : La réalisation par le modèle HMM *H* d'un processus Markovien de durée T est décrite par :

- $Q = q_1\ q_2\ \ldots\ q_T$, un chemin *a priori* caché parmi les N états,

- $O = O_1\ O_2\ \ldots\ O_T$, une suite d'observations appartenant à l'alphabet de M symboles.

À partir d'une suite d'observations *O* supposées émises par un modèle, trois problèmes de base doivent être résolus à fin de pouvoir exploiter le modèle H :

- ***Le problème d'évaluation :*** L'évaluation de la probabilité que la suite des observations ait été émise par un modèle. Lorsque plusieurs modèles existent, cette évaluation permet le choix du modèle le plus probable. Ce problème est efficacement résolu par l'algorithme *Forward* [4].

- ***Le problème de décodage ou de reconnaissance :*** La recherche de la séquence d'états d'un modèle ayant produit la séquence des observations, c'est à dire la recherche de la séquence cachée de plus forte probabilité et qui explique mieux ces observations. Pour résoudre ce problème efficacement, nous utilisons l'algorithme de *Viterbi* [4].

- ***Le problème de re-estimation ou d'apprentissage :*** L'apprentissage des paramètres d'un modèle. À partir d'un modèle donné *a priori* et d'observations supposées émises par ce modèle, nous cherchons les probabilités de transition et d'émission maximisant la vraisemblance des observations. La maximisation de la vraisemblance d'un modèle probabiliste est aujourd'hui effectuée par des algorithmes efficaces, comme l'algorithme *EM* (*Expectation Maximization*) ou l'algorithme de *Baum-Welch* [4].

3 Formalismes de spécification

Les model-checkers probabilistes les plus connus utilisent les logiques qui permettent de spécifier et de vérifier les systèmes probabilistes. Ces logiques sont : PCTL [12], CSL [13], PRCTL [14], CSRL [15, 16], QuaTEx [17], aCSL [18].

Dans cette section, nous présentons les différents types de propriétés, puis la logique PCTL [12] étendue à la logique POCTL [11].

3.1 Quelques propriétés

Nous décrivons quelques propriétés courantes que nous allons utiliser à des fin de vérification.

- ***Propriété d'atteignabilité*** : énonce qu'une certaine situation peut être atteinte.

- ***Propriété de sûreté*** : affirme que rien de mauvais ne peut se produire.

- ***Propriété de vivacité*** : affirme que quelque chose de bon peut inévitablement se produire.

- ***Propriété d'équité*** : énonce que, sous certaines conditions, quelque chose aura lieu (ou n'aura pas lieu) un nombre infini de fois.

3.2 La logique PCTL

PCTL, introduite par H. Hansson et B. Jonsson [5] est une modification probabiliste de CTL (*Computational Tree Logic*) [2], où le quantificateur (∀) "*Pour tous les chemins*" et (∃) "*Existe un chemin*" a été remplacé par un opérateur probabiliste qui nécessite une probabilité donnée sur un certain ensemble de chemins qui est l'opérateur probabiliste $P_{\sim} p$; où $P \in [0.1]$ est une liaison de probabilité, et $\sim \in \{>, <, \leq, \geq\}$.

3.2.1 La syntaxe de PCTL :

La logique PCTL, autant qu'une prolongation probabiliste de la logique CTL, distingue les formules d'états, des formules de chemins.

Les formules d'états expriment les propriétés et les comportements attribuent aux états, et les formules de chemins expriment les propriétés et les comportements attribuent aux chemins [19].

Soit *AP* un ensemble fini de propositions atomiques. La syntaxe de PCTL est donnée dans la figure 2.2 :

$$\phi := true \mid a \mid \neg \phi \mid \phi_1 \wedge \phi_2 \mid P_{\sim p}(\psi)$$

$$\psi := X\phi \mid \phi_1 \, U \, \phi_2 \mid \phi_1 \, U^{\leq k} \, \phi_2$$

Où : $a \in AP$, $k \in N$, $0 \leq p \leq 1$ et $\sim \in \{>, \geq, <, \leq\}$

Figure 2.2 : La syntaxe de la logique PCTL

Cette présentation s'appelle la présentation Backus-Naur [2]. Elle signifie que les formules de la logique PCTL sont construites récursivement à partir de cette syntaxe de la manière suivante :

- ✓ **Formules d'états**

a) a est une formule d'état.

b) Si ϕ est une formule d'état, alors : $\neg \phi$ est une formule d'état.

c) Si ϕ_1 et ϕ_2 sont des formules d'états, alors $\phi_1 \wedge \phi_2$ est une formule d'état.

d) Si ψ est une formule de chemin, $P_{\sim p}$ (ψ) est une formule d'état.

e) Rien d'autre n'est une formule d'état.

✓ **Formules de chemin**

a) Si ϕ est une formule d'état, alors $X\phi$ est une formule de chemin.

b) Si ϕ_1 et ϕ_2 sont des formules d'états, alors $\phi_1 \, U \, \phi_2$ est une formule de chemin.

c) Si ϕ_1 et ϕ_2 sont des formules d'états et k ∈ N, alors $\phi_1 \, U^{\leq k} \, \phi_2$ est une formule de chemin.

d) Rien d'autre n'est une formule de chemin.

3.2.2 La sémantique de PCTL

Soit *AP* un ensemble fini de propositions atomiques. La sémantique de PCTL est donnée dans la figure 2.3 :

Sémantique pour les formules d'états

$s \models true$	*pour tout s*	
$s \models a$	*ssi*	$a \in L(s)$
$s \models \neg \phi$	*ssi*	$s \not\models \phi$
$s \models (\phi_1 \wedge \phi_2)$	*ssi*	$s \models \phi_1 \wedge s \models \phi_2$
$s \models P_{\sim p}(\psi)$	*ssi*	$Prob(\{\pi \in Paths(s) \mid \pi \models \psi\}) \sim p$

Sémantique pour les formules de chemins

$\pi \models X \phi$	*ssi*	$\pi = s_0 s_1 s_2 ...$ et $s_1 \models \phi$
$\pi \models \phi_1 \, U \, \phi_2$	*ssi*	$\pi = s_0 s_1 s_2 ...$ et $\exists k$ *tq.* $S_k \models \phi_2 \wedge \forall j < k, s_j \models \phi_1$
$\pi \models \phi_1 \, U^{\leq k} \, \phi_2$	*ssi*	$\exists i \leq k$ tq. $s_i \models \phi_2 \wedge s_j \models \phi_1, \forall j < i$

Figure 2.3: La sémantique de PCTL

Les opérateurs booléens true, $\wedge$ et la négation sont interprétés de la façon habituelle. Par exemple, la formule $\phi_1 \wedge \phi_2$ est valide dans un état s'il satisfait à la fois ϕ_1 et ϕ_2.

La formule $P_{\sim p}(\psi)$ est satisfaite dans un état s, si et seulement si *Prob* ($\{ \pi \in \text{Paths(s)} \mid \pi \models \psi \}$) est comprise dans l'intervalle défini par $\sim p$. Par exemple $P_{\geq p}(X\,\phi)$ est satisfaite par le modèle M si la probabilité de l'ensemble des chemins partant de l'état initial s_0 et dont le prochain état s_1 satisfait ϕ, est supérieure ou égale à *p* [19].

3.3 La logique POCTL :

Dans un modèle HMM, les états sont cachés, et les processus cachés produisent un ordre des observations. La logique POCTL présenté dans [11], est une prolongation de la logique PCTL [12]. Avec POCTL nous pouvons énoncer des propriétés sur les HMMs telles que " *Il y a au moins une probabilité de 90 % que le modèle produit une séquence d'observations donnée* ".

POCTL est fondamentalement une prolongation de PCTL où l'opérateur next est équipé d'une contrainte d'observation. D'autre part, POCTL peut être considéré également comme une variante de la logique temporelle ACTL (*Action Computation Tree Logic*), présentée par De Nicola et autres [20], dans lequel l'opérateur *next* habituel est prolongé pour contraindre l'étiquette d'action de la transition. POCTL se compose des formules d'état, des formules de chemin et des formules d'état de croyance qui nous permettent de spécifier des propriétés sur les HMMs. De cette façon, nous pouvons énoncer des propriétés sur des observations, par exemple, $X_o\varphi$ signifie que la prochaine observation est *o* et le chemin suivant satisfait φ [11].

3.3.1 La syntaxe de POCTL

Soit H= (S, P, L,θ, B, π) est un HMM avec $o \in O$. La syntaxe de la logique POCTL est définie comme suit :

$$\phi := a \mid \phi_1 \wedge \phi_2 \mid \neg \phi \mid \varepsilon$$

$$\psi := X_o \phi \mid \phi_1 \, U^{\leq k} \, \phi_2$$

$$\varepsilon := P_{\sim p}(\psi) \mid \neg \varepsilon \mid \varepsilon_1 \wedge \varepsilon_2$$

Où : $k \in N$, $0 \leq p \leq 1$ et $\sim \in \{>, \geq, <, \leq\}$

Figure 2.4 : La syntaxe de la logique POCTL

La syntaxe de POCTL se compose des formules d'état, des formules de chemin et des formules d'état de croyance. Nous employons ϕ pour la formule d'état et ψ pour la formule de chemin et a formule ε s'appelle la formule d'état de croyance. Pour un HMM, nous sommes incertains au sujet de l'état actuel, mais nous connaissons toujours l'état courant de croyance. Par conséquent, nous voulons savoir si quelques propriétés (probabilistes) sont valides dans des états de croyance. Nous considérons la propriété : " *Il y a au moins une probabilité de 90 % que le modèle produit l'ordre des observations* $O = \{o_0, o_1, \ldots\ldots, o_n\}$ ". Cette propriété peut être exprimée en POCTL comme suit : $P_{\geq 0.9}(X_{o0} X_{o1} \ldots X_{on})$ comme indiqué par Rabiner dans [4]. Ceci peut être exprimé par une formule d'état de croyance : $\varepsilon = P_{\geq 0.9}(X_{o0} X_{o1} \ldots. X_{on})$ [11].

3.3.2 La sémantique de POCTL

Soit $H = (S, P, L, \theta, B, \pi)$ un HMM avec $s \in S$ et $\sigma \in Path$. La sémantique de POCTL (figure 2.5) est définie par une relation de satisfaction (dénotée par $\models$) entre un état s et une formule d'état ϕ, ou entre un chemin σ et une formule de chemin ψ, ou entre un état de croyance b et une formule d'état de croyance ε. Nous écrivons $H, s \models \phi$; $H, \sigma \models \psi$ et $H, b \models \varepsilon$ si l'état s, le chemin σ et l'état de croyance b satisfont la formule d'état ϕ, la formule de chemin ψ et la formule d'état de croyance ε, respectivement [11].

$s \models a$	*ssi*	$a \in L(s)$
$s \models \phi_1 \wedge \phi_2$	*ssi*	$s \models \phi_1 \wedge s \models \phi_2$
$s \models \neg \phi$	*ssi*	$s \models$ non ϕ
$s \models \varepsilon$	*ssi*	$b_s \models \varepsilon$
$\sigma \models X_o \phi$	*ssi*	$\sigma_o[0] \models o \wedge \sigma[1] \models \phi$
$\sigma \models \phi_1 U^{\leq k} \phi_2$	*ssi*	$\exists\, 0 \leq j \leq k\ (\sigma[j] \models \phi_2 \wedge \forall\, i<j: \sigma[i] \models \phi_1)$
$b \models P_{\sim p}(\psi)$	*ssi*	$p_b\{\sigma / \sigma \models \psi\} \sim p$
$b \models \neg\varepsilon$	*ssi*	$b \not\models \varepsilon$
$b \models \varepsilon_1 \wedge \varepsilon_2$	*ssi*	$b \models \varepsilon_1 \wedge b \models \varepsilon_2$

Figure 2.5 : La sémantique de POCTL

Un chemin satisfait le nouvel opérateur *X* s'il commence par l'observation o et le suffixe σ[1] satisfait ϕ. Soit Ω un ensemble d'observations, c.-à-d. $\Omega \subseteq \theta$.

Nous avons évidemment $\sigma \models X_\Omega \phi$ ssi $\sigma_o[0] \in \Omega \wedge \sigma[1] \models \phi$.

Le prochain opérateur habituel peut être décrit $X\phi \equiv X_\theta \phi$. Ainsi, la logique PCTL peut être considérée en tant que logique secondaire de POCTL [11].

4 Algorithmes de model-checking probabiliste

Il existe trois algorithmes de model-checking probabiliste : PCTL sur les DTMCs, PCTL sur les MDPs et CSL sur les CTMCs. Un algorithme de model-checking pour PCTL ou CSL prend un modèle du type approprié, d'une formule Ø dans la logique, et renvoie l'ensemble *Sat*(Ø) contenant les états du modèle qui satisfont Ø. La structure globale de l'algorithme de model-checking probabiliste est la même dans chacun des trois cas, et provient de l'algorithme de model-checking original de CTL présenté dans [21].

De nombreuses méthodes ont été proposées pour vérifier des formules d'une logique temporelle sur des systèmes probabilistes, voire concurrents probabilistes. Vardi [22] a ainsi mis au point une technique à base d'automates pour la vérification de propriétés qualitatives exprimant le

fait qu'une formule du temps linéaire est satisfaite avec probabilité 0 ou 1 (propriétés dites *qualitatives*).

Courcoubetis et Yannakakis [23] ont étudié la vérification probabiliste de propriétés *quantitatives* (probabilité de satisfaction arbitraire) également exprimées dans le cadre du temps linéaire. Pour le cas des systèmes totalement probabilistes (DTMCs) [2, 3], la complexité en temps de leur méthode, est polynomiale dans la taille de l'espace des états et exponentielle dans la taille de la formule. Dans le cas concurrent, la complexité en temps est linéaire dans la taille du système, et doublement exponentielle dans la taille de la formule.

Hansson et Jonsson [24] ont introduit la logique PCTL et proposé un algorithme de model-checking pour les formules de cette logique exprimant des propriétés sur des systèmes totalement probabilistes. Leur algorithme combine un calcul d'accessibilité (*comme dans le cas des systèmes non probabilistes*) et la résolution de systèmes d'équations linéaires pour calculer la probabilité des formules contenant l'opérateur *Until*.

Pour les systèmes concurrents probabilistes, Bianco et de Alfaro [25] ont montré, dans l'un de leurs articles, que les probabilités minimales et maximales de satisfaction d'une formule construite avec *Until* pouvaient être calculées en résolvant un problème d'optimisation linéaire. La complexité en temps de leur algorithme est polynomiale en la taille du système que l'on considère, et linéaire dans la taille de la formule que l'on vérifie. Depuis quelques temps, des méthodes qui tournent autour de la notion du test statistique d'hypothèses apparaissent dans la littérature.

Ainsi, dans un article récent [26], Younes et Simmons décrivent une méthode pour vérifier des propriétés de systèmes à évènements discrets. De manière similaire, la vérification qualitative de propriété a été considérée pour les modèles qui montrent le comportement probabiliste et non déterministe, c.-à-d. les modèles comme les (MDPs) [27, 28, 29, 30, 31]. Une des différences principales ici est qu'une notion de l'équitabilité doit être considérée.

La plus difficile est la vérification des caractéristiques quantitatives. Pour ces derniers, nous devons calculer la probabilité exacte qu'une propriété donnée est satisfaite. Le cas pour DTMCs a été considéré par Courcoubetis et Yannakakis [31, 32], Hansson et Jonsson [24], et Aziz et autres. [33].

La vérification des propriétés quantitatives pour (MDPs) a été considérée par Courcoubetis et Yannakakis [34] et Bianco et de Alfaro [25]. Le model-checking de PCTL est montré pour

réduire la solution d'un problème linéaire d'optimisation. Bianco et De Alfaro [25] ont également présenté un algorithme pour le PCTL.

Dans [35], Baier et Kwiatkowska ont montré comment les algorithmes pour le model-checking des propriétés quantitatives peuvent être prolongées pour incorporer une notion d'équitabilité basée sur celle de Vardi [29]. Une autre amélioration de leur algorithme a été donnée par Baier dans [36].

Dans [31], Aziz et autres ont proposé la logique CSL pour fournir des moyens d'indiquer formellement des propriétés de CTMCs. CSL peut être vu comme prolongation de PCTL. Elle présente l'opérateur Until borné qui peut être employé pour raisonner au sujet du comportement en temps réel de CTMC.

Baier et autres [32] ont alors prolongé CSL en ajoutant un opérateur à la raison sur le comportement équilibré de CTMC et a présenté le premier modèle examinant l'algorithme pour assurer la logique. Des probabilités équilibrées sont calculées de la manière habituelle en résolvant un système d'équation linéaire. Pour l'opérateur Until borné, ils ont proposé un algorithme basé sur une méthode itérative pour la solution approximative d'un système d'équation d'intégrales de Volterra. Dans [37], Baier et autres ont présenté une méthode alternative de model-checking de l'opérateur Until borné de CSL en employant une conversion en analyse passagère. Ceci permet d'employer des techniques de calcul standards et efficaces. L'algorithme a été amélioré plus loin par Katoen et autres dans [38].

5 Conclusion

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 implantant des model-checkers capables de répondre à la question : « *le modèle du système satisfait-il la propriété ?* », 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 les modèles dont le nombre d'états est fini, le principal problème pour l'utilisation de model-checking est connu sous le nom d'explosion combinatoire du nombre d'états. La modélisation des systèmes réels (qui peuvent être complexes) conduit généralement à la création d'un nombre d'états tel

qu’il devient impossible de les conserver tous 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 des systèmes de grande taille.

Les HMMs connaissent un essor important. Ceci, en raison de nombreux avantages qu’ils offrent dans la possibilité d’intégrer correctement plusieurs sources de connaissance grâce à l’usage d’une théorie, telle que celle des probabilités, l’existence d’algorithmes d’apprentissage des modèles et la facilité de représenter des régularités statistiques de l’application à traiter.

Les logiques temporelles fournissent une manière de raisonnement au sujet du comportement qui change au cours du temps. Leur application dans l'informatique a été introduite par Pnueli qui a argué le fait que les techniques existantes pour la vérification formelle n'étaient pas proportionnées pour les systèmes embarqués. Nous avons adopté dans notre approche la logique temporelle POCTL. D'une part, POCTL est fondamentalement une prolongation de PCTL où l’opérateur next est équipé d'une contrainte d'observation. D'autre part, POCTL peut également être considérée comme variante de la logique temporelle ACTL, dans lequel l’opérateur next habituel est prolongé pour contraindre l'étiquette d'action de la transition.

Dans le chapitre suivant, nous allons voir l'un des outils les plus connus qui supportent le Model-checking probabiliste.

Chapitre 03 : Présentation du PRISM

1 Introduction

PRISM est un vérificateur de modèle probabiliste (model-checker), un outil pour la modélisation formelle et l'analyse de systèmes qui présentent un comportement aléatoire ou probabiliste. Il a été utilisé pour analyser des systèmes de nombreux domaines d'application différents, y compris les protocoles de communication et multimédias, les algorithmes distribués randomisés, les protocoles de sécurité, les systèmes biologiques et bien d'autres.

PRISM peut construire et analyser plusieurs types de modèles probabilistes :

• Discrete-Time Markov Chains (DTMCs).

• Continuous-Time Markov Chains (CTMCs).

• Markov Decision Processes (MDPs).

• Probabilistic Automata (PAs).

• Probabilistic Timed Automata (PTAs).

Ainsi que des extensions de ces modèles avec des coûts et des récompenses.

Les modèles sont décrits en utilisant le langage de PRISM, un langage simple basé sur l'état. PRISM prend en charge l'analyse automatisée d'un large éventail de propriétés quantitatives de ces modèles, par ex. "quelle est la probabilité qu'une défaillance provoquant l'arrêt du système dans les 4 heures?", "quelle est la pire probabilité que le protocole se termine par une erreur, sur toutes les configurations initiales possibles?", "quelle est la taille attendue de la file d'attente de messages après 30 minutes? ", ou" quel est le temps le plus défavorable prévu pour que l'algorithme se termine? ". Le langage de spécification de propriété intègre les logiques temporelles PCTL, CSL, LTL et PCTL*, ainsi que des extensions pour les spécifications quantitatives et les coûts / récompenses.

PRISM incorpore des structures de données symboliques et des algorithmes de pointe, basés sur les BDDs (Binary Decision Diagrams) et les MTBDD (Multi-Terminal Binary Decision Diagrams) . Il comprend également un moteur de simulation d'événement discret, fournissant un support pour la vérification de modèle approximatif / statistique, et des implémentations de

diverses techniques d'analyse différentes, telles que le raffinement d'abstraction quantitative et la réduction de symétrie.

PRISM est gratuit et open source, publié sous la licence publique générale GNU (GPL).

2 Une brève histoire

- Fin des années 80, début des années 90 : première théorie sous-jacente développée.
- Développé à Birmingham / Oxford Université, depuis 1999.
- 2001 : première sortie publique officielle de PRISM.
- 2011 : version 4.0 - systèmes temps réel probabilistes.
- 2013 : PRISM-games - jeux multi-joueurs stochastiques.

3 Thèmes et tendances

Thèmes dans le développement de PRISM :

- De la théorie à la pratique (et de la pratique à la théorie).
- Large collaboration (théorie, algorithmes, études de cas).
- Logiciels (et données) open source.
- Chevauche / interagit avec de nombreuses autres disciplines.

Les tendances :

- Amélioration de l'évolutivité vers des modèles plus grands.
- Classes de modèles de plus en plus expressives / puissantes.
- Des problèmes de vérification aux problèmes de contrôle.
- Une gamme toujours plus large de domaines d'application (algorithmes distribués aléatoires, réseaux / protocoles de communication, sécurité informatique, performance / fiabilité, systèmes biologie Calcul ADN, robotique et véhicules autonomes ,dispositifs portables / implantables , Vérification du stimulateur cardiaque).

4 Language de Prism :

Les composants fondamentaux du langage de PRISM sont les modules et les variables. Un modèle est composé d'un certain nombre de modules qui peuvent interagir les uns avec les

autres. Un module contient un certain nombre de variables locales. Les valeurs de ces variables à un instant donné constituent l'état du module. L'état global de l'ensemble du modèle est déterminé par l'état local de tous les modules. Le comportement de chaque module est décrit par un ensemble de commandes. Une commande prend la forme :

```
[] guard -> prob_1: update_1 + ... + prob_n: update_n;
```

La garde est un prédicat sur toutes les variables du modèle (y compris celles appartenant à d'autres modules). Chaque mise à jour décrit une transition que le module peut effectuer si la garde est vraie. Une transition est spécifiée en donnant les nouvelles valeurs des variables dans le module, éventuellement en fonction d'autres variables. Chaque mise à jour se voit également attribuer une probabilité (ou dans certains cas un taux) qui sera attribuée à la transition correspondante.

Indépendamment du choix des moteurs, PRISM propose souvent plusieurs méthodes de solution différentes qui peuvent être utilisées pour le calcul des probabilités et des coûts / récompenses attendus lors de la vérification du modèle. Beaucoup de ces méthodes, mais pas toutes, sont des méthodes numériques itératives. Le choix de la méthode (et de ses paramètres) dépend du type d'analyse en cours (c'est-à-dire du type de modèle et de propriété).

4.1 Identifiants :

Les noms donnés aux modules et aux variables sont appelés identificateurs. Les identificateurs peuvent être composés de lettres, de chiffres et du caractère de soulignement, mais ne peuvent pas commencer par un chiffre, c'est-à-dire qu'ils doivent satisfaire l'expression régulière, et sont à la casse- sensible. De plus, les identifiants ne peuvent pas être l'un des suivants, qui sont tous des mots-clés réservés dans PRISM: A, bool, clock, const, ctmc, C, double, dtmc, E, endinit, endinvariant, endmodule, endrewards, endsystem, false, formula, filter, func, F, global, G, init, invariant, I, int, label, max, mdp, min, module, X, nondeterministic, Pmax, Pmin, P, probabilistic, prob, pta, rate, rewards, Rmax, Rmin, R, S, stochastic, system, true, U, W.

4.2 Expression :

Les expressions peuvent contenir des valeurs littérales (12, 3.141592, vrai, faux, etc.), des identificateurs (correspondant à des variables, des constantes, etc.) et des opérateurs de la liste suivante :

- - (unaire moins).
- *, / (multiplication, division).
- +, - (addition, soustraction).
- <, <=,> =,> (opérateurs relationnels).
- =,! = (opérateurs d'égalité).
- ! (négation).
- & (conjonction).
- | (disjonction).
- <=> (si et seulement si).
- => (implication).
- ? (évaluation de la condition : condition? a: b signifie "si la condition est vraie alors a sinon b").

Tous ces opérateurs sauf « ? » sont associatifs à gauche (c'est-à-dire qu'ils sont évalués de gauche à droite). La priorité des opérateurs se trouve dans la liste ci-dessus, les opérateurs les plus fortement liés en premier. Les opérateurs sur la même ligne (par exemple + et -) ont la même priorité.

Une grande partie de la notation des expressions est donc essentiellement équivalente à celle de C / C ++ ou Java. Une exception notable à cela est que l'opérateur de division / effectue toujours une division en virgule flottante, pas en entier, c'est-à-dire que le résultat de 22/7 est 3,142857 ... pas 3. Toutes les expressions doivent s'évaluer correctement en termes de type (entier, double ou Booléen).

Les expressions peuvent utiliser plusieurs fonctions intégrées :

- min (...) et max (...), qui sélectionnent respectivement la valeur minimale et maximale de deux nombres ou plus.

- floor (x) et ceil (x), qui arrondissent respectivement x vers le bas et vers le haut à l'entier le plus proche.
- round (x), qui arrondit x à l'entier le plus proche (notez que dans un tie-break, on arrondit toujours vers le haut, par exemple round (-1,5) donne -1 et non -2).
- pow (x, y) qui calcule x à la puissance y.
- mod (i, n) pour les opérations modulo entières.
- log (x, b), qui calcule le logarithme de x en base b.

Des exemples de leur utilisation sont : min(x+1, x_max), max(a,b,c), floor(13.5), ceil(13.5), round(13.5), pow(9.0, 0.5), mod(1977, 100), log(123, 2.71828183).

Remarque : Pour la compatibilité avec les anciennes versions de PRISM, toutes les fonctions peuvent également être exprimées via le mot-clé func, par ex. func (floor, 13,5).

4.3 Frais et récompenses (en englais Costs and rewards) :

Nous augmentons les modèles avec des récompenses (ou, au contraire, des frais) qui sont des quantités à valeur réelle affectées aux états et / ou transitions. Ceux-ci peuvent avoir un large éventail d'interprétations possibles. Par exemple : temps écoulé, consommation électrique, taille de la file d'attente des messages, nombre de messages envoyés avec succès, bénéfice net,…

4.3.1 Frais? ou récompenses?

Mathématiquement, aucune distinction entre les récompenses et les coûts, lors de l'interprétation, nous supposons qu'il est souhaitable de minimiser les frais et pour maximiser les récompenses.

4.3.2 Propriétés :

Dans la figure 3.1, on va savoir raison de la récompense cumulative / instantanée attendue

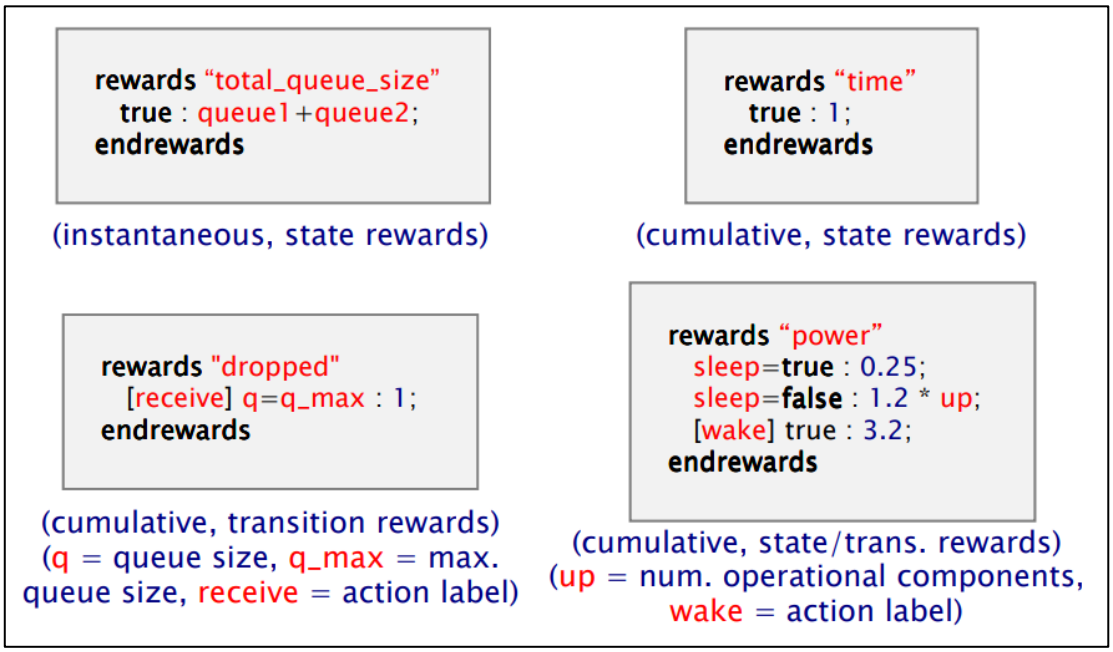

Figure 3.1 : les raison de la récompense (rawards)

4.4 Spécification de la propriété :

Langage de spécification de propriété basé sur la logique temporelle : subsume PCTL, CSL, LTL probabiliste, PCTL *, …

Simple exemples :

- P≤0.01 [F "crash"] :C'est-à-dire la probabilité d'un accident (crach) est d'au plus 0,01 S>0.999 ["up"])"long-run probability of availability is >0.999".

Concentrez-vous généralement sur les propriétés quantitatives (numériques) :

- P =? [F "crash"] «Quelle est la probabilité d'un crash? ».

Les propriétés peuvent combiner des aspects numériques + exhaustifs.

- Pmax=? [F≤10 "fail"] : " dans le pire des cas, une panne se produisant dans les 10 secondes, pour toute planification possible des composants du système

5 Des exemples de modèles probabilistes en langage de Prism :

5.1 Exemple du modèle DTMC :

Algorithme probabiliste simple dés de Knuth et Yao. Pour émuler un dé à 6 faces avec une pièce juste. Voici une illustration graphique de l'algorithme dans la figure 3.2 :

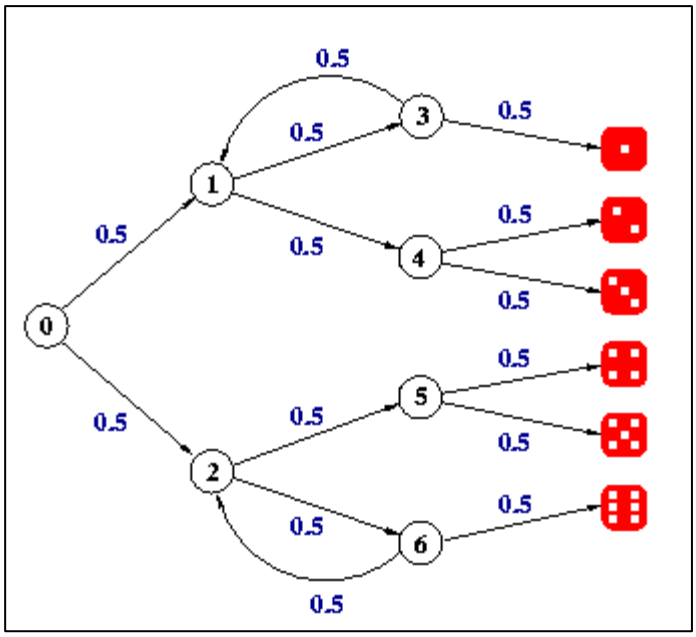

Figure 3.2 : model DTMC de dés

L'idée de l'algorithme est la suivante. À partir de l'étape 0 (c'est-à-dire le cercle étiqueté 0), à chaque étape une pièce est lancée et il y a 50% de chances de prendre chacun des deux choix possibles. L'algorithme se termine lorsque vous atteignez l'une des valeurs du dé sur le côté droit.

Le code PRISM pour modéliser cet algorithme est illustré ci-dessous dans la figure 3.3 :

```
dtmc

module die

	// état local

	s : [0..7] init 0;
	//  valeur du dé

	d : [0..6] init 0;

	[] s=0 -> 0.5 : (s'=1) + 0.5 : (s'=2);
	[] s=1 -> 0.5 : (s'=3) + 0.5 : (s'=4);
	[] s=2 -> 0.5 : (s'=5) + 0.5 : (s'=6);
	[] s=3 -> 0.5 : (s'=1) + 0.5 : (s'=7) & (d'=1);
	[] s=4 -> 0.5 : (s'=7) & (d'=2) + 0.5 : (s'=7) & (d'=3);
	[] s=5 -> 0.5 : (s'=7) & (d'=4) + 0.5 : (s'=7) & (d'=5);
	[] s=6 -> 0.5 : (s'=2) + 0.5 : (s'=7) & (d'=6);
	[] s=7 -> (s'=7);

endmodule
```

Figure 3.3 : code prism de l'algorithme de dés

La première ligne indique que ce modèle est une chaîne de Markov en temps discret (DTMC). Les lignes restantes décrivent un seul module PRISM, que nous utilisons pour modéliser le comportement de l'algorithme. Nous utilisons deux variables pour représenter l'état du système : s, indiquant quelle étape de l'algorithme est en cours d'exécution (c'est-à-dire quel cercle dans le diagramme ci-dessus), et d, indiquant la valeur du dé (0 signifie qu'aucune valeur n'a été choisi encore).

Nous allons maintenant utiliser PRISM pour effectuer une analyse du modèle par exemple choisie cette propriété (P=? [F s=7 & d=5]) Ce qui signifie quelle est la probabilité À la fin à condition de s=7 et d = 5, Et lorsque lancer la vérification, nous obtenir la valeur de probabilité

P= 0.16666666865348816 illustré ci-dessous dans la figure 3.4.

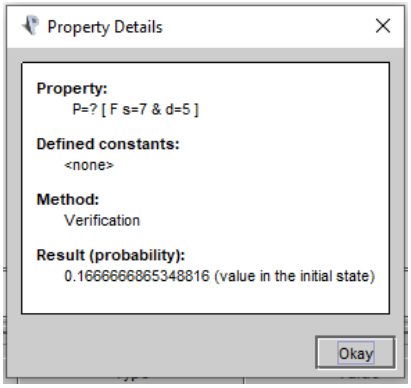

Figure 3.4: vérification de propriété (P=? [F s=7 & d=5]).

5.1 Exemple du modèle CTMC :

Nous présentons maintenant un deuxième exemple: un CTMC qui modélise une file d'attente de travaux à N emplacements et un serveur qui supprime les travaux de la file d'attente et les traite.

Le code PRISM pour modéliser cet algorithme est illustré ci-dessous dans la figure 3.5:

```
ctmc

const int N = 10;
const double mu = 1/10;
const double lambda = 1/2;
const double gamma = 1/3;

module queue
      q : [0..N];

      [] q<N -> mu:(q'=q+1);
      [] q=N -> mu:(q'=q);
      [serve] q>0 -> lambda:(q'=q-1);
endmodule

module server
      s : [0..1];

      [serve] s=0 -> 1:(s'=1);
      [] s=1 -> gamma:(s'=0);
endmodule
```

Figure 3.5 : serveur traiter file d'attente

Cet exemple présente également un certain nombre d'autres concepts du langage PRISM, notamment les étiquettes d'action et la synchronisation (**[**serve**]** s=0 -> 1**:(**s'=1**)**

dans le module server et **[**serve**]** q>**0** -> lambda**:(**q'=q-1**)** dans le module queue exécutent en même temps car ils y a le même étiquette d'action qui est : serve) .

5.2 Exemple du modèle MDP :

Cet exemple applique PRISM à la spécification et à l'analyse d'un modèle de processus de décision de Markov (MDP) qui est Le problème d'alimentation des philosophes.

5.2.1 Le problème d'alimentation des philosophes :

Le problème d'alimentation des philosophes est un exemple d'une grande classe de problèmes de concurrence qui tentent de gérer l'allocation d'un nombre défini de ressources entre plusieurs processus. Le problème vient d'Edsger Dijkstra, qui en 1971 a posé une question d'examen où cinq ordinateurs se disputaient l'accès à cinq lecteurs de bande partagés. Le problème a ensuite été raconté par Tony Hoare comme Le problème d'alimentation des philosophes.

Le problème suppose un certain nombre de philosophes, assis autour d'une table circulaire, qui passent leur vie à alterner entre penser et manger. Il y a une fourchette placée entre chaque paire de philosophes voisins à laquelle les deux peuvent accéder et un philosophe doit être en possession des deux fourchettes pour manger. Un philosophe ne peut prendre qu'une fourchette à la fois et lorsqu'un philosophe a fini de manger, les deux fourchettes sont replacées sur la table et il / elle revient à la réflexion. Remarquez que, puisque les philosophes partagent des fourchettes, il n'est pas possible pour tous de manger en même temps.

Ci-dessous, la figure 3.6 nous représentons graphiquement le cas où cinq philosophes sont assis autour de la table.

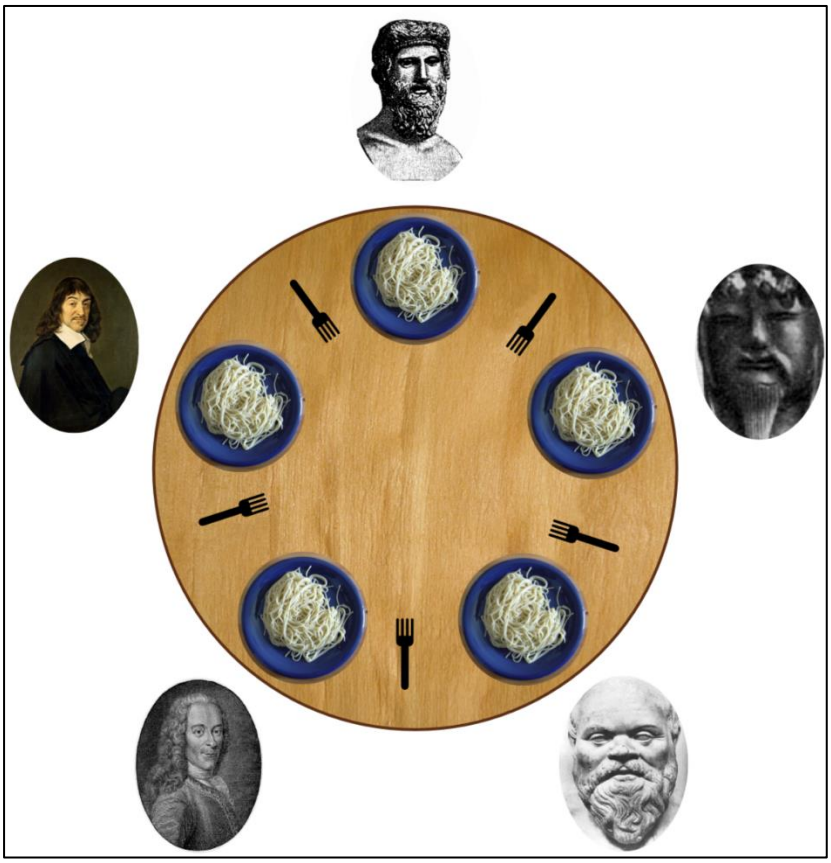

Figure 3.6 : table de philosophes

Une solution à ce problème est un algorithme à suivre pour chacun des philosophes qui garantira que :

- Il n'y aura jamais de situation d'impasse (c'est-à-dire où aucun philosophe ne peut manger).
- il n'y aura jamais de situation de manque de ressources (c'est-à-dire où un philosophe veut manger mais ne le fait jamais).
- Pour un exemple de situation qui peut conduire à une impasse, supposons que tous les philosophes aient faim en même temps et tendent la main et prennent leur fourchette gauche, alors si tous atteignent la fourche droite, personne ne peut bouger.

5.2.2 Une solution aléatoire

Nous considérons maintenant la solution aléatoire de Lehmann et Rabin à ce problème. Cela garantit que le système est sans impasse, c'est-à-dire que, si un philosophe a faim, un philosophe finit par manger.

Le modèle PRISM dans le cas de trois philosophes est donné ci-dessous :

```
Mdp

// formulae

// left fork free (left neighbour is philosopher 2)

formula lfree = (p2>=0 & p2<=4) | p2=6 | p2=10;

// right fork free (left neighbour is philosopher 3)

formula rfree = (p3>=0 & p3<=3) | p3=5 | p3=7 | p3=11;

module phil1

        p1: [0..11];

        [] p1=0 -> (p1'=1); // trying

        [] p1=1 -> 0.5 : (p1'=2) + 0.5 : (p1'=3); // draw randomly

        [] p1=2 &  lfree -> (p1'=4); // pick up left

        [] p1=3 &  rfree -> (p1'=5); // pick up right

        [] p1=4 &  rfree -> (p1'=8); // pick up right (got left)

        [] p1=4 & !rfree -> (p1'=6); // right not free (got left)

        [] p1=5 &  lfree -> (p1'=8); // pick up left (got right)

        [] p1=5 & !lfree -> (p1'=7); // left not free (got right)

        [] p1=6  -> (p1'=1); // put down left

        [] p1=7  -> (p1'=1); // put down right

        [] p1=8  -> (p1'=9); // move to eating (got forks)

        [] p1=9  -> (p1'=10); // finished eating and put down left

        [] p1=9  -> (p1'=11); // finished eating and put down right

        [] p1=10 -> (p1'=0); // put down right and return to think

        [] p1=11 -> (p1'=0); // put down left and return to think

endmodule

// construct further modules through renaming

module phil2 = phil1 [ p1=p2, p2=p3, p3=p1 ] endmodule

module phil3 = phil1 [ p1=p3, p2=p1, p3=p2 ] endmodule


// labels

// a philosopher is hungry

label "hungry" = ((p1>0)&(p1<8))|((p2>0)&(p2<8))|((p3>0)&(p3<8));

 // a philosopher is eating

label "eat" =
((p1>=8)&(p1<=9))|((p2>=8)&(p2<=9))|((p3>=8)&(p3<=9));
```

```
// philosopher 1 is
hungry

label "hungry1" =
(p1>0)&(p1<8);

// philosopher 2 is
hungry

label "hungry2" =
(p2>0)&(p2<8);

// philosopher 3 is
hungry

label "hungry3" =
(p3>0)&(p3<8);

 // philosopher 1 is
eating

label "eat1" =
(p1>=8)&(p1<=9);

 // philosopher 2 is
eating

label "eat2" =
(p2>=8)&(p2<=9);

 // philosopher 3 is
eating

label "eat3" =
(p3>=8)&(p3<=9);
```

Figure 3.7 : model mdp sur prism de table de philosophes

Ce modèle est un processus de décision de Markov (MDP), comme indiqué par le mot-clé au début du modèle. Nous avons trois modules, représentant les trois philosophes.

Notez que, après qu'un philosophe mange (par exemple lorsque p1 = 9 pour le philosophe 1), le choix non déterministe de la fourchette qu'il pose en premier est représenté par deux commandes activées (c'est-à-dire qu'il y a plusieurs commandes avec une garde qui se chevauchent p1 = 9).

Notez également l'utilisation des formules PRISM (lfree et rfree) pour représenter des expressions qui sont réutilisées plusieurs fois dans le fichier. Dans ce modèle, ces formules sont utilisées pour identifier les états du voisin gauche (ou droit) d'un philosophe dans lequel il n'utilise pas sa fourchette. En conséquence, cela nécessite de renommer chacune des variables p1, p2 et p3 lors de la construction des modules pour les philosophes 2 et 3. Ce changement de nom correspond au fait que les voisins gauche et droit du philosophe 2 et 3 diffèrent de celui du philosophe 1.

Les étapes de base de l'algorithme pour chaque philosophe sont :

1. passer de la pensée à la faim (pour simplifier l'analyse, nous supposons que les philosophes ont toujours faim) ;
2. quand vous avez faim, choisissez au hasard d'essayer de prendre la fourche gauche ou droite ;
3. attendez que la fourche soit abaissée, puis prenez-la ;
4. si l'autre fourchette est libre, prenez-la ; sinon, abaissez la fourche d'origine (et revenez à l'étape 1);
5. manger (cas en possession des deux fourchettes) ;
6. lorsque vous avez fini de manger, abaissez les deux fourchettes dans n'importe quel ordre et recommencez à réfléchir.

La figure 3.8 ci-dessous clarifier une simulation à valeur de temps avec prism pour différentes situations des philosophes(0..11).

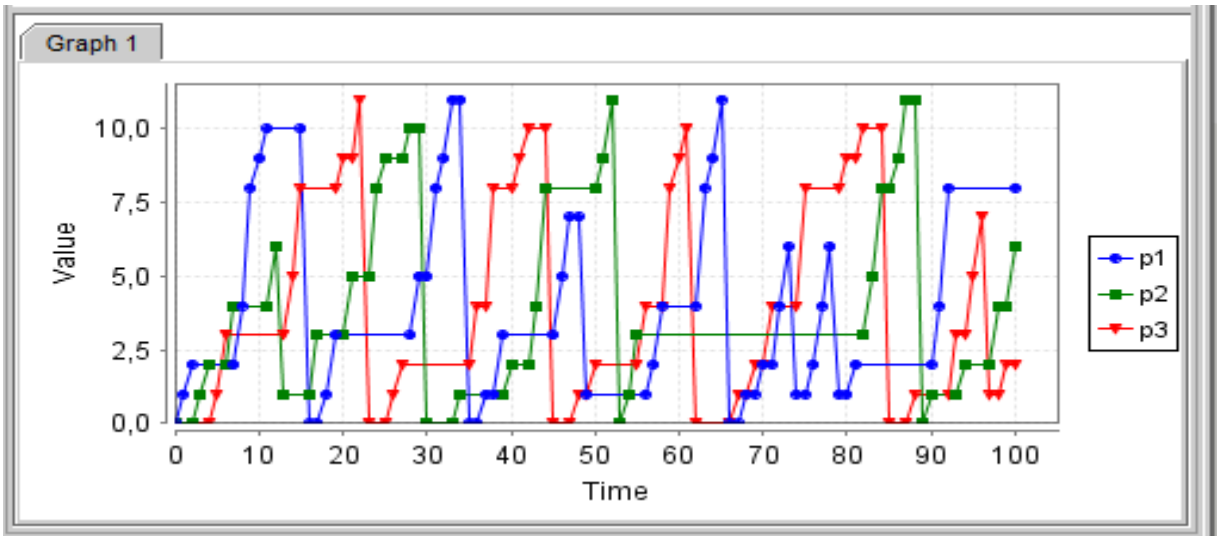

Figure 2.8 : une simulation à valeur de temps de situation des philosophes

- Le courbe bleu : simulation de situation de philosophe 1.
- Le courbe vert : simulation de situation de philosophe 2.
- La courbe rouge : simulation de situation de philosophe 3.

5.2.3 Vérification des propriétés avec Prism :

- Propriété 01 : Il n'y a pas de famine pour tout le monde. S'il y a faim, il y a un processus d'alimentation (propriété d'impasse) décrit comme suit :

filter (forall, "hungry" => P>=1 [F "eat"]).

- Propriété 02 : il y a de philosophe faim qui mange toujours (propriété de famine du voisin) décrit comme suit :

Pour le philosophe 1 : filter (forall, "hungry1" => P>=1 [F "eat1"]).

Pour le philosophe 2 : filter (forall, "hungry2" => P>=1 [F "eat2"]).

Pour le philosophe 3 : filter (forall, "hungry3" => P>=1 [F "eat3"]).

Ci dissous, la figure 3.9 nous montre les résultats de vérification de ces propriétés par Prism.

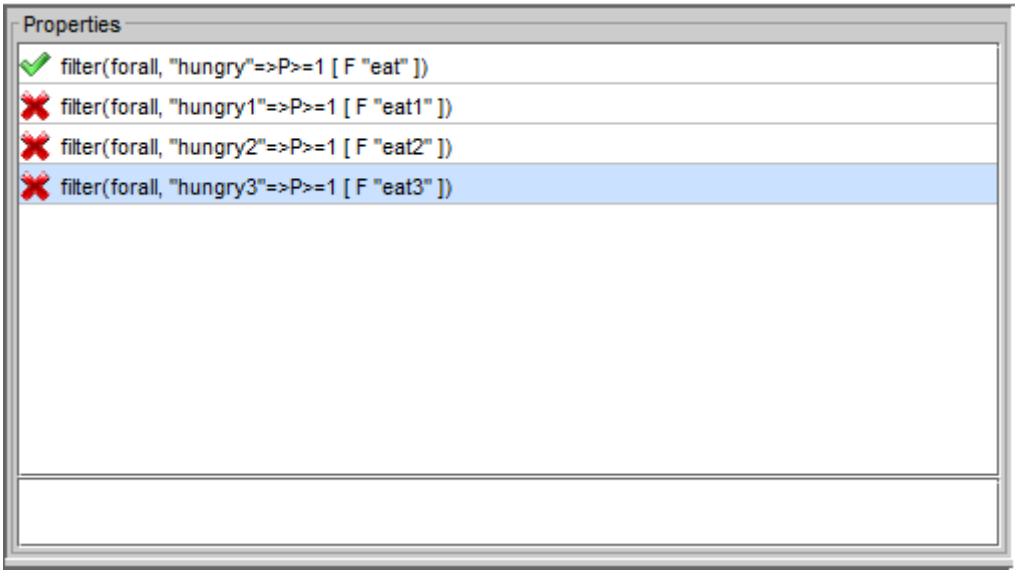

Figure 3.9 : Résultats de vérification des propriétés par Prism

En remarque que la propriété 01 vrai et la propriété 02 est faux pour tous les philosophes. Logiquement les résultats sont vrais car il n'y a pas de philosophe qui ne mange pas (propriété 02) et en cas de faim, il y a au moins un philosophe qui mange.

5.3 Exemple PTA:

Jusqu'à présent, nous nous sommes principalement concentré sur trois types de modèles : DTMC, MDP et CTMC. PRISM prend également en charge un quatrième les automates temporisés probabilistes (PTA), qui étendent les MDP avec la capacité de modéliser le comportement en temps réel. Ceci est fait dans le style des automates chronométrés [AD94], en ajoutant des horloges à valeur réelle qui augmentent avec le temps et peuvent être réinitialisées

Avant de décrire comment les fonctionnalités PTA sont incorporées dans le langage de modélisation PRISM, nous donnons un exemple simple. Voici un petit PTA.

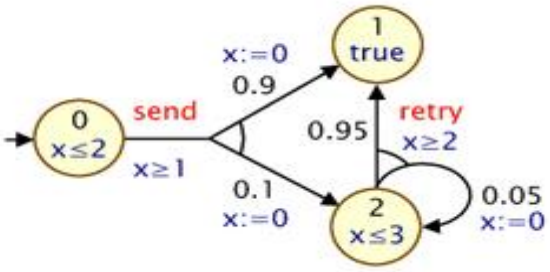

Figure 3.10 : exemple de automate temporisé probabiliste (PTA).

est ci-dessous ce model prism

```
pta

module M

    s :[0..2] init 0;
    x : clock;

    invariant
        (s=0 => x<=2)&(s=2 => x<=3)
    endinvariant

    [send] s=0 & x>=1 -> 0.9:(s'=1)&(x'=0)+ 0.1:(s'=2)&(x'=0);
    [retry] s=2 & x>=2 -> 0.95:(s'=1)+ 0.05:(s'=2)&(x'=0);

endmodule
```

Pour modéliser les PTA dans PRISM, il existe un nouveau type de données, **clock**, utilisé pour les variables qui sont des horloges. Celles-ci doivent être locales à un module particulier et non globales. D'autres types de variables PRISM peuvent être définis de la manière habituelle. Dans l'exemple ci-dessus, nous n'utilisons qu'une seule variable entière s pour représenter les emplacements des PTA.

Dans un PTA, les transitions peuvent inclure une garde, qui contraint le moment où elle peut se produire en fonction de la valeur actuelle des horloges, et des réinitialisations, qui spécifient que les valeurs d'une horloge doivent être définies sur une nouvelle valeur (entière). Celles-ci sont toutes deux spécifiées dans les

commandes PRISM de la manière habituelle : voir, par exemple, l'inclusion de x> = 1 dans la garde pour la commande send-étiquetée et les mises à jour du formulaire (x '= 0) qui réinitialisent **clock** x à 0.

L'autre ajout est une construction **invariant**, qui est utilisée pour spécifier une expression décrivant les invariants de **clock** pour chaque module PRISM. Celles-ci imposent des restrictions sur les valeurs autorisées des variables d'horloge, en fonction des valeurs des autres variables non de **clock**. La construction invariante doit apparaître entre les déclarations de variables et les commandes du module. Souvent, les invariants d'horloge sont décrits séparément pour chaque emplacement PTA; par conséquent, l'invariant prendra souvent la forme d'une conjonction d'implications, comme dans l'exemple de modèle ci-dessus, mais des expressions plus générales sont également autorisées. Dans l'exemple, l'horloge x doit satisfaire x <= 2 ou x <= 3 lorsque les variables locales s sont respectivement 0 ou 2. Si s vaut 1, il n'y a pas de restriction (puisque l'invariant est effectivement vrai dans ce cas).

6 Conclusion

Dans ce chapitre, nous avons pu en apprendre davantage sur Prism et ce que nous pouvons faire en l'utilisant à partir construire et analyser plusieurs types de modèles probabilistes (CTMCS, DTMCS, MDP, PAS, PTAS), ainsi que sur le langage qu'il utilise à travers d'Identifiants et expressions. Nous avons également fourni un ensemble d'exemples de types modèles probabilistes et vérifié des propriétés et faire une simulation de modèle.

Prism a un site Web officiel « https://www.prismmodelchecker.org/ » à partir duquel nous pouvons le télécharger ou pour rechercher plus d'informations.

Maintenant, dans le chapitre suivant nous ferons une étude de cas de Prism sur un système embarqué choisi.

Chapitre 04 : Étude de cas

1 Introduction

Depuis quelques années, de nombreux outils ont vu le jour et ont permis de réaliser de nombreuses études de cas. Notre objectif est de vérifier un vrai système par la réalisation d'une étude de cas de l'outil qualifié le plus célèbre et le plus utilisé dans le domaine de vérification à savoir PRISM.

Lors d'une étude de cas, la partie « modélisation » est difficile : il n'y a pas vraiment de règles générales, mais elle se fait au cas par cas. Un cahier des charges décrivant le système réel étant fourni, la première étape consiste à réaliser une première modélisation assez grossière du système : cela sert à se rendre compte quelle classe de modèles paraît bien adaptée à la représentation du système et cela permet aussi éventuellement de choisir déjà un outil qui permettra d'implémenter la modélisation. La deuxième phase consiste à faire une modélisation complète du système, en l'adaptant éventuellement aux choix qui viennent d'être faits.

Dans ce chapitre, nous présentons une étude de cas [39] d'un système embarqué réel intéressant qui est le télescope spatial Hubble [40] (en anglais, *Hubble Space Telescope* ou HST). Cet exemple [41] (une adaptation de [42]) modélise le comportement d'échec du télescope spatial *Hubble*. Nous commençons par une présentation du télescope HST, puis nous décrivons le système réel, constitué de composants qui peuvent probablement échouer. Une fois que le système est compris, nous faisons la modélisation de son comportement d'échec, de sorte que nous puissions prévoir son comportement et poser la question comme : « Quelle est la probabilité que la prochaine observation est une observation d'un échec de gyroscope ? ».

2 Le télescope spatial Hubble

Le télescope spatial *Hubble* (En anglais, *Hubble Space Telescope* ou *HST*) est un télescope en orbite à environ 600 kilomètres d'altitude, il effectue un tour complet de la Terre toutes les 100 minutes. Il est nommé en l'honneur de l'astronome Edwin Hubble. Son lancement, effectué le 24 avril 1990 par une navette spatiale, est le fruit d'un long travail de recherche de la NASA et de l'ESA.

2.1 Description technique

Le télescope Hubble pèse environ 11 tonnes, fait 13,2 mètres de long, a un diamètre maximum de 4,2 mètres et a coûté 2 milliards de dollars US. C'est un télescope réflecteur à deux miroirs ; le miroir primaire a un diamètre d'environ 2,4 mètres. Il est couplé à divers spectromètres et trois caméras : la première, champ étroit pour les objets faiblement lumineux, la seconde, à large champ pour les images planétaires et la troisième pour l'infrarouge. Le télescope a été lancé le 24 avril 1990 par la mission STS-31 de la navette spatiale Discovery. Ce lancement avait déjà été retardé en 1986 à cause de la catastrophe de la navette spatiale Challenger en janvier de la même année.

2.2 Missions d'entretien

Le télescope a été visité quatre fois par des astronautes opérant à partir de navettes spatiales afin de corriger les défauts de fonctionnement et d'installer de nouveaux équipements. De plus, en raison du freinage atmosphérique, le télescope perd lentement de l'altitude (et gagne de la vitesse). Nous profitons donc de chacune de ces visites d'entretien pour replacer le télescope à une orbite plus haute à l'aide de la navette.

3 Le Système réel

Le système a une unité de direction avec six gyroscopes, qui sont utilisés pour viser le télescope. La redondance est une question importante à la conception des systèmes embarqués tels que les télescopes spatiaux, puisqu'il est évident que l'exécution des réparations ne soit pas insignifiante. C'est pourquoi le télescope est conçu de telle manière qu'il reste en fonction avec la pleine exactitude quand seulement trois de ses six gyroscopes sont opérationnels. Au-dessous de trois gyroscopes opérationnels, le télescope se transforme en mode *sommeil*, signifiant que les réparations sont nécessaires. Si aucun des gyroscopes n'est opérationnel, le télescope se brise. Aucune mission de réparation ne sera entreprise, tant qu'il y a plus de deux gyroscopes opérationnels [41].

4 Modélisation avec le modèle HMM

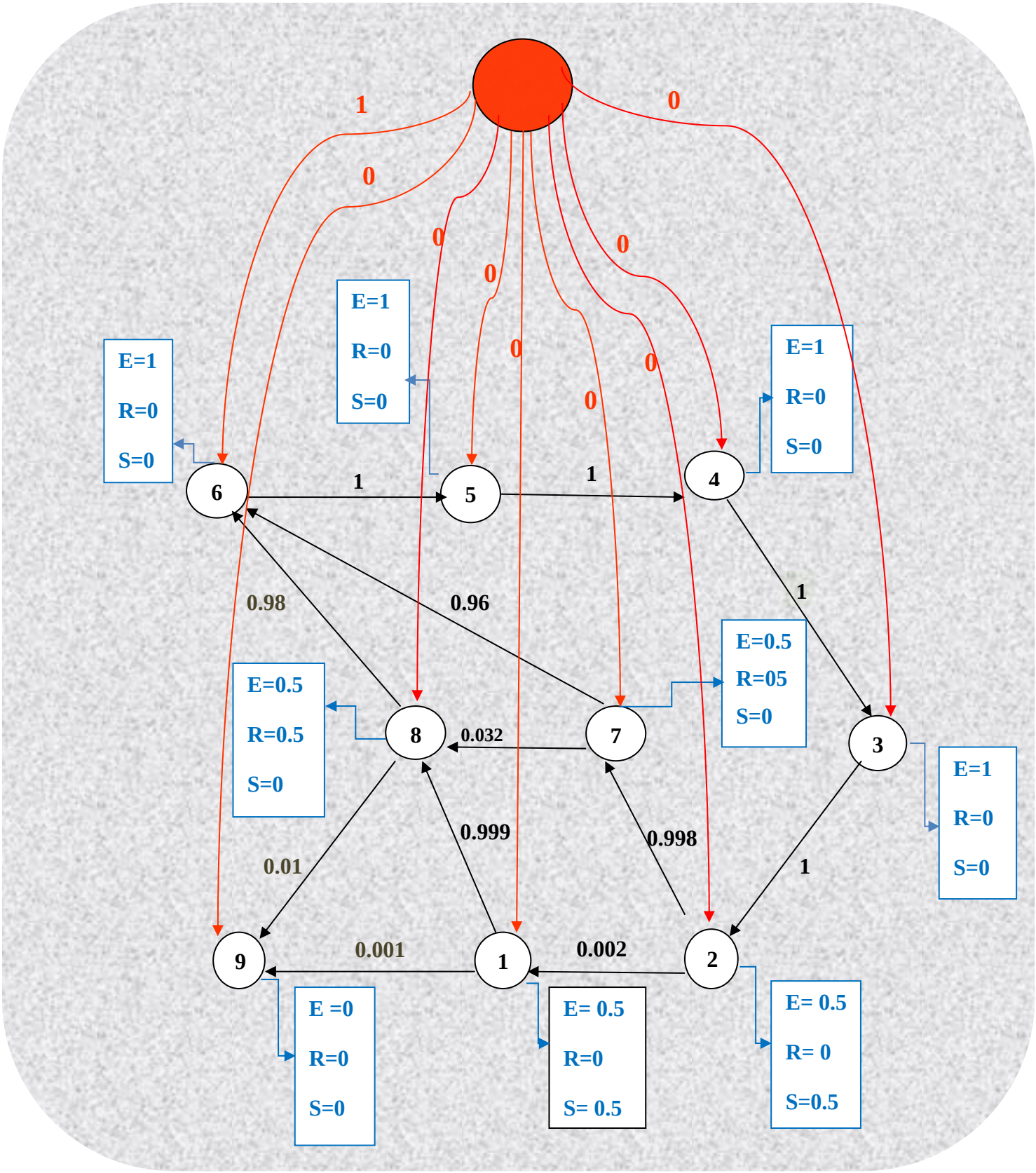

Figure 4.1 : Le modèle HMM du télescope spatial Hubble (HST)

A partir du DTMC du HST présenté dans [41], nous avons construit son modèle HMM illustré en figure 4.1. Le modèle a un total de neuf états.

- L'ensemble de tous les états est : S = {1, 2, 3, 4, 5, 6, 7, 8, 9}.
- Les propositions atomiques qui sont « true » dans chaque état :

  Les états 1 à 6 : les propositions atomiques égalent le nombre de gyroscopes opérationnels.

  - S_1 : un
  - S_2 : deux
  - S_3 : trois
  - S_4: quatre
  - S_5: cinq
  - S_6: six.

- Les états 7, 8, et 9 sont marqués sommeil2, sommeil1 et accident.

  - S_7 : sommeil2 signifie que le télescope est en mode *sommeil* de niveau 2.
  - S_8 : sommeil1 signifie que le télescope est en mode *sommeil* de niveau 1.
  - S_9 : accident signifie que le télescope est en *accident* (destruction).

- L: est une fonction associant à chaque état s, L(s) l'ensemble des propositions atomiques vérifiées par s :

  $L(S_1) = \{un\}$, $L(S_2) = \{deux\}$, $L(S_3) = \{trois\}$, $L(S_4) = \{quatre\}$, $L(S_5) = \{cinq\}$, $L(S_6) = \{six\}$, $L(S_7) = \{sommeil2\}$, $L(S_8) = \{sommeil1\}$, $L(S_9) = \{accident\}$.

  Elle est représentée par la matrice L définie comme suite (figure 4.2) :

L =	un	deux	trois	quatre	cinq	six	sommail2	sommail1	accident
S_1	1	0	0	0	0	0	0	0	0
S_2	0	1	0	0	0	0	0	0	0
S_3	0	0	1	0	0	0	0	0	0
S_4	0	0	0	1	0	0	0	0	0
S_5	0	0	0	0	1	0	0	0	0
S_6	0	0	0	0	0	1	0	0	0
S_7	0	0	0	0	0	0	1	0	0
S_8	0	0	0	0	0	0	0	1	0
S_9	0	0	0	0	0	0	0	0	1

Figure 4.2 : La matrice L du HMM du télescope HST

- L'ensemble des observations θ = {E, R, S}.

  Les observations sont :

  E (Echec) : représente une observation d'un échec d'un gyroscope.
  R (Réparation) : représente une observation d'une mission de réparation entreprise avec succès.

  S (Sommeil) : représente une observation qui signifie que le télescope va entrer dans le mode de *sommeil* indiquant que les réparations sont nécessaires.

- La matrice P de probabilités de transition entre les états est comme suite (figure 4.3) :

$$P = \begin{array}{c} \\ S_1 \\ S_2 \\ S_3 \\ S_4 \\ S_5 \\ S_6 \\ S_7 \\ S_8 \\ S_9 \end{array} \begin{array}{c} \begin{array}{ccccccccc} S_1 & S_2 & S_3 & S_4 & S_5 & S_6 & S_7 & S_8 & S_9 \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.999 & 0.001 \\ 0.002 & 0 & 0 & 0 & 0 & 0 & 0.998 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.968 & 0 & 0.032 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.984 & 0 & 0 & 0.016 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

Figure 4.3- La matrice P du HMM du HST

La matrice μ de probabilités des observations est définie comme suite (figure 4.4) :

$$\mu = \begin{pmatrix} 0.5 & 0 & 0.5 \\ 0.5 & 0 & 0.5 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0.5 & 0.5 & 0 \\ 0.5 & 0.5 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

Figure 4.4 : La matrice μ du HMM correspondante

- π est le vecteur de distribution des probabilités de transitions initiales. L'état 6 est le seul état initial où tous les gyroscopes sont opérationnels ; c'est également l'état du télescope après une mission réussie de réparation. Le vecteur π s'écrit (figure 4.5) :

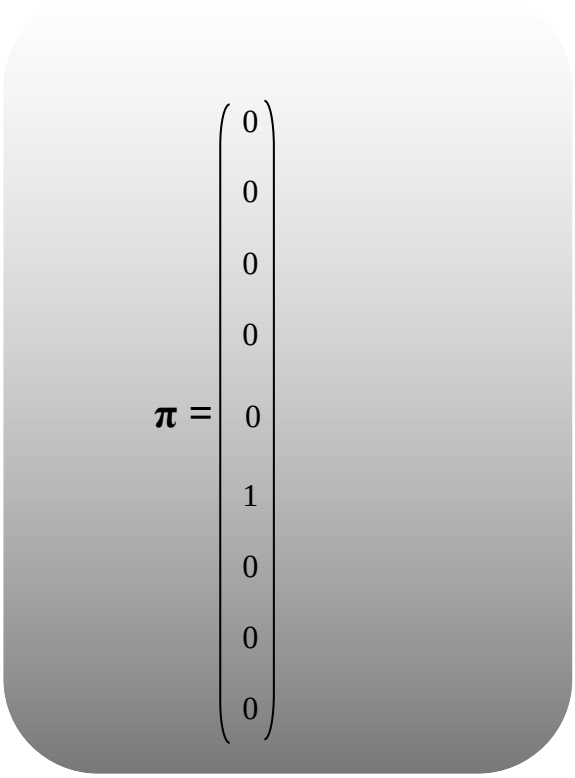

Figure 4.5 : Le vecteur π du HMM du télescope HST

5 Choix des propriétés à vérifier

Parmi les propriétés vérifiées par notre outil, nous avons choisi de présenter les trois propriétés suivantes :

- **La propriété 1 :**

  « La probabilité que la prochaine observation est une observation d'un échec de gyroscope et le chemin suivant satisfait que le nombre des gyroscopes opérationnels est différent de six, est inférieure ou égale à 1. ». C'est une propriété d'*atteignabilité*.

- **La propriété 2 :**

  « La probabilité que la prochaine observation est une observation d'un échec de gyroscope et le chemin suivant satisfait que le télescope se brisera est supérieure ou égale à 0 ». C'est une propriété de *sûreté*.

- **La propriété 3 :**

  « La probabilité que la prochaine observation est une observation d'un échec de gyroscope et le chemin suivant satisfait que le télescope se brisera est inférieure ou égale à 0.008. c.à.d. la probabilité que le télescope se brisera est inférieure ou égale à 0.008 ». C'est une propriété de *sûreté*.

6 Spécification des propriétés choisies en POCTL

Nous avons spécifié chaque propriété en suivant la syntaxe de logique POCTL comme suit :

- **La propriété 1 :** $P_{<= 1}\,[X_E\ \ !(six)]$
- **La propriété 2 :** $P_{>=0}\,[X_E\ \ (accident)]$
- **La propriété 3 :** $P_{<= 0.008}\,[X_E\ (accident)]$

7 Intégration du modèle HMM de HST dans PRISM :

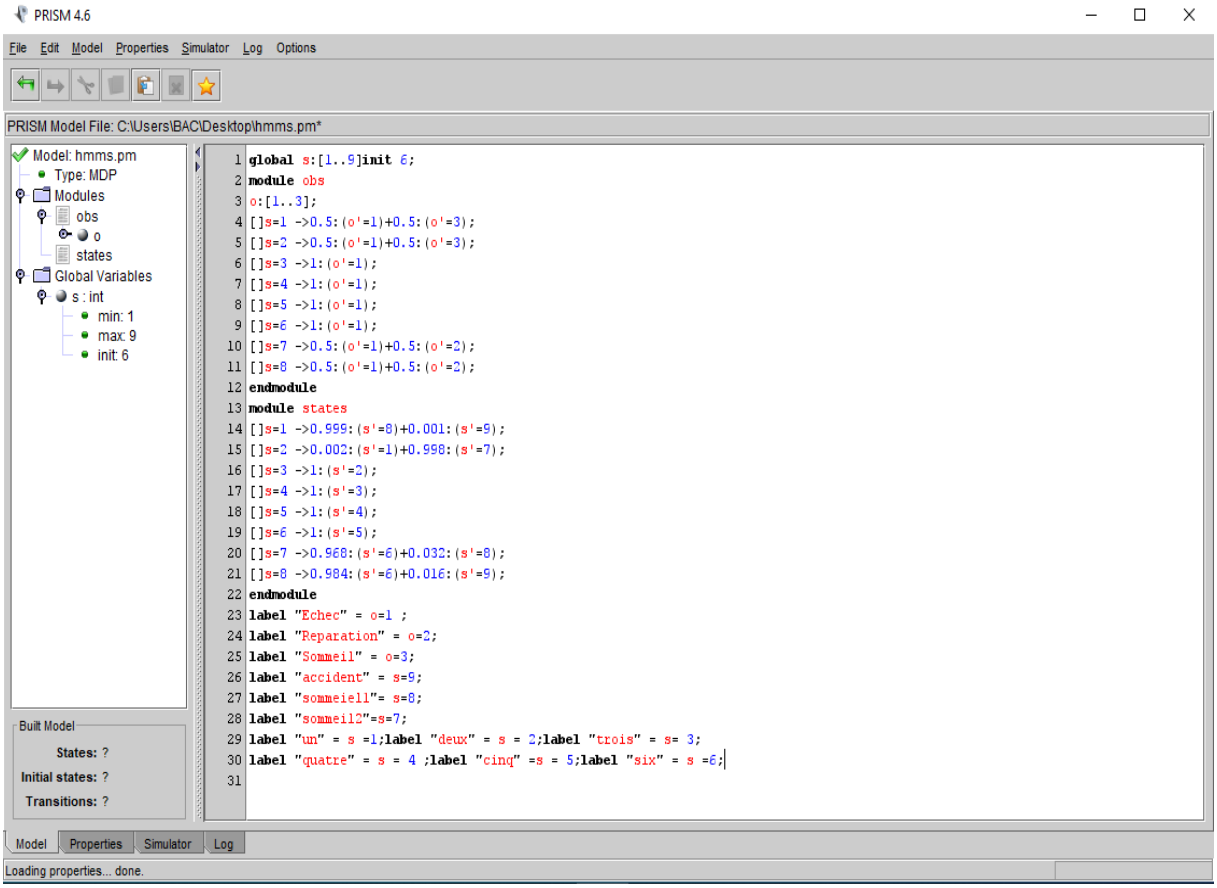

Figure 4.6: Le HMM du télescope HST intégré dans PRISM.

7.1 Explication de code :

1 er ligne : déclaration s global variable de type tableau de taille 9 du 1 au 9 initialiser a 6(déclaration de 9 states de hmm HST) [s global car utiliser en model observation (2…12) et en model state (13…22); initialiser a 6 car state 6 la seule initiale dans hmm(figure 4.5].

2^{eme} …12^{eme} ligne (model observation) : c'est une traduction de matrice µ de probabilités des observations définie en (figure 4.4) le variable o exprime les observation (o=1[observation d'un échec « ligne 23 »], o=2[observation d'un réparation « ligne 24 »], o=3[observation d'un sommeil « ligne 25 ») par exemple dans la ligne 5 :si s= 1 alors o=1(« échec »)avec probabilité de 0.5 ou o =3(« sommeil »)avec probabilité de 0.5.

15^{eme} …. 24^{eme} (model state) : c'est une traduction de matrice P de probabilités de transition entre les états définis en (figure 4.3) par exemple dans la ligne 16 :si s= 1 alors il va transite vers s=8 avec probabilité de 0.999 ou vers s =9 avec probabilité de 0.001.

26^{eme} …. 30^{eme} : c'est une traduction de matrice L du HMM du télescope *HST* définie en (Figure 4.2) par exemple label "accident" = s=9⇔L(S_9)=accident.

8 Intégration des propriétés choisies à vérifier dans PRISM

propriété en logique POCTL	propriété en langage de PRISM
La propriété 1 : $P_{<= 1}$ [X_E !(six)]	"property01" : P<=1 [(X "Echec")&(G (!"six"))]
La propriété 2 : $P_{>=0}$ [X_E (accident)]	"property02" : P>=0 [(X "Echec")&(G ("accident"))]
La propriété 3 : $P_{<= 0.008}$ [X_E (accident)]	"property03":P<=0.008[(X"Echec")&(G("accident"))]

Tableau 4.1 : propriétés en logique POCTL et leur Traduction en PRISM

8.1 Une simulation de modèle en PRISM :

Une simulation du modèle HMM de HST à valeur de temps (simulation des êtas et d'observations définis comme suite (figure 4.7) :

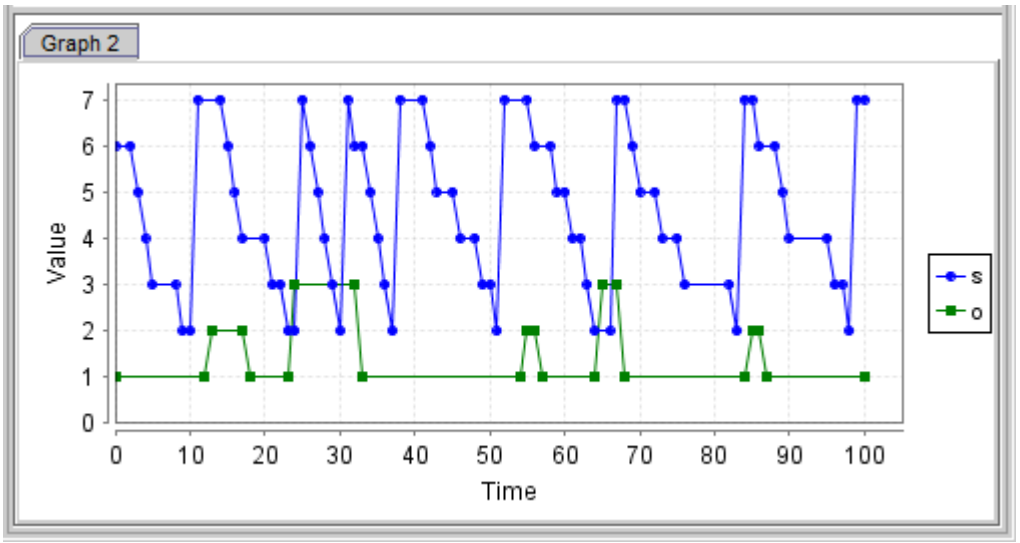

Figure 4.7 :Une simulation de modèle en prism

9 Vérification des propriétés

9.1 Vérification manuelle des propriétés :

- **La propriété 1 :** $P_{<= 1}$ [X_e !(six)].

s satisfait P<=1 (Xe !(six)) ssi ps(Xe !(six)) <=1.

On a: ps(XΩ□) = μs(Ω) . ∑s'□ sat (□) p(s,s') , □!=six donc s' !={s6}

Dans ce qui suit, chaque ligne correspond à la satisfaction de la propriété au sein d'un état (si, i=1-9)

ps1 [Xe !(six)] = μs1(e) . ∑s'□ sat !(six) p(s1,s')= 0.5 (0.001+ 0.999) = 0.5 <= 1 .

ps2 [Xe !(six)] = μs2(e) . ∑s'□ sat !(six) p(s2,s') = 0.5 (0.002+0.998) = 0.5<= 1.

ps3 [Xe !(six)] = μs3(e) . ∑s'□ sat !(six) p(s3,s') = 1 (1) = 1 <= 1.

ps4 [Xe !(six)] = μs4(e) . ∑s'□ sat !(six) p(s4,s') = 1 (1) = 1 <= 1.

ps5 [Xe !(six)] = μs5(e) . ∑s'□ sat !(six) p(s5,s')= 1 (1) = 1 <= 1.

ps6 [Xe !(six)] = μs6(e) . ∑s'□ sat !(six) p(s6,s') = 1 (1) = 1 <= 1.

ps7 [Xe !(six)] = μs7(e) . ∑s'□ sat !(six) p(s7,s') = 0.5 (0.032) = 0.016 <= 1.

ps8[Xe !(six)] = μs8(e) . ∑s'□ sat !(six) p(s8,s') = 0.5 (0.016) = 0.008 <= 1.

ps9[Xe !(six)] = μs9(e) . ∑s'□ sat !(six) p(s9,s') = 0 (0)= 0 <= 1.

Nous venons de voir que la formule $P_{<= 1}$ [X_e !(six)]. est satisfaite dans tous les états du modèle (si, i=1 ... 9). Ainsi la formule (X_e !(six)) voire la propriété (atteignabilité) est vérifiée pour tout le modèle, et par conséquent nous déduisons la atteignabilité de fonctionnement du système.

- **La propriété 2 :** $P_{>=0}$ [X_E (accident)]

s satisfait P>= 0 (Xe accident) ssi ps(Xe accident) >= 0.

On a: ps(XΩ□) = μs(Ω) . ∑s'□ sat (□) p(s,s') , □= accident donc s'= {s9}.

Dans ce qui suit, chaque ligne correspond à la satisfaction de la propriété au sein d'un état (si, i=1-9)

ps1 [Xe (accident)] = μs1(e) . ∑s'□ sat (accident) p(s1,s')= 0.5 (0.001) = 0.0005 >= 0.

ps2 [Xe (accident)] = μs2(e) . ∑s'□ sat (accident) p(s2,s') = 0.5 (0) = 0 >= 0.

ps3 [Xe (accident)] = μs3(e) . ∑s'□ sat (accident) p(s3,s') = 1 (0) = 0 >= 0.

ps4 [Xe (accident)] = μs4(e) . ∑s'□ sat (accident) p(s4,s') = 1 (0) = 0 >= 0.

ps5 [Xe (accident)] = μs5(e) . ∑s'□ sat (accident) p(s5,s')= 1 (0) = 0 >= 0.

ps6 [Xe (accident)] = µs6(e) . ∑s'□ sat (accident) p(s6,s') = 1 (0) = 0 >= 0.

ps7 [Xe (accident)] = µs7(e) . ∑s'□ sat (accident) p(s7,s') = 0.5 (0) = 0 >= 0.

ps8[Xe (accident)] = µs8(e) . ∑s'□ sat (accident) p(s8,s') = 0.5 (0.016) = 0.008 >= 0.

ps9[Xe (accident)] = µs9(e) . ∑s'□ sat (accident) p(s9,s') = 0 (0)= 0 >= 0.

Nous venons de voir que la formule P>= 0 [Xe (accident)] est satisfaite dans tous les états du modèle (si, i=1 ... 9). Ainsi la formule (Xe accident) voire la propriété (sûreté) est vérifiée pour tout le modèle, et par conséquent nous déduisons la sûreté de fonctionnement du système.

- **La propriété 3 :** $P_{<= 0.008}$ [X_E (accident)]

s satisfait P<=0.008 (Xe accident) ssi ps(Xe accident) <= 0.008.

On a: ps(XΩ□) = µs(Ω) . ∑s'□ sat (□) p(s,s') , □= accident donc s'= {s9}

Dans ce qui suit, chaque ligne correspond à la satisfaction de la propriété au sein d'un état (si, i=1-9)

ps1 [Xe (accident)] = µs1(e) . ∑s'□ sat (accident) p(s1,s')= 0.5 (0.001) = 0.0005 <= 0.008 .

ps2 [Xe (accident)] = µs2(e) . ∑s'□ sat (accident) p(s2,s') = 0.5 (0) = 0 <= 0.008.

ps3 [Xe (accident)] = µs3(e) . ∑s'□ sat (accident) p(s3,s') = 1 (0) = 0 <= 0.008.

ps4 [Xe (accident)] = µs4(e) . ∑s'□ sat (accident) p(s4,s') = 1 (0) = 0 <= 0.008.

ps5 [Xe (accident)] = µs5(e) . ∑s'□ sat (accident) p(s5,s')= 1 (0) = 0 <= 0.008.

ps6 [Xe (accident)] = µs6(e) . ∑s'□ sat (accident) p(s6,s') = 1 (0) = 0 <= 0.008.

ps7 [Xe (accident)] = µs7(e) . ∑s'□ sat (accident) p(s7,s') = 0.5 (0) = 0 <= 0.008.

ps8[Xe (accident)] = µs8(e) . ∑s'□ sat (accident) p(s8,s') = 0.5 (0.016) = 0.008 <= 0.008.

ps9[Xe (accident)] = µs9(e) . ∑s'□ sat (accident) p(s9,s') = 0 (0)= 0 <= 0.008

Nous venons de voir que la formule P<= 0.008 [Xe (accident)] est satisfaite dans tous les états du modèle (si, i=1 ... 9). Ainsi la formule (Xe accident) voire la propriété (sûreté) est vérifiée pour tout le modèle, et par conséquent nous déduisons la sûreté de fonctionnement du système.

9.2 Vérification automatique des propriétés par PRISM

Après implémentation et lancement de la vérification sur les mêmes modèles et propriétés, nous obtenons pour chaque propriété les fenêtres montrées par les figures 7.1 ;7.2 ;7.3.

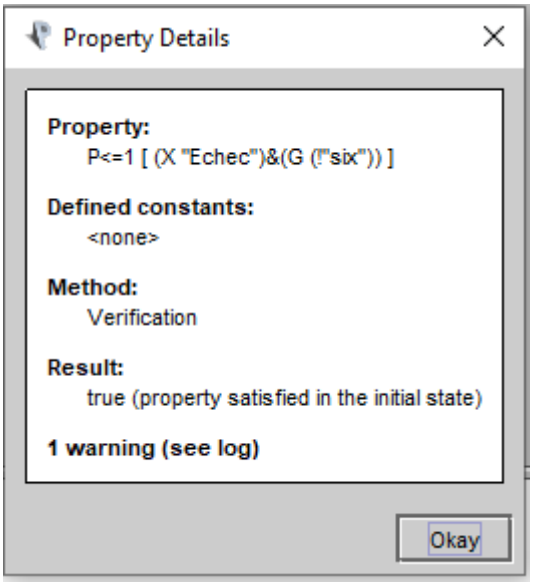

Figure 4.8 :vérification de propriété 01 «P<= 1 [Xe !(six)] ».

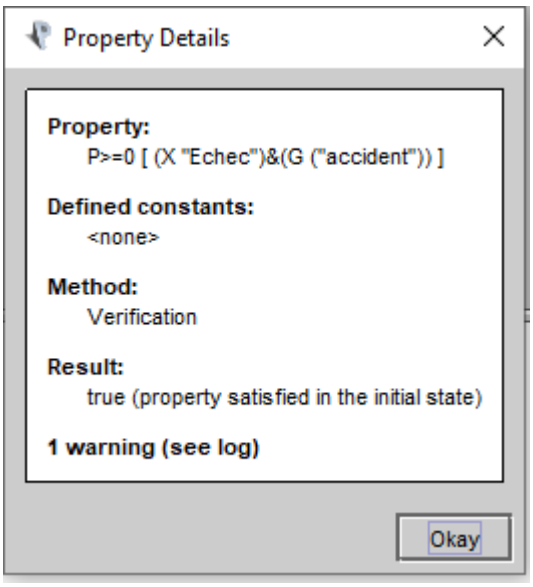

Figure 3.9 : vérification de propriété 02 « P>= 0 (Xe accident) ».

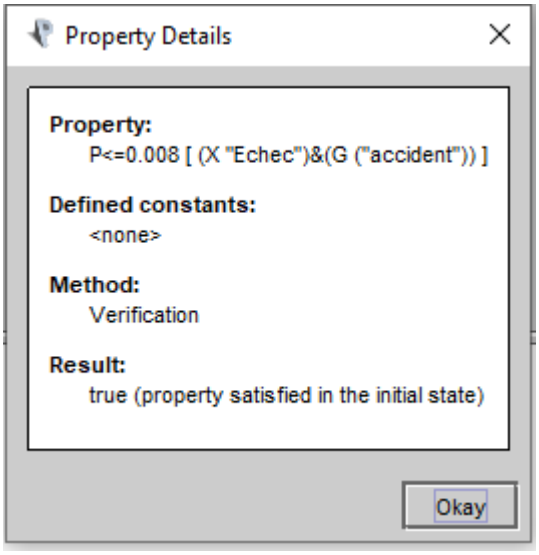

Figure 4.10 : vérification de propriété 03 « P<=0.008 (Xe accident) ».

Solon les figure 4.8 , 4.9, 4.10, on peut dire que les trois propriétés sont vérifiées

Remarque : Sur la dernière ligne dans le figure 4.8 « 1warning (see log)» En regardant le journal ,vous obtenez :Checking for probability <=1.0 - formula trivially satisfies all states. Sur la dernière ligne dans le figure 7.9 « 1warning (see log) » En regardant le journal, vous obtenez : Checking for probability >=0.0 - formula trivially satisfies all states.

10 Résultats et discussions

Nous avons confronté les résultats de vérification des propriétés choisis sur le système embarqué HST, à une double comparaison: la première aux résultats obtenus manuellement et la seconde à ceux obtenus par l'outil PRISM. Comme il est présenté ci-dessus, les résultats de la vérification manuelle coïncident avec ceux de la vérification automatique, ce qui nous a permis d'affirmer que l'implémentation de notre étude de cas est correcte et valide.

11 Conclusion

Dans ce chapitre, nous avons présenté une étude de cas du model-checker PRISM sur un système embarqué réel très important qui est le (*HST : Hubble Space Telescope*) et les résultats obtenus sont encourageants.

Nous avons construit le modèle HMM du HST en basant sur son modèle DTMC présenté dans [28] et en lui ajoutant l'ensemble des observations θ, la matrice B de distribution des probabilités d'observations, le vecteur π de distribution des probabilités de transitions initiales. L'étape de modélisation est la partie la plus difficile lors d'une étude de cas, car les classes de modèles implémentées dans l'outil ne permettent pas toujours d'exprimer le système en entier, mais uniquement une simplification de ce système (ce que l'on appelle une *abstraction*). Après avoir implémenté le modèle et la propriété dans l'outil, théoriquement, il ne reste plus qu'à appuyer sur le bouton de l'outil pour obtenir le résultat voulu, à savoir la réponse à la question « est-ce que le modèle vérifie la spécification ? ». Cependant, les limitations dues aux machines utilisées peuvent faire échouer ce calcul (par exemple à cause d'un manque de mémoire). Il faut alors encore simplifier la modélisation (tout en préservant les caractéristiques importantes du système) et recommencer.

Dans l'étape de spécification des propriétés, nous avons choisi de présenter trois propriétés qui nous ont permis de vérifier le bon fonctionnement du système, puis nous avons spécifié ces propriétés en logique POCTL. Toutes les propriétés choisies sont du type Next suivies d'une observation qui est le cas qui démontre notre approche par rapport aux autres model-checkers, car pour un HMM donné, nous sommes souvent intéressés par les propriétés du processus stochastique fondamental que nous pouvons considérer comme un modèle DTMC. En outre, nous nous sommes intéressés également au raisonnement sur des propriétés de l'autre ensemble de processus stochastiques qui produisent des observations.

Dans la dernière étape de model-checking qui est la vérification des propriétés, nous avons vérifié la satisfiabilité ou non des trois propriétés spécifiées en POCTL dans la troisième section sur le modèle HMM construit dans la deuxième section, manuellement en utilisant l'algorithme de model-checking pour POCTL sur le modèle HMM, et automatiquement en utilisant PRISM pour valider le bon fonctionnement du système HST.

Conclusion générale

La liste des catastrophes connues de systèmes embarqués est très longue. L'étude de cas nous permet de réduire la valeur d'erreur et les catastrophe à travers l'utilisation d'un outil qui supporte le model checking probabiliste (PRISM).

Dans ce travail, nous nous sommes penchés sur un aspect de vérification automatique à base de modèle probabiliste, une technique plus générale et réaliste, qui continue à attirer davantage d'adeptes dans l'univers de la vérification probabiliste.

Ce travail méthodiquement mené, a permis :

- D'introduire pour la première fois le formalisme mathématique « HMM » dans le domaine de la vérification probabiliste car d'une part, les HMMs sont des outils statistiques d'apprentissage simple, riches en propriétés, fondés sur des bases mathématiques solides et ils permettent de modéliser divers systèmes probabilistes à partir d'un ensemble d'observations et dont le nombre d'états du système peut ne pas être connu. Et d'autre part, leur utilisation dans le domaine informatique est très répandue.

- D'enrichir les études de cas de PRISM.

- Intégration pour la première fois dans PRISM le modèle HMM comme un nouveau formalisme de modélisation et de la logique POCTL adaptée de PCTL, comme un nouveau formalisme de spécification. Il n'existe aucun model-checker probabiliste connu utilisant le HMM et le POCTL.

Plusieurs pistes de recherches peuvent être explorées afin d'approfondir le travail présenté dans ce mémoire. Nous avons plus particulièrement identifié quelques points qui nous paraissent intéressants :

- Ajout de nouvelles fonctionnalités à l'outil PRISM.
- Extension de la vérification à d'autres propriétés.
- Réalisation d'autres études de cas sur d'autres systèmes embarqués réels.
- Aller à une plateforme de vérification des systèmes probabilistes.
- Le développement de langages de description de systèmes, car la véritable difficulté dans le processus de model-checking réside dans la tâche de modélisation.

Références bibliographiques

[1] Les systèmes embarqués Introduction (Richard Grisel – Professeur des Universités – Université de RouenNacer Abouchi – Professeur – ESCPE Lyon)

[2] Christel B., Joost-Pieter K., & al., 2008. Principles of model checking, volume 26202649. MIT press Cambridge.

[3] Parker D., 2002. Implementation of symbolic model checking for probabilistic systems, these de doctorat, University of Birmingham.

[4] Rabiner L.R., 1989. A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition, Proceedings of the IEEE, Vol. 77(2), 257-286.

[5] Jurafsky D. & Martin J.H., 2000. Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition. Prentice Hall.

[6] Vlontzos J. A. & Kung S. Y., October 1992. Hidden Markov models for character recognition. IEEE Transactions on Image Processing, Vol. (1), 539–543.

[7] Birney E., 2001. Hidden Markov models in biological sequence analysis. IBM Journal of Research and Development, Vol. 45(3), 449–454.

[8] Pevzner P.A., 2000. Computational Molecular Biology: An Algorithmic Approach. The MIT Press.

[9] Salamatian K. & Vaton S., 2001. Hidden markov modeling for network communication channels. In SIGMETRICS, ACM Press, 92–101.

[10] Francois J.M. & Leduc G., 2005. Mobility prediction's influence on QoS in wireless networks: A study on a call admission algorithm. In 3rd International Symposium on Modeling and Optimization in Mobile, Ad-Hoc and Wireless Networks, IEEE Computer Society, 238–247.

[11] Zhang L., Hermanns H. & Jansen D. N., 2005. Logic and Model Checking for Hidden Markov Models. Proceeding of FORTE, LNCS 3731, 98 -112.

[12] Hansson H. & Jonsson B., 1994. A logic for reasoning about time and probability, Formal Aspects of Computing, Vol. 6(5), 512-535.

[13] Christel B., Boudewijn H., Holger H. & Joost-Pieter K., 2003. Model-checking algorithms for continuous-time Markov chains. IEEE Transactions on Software Engineering, Vol. 29(6), 524–541.

[14] Andova S., Holger H. & Joost-Pieter K., 2003. Discrete-time rewards model-checked. In Kim G. Larsen and Peter Niebert, editors, Formal Modeling and Analysis of Timed Systems: First International Workshop (FORMATS'03), Vol. 2791 of LNCS, Berlin, Springer, 88–104.

[15] Baier C., Boudewijn R. H., Holger H. & Joost-Pieter K., 2000. On the logical characterisation of performability properties. In Ugo Montanari, Jos´e D. P. Rolim, and Emo Welzl, editors, Proc. 27th International Colloquium on Automata, Languages and Programming (ICALP '00), Vol. 1853 of LNCS, Berlin, Springer, 780–792.

[16] Boudewijn H., Lucia C., Holger H., Joost-Pieter K. & Christel B., 2002. Model-checking performability properties. In International Conference on Dependable Systems and Networks (DSN'02), Washington D. C., IEEE CS Press, 103–112.

[17] Gul A. A., José M. & Koushik S., 2005. Pmaude: Rewrite-based specification language for probabilistic object systems. Electronic Notes in Theoretical Computer Science, Vol. 153(2), 213–239.

[18] Holger H., Joost-Pieter K., Joachim M. K. & Markus S., 2000. Towards model checking stochastic process algebra. In Wolfgang Grieskamp, Thomas Santen, and Bill Stoddart, editors, 2nd International Conference on Integrated Formal Methods (IFM'00), Vol. 1945 of LNCS, Berlin, Springer, 420–439.

[19] Muhimpundu J., 2014. Analyse de l'erreur dans la vérification probabiliste, Maîtrise en informatique, Université Laval, Québec, Canada. 106p.

[20] Nicola R.D. & Vaandrager F.W., 1990. Action versus state based logics for transition systems. Semantics of Systems of Concurrent Processes, /LNCS 469, 407-419.

[21] Clarke E., E. Emerson, & Sistla A., 1986. Automatic verification of finite-state concurrent systems using temporal logics. ACM Transactions on Programming Languages and Systems, Vol. 8(2), 244–263.

[22] Vardi M. Y., 2006. From Church and Prior to PSL. In 25MC.

[23] Courcoubetis C. & Yannakakis M., 1995. The Complexity of Probabilistic Verification. Journal of the ACM, VOL. 42(4), 857–907.

[24] Hansson H. & Jonsson B., 1994. A logic for reasoning about time and reliability. Formal Aspects of Computing, Vol. 6(5), 512–535.

[25] Bianco A. & De Alfaro L., 1995. Model checking of probabilistic and nondeterministic systems. In P. Thiagarajan, editor, Proc. 15th Conference on Foundations of Software Technology and Theoretical Computer Science, vol. 1026 of LNCS, Springer, 499–513.

[26] Younes H. L. S. & Simmons R. G., 2002. Probabilistic verification of discrete event systems using acceptance sampling. In Ed Brinksma and Kim Guldstrand Larsen, editors, Proc. 14th International Conference on Computer Aided Verification (CAV'02), Vol. 2404 of LNCS, Berlin, Springer, 223–235.

[27] Hart S., Sharir M. & Pnueli A., 1983. Termination of probabilistic concurrent programs. ACM Transactions on Programming Languages and Systems, Vol. 5(3), 356–380.

[28] Pnueli A., 1983. On the extremely fair treatment of probabilistic algorithms. In Proc. 15th Annual ACM Symposium on Theory of Computing, 278– 290.

[29] Vardi M. Y., 1985. Automatic verification of probabilistic concurrent finite state programs. In Proc. 26th Annual Symposium on Foundations of Computer Science (FOCS'85), IEEE Computer Society Press, 327–338.

[30] Pnueli A. & Zuck L., 1986. Verification of multiprocess probabilistic protocols. Distributed Computing, Vol. 1(1), 53–72.

[31] Aziz A., Sanwal K., Singhal V. & Brayton R., 1996. Verifying continuous time Markov chains. In R. Alur and T. Henzinger, editors, Proc. 8th International Conference on Computer Aided Verification (CAV'96), vol. 1102 of LNCS, Springer, 269–276.

[32] Baier C., Katoen J.-P. & Hermanns H., 1999. Approximate symbolic model checking of continuous-time Markov chains. In J. Baeten and S. Mauw, editors, Proc. 10th International Conference on Concurrency Theory (CONCUR'99), vol. 1664 of LNCS, Springer, 146–161.

[33] Aziz A., Singhal V., Balarin F., Brayton R. & Sangiovanni-Vincentelli A., 1995. It usually works: The temporal logic of stochastic systems. In P. Wolper, editor, Proc. 7th International Conference on Computer Aided Verification (CAV'95), vol. 939 of LNCS,. Springer, 155–165.

[34] Courcoubetis C. & Yannakakis M., 1990. Markov decision processes and regular events. In M. Paterson, editor, Proc. 17th International Colloquium on Automata, Languages and Programming (ICALP'90), Vol. 443 of LNCS, Springer, 336–349.

[35] Baier C. & Kwiatkowska M. Z., 1998. Model checking for a probabilistic branching time logic with fairness. Distributed Computing, Vol.11(3), 125–155.

[36] Baier C., 1998. On algorithmic verification methods for probabilistic systems. habilitation thesis. University of Mannheim.

[37] Baier C., Boudewijn R. H., Holger H. & Joost-Pieter K., 2000. Model checking continuous-time Markov chains by transient analysis. Proc. 12th International Conference on Computer Aided Verification (CAV'00), In E. Allen Emerson and A. Prasad Sistla, editors, Vol. 1855 of LNCS, Berlin, Springer, 358–372.

[38] Joost-Pieter K., Kwiatkowska M. Z., Gethin N. & David P., 2001. Faster and symbolic CTMC model checking. In Luca de Alfaro and Stephen Gilmore, editors, Proc. 1st Joint International Workshop on Process Algebra and Probabilistic Methods, Performance Modeling and Verification (PAPM/PROBMIV'01), Vol. 2165 of LNCS, Berlin, Springer, 23–38.

[39] Aziz A., Singhal V., Balarin F., Brayton R. & Sangiovanni-Vincentelli A., 1995. It usually works: The temporal logic of stochastic systems. In P. Wolper, editor, Proc. 7th International Conference on Computer Aided Verification (CAV'95), vol. 939 of LNCS,. Springer, 155–165.

[40] Courcoubetis C. & Yannakakis M., 1990. Markov decision processes and regular events. In M. Paterson, editor, Proc. 17th International Colloquium on Automata, Languages and Programming (ICALP'90), Vol. 443 of LNCS, Springer, 336–349.

[41] Baier C. & Kwiatkowska M. Z., 1998. Model checking for a probabilistic branching time logic with fairness. Distributed Computing, Vol.11(3), 125–155.

[42] Baier C., 1998. On algorithmic verification methods for probabilistic systems. habilitation thesis. University of Mannheim.

[43] Baier C., Boudewijn R. H., Holger H. & Joost-Pieter K., 2000. Model checking continuous-time Markov chains by transient analysis. Proc. 12th International Conference on Computer Aided Verification (CAV'00), In E. Allen Emerson and A. Prasad Sistla, editors, Vol. 1855 of LNCS, Berlin, Springer, 358–372.

[44] Joost-Pieter K., Kwiatkowska M. Z., Gethin N. & David P., 2001. Faster and symbolic CTMC model checking. In Luca de Alfaro and Stephen Gilmore, editors, Proc. 1st Joint International Workshop on Process Algebra and Probabilistic Methods, Performance Modeling and Verification (PAPM/PROBMIV'01), Vol. 2165 of LNCS, Berlin, Springer, 23–38.

[39] Ferroum A. & Boudour R., 2016. A tool for automatic verification of Embedded systems.3rd International Conference on Embedded Systems in Telecommunications and Instrumentation (ICESTI'16), Annaba, Algeria.

[40] Tatarewicz J.N., 2001. The Hubble Space Telescope Servicing Mission. In: From engineering science to big science.

(Eds) J. Henry – 365-396.

[41] Oldenkamp H.A., 2007. Probabilistic model checking: a comparison of tools. Master's thesis, University of Twente, Netherlands. 108p.

[42] Segala R., 1995. Modelling and Verification of Randomized Distributed Real Time Systems, Ph.D. thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, USA. 283p.

Résumé :

Dans ce mémoire, nous avons défini un système embarqué, ses types, domaines d'utilisation, facilité et services qu'il fournit dans la vie quotidienne, puis nous avons expliqué un ensemble de types de modèles de probabilités aléatoires à travers lesquels nous pouvons modéliser le comportement de ce système, puis nous avons appris à connaître l'outil PRISM avec son langage et son mode d'utilisation, ce qui nous permet de vérifier, et en analysant, simulant le comportement de ces différents modèles, nous avons donc abordé l'étude du système embarqué, qui est le télescope Hubble, en utilisant l'un des modèles aléatoires de probabilité, le modèle de Markov caché à l'aide de l'outil PRISM.

Abstarct:

In this memory, we have defined an embedded system, its types, areas of use, ease and services that it provides in daily life, then we have explained a set of types of random probability models through which we can model the behavior of this system, then we got to know the PRISM tool with its language and its mode of use, which allows us to verify, and by analyzing, simulating the behavior of these different models, we therefore approached the study of the on-board system, which is the Hubble Telescope, using one of the random probability models, the Markov model hidden using the PRISM tool.

ملخص

في هذه المذكرة قمنا بالتعريف بنظام اومباركي أنواعه، مجالات استخدامه، التسهيلات والخدمات التي يقدمها في الحياة اليومية ثم قمنا بشرح مجموعة من أنواع النماذج الإحتمالية العشوائية التي يمكننا من خلالها نمذجة سلوك هذا النظام ثم تعرفنا على الأداة بريسم لغته و طريقة استعماله والذي يمكننا من التحقق و تحليل و القيام بمحاكاة سلوك مختلف هذه النماذج فتطرقنا لدراسة نظام اومباركي وهو تلسكوب هوبل باستخدام أحد النماذج العشوائية الإحتمالية نموذج ماركوف المخفي باستعمال الأداة بريسم