Titre: Title:	Vérification des systèmes de sécurité probabilistes : restriction de l'attaquant
Auteur: Author:	Alain Freddy Kiraga
Date:	2010
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Kiraga, A. F. (2010). Vérification des systèmes de sécurité probabilistes : restriction de l'attaquant [Master's thesis, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/349/

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/349/
Directeurs de recherche: Advisors:	John Mullins
Programme: Program:	Génie informatique

UNIVERSITÉ DE MONTRÉAL

VÉRIFICATION DES SYSTÈMES DE SÉCURITÉ PROBABILISTES : RESTRICTION DE L'ATTAQUANT

ALAIN FREDDY KIRAGA
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
JUILLET 2010

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

VÉRIFICATION DES SYSTÈMES DE SÉCURITÉ PROBABILISTES : RESTRICTION DE L'ATTAQUANT

présenté par : KIRAGA, Alain Freddy
en vue de l'obtention du diplôme de : Maîtrise ès Sciences Appliquées
a été dûment accepté par le jury d'examen constitué de :

M. GALINIER, Philippe, Doct., président.
M. MULLINS, John, Ph.D., membre et directeur de recherche.
M. TAHAR, Sofiène, Ph.D., membre.

À mon père et ma mère

REMERCIEMENTS

Je tiens premièrement à exprimer mes plus sincères remerciements et ma gratitude envers mon directeur de recherche M. John MULLINS pour m'avoir accueilli et initié à la recherche dans son laboratoire CRAC. Son aide, son encadrement, son soutien continu et ses précieux conseils m'ont été d'une utilité inestimable tout au long de mon projet de maîtrise. Mes remerciements s'adressent également au président du jury M. Philippe GALINIER ainsi qu'au membre du jury M. Sofiène TAHAR pour avoir accepté d'évaluer ce mémoire de Maîtrise.

Je tiens aussi à remercier Dave PARKER de l'équipe de développement du model checker PRISM pour m'avoir aidé et guidé dans la compréhension du code source de PRISM. Ses conseils ont été incommensurables pour réussir à y intégrer notre application. J'aimerais également remercier Mark KATTENBELT du "Computing Laboratory, University of Oxford" pour ses conseils qui m'ont permis de venir à bout de ces terribles problèmes de fuite de mémoire liés au package CUDD.

Grand Merci aussi à mes parents pour leur amour et surtout pour m'avoir toujours encouragé et soutenu dans mes études. Enfin, je tiens à adresser mes remerciements à tous ceux dont je n'ai pas cité les noms et qui m'ont assisté d'une manière ou d'une autre durant la réalisation de ce projet.

RÉSUMÉ

L'analyse des protocoles de sécurité présentant un comportement probabiliste et non déterministe nécessite, afin d'évaluer les propriétés probabilistes, l'introduction d'un objet appelé *ordonnanceur* résolvant tout le non déterminisme existant dans le modèle. La sécurité des protocoles étant évaluée dans un environnement hostile, il est tout naturel que l'ordonnanceur soit considéré sous le contrôle de l'adversaire ayant le contrôle complet de tout le réseau de communications. Cette considération confère généralement une puissance trop forte à l'adversaire du fait que certains choix résultent des actions internes devant rester invisibles à l'adversaire. Nous proposons une méthode de restriction des ordonnanceurs basée sur la relation d'équivalence observationnelle définie sur l'ensemble des actions du protocole modélisé sous-forme de processus de décision markovien (MDP) en définissant deux niveaux d'ordonnancement. L'ordonnanceur interne ou coopératif résout à l'avance le non déterminisme entre les actions observationnellement équivalentes avec une distribution de probabilités uniforme et le reste du non déterminisme dans le modèle est résolu par les ordonnanceurs externes modélisant ceux considérés sous le contrôle de l'attaquant lors de l'analyse du protocole. Nous implémentons ensuite cette méthode dans le *model checker* PRISM en utilisant des algorithmes basés sur les structures de données symboliques tout en préservant les fonctionnalités initiales de PRISM. Enfin, nous illustrons notre méthode par des études de cas de protocoles de sécurité, le *protocole de dîner de cryptographes* de David Chaum et le *protocole de signature de contrat* de Michael Rabin.

Mots-clés : Protocole de sécurité, système probabiliste, non déterminisme, ordonnanceur, processus de décision markovien, model checking probabiliste, diagramme de décision.

ABSTRACT

Analysis of security protocols which exhibit both probabilistic and non-deterministic behavior require an object called *scheduler* to solve all non-determinism in the model for evaluating probabilistic properties. In the context of analysis of security protocols running into a hostile environment, it's quite natural to consider the scheduler to be under control of the adversary having full control of all the communications network. This assumption gives the adversary an unreasonably strong power of the fact that certain choices result from internal actions which must remain invisible to the attacker. We propose a method for limiting the power of the schedulers based on an observational equivalence relation defined on the actions set of the protocol modeled as Markov decision process (MDP) by defining two levels of scheduling. A (static) internal or cooperative scheduler resolves uniformly the non-determinism over observationally equivalent actions and the remaining non-determinism in the model is resolved by external or adversarial schedulers which model partial capabilities of the adversary during the security analysis of the protocol. We then implement our method in PRISM *model checker* by using algorithms based on symbolic data structures and by preserving basic functionalities of PRISM. Finally we illustrate our idea by applying the method on David Chaum's *dining cryptographers protocol* and Michael Rabin's *probabilistic contract signing protocol*.

Keywords: Security protocol, probabilistic system, non-determinism, scheduler, Markov decision process, probabilistic model checking, decision diagram.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES SIGLES ET ABRÉVIATIONS

BDD	Binary Decision Diagram
CCS	Calculus of Communicating Systems
CSL	Continuous Stochastic Logic
CSP	Communicating Sequential Processes
CTL	Computation Tree Logic
CTMC	Continuous-Time Markov Chain
CUDD	Colorado University Decision Diagram
DCP	Dining Cryptographers Protocol
DTMC	Discrete-Time Markov Chain
IOA	Input/Output Automata
JNI	Java Native Interface
LTS	Labeled Transition System
MDP	Markov Decision Process
MTBDD	Multi-Terminal Binary Decision Diagram
NSA	National Security Agency
PCTL	Probabilistic Computation Tree Logic
PEPA	Performance Evaluation Process Algebra
PIOA	Probabilistic Input/Output Automata
PPC	Probabilistic Polynomial-time process Calculus
Task-PIOA	Task-structured Probabilistic Input/Output Automata

CHAPITRE 1

INTRODUCTION

La sécurité de l'information est devenue primordiale avec l'essor de plus en plus grandissant des technologies de l'information qui accroît la dépendance de notre société à des échanges d'informations par ordinateurs. Les informations sensibles telles que les numéros de cartes de crédit, les informations médicales, les informations de vote, etc. transitent tous les jours sur l'Internet qui s'est imposé au fil des années comme un outil incontournable de communication et d'échange. Malheureusement, malgré que l'Internet soit un outil pratique, il présente des faiblesses de taille en matière de sécurité. Les échanges sur Internet s'effectuant à travers des canaux non sécurisés, les personnes ou organisations malveillantes peuvent profiter de ces failles pour intercepter les numéros de cartes de crédit, voler les informations privées, modifier les votes, etc. Ces informations doivent donc être protégées adéquatement.

L'efficacité d'un système offrant des mécanismes de protection est relative aux réponses que le concepteur donne aux questions suivantes : "*Quel(s) objectif(s) le système est-il supposé atteindre en terme(s) de sécurité ?*", "*le système assure la sécurité contre qui ou contre quoi ?*". La première question nécessite de spécifier les propriétés que le système doit satisfaire et la seconde exige de définir un modèle permettant de spécifier l'interaction de tous les composants du système et de l'environnement ainsi que les menaces considérées. La définition d'un modèle muni d'une sémantique bien définie nous permet de vérifier formellement les propriétés de sécurité vis-à-vis du système et son contexte.

1.1 Propriétés de sécurité et protocoles de sécurité

Dans cette section, nous présentons les principales propriétés de sécurité qui sont généralement désirables lors des communications à travers les réseaux puis nous allons essayer de donner une définition informelle d'un protocole de sécurité. Les propriétés de sécurité peuvent se regrouper en six grandes catégories :

- **Confidentialité** : C'est la mieux connue et la plus désirée des propriétés de sécurité. Elle assure qu'une personne non autorisée ne puisse prendre connaissance d'une information sensible devant rester secrète. Nous pouvons citer comme exemples les numéros de cartes de crédit ou les informations médicales qui doivent toujours rester secrets durant des transactions électroniques.

- **Authenticité** : Cette propriété assure que l'origine d'une information est identifiée. Par exemple lors du téléchargement d'une application sur Internet, il faut toujours s'assurer que l'application provient d'une source de confiance et qu'elle n'a pas été remplacée par une application malveillante qui usurpe son identité. Dans certains cas l'authenticité doit être dépendante du temps, c.-à-d. que chaque message doit avoir une durée de vie pour éviter que des messages périmés puissent être réenvoyés.

- **Intégrité** : Cette propriété assure que les données contenues dans un message sont complètes et exactes après que ce message ait transité d'un expéditeur vers un récépteur. L'intégrité doit alors être maintenue afin de prévenir toute falsification du message par des intrus.

- **Non-répudiation** : Elle assure qu'une personne ne peut nier être l'auteur d'un message qu'elle a envoyé ou le récipiendaire d'un message qu'elle a reçu durant un échange. Considérons l'exemple d'un achat d'une application logicielle sur Internet. Cette propriété assure que l'acheteur ne peut frauduleusement nier avoir reçu la bonne application.

- **Anonymat** : L'anonymat est une propriété permettant de garantir qu'un récepteur d'un message ou n'importe quel observateur ne puisse connaître l'identité de l'auteur du message. Cette propriété doit être garantie par exemple par les systèmes de vote électronique afin d'éviter une quelconque inférence possible entre un électeur et son vote.

- **Équité** : Cette propriété est indispensable pour une transaction entre personnes ou entités ne se faisant pas confiance afin de prévenir toute situation d'arnaque. Elle assure par exemple que lors d'une signature de contrat électronique, il ne puisse pas y avoir une telle situation où une fois qu'un des participants reçoit un contrat signé par l'autre partenaire, il refuse d'envoyer le contrat portant sa signature.

Afin d'assurer ces différentes propriétés de sécurité, des mécanismes associés à des outils *cryptographiques* tels que les fonctions de chiffrement/déchiffrement, les fonctions de hachage cryptographiques ou les signatures digitales doivent être mis en oeuvre. Nous assumons que le lecteur connaît quelques notions de la cryptographie. Toutefois, le lecteur intéressé peut consulter les détails sur les techniques cryptographiques dans Stallings (2002).

Nous définissons *les protocoles de sécurité* ou *protocoles cryptographiques* comme étant des petites applications distribuées utilisant des primitives cryptographiques afin d'assurer les propriétés de sécurité dans n'importe quel environnement hostile. La fiabilité de tels systèmes

dépend de l'efficacité des techniques d'analyse et de validation utilisées. Nous nous intéressons particulièrement aux méthodes formelles sur lesquelles repose notre mémoire.

1.2 Méthodes formelles et adversaire à la Dolev-Yao

Les systèmes exigeant une haute fiabilité comme les systèmes de navigation des avions ou engins spatiaux, les protocoles de sécurité assurant les transactions bancaires, etc. sont difficiles à évaluer en utilisant des techniques classiques de test ou de simulation. Les techniques de vérification formelle sont utilisées afin de garantir un haut degré de fiabilité de tels systèmes. Ces techniques basées sur les mathématiques assurent une vérification rigoureuse et comportent trois étapes essentielles :

- **Modélisation du système** : La modélisation est incontestablement la première étape dans la vérification formelle. Elle consiste à répondre à la question suivante :"*Comment représenter formellement le système à analyser* ?". La réponse à cette question consiste à convertir un système réel en un modèle mathématique en utilisant des objets suffisamment expressifs décrivant de façon abstraite tous les aspects du système. Plusieurs familles de modèles formels existent dont les modèles *opérationnels* parmi lesquels les automates (LTS (*Labeled Transition System*) de Keller (1976), IOA (*Input/Output Automata*) de Lynch et Tuttle (1989), PIOA (*Probabilistic Input/Output Automata*) de Segala (1995), etc.) qui modélisent l'évolution du système en termes d'*états* et de *transitions*.

- **Spécification des propriétés** : La spécification des propriétés permet de répondre à la question : "*Comment exposer les propriétés que le système doit satisfaire* ?". Répondre à cette question revient à définir un formalisme permettant de décrire les propriétés des systèmes. Nous pouvons citer les logiques temporelles qui sont les plus populaires dans la spécification des propriétés des systèmes mais aussi les équivalences de comportement des systèmes (simulation Milner (1989), bisimulation Park (1981), équivalence de traces Hoare (1985), etc.).

- **Validation** : La validation prouve si oui ou non une spécification de propriété est vérifiée sur un système représenté par un modèle. Cela revient à répondre à la question suivante : "*Étant donnée une représentation d'un système sous forme d'un modèle, comment pouvons-nous vérifier les différentes propriétés que doit satisfaire le système* ?". Les techniques de vérification peuvent être manuelles ou complètement automatisées. Parmi les méthodes automatisables, nous pouvons citer le *model checking* proposé initialement par Clarke et Emerson (1984) et qui vérifie la validité de la propriété exprimée

en logique temporelle dans tous les états du système. Des outils automatisés tels que SPIN de Holzmann (1997), Murφ de Mitchell *et al.* (1997) ou PRISM de Kwiatkowska *et al.* (2002) utilisent cette approche de vérification.

Les méthodes formelles se sont progressivement implantées dans les processus de développement des applications industrielles. Les mémoires des ordinateurs n'étant pas illimitées, des techniques de représentation symbolique et compacte utilisant les *diagrammes de décision* en l'occurrence des BDDs (*Binary Decision Diagrams*) Bryant (1986) et MTBDDs (*Multi-Terminal Binary Decision Diagrams*) Fujita *et al.* (1997) permettent de faire face au problème d'explosion combinatoire d'états lors de l'analyse de grands systèmes. L'application des méthodes formelles dans la vérification des systèmes de sécurité est apparue depuis les années 80. Toutefois, le contexte de leur vérification est quelque peu différent des systèmes classiques car il doit tenir compte de l'environnement hostile dans lequel évolue le système. Autrement dit, la propriété de sécurité doit rester valide sur le système évoluant dans n'importe quel environnement.

L'environnement hostile suppose une entité malicieuse qui cherche à casser la sécurité du système. Nous considérons le modèle de l'adversaire décrit par Dolev et Yao (1983) et qui est assumé, avec quelques fois des extensions, par l'ensemble de la communauté utilisant les méthodes formelles dans la vérification des protocoles de sécurité. Le modèle à la Dolev-Yao suppose un adversaire ayant le contrôle complet de tout le réseau de communications, pouvant intercepter tous les messages, introduire de nouveaux messages à partir de sa base de connaissance, changer l'ordre des messages, etc.. L'adversaire est considéré aussi comme pouvant être n'importe quel participant parmi ceux impliqués dans l'échange à travers le protocole. L'approche formelle dans les protocoles de sécurité consiste donc à abstraire les primitives cryptographiques en un système formel de règles et à faire l'hypothèse de la cryptographie parfaite en rejettant toute forme de *cryptanalyse* (c.-à-d. il est impossible de déchiffrer un message chiffré ou avoir connaissance des informations sur le message en clair à partir du message chiffré sans connaître la clé de déchiffrement). Cette abstraction quoique simplificatrice ouvre toujours la voie à l'automatisation de l'analyse. Les autres techniques que nous pouvons citer dans l'approche formelle sont les logiques de croyances et le *theorem proving* Paulson (1989). Les logiques de croyances tels que la logique BAN Burrows *et al.* (1990) permettent de déduire les croyances des principaux participants au protocole concernant l'origine et l'usage de l'information. La logique BAN est utilisée pour déterminer lesquelles de ces croyances peuvent être déduites à partir d'une idéalisation du protocole. Ce modèle ne dicte pas quelles croyances un protocole doit vérifier, c'est plutot à celui qui

analyse le protocole de décider lesquelles doivent être satisfaites et de déterminer si le protocole permet de le garantir. Le *theorem proving* qui est une technique semi-automatique de déduction de preuves de théorèmes, peut également être utilisé pour vérifier les propriétés de protocoles de sécurité. La technique repose sur les concepts de traces qui sont des événements qui surviennent sur le réseau lors de l'exécution du protocole Giampaolo (2000). Le système et les propriétés de sécurité sont exprimés en logique mathématique donné par un système formel basé sur un ensemble d'axiomes et de règles d'inférence. Les preuves de propriétés sont effectuées par induction sur une trace générique du modèle représentant la propriété du protocole. Des outils interactifs tels que Isabelle Paulson (1994) et HOL Gordon et Melham (1993) supportent la modélisation inductive des protocoles en utilisant la logique d'ordre supérieur. Différentes méthodes de *theorem proving* sont discutés en détail dans Gawanmeh (2008).

Une autre approche utilisée dans l'analyse des protocoles de sécurité est l'approche *computationnelle* Goldwasser *et al.* (1988) qui traite les primitives cryptographiques comme des algorithmes opérant sur les chaînes de bits et l'adversaire comme une *machine de Turing* opérant en temps polynomial. Cette approche traite correctement et efficacement les primitives cryptographiques mais malheureusement les preuves de sécurité sont souvent manuelles et très difficiles à vérifier.

1.3 Probabilité vs. non déterminisme

Certains protocoles de sécurité utilisent des mécanismes aléatoires pour atteindre leurs objectifs de sécurité. De tels protocoles sont appelés des *protocoles de sécurité probabilistes*. La validation des protocoles de sécurité probabilistes exige alors un formalisme permettant de représenter l'aspect *probabiliste* de ces systèmes. Ce formalisme doit également être capable de modéliser le *non déterminisme*. La notion de non déterminisme est essentielle pour modéliser les phénomènes tels que :

- Les actions que l'environnement peut exercer sur le système ;
- La liberté d'implémentation du système ;
- La concurrence entre processus (entités) du système où le choix d'un processus effectuant une action est libre.

Pour éclaircir la différence entre les notions de probabilité et de non déterminisme, considérons l'exemple de jet d'une pièce de monnaie de la Figure 1.1. Cet exemple montre les probabilités

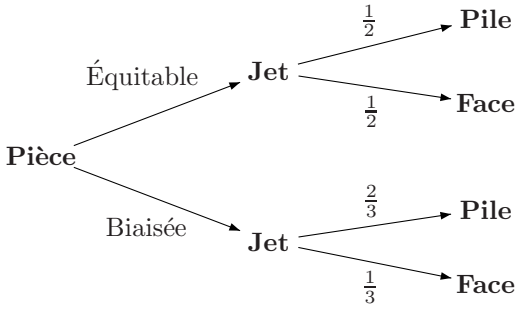


Figure 1.1 Jet d'une pièce de monnaie.

de voir apparaître les événements *Pile* ou *Face* selon que la pièce jetée est *équitable* ou *biaisée*. La probabilité de choisir une quelconque pièce (équitable ou biaisée) n'étant pas connue, cette situation est modélisée par le non déterminisme. Sans éliminer ce non déterminisme, il est impossible de donner par exemple la probabilité exacte associée à l'événement *Pile* mais nous pouvons affirmer que la probabilité de voir apparaître *Pile* est située entre $\frac{1}{2}$ et $\frac{2}{3}$ (c.-à-d. dans l'intervalle $[\frac{1}{2},\frac{2}{3}]$). Afin de pouvoir évaluer les propriétés probabilistes sur de tels systèmes, il faut d'abord éliminer tout le non déterminisme présent dans le modèle grâce à un objet appelé *ordonnanceur*.

1.4 Approches de vérification des systèmes probabilistes

Plusieurs techniques sont proposées dans la littérature pour la vérification de systèmes probabilistes. Des algorithmes de *model checking* probabilistes ont été proposés dans Baier et Kwiatkowska (1998); Bianco et de Alfaro (1995) et étendent l'algorithme de Clarke et Emerson (1984) en y intégrant les mécanismes aléatoires. La technique prend un modèle exprimé sous-forme d'un système de transitions et d'une formule exprimée en logique temporelle probabiliste puis retourne le résultat de vérification sous forme d'un ensemble d'états satisfaisant la formule. Différents *model checkers* probabilistes ont été proposés (p.ex. PRISM Kwiatkowska *et al.* (2002), ETMCC (*Erlangen-Twente Markov Chain Checker*) Hermanns *et al.* (2000), MRMC (*Markov Reward Model Checker*) Katoen *et al.* (2005), YMER Etessami *et al.* (2005), VESTA Sen *et al.* (2005)). Une étude comparative des ces différents outils est exposée dans Oldenkamp (2007). Parmi tous ces outils, notre choix a porté sur PRISM, le seul supportant la notion de non déterminisme qui est au centre de la problématique de notre projet.

La technique de *theorm proving* Paulson (1989) est également utilisée pour l'analyse des

systèmes probabilistes avec l'introduction de la théorie des probabilités dans la logique Bialas (1990). Elle nécessite la formalisation des variables aléatoires utilisées pour décrire le système. Parmi les logiques supportées par le *theorem proving*, seule la logique d'ordre supérieure permet de formaliser l'analyse des systèmes probabilistes. Cette technique est basée sur la modélisation du système à analyser et des propriétés à vérifier en termes de logique d'ordre supérieure. L'étape suivante consiste à vérifier les théorèmes correspondant aux propriétés probabilistes du modèle à l'aide d'un *theorem prover*. L'expressivité en logique d'ordre supérieure permet de venir à bout de la limitation liée aux systèmes très larges comme l'explosion combinatoire dans le cadre du *model checking* probabiliste. Plusieurs approches de *theorem proving* sur les systàmes probabilistes ont été proposées (p.ex. Hurd *et al.* (2005), Audebaud et Paulin-Mohring (2006), etc.) et une étude comparative des différentes approches est exposée dans Hasan (2008).

Différentes autres technniques sont utilisées pour la vérification des systèmes probabilistes. La méthode de vérification basée sur les équivalences de test a été formalisée par De Nicola et Hennessy (1983). L'idée derrière cette méthode est de définir le comportement du système en terme de capacité à passer des tests. Cela nécessite de définir la notion de *test* et la manière dont le test intéragit avec le système étudié. Le test est un processus spécial pouvant effectuer une action spéciale représentant le succès. Si les intéractions du système avec le test se terminent dans un état de succès, le système se comporte de façon désirée et dans le cas contraire le système possède des défauts de spécification à corriger. Une extension probabiliste de cette méthode a été proposée par Christoff (1990) en définissant trois équivalences de test, chacune basée sur la probabilité des traces issues de l'intéraction entre le système et le test. Le comportement du système testé est ainsi évalué en terme de probabilités que le système réussit à passer des tests. Nous pouvons citer également d'autres méthodes de vérification telles que la simulation probabilistes Segala et Turrini (2007), la bisimulation probabiliste Larsen et Skou (1989) et l'analyse de traces Segala (1995).

1.5 Problématique et objectifs du mémoire

D'une part, l'évaluation des propriétés probabilistes des systèmes nécessite la résolution (élimination) de tout le non déterminisme existant dans les modèles représentant ces systèmes. D'autre part, l'évaluation des propriétés de sécurité des protocoles suppose un environnement hostile où l'adversaire contrôle tout le réseau de communications. Il est ainsi tout à fait normal que l'ordonnanceur soit considéré sous le contrôle de l'adversaire lors de la vérification formelle des protocoles de sécurité. Or dans certains cas, le non déterminisme pouvant résulter des actions internes (invisibles à l'environnemment externe) des agents participant dans

le protocole, l'attribution du contrôle du tel non déterminisme à l'adversaire lui confère une puissance trop forte impossible à atteindre dans les conditions réelles d'utilisation du protocole. Cette considération a pour conséquences de fausser la vérification de certains protocoles de sécurité durant la phase de validation.

L'objectif principal de notre projet consiste à réaliser dans le cadre d'un outil de *model checking* tout usage un modèle d'attaquant calibré pour les protocoles de sécurité probabilistes afin de l'utiliser pour la vérification de leurs propriétés de sécurité. Notre objectif principal se subdivise en trois objectifs spécifiques :

1. Adapter dans le cadre des automates probabilistes, le modèle de restriction des ordonnanceurs défini par Hamadou et Mullins (2010) sur les algèbres de processus probabilistes étendus avec des primitives cryptographiques.
2. Implanter le modèle de restriction de l'adversaire proposé dans l'outil PRISM basé sur les automates probabilistes.
3. Évaluer le modèle implanté.

1.6 Calibration de la puissance des ordonnanceurs

Le problème des ordonnanceurs trop forts pour l'analyse de certains protocoles de sécurité a fait l'objet de plusieurs travaux dans la communauté de vérification formelle des protocoles de sécurité. Ces travaux ont proposé différentes méthodes pour calibrer la puissance des ordonnanceurs afin d'éviter d'attribuer une puissance excessive à l'adversaire impossible à atteindre dans les conditions réelles d'utilisation de ces protocoles.

Canetti *et al.* (2006) montrent dans leurs travaux que les techniques de vérification des algorithmes distribués peuvent être appliquées pour analyser les protocoles de sécurité mais en y ajoutant certains mécanismes modélisant efficacement les secrets et limitant la connaissance ainsi que la puissance de calcul des attaquants. Les auteurs proposent un modèle d'automates probabilistes appelé *Task*-PIOA (*Task-structured Probabilistic Input/Output Automata*) une extension des PIOAs de Segala (1995) intégrant un mécanisme de réduction de la puissance des ordonnanceurs en utilisant une approche basée sur la division des actions de chaque composant du système en classes d'équivalence appelées "*tasks*". L'ordre d'exécution de ces *tasks* est décidé à l'avance par un ordonnanceur hors de contrôle de l'adversaire appelé "*tasks scheduler*". Le reste du non déterminisme à l'intérieur de chaque *task* est résolu par un second ordonnanceur appelé "*adversarial scheduler*" modélisant l'adversaire classique à la Dolev-

Yao dans la vérification des protocoles de sécurité. L'*adversarial scheduler* possède alors une connaissance limitée à propos des autres composants du fait que seules les informations qu'ils communiquent durant l'exécution lui sont visibles. Les auteurs définissent comme méthode de vérification, la relation d'implémentation permettant de simuler tout comportement possible d'un protocole réel à l'aide d'un système abstrait contenant une fonctionnalité idéale.

D'autres travaux réalisés par Scedrov *et al.* (2006) proposent un modèle d'algèbres de processus appelé PPC (*Probabilistic Polynomial-time process Calculus*) une variante de CCS (*Calculus of Communicating Systems*) de Milner (1989) permettant de spécifier les protocoles et leurs composants tout en limitant la puissance des ordonnanceurs. Les termes de leur langage ont été restreints et ne peuvent représenter que des protocoles probabilistes dont les calculs s'effectuent en temps polynomial. Les auteurs proposent une méthode de spécification des propriétés utilisant l'équivalence observationnelle sur les termes et les preuves de sécurité sont basées sur la relation de bisimulation. La sémantique opérationnelle de PPC est définie de telle sorte que les calculs internes sont toujours priorisés par rapport aux communications externes et si le non déterminisme apparaît entre les transitions internes, il est résolu avec une distribution uniforme. Cette approche empêche l'ordonnanceur de contrôler les actions internes du protocole mais aussi peut beaucoup l'affaiblir et ainsi ne pas révéler certaines failles durant l'analyse du protocole.

Hamadou et Mullins (2010) proposent le modèle d'algèbres de processus PPC_σ une variante de PPC de Scedrov *et al.* (2006) utilisant les étiquettes pour indexer les actions. Chaque action partielle est indexée par une étiquette dénotant le composant auquel elle appartient et les actions de communications par une paire d'étiquettes dénotant les composants auxquels les actions complémentaires partielles appartiennent. Pour différencier les actions du protocole et celles de l'attaquant, ils partitionnent l'ensemble des étiquettes en deux sous-ensembles disjoints. Pour restreindre la puissance de l'adversaire, les auteurs proposent des ordonnanceurs qui choisissent un ensemble de transitions devant rester indistinguables par l'attaquant. Cette technique consiste à définir deux niveaux d'ordonnanceurs suivant une relation d'équivalence comme dans Canetti *et al.* (2006). A la différence de ces derniers, une fois défini les classes d'actions indistinguables par l'adversaire, les ordonnanceurs internes résolvent le non déterminisme dans chaque classe avec une distribution de probabilités prédéterminée, généralement la distribution uniforme. Le reste du non déterminisme dans le modèle est résolu par les ordonnanceurs externes considérés sous le contrôle de l'adversaire.

D'autres travaux dans le même sens ont été réalisés par Chatzikokolakis et Palamidessi (2007). Les auteurs proposent le langage CCS_σ, une extension probabiliste du langage CCS de Milner

(1989) dans lequel l'ordonnanceur est spécifié syntaxiquement pour ne pas révéler les choix probabilistes du protocole. Ils utilisent une technique étiquetant les actions du protocole comme dans Hamadou et Mullins (2010) afin d'indiquer les choix visibles à l'adversaire et ceux qui ne le sont pas. L'ordonnanceur se comporte ainsi comme un processus et son rôle est de guider l'exécution du protocole en utilisant les étiquettes. La composition parallèle entre le protocole et l'ordonnanceur syntaxique est complètement probabiliste permettant ainsi l'évaluation des propriétés du protocole. Leur méthode d'évaluation est basée sur les équivalences de tests. Ce système d'étiquetage permet à l'ordonnanceur syntaxique de résoudre tout le non déterminisme sans faire appel à l'ordonnanceur sémantique au niveau de l'automate obtenu via la sémantique opérationnelle de leur algèbre de processus.

D'autres travaux intéressants ont été effectués par Garcia *et al.* (2007) qui proposent, lors de l'analyse des protocoles de sécurité, de restreindre la classe de tous les ordonnanceurs possibles à la classe des *ordonnanceurs admissibles* qui ordonnancent en fonction de l'historique des informations observables du système (c.-à-d. sans les informations cachées comme les actions internes dont l'ordonnanceur ne devrait pas avoir accès). Toutesfois, les auteurs n'établissent pas clairement une méthode permettant de définir cet ensemble d'ordonnanceurs admissibles.

Toutes les approches proposées par ces différents travaux cités pour résoudre le problème de la puissance des ordonnanceurs se limitent à l'aspect théorique et aucun d'eux n'a proposé un outil de vérification automatisé des systèmes probabilistes intégrant cette fonctionnalité de restriction de la puissance des ordonnanceurs. Notre projet s'efforce de poser des briques dans ce sens.

1.7 Méthodologie et contribution

La modélisation d'applications probabilistes et concurrentes exige un formalisme supportant à la fois l'aspect probabiliste et non déterministe de certains protocoles de sécurité. Parmi les formalismes utilisés par PRISM seuls les MDPs (*Markov Decision Processes*) Baier (1998); Baier et Katoen (2008) supportent les deux aspects. Nous construisons donc notre modèle de restriction de l'adversaire sur ces MDPs. Une fois le modèle de restriction défini, l'étape suivante consiste à l'implanter dans l'outil PRISM. L'implantation du modèle de restriction dans PRISM doit respecter les conditions suivantes :

- Cette fonctionnalité doit être optionnelle afin de préserver le comportement original de PRISM. Elle doit ainsi être activable et désactivable depuis les interfaces externes ;

- Les propriétés probabilistes doivent être évaluées sur le modèle issu de la restriction de l'adversaire avec les mêmes algorithmes de *model checking* implantés dans PRISM ;
- L'implémentation du modèle de restriction de l'adversaire doit respecter les principes de représentation symbolique utilisant les diagrammes de décision (BDDs et MTBDDs) puisque les algorithmes de *model checking* de PRISM sont définies sur ces structures symboliques.

La dernière étape consiste à évaluer le modèle sur études de cas de protocoles de sécurité probabilistes. Ces protocoles sont spécifiés avec le langage de spécification de PRISM puis les propriétés à vérifier avec la logique temporelle PCTL (*Probabilistic Computation Tree Logic*). Une étude comparative est ensuite effectuée sur chaque cas de protocole en considérant deux situations : adversaire trop fort (sans restriction) et adversaire réaliste (en tenant compte du modèle de restriction des ordonnanceurs proposé).

A notre connaissance, tous les travaux reliés au problème de la puissance des ordonnanceurs pour l'évaluation des protocoles de sécurité proposent des solutions où les preuves de propriétés s'effectuent manuellement dans un cadre théorique. Notre principale contribution consiste en la réalisation de la première solution automatisée intégrant un modèle de restriction de l'adversaire dans un outil automatisé de vérification des systèmes probabilistes.

1.8 Organisation du mémoire

Le reste du mémoire est organisé comme suit : Le Chapitre 2 présente les systèmes de transitions probabilistes dont les MDPs et DTMCs que nous utilisons comme formalismes de modélisation des protocoles, le *model checking* PCTL utilisé dans leur vérification ainsi que les structures de données symboliques utilisées par PRISM dans leur représentation en mémoire. Le Chapitre 3 est consacré à la présentation du modèle de sécurité résultant de l'adjonction au MDP de la fonction d'observation de l'adversaire en vue de restreindre sa puissance ainsi que les algorithmes l'implantant dans l'outil PRISM. Dans le Chapitre 4, une évaluation du modèle proposé est effectuée sur des études de cas de protocoles de sécurité. Enfin, le Chapitre 5 présente la conclusion et les travaux futurs.

CHAPITRE 2

SYSTÈMES DE TRANSITIONS PROBABILISTES ET REPRÉSENTATION SYMBOLIQUE

Dans ce chapitre, nous introduisons d'abord les systèmes de transitions probabilistes et les principes du *model checking* probabiliste que nous utilisons dans l'analyse des protocoles de sécurité. Ensuite, nous procédons à un bref passage en revue des langages de spécification dans le *model checker* PRISM et enfin, nous présentons les principes des structures de données de base utilisées dans PRISM appelées diagrammes de décision.

2.1 Rappel sur les notions des probabilités

Dans cette section nous présentons une brève introduction sur quelques notions et terminologies standards utilisées dans la théorie des probabilités. Ces concepts sont importants pour l'étude et l'analyse des systèmes probabilistes que nous exposons dans ce chapitre. Ils sont tirés de Reischer *et al.* (2002); Bagacher (2007). Les preuves de propositions que nous présentons dans cette section peuvent être trouvées dans ces deux manuels ou tout autre traitant la théorie des mesures.

2.1.1 Espace mesurable

Soit un ensemble non vide Ω. Une *tribu* ou *σ-algèbre* sur Ω est un ensemble non vide $\mathcal{F} \subseteq \mathcal{P}(\Omega)$ où $\mathcal{P}(\Omega)$ est l'ensemble des parties de Ω tel que :

1. $\Omega \in \mathcal{F}$
2. $\mathrm{A} \in \mathcal{F}$ implique $\bar{\mathrm{A}} \in \mathcal{F}$
3. si $\{\mathrm{A}_i\}_{i\in\mathbb{N}}$ est une suite d'éléments de $\mathcal{F}$, alors $\cup_i \mathrm{A}_i \in \mathcal{F}$

Les trois conditions impliquent qu'une *σ-algèbre* est stable par rapport à la réunion et à la complémentarité. $\{\emptyset, \Omega\}$ est la plus petite tribu sur Ω tandis que l'ensemble $\mathcal{P}(\Omega)$ des parties de Ω est la plus grande tribu sur Ω. Les éléments d'une tribu sont appelés des *ensembles mesurables*.

Nous appellons *espace mesurable*, tout couple $(\Omega, \mathcal{F})$ formé d'un ensemble non vide Ω et d'une

tribu $\mathcal{F}$ sur Ω.

Soit $\mathcal{C}$ une collection non vide de sous-ensembles de Ω. La σ-algèbre engendrée par $\mathcal{C}$ est la plus petite σ-algèbre contenant $\mathcal{C}$. Elle est dénotée par $\sigma(\mathcal{C})$. La collection $\mathcal{C}$ est appelée *générateur* pour $\sigma(\mathcal{C})$.

2.1.2 Mesure de probabilité et espace probabilisé

Soit $\mathcal{C}$ une collection de sous-ensembles de Ω. Une *mesure* μ sur $\mathcal{C}$ est une fonction qui assigne une valeur réelle positive à chaque élément de $\mathcal{C}$ telle que :

1. Si $\emptyset$ est un élément de $\mathcal{C}$, alors $\mu(\emptyset) = 0$
2. Si $(\mathrm{C}_i)_{i\in\mathbb{N}}$ forme une séquence d'éléments de $\mathcal{C}$ deux à deux disjoints et $\cup_i \mathrm{C}_i$ est un élément de $\mathcal{C}$, alors $\mu(\cup_i \mathrm{C}_i) = \sum_i \mu(\mathrm{C}_i)$

Si $(\Omega, \mathcal{F})$ est un espace mesurable, alors une mesure sur $\mathcal{F}$ est appelée *mesure* sur $(\Omega, \mathcal{F})$. Une mesure sur une collection $\mathcal{C}$ de sous-ensembles de Ω est *finie* si la mesure de chaque élément de $\mathcal{C}$ est finie.

Un *espace mesuré* est un triplet $(\Omega, \mathcal{F}, \mu)$ où $(\Omega, \mathcal{F})$ est un espace mesurable et μ une mesure sur $(\Omega, \mathcal{F})$. L'espace mesuré est considéré *discret* si $\mathcal{F} = 2^\Omega$ et si la mesure de chaque ensemble mesurable est la somme des mesures de ses éléments.

Un *espace probabilisé* est un triplet $(\Omega, \mathcal{F}, \mathrm{P})$ où $(\Omega, \mathcal{F})$ est un espace mesurable et P une mesure sur $(\Omega, \mathcal{F})$ telle que $\mathrm{P}(\Omega) = 1$. La mesure P est appelée aussi *mesure de probabilité discrète* ou *distribution de probabilités*. Les éléments de $\mathcal{F}$ sont appelés des *événements*. L'ensemble de toutes les distributions de probabilités définies sur Ω est noté $\mathfrak{D}(\Omega)$.

Proposition 2.1.1 *Soit $\{(\Omega_i, \mathcal{F}_i, \mathrm{P}_i)\}_{i\geq 0}$ un ensemble d'espaces probabilisés et soit $\{a_i\}_{i\geq 0}$ un ensemble de nombres réels dans $[0, 1]$ tels que $\sum_{i\geq 0} a_i = 1$. La combinaison $\sum_{i\geq 0}(\Omega_i, \mathcal{F}_i, \mathrm{P}_i)$ correspondant au triplet $(\Omega, \mathcal{F}, \mathrm{P})$ où $\Omega = \cup_{i\geq 0}\Omega_i$, $\mathcal{F} = 2^\Omega$, et $\forall x \in \Omega$, $\mathrm{P}(x) = \sum_{i\geq 0|x\in\Omega_i} a_i \mathrm{P}_i(x)$, est un espace probabilisé.*

2.1.3 Fonction mesurable

Soient $(\Omega, \mathcal{F})$ et $(\Omega', \mathcal{F}')$ deux espaces mesurables. Une fonction $f : \Omega \to \Omega'$ est appelée *fonction mesurable* de $(\Omega, \mathcal{F})$ vers $(\Omega', \mathcal{F}')$ si pour chaque élément C de $\mathcal{F}'$, l'image inverse de C notée $f^{-1}(\mathrm{C})$ est un élément de $\mathcal{F}$.

Soient μ une mesure sur $(\Omega, \mathcal{F})$ et f une fonction mesurable de $(\Omega, \mathcal{F})$ vers $(\Omega', \mathcal{F}')$. La fonction μ' définie sur $\mathcal{F}'$ telle que $\mu'(\mathrm{C}) = \mu(f^{-1}(\mathrm{C}))$ pour chaque élément $\mathrm{C} \in \Omega'$ est une mesure sur $(\Omega', \mathcal{F}')$, notée également $f(\mu)$, appelée mesure induite par la fonction f.

Proposition 2.1.2 *Soit f une fonction mesurable de $(\Omega_1, \mathcal{F}_1)$ vers $(\Omega_2, \mathcal{F}_2)$ et soit g une fonction mesurable de $(\Omega_2, \mathcal{F}_2)$ vers $(\Omega_3, \mathcal{F}_3)$. Alors $f \circ g$ est une fonction mesurable de $(\Omega_1, \mathcal{F}_1)$ vers $(\Omega_3, \mathcal{F}_3)$.*

2.1.4 Espace mesuré produit

Soient $(\Omega_1, \mathcal{F}_1, \mu_1)$ et $(\Omega_2, \mathcal{F}_2, \mu_2)$ deux espaces mesurés et le produit cartésien $\Omega_1 \times \Omega_2 = \Omega$. Nous appelons *pavés mesurables* de Ω les sous-ensembles de la forme $\mathrm{A}_1 \times \mathrm{A}_2$ tels que $\forall i = 1, 2$, $\mathrm{A}_i \in \mathcal{F}_i$. La tribu $\mathcal{F}$ sur Ω, notée $\mathcal{F}_1 \otimes \mathcal{F}_2$, engendrée par les pavés mesurables de Ω s'appelle la *tribu produit*.

Proposition 2.1.3 *Soient $(\Omega_1, \mathcal{F}_1, \mu_1)$ et $(\Omega_2, \mathcal{F}_2, \mu_2)$ deux espaces mesurés où μ_1 et μ_2 sont des mesures finies. Alors il existe une mesure unique sur $\mathcal{F}_1 \otimes \mathcal{F}_2$ notée $\mu_1 \otimes \mu_2$ telle que $\forall \mathrm{A}_1 \in \mathcal{F}_1$ et $\forall \mathrm{A}_2 \in \mathcal{F}_2$, $\mu_1 \otimes \mu_2(\mathrm{A}_1 \times \mathrm{A}_2) = \mu_1(\mathrm{A}_1) \cdot \mu_2(\mathrm{A}_2)$.*

Un *espace mesuré produit* de deux espaces mesurés $(\Omega_1, \mathcal{F}_1, \mu_1)$ et $(\Omega_2, \mathcal{F}_2, \mu_2)$, noté $(\Omega_1, \mathcal{F}_1, \mu_1) \otimes (\Omega_2, \mathcal{F}_2, \mu_2)$, est l'espace mesuré $(\Omega, \mathcal{F}, \mu)$ tel que $\Omega = \Omega_1 \times \Omega_2$, $\mathcal{F} = \mathcal{F}_1 \otimes \mathcal{F}_2$ et $\mu = \mu_1 \otimes \mu_2$.

2.2 Les systèmes de transitions probabilistes

Comme nous l'avons déjà signalé dans l'introduction de notre mémoire, l'étude des systèmes probabilistes nécessite l'adoption d'un formalisme pouvant nous permettre de modéliser l'aspect aléatoire des systèmes. Plusieurs modèles ont été proposés dont les systèmes de transitions probabilistes de Glabbeek *et al.* (1990) classifiés en trois catégories : les modèles réactifs, génératifs et stratifiés. Parmi ces trois types de modèles seuls les modèles réactifs supportent les aspects probabiliste et non déterministe. Les modèles de *chaînes de Markov concurrentes* de Vardi (1985) supportent également les deux aspects et distinguent les états probabilistes des états non déterministes. Les états probabilistes sont du style des modèles génératifs de Glabbeek *et al.* (1990) et les états non déterministes sont du style de ceux des systèmes de transitions étiquetés non probabilistes de Keller (1976). Un autre modèle supportant les probabilités et le non déterminisme est le modèle de PIOA de Segala (1995) qui étend le modèle de IOA de Lynch et Tuttle (1989) avec des notions de probabilités. Dans les PIOAs de Segala, chaque composant du système est modélisé sous-forme d'un automate avec des transitions probabilistes et le système entier résulte de la composition parallèle entre les automates représentant les processus et les automates représentant les canaux de

communications entre processus.

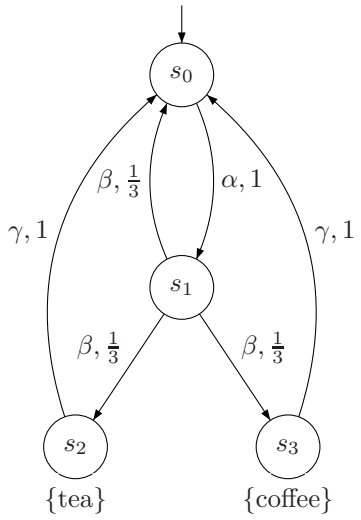
Dans la suite de ce mémoire, nous nous intéressons particulièrement aux *chaînes de Markov* et aux *processus de décision markoviens* Baier (1998); Parker (2002); Baier et Katoen (2008) qui sont supportés par le *model checker* PRISM.

2.2.1 Chaînes de Markov à temps discret

Les *chaînes de Markov à temps discret* (DTMC) sont des systèmes de transitions où la transition d'un état vers un autre est choisie suivant une certaine distribution de probabilités. Une DTMC peut être définie comme un tuple $(S, s_{init}, \Delta, L, Act)$ où :

- S est un ensemble fini d'états ;
- s_{init} est un état initial ;
- Act est un ensemble fini d'actions ;
- $\Delta : S \longrightarrow Act \times \mathfrak{D}(S)$ est une *fonction de transitions* ;
- $L : S \longrightarrow 2^{AP}$ est une *fonction d'étiquetage* où AP est l'ensemble des propositions atomiques valides dans les états.

Par abus de notation $\mu(s')$, où $\Delta(s) = (\alpha, \mu)$, est dénoté par $\Delta(s, \alpha, s')$. Les propositions atomiques jouent le même rôle que dans les systèmes de transitions non probabilistes de Keller (1976). Elles permettent d'indiquer quels états satisfont quelles propriétés parmi un ensemble de propriétés d'intérêt. Un exemple de DTMC et sa fonction de transitions Δ est donné à la Figure 2.1. Il s'agit d'un système de transitions probabiliste simple modélisant une machine distributrice de café et de thé. Dans ce petit modèle, nous assumons que le prix d'une tasse de café et de thé est le même. L'état initial s_0 modélise l'état de la machine au moment où elle est prête à recevoir de la monnaie. Une fois la monnaie insérée, la machine passe à l'état prêt au service (état s_1) avec probabilité 1. A ce niveau, le ditributeur choisit l'état suivant probabilistiquement. Il peut soit restituer la monnaie avec probabilité $\frac{1}{3}$, soit servir du café avec probabilité $\frac{1}{3}$ ou soit servir du thé avec probabilité $\frac{1}{3}$. Les propriétés atomiques *tea* dans l'état s_2 et *coffee* dans l'état s_3 modélisent respectivement le service de thé et de café par la machine. Une fois le service effectué, la machine retourne à l'état initial en attente de la monnaie pour un nouveau service. Certes, cet exemple paraît très simple mais prouve l'élégance du formalisme à représenter des systèmes même complexes.



$$\Delta = \begin{pmatrix} 0 & 1 & 0 & 0 \\ \frac{1}{3} & 0 & \frac{1}{3} & \frac{1}{3} \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \begin{matrix} \alpha \\ \beta \\ \gamma \\ \gamma \end{matrix}$$

Figure 2.1 DTMC modélisant un distributeur de café et de thé et sa matrice des transitions

Chemin d'exécution dans une DTMC

L'exécution d'un système modélisé sous-forme d'une *chaîne de Markov* est représentée par un *chemin d'exécution* à travers le modèle. Un chemin d'exécution noté π est une séquence non vide d'états et d'actions $s_0\,\alpha_0\,s_1\,\alpha_1\,s_2\,\alpha_2\,s_3\cdots$ où $s_i \in S$ et $\Delta(s_i, \alpha_i, s_{i+1}) > 0$, $\forall i \geq 0$. L'élément $\pi(i)$ représente l'état s_i dans le chemin d'exécution π. Un chemin d'exécution peut être fini ou infini. Étant donnée une chaîne de Markov M, nous dénotons par $Path(\mathrm{M})$ l'ensemble de tout les chemins d'exécution, $Path^s(\mathrm{M})$ l'ensemble de tous les chemins d'exécution partant de l'état s et $Path^s_{fin}(\mathrm{M})$ l'ensemble des chemins d'exécution finis partant de l'état s. Dans la mesure où un chemin représente une exécution d'un système, déterminer le comportement probabiliste d'une certaine exécution nécessite de calculer la probabilité avec laquelle le chemin d'exécution a été pris. Hansson et Jonsson (1994); Baier (1998); Parker (2002) définissent pour chaque état $s \in S$ une *mesure de probabilité* pour tous les chemins d'exécution partant de cet état. Cette mesure est simple à définir puisqu'elle est induite par la fonction de transitions Δ. Dans le cas d'un chemin d'exécution fini $\pi = s_0\,\alpha_0\,s_1\,\alpha_1\,s_1\cdots\alpha_{n-1}\,s_n$, la probabilité de π est définie comme suit :

$$\mathrm{P}(\pi) = \begin{cases} 1 & \text{si } \pi \text{ a un seul état} \\ \Delta(s_0, \alpha_0, s_1) \cdot \Delta(s_1, \alpha_1, s_2) \cdots \Delta(s_{n-1}, \alpha_{n-1}, s_n) & \text{sinon} \end{cases}$$

Par souci de simplicité, nous omettons par la suite les actions dans la notation d'un chemin

d'exécution sur une DTMC du fait que dans chaque état, il n'y a qu'une seule action disponible. Considérons le chemin $\pi = s_0\, s_1\, s_0\, s_1\, s_2$ dans l'exemple de la Figure 2.1. La probabilité correspondant à ce chemin est calculée ci-dessous :

$$\begin{aligned} \mathrm{P}(\pi) &= \Delta(s_0, \alpha, s_1) \cdot \Delta(s_1, \beta, s_0) \cdot \Delta(s_0, \alpha, s_1) \cdot \Delta(s_1, \beta, s_2) \\ &= 1 \cdot \frac{1}{3} \cdot 1 \cdot \frac{1}{3} \\ &= \frac{1}{9} \end{aligned}$$

L'évaluation de la satisfaction d'une spécification de propriété par une chaîne de Markov M nécessite l'identification de tous les chemins d'exécution partant de l'état initial du système et satisfaisant la propriété. Segala (1995); Baier (1998); Parker (2002) définissent la notion de "*cylinder set*" correspondant à un chemin d'exécution fini π et noté $C(\pi)$ comme étant l'ensemble de tous les chemins d'exécution ayant comme préfixe le chemin d'exécution fini π. Considérons le chemin d'exécution fini $\pi = s_0\, s_1\, s_0$ sur la chaîne de Markov de la Figure 2.1. Le chemin infini $(s_0\, s_1)^\omega$ constitue un chemin d'exécution de $C(\pi)$ n'aboutissant jamais au service de thé ou de café et retournant toujours la monnaie insérée.

La notion de *cylinder set* permet de définir une mesure de probabilité sur l'ensemble de tous les chemins d'exécution de la chaîne de Markov M. Soit la *σ-algèbre* Σ_s associée à la chaîne de Markov M comme étant la plus petite *σ-algèbre* contenant tous les *cylinder sets* $C(\pi)$ où π est un chemin de l'ensemble $Path^s_{fin}$. Une mesure de probabilité P^M_s peut être définie sur la *σ-algèbre* Σ_s comme une mesure unique avec $\mathrm{P}^M_s(C(\pi)) = \mathrm{P}(\pi)$ Kemeny *et al.* (1966). Cette mesure nous permet de quantifier la probabilité avec laquelle une chaîne de Markov M satisfait une certaine propriété. Considérons l'événement *ev* associé à la σ-algèbre Σ_{s_0} de M à la Figure 2.1 où le café est servi après au plus deux retours de monnaie. La probabilité correspondant à cette événement revient à calculer la somme des probabilités associées aux *cylinder sets* $C(\pi_1)$, $C(\pi_2)$ et $C(\pi_3)$ où $\pi_1 = s_0\, s_1\, s_2$, $\pi_2 = s_0\, s_1\, s_0\, s_1\, s_2$ et

$\pi_3 = s_0\, s_1\, s_0\, s_1\, s_0\, s_1\, s_2$, ce qui équivaut à :

$$\begin{aligned}
\mathrm{P}(ev) &= \mathrm{P}(C(\pi_1)) + \mathrm{P}(C(\pi_2)) + \mathrm{P}(C(\pi_3)) \\
&= \Delta(s_0, \alpha, s_1) \cdot \Delta(s_1, \beta, s_2) \\
&+ \Delta(s_0, \alpha, s_1) \cdot \Delta(s_1, \beta, s_0) \cdot \Delta(s_0, \alpha, s_1) \cdot \Delta(s_1, \beta, s_2) \\
&+ \Delta(s_0, \alpha, s_1) \cdot \Delta(s_1, \beta, s_0) \cdot \Delta(s_0, \alpha, s_1) \cdot \Delta(s_1, \beta, s_0) \cdot \Delta(s_0, \alpha, s_1) \cdot \\
&\quad \Delta(s_1, \beta, s_2) \\
&= 1 \cdot \frac{1}{3} + 1 \cdot \frac{1}{3} \cdot 1 \cdot \frac{1}{3} + 1 \cdot \frac{1}{3} \cdot 1 \cdot \frac{1}{3} \cdot 1 \cdot \frac{1}{3} \\
&= \frac{1}{3} + \frac{1}{9} + \frac{1}{27} \\
&= \frac{13}{27}
\end{aligned}$$

En procédant aux calculs de probabilités associées aux événements de cette façon, certains peuvent nécessiter d'effectuer des sommes infinies, ce qui devient pratiquement infaisable. D'où la nécessité de spécifier de tels événements en utilisant la logique temporelle dont la vérification s'effectue avec des techniques efficaces que nous présentons dans les prochaines sections.

Spécification des propriétés sur une DTMC

Nous présentons un formalisme permettant de spécifier des propriétés sur une chaîne de Markov. Il s'agit de la logique temporelle probabiliste PCTL de Hansson et Jonsson (1994). PCTL est une extension probabiliste de la logique arborescente CTL (*Computation Tree Logic*) de Clarke *et al.* (1986). La principale différence entre les deux logiques réside dans la présence de l'opérateur probabiliste $\mathcal{P}_{\bowtie p}(\varphi)$ pour les formules PCTL où φ est une formule de chemin et $\bowtie p$ est une expression avec $\bowtie$ un opérateur de l'ensemble $\{<, \leq, >, \geq\}$ et $p \in [0, 1]$.

Les formules PCTL se classent en deux catégories :

- Les formules d'état
- Les formules de chemin

Les formules d'état sur un ensemble AP de propositions atomiques suivent la syntaxe ci-dessous :

$$\Phi ::= \ vrai \mid a \mid \Phi_1 \ \wedge \ \Phi_2 \mid \neg\, \Phi \mid \mathcal{P}_{\bowtie p}(\varphi)$$

où $a \in AP$ l'ensemble des propositions atomiques, $\bowtie \in \{<, \leq, >, \geq\}, p \in [0,1]$ et φ est une formule de chemin.

Quant aux formules de chemin la syntaxe suit :

$$\varphi ::= \mathbf{X}\Phi \mid \Phi_1 \, \mathcal{U} \, \Phi_2 \mid \Phi_1 \, \mathcal{U}^{\leq n} \, \Phi_2$$

où Φ, Φ_1 et Φ_2 sont des formules d'état et $n \in \mathbb{N}$

Un état satisfait $\mathcal{P}_{\bowtie p}(\varphi)$ si la probabilité de choisir un chemin partant de cet état et satisfaisant la formule de chemin φ est située dans l'intervalle représenté par $\bowtie p$. La formule $\mathbf{X}\Phi$ est vraie dans un état s de l'ensemble S des états du modèle si Φ est vraie dans l'état suivant ; $\Phi_1 \, \mathcal{U} \, \Phi_2$ est vraie dans un état s si Φ_2 est vraie dans un futur état du chemin partant de s et Φ_1 est vraie dans tous les autres états du chemin avant l'état satisfaisant Φ_2. $\Phi_1 \, \mathcal{U}^{\leq n} \, \Phi_2$ a la même signification que précédemment sauf que Φ_2 doit être satisfait après au plus n pas. La satisfaction d'une formule Φ par un état s est notée par $s \models \Phi$. Formellement nous résumons la relation de satisfaction des formules PCTL sur un système modélisé sous-forme d'une chaîne de Markov M comme suit :

- Pour un état $s \in S$:

$$\begin{array}{lll}
s \models vrai & & \forall s \in S \\
s \models a & \iff & a \in L(s) \\
s \models \neg\Phi & \iff & s \not\models \Phi \\
s \models \Phi_1 \wedge \Phi_2 & \iff & s \models \Phi_1 \text{ et } s \models \Phi_2 \\
s \models \mathcal{P}_{\bowtie p}(\varphi) & \iff & \mathrm{P}(\{\pi \in Path_s(\mathrm{M}) \mid \pi \models \varphi\}) \bowtie p
\end{array}$$

où P est calculée grâce à la *mesure de probabilité* sur l'ensemble des chemins de M partant de s.

- Pour un chemin $\pi \in Path(\mathrm{M})$:

$$\begin{array}{lll}
\pi \models \mathbf{X}\Phi & \iff & \pi(1) \models \Phi \\
\pi \models \Phi_1 \mathcal{U} \Phi_2 & \iff & \exists j \geq 0 \mid (\pi(j) \models \Phi_2 \wedge (\forall 0 \leq k < j : \ \pi(k) \models \Phi_1)) \\
\pi \models \Phi_1 \, \mathcal{U}^{\leq n} \, \Phi_2 & \iff & \exists 0 \leq j \leq n \mid (\pi(j) \models \Phi_2 \wedge (\forall 0 \leq k < j : \ \pi(k) \models \Phi_1))
\end{array}$$

Nous pouvons aussi facilement dériver à partir de cette syntaxe d'autres opérateurs classiques de la logique temporelle **F** (*fatalement*) et **G** (*toujours*). En effet :

$$\begin{aligned}
\mathbf{F}\Phi &\equiv vrai\ \mathcal{U}\ \Phi \\
\mathbf{F}^{\leq n}\Phi &\equiv vrai\ \mathcal{U}^{\leq n}\ \Phi \\
\mathbf{G}\Phi &\equiv \neg\mathbf{F}\neg\Phi \\
\mathbf{G}^{\leq n}\Phi &\equiv \neg\mathbf{F}^{\leq n}\neg\Phi
\end{aligned}$$

Considérons par exemple la propriété suivante sur la DTMC de la Figure 2.1 :

> "*La probabilité que le café ou le thé soit servi par la machine est supérieure ou égale à un tiers*"

Cette propriété peut s'exprimer en PCTL simplement comme $\mathcal{P}_{\geq \frac{1}{3}}\mathbf{F}(coffee \vee tea)$ et se vérifie en utilisant l'un des algorithmes de *model checking* de Hansson et Jonsson (1994); Baier (1998) que nous discutons brièvement à la Section 2.2.3.

2.2.2 Processus de décision markovien

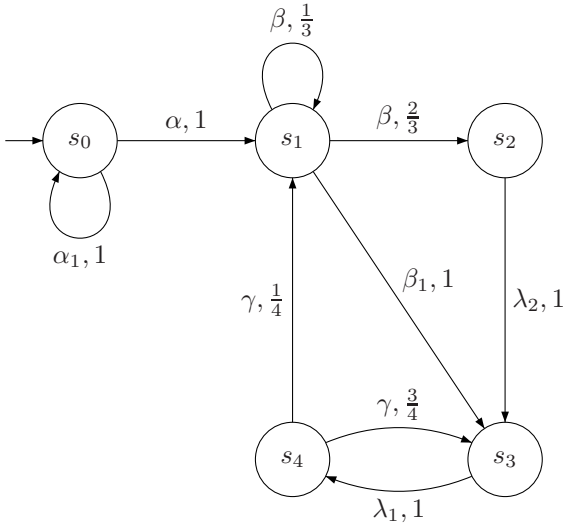
Contrairement aux chaînes de Markov, les *processus de décision markoviens* (MDP) supportent à la fois le comportement non déterministe et probabiliste des systèmes. Ce modèle est plus expressif que la DTMC car il permet de modéliser facilement la communication entre les processus probabilistes tels que les agents ou participants de la plupart des protocoles de sécurité.

Un processus de décision markovien Mp est un tuple $(S, s_{init}, \Delta, L, Act)$ où :

- S est un ensemble fini d'états ;
- s_{init} est un état initial ;
- Act est un ensemble fini d'actions ;
- $\Delta : S \times Act \longrightarrow \mathfrak{D}(S) \cup \{\surd\}$ est une fonction de transitions telle que $\forall \alpha \in Act, \forall s \in S :\ \Delta(s,\alpha) = \begin{cases} \surd & \text{si } \alpha \notin Act(s) \\ \mu \text{ sur } S & \text{sinon} \end{cases}$
- $L : S \longrightarrow 2^{AP}$ est une fonction d'étiquetage où AP est l'ensemble des propositions atomiques valides dans les états.

Nous dénotons par $Act(s)$ l'ensemble des actions disponibles dans l'état s. Dans cette définition, nous retrouvons les mêmes éléments que dans les DTMCs exceptée la fonction Δ qui est étendue en permettant l'expression du non déterminisme dans les MDPs. La fonction Δ du MDP associe une distribution de probabilités à chaque action de l'ensemble d'actions disponibles dans un état. Chaque distribution représente les probabilités de transitions d'un état vers un ensemble d'états comme dans les DTMCs.

La Figure 2.2 représente un exemple de processus de décision markovien. Dans ce système



$$\Delta = \left(\begin{array}{ccccc} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ \hline 0 & \frac{1}{3} & \frac{2}{3} & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ \hline 0 & 0 & 0 & 1 & 0 \\ \hline 0 & 0 & 0 & 0 & 1 \\ \hline 0 & \frac{1}{4} & 0 & \frac{3}{4} & 0 \end{array}\right) \begin{array}{l} \alpha_1 \\ \alpha \\ \beta \\ \beta_1 \\ \lambda_2 \\ \lambda_1 \\ \gamma \end{array}$$

Figure 2.2 MDP à 5 états et la fonction de transitions correspondante

de transitions, deux états s_0 et s_1 possèdent plus d'une action, chacune associée à une distribution de probabilités. Dans l'état s_0 le non déterminisme apparaît entre les actions α et α_1. Quant à l'état s_1, il apparaît entre les actions β et β_1. La fonction Δ est représentée dans la Figure 2.2 sous-forme d'une superposition de 5 matrices, chacune correspondant à un état parmi les états du modèle. Intuitivement, le nombre de lignes dans chaque matrice indique le nombre de distributions de probabilités, chacune correspondant à une action parmi les actions du système, dans l'état correspondant. Les transitions non déterministes existent dans les états correspondant aux matrices ayant plus d'une ligne.

Chemin d'exécution dans un MDP

Pour déterminer le chemin d'exécution dans un MDP, les résolutions des choix *non déterministes* et *probabilistes* doivent être efféctuées. Un *chemin d'exécution* se présente sous-forme

d'une séquence non vide de la forme :

$$\pi = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} s_2 \xrightarrow{\alpha_2} s_3 \xrightarrow{\alpha_3} \cdots$$

où $\forall i \geq 0$, α_i est l'action exécutée à l'état s_i du MDP.

Contrairement aux chaînes de Markov où nous pouvons définir une *σ-algèbre* et une *mesure de probabilité* sur l'ensemble des chemins, les processus de décision markoviens n'offrent pas directement cette possibilité du fait de la présence du non déterminisme où aucune contrainte n'est imposée pour sa résolution. Plusieurs résolutions sont ainsi possibles et à chaque résolution correspond une mesure de probabilité. La première étape avant de définir une mesure de probabilité consiste alors à résoudre le non déterminisme dans le MDP. Cette résolution est effectuée par un *ordonnanceur* appelé aussi *stratégie* ou *adversaire* qui choisit une action parmi plusieurs actions possibles (actives) dans chaque état.

Ordonnanceur (adversaire)

L'analyse des propriétés du MDP passe par la résolution de tout le non déterminisme présent dans le modèle grâce à un ordonnanceur. Soit le MDP $\mathrm{M}p = (S, s_{init}, \Delta, L, Act)$. Un ordonnanceur sur $\mathrm{M}p$ est défini comme une fonction

$$\zeta : \ S^+ \longrightarrow Act$$

telle que $\forall s_0 s_1 \cdots s_n \in S^+, \ \zeta(s_0 s_1 \cdots s_n) \in Act(s_n)$.

Cet ordonnanceur est considéré *historiquement dépendant* puisque son choix dans l'état s_n est effectué en fonction de l'historique de l'exécution du système depuis l'état initial s_0 jusqu'à l'état s_n. Nous dénotons par $Scheds$ l'ensemble de tous les ordonnanceurs possibles sur le MDP. Une fois tout le non déterminisme résolu dans le MDP, le système est purement probabiliste et devient une chaîne de Markov sous-forme d'un *arbre d'exécution* où les états sont constitués par des séquences d'états du MDP.

Formellement, nous définissons une telle chaîne de markov obtenue à partir du MDP $\mathrm{M}p = (S, s_{init}, \Delta, L, Act)$ sous l'ordonnanceur ζ comme étant un tuple $\mathrm{M}p_\zeta = (S^+, s_{init}, \Delta_\zeta, L', Act)$ tel que :

$$\forall t = s_0\, s_1 \cdots s_n \in S^+ \ et \ \alpha \in Act, \ \text{nous avons} \begin{cases} \Delta_\zeta(t, \alpha, s_{n+1}) = \Delta(s_n, \alpha, s_{n+1}) \\ L'(t) = L(s_n) \end{cases}$$

La Figure 2.3 représente une portion de l'arbre d'exécution du MDP de la Figure 2.2 sous

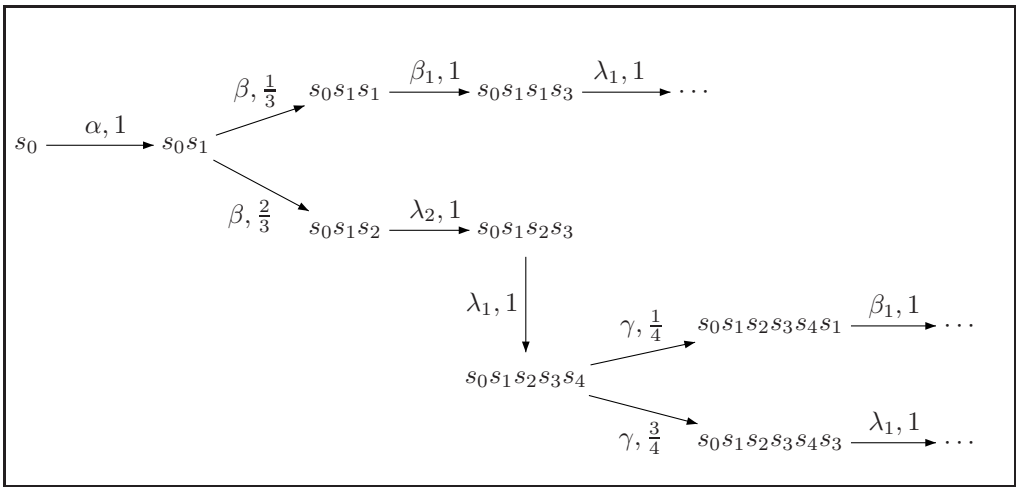


Figure 2.3 Arbre d'exécution du MDP sous l'ordonnanceur ζ_1

l'ordonnanceur ζ_1 qui choisit toujours l'action α dans l'état s_0 et l'action β ou β_1 dans l'état s_1 selon que le nombre d'états s_1 dans l'historique est impair ou non.

En dénotant $Path^s(\mathrm{M}p)$ l'ensemble de tous les chemins partant d'un état s du MDP Mp, nous définissons $Path^s_\zeta(\mathrm{M}p)$ comme étant le sous-ensemble des chemins partant de s et après que tout le non déterminisme soit résolu par l'ordonnanceur ζ. Pour chaque état s, une mesure de probabilité $\mathrm{P}_s^{\mathrm{M}p_\zeta}$ sur le MDP sous l'ordonnanceur ζ peut être définie sur l'espace $Path^\zeta_s$.

Specification des propriétés sur un MDP

Comme pour les DTMCs dans la Section 2.2.1, le langage PCTL est utilisé pour spécifier les propriétés sur les MDPs. La seule différence réside dans le fait que pour les MDPs, la relation de satisfaction est définie en quantifiant sur tous les ordonnanceurs de l'ensemble $Scheds$. En effet, nous résumons la relation de satisfaction dans un état s d'une formule d'état sur un MDP comme suit :

$$
\begin{aligned}
&s \models_{Scheds} vrai && && \forall s \in S \\
&s \models_{Scheds} a && \iff && a \in L(s) \\
&s \models_{Scheds} \neg\Phi && \iff && s \not\models_{Scheds} \Phi \\
&s \models_{Scheds} \Phi_1 \ \wedge \ \Phi_2 && \iff && s \models_{Scheds} \ \Phi_1 \ \wedge \ s \models_{Scheds} \Phi_2 \\
&s \models_{Scheds} \mathcal{P}_{\bowtie p}(\varphi) && \iff && \mathrm{P}(\{\pi \in Path^s_\zeta(\mathrm{M}p) |\ \pi \models \varphi\}) \bowtie p \ \forall \zeta \in Scheds
\end{aligned}
$$

où P est calculé grâce à la *mesure de probabilité* sur l'ensemble des chemins de Mp partant de l'état s et sous la résolution de l'ordonnanceur ζ de l'ensemble $Scheds$.

Quant aux formules de chemins, elles sont syntaxiquement et sémantiquement les mêmes que dans les DTMCs, et c'est pour cette raison que nous ne les reprenons pas dans cette section.

2.2.3 Model checking probabiliste pour MDPs et DTMCs

Le *model checking* PCTL sur MDP Bianco et de Alfaro (1995) et DTMC Hansson et Jonsson (1994) est une extension probabiliste de celui présenté par Clarke *et al.* (1986) sur CTL. Pour les formules sans opérateur probabiliste, le *model checking* s'effectue de la même manière sur les DTMCs et MDPs. Il s'agit de déterminer l'ensemble des états satisfaisant la formule. Cette procédure consiste à décomposer la formule et à l'organiser sous-forme d'un arbre Clarke *et al.* (1986). La racine constitue la formule complète, les noeuds intermédiaires sont des sous-formules et les feuilles sont des propositions atomiques ou des valeurs booléennes "*vrai*". La détermination des états satisfaisant la formule se fait des feuilles vers la racine et à la fin le résultat est trivial.

Pour les formules comportant l'opérateur probabiliste le *model checking* nécessite, pour

Tableau 2.1 Méthodes de calcul des probabilités de satisfaction pour les formules de chemin

Type de modèle	$\mathbf{X}\Phi$	$\Phi_1 \mathcal{U} \Phi_2$	$\Phi_1 \mathcal{U}^{\leq n}\ \Phi_2$
DTMC	Une multiplication de matrice/vecteur	Méthodes directes ou méthodes itératives	n multiplications de matrice/vecteur
MDP	Deux opérations : - Multiplication de matrice/vecteur - Opération de *max* ou de *min* dans chaque état	Optimisation linéaire par la méthode du *simplexe*	n itérations de : - Multiplication de matrice/vecteur - Opération de *max* ou de *min* dans chaque état

chaque état, le calcul de la probabilité qu'un chemin partant de l'état satisfait la formule. Le calcul de ces probabilités s'opère différemment suivant la nature de la formule de chemin. Le Tableau 2.1 résume les méthodes de calcul des probabilités utilisées pour les trois types

de formules de chemin selon que le modèle est une DTMC ou un MDP. La grande différence entre les deux c'est que pour les MDPs nous avons besoin de calculer la probabilité *maximale* ou la probabilité *minimale*, dépendamment de l'inégalité de la formule et en considérant tous les ordonnanceurs, qu'un chemin partant de l'état satisfait la formule. Une fois calculées les probabilités pour tous les états, la détermination de l'ensemble des états satisfaisant la formule est trivial. Il suffit de comparer les probabilités calculées avec l'inégalité de la formule.

Exemple : Considérons le MDP de la Figure 2.2 tel que $AP = \{a, b\}$, $L(s_0) = \{\}$, $L(s_1) = \{\}$, $L(s_2) = \{a\}$, $L(s_3) = \{b\}$ et $L(s_4) = \{\}$. Nous voulons calculer les états satisfaisant la formule $\mathcal{P}_{\leq \frac{1}{4}}(\mathbf{X}(a \vee b))$.

Soit $\phi = a \vee b$. Les états où $\mathbf{X}\phi$ est vérifiée sont s_1, s_2 et s_4. Dans s_1 la probabilité associée à $\mathbf{X}\phi$ est $\frac{2}{3}$ si l'ordonnanceur choisit l'action β et 1 si l'ordonnanceur choisit l'action β_1. La probabilité maximale que s_1 satisfait $\mathbf{X}\phi$ en considérant tous les ordonnanceurs est donc 1. Dans l'état s_2 la probabilité associée à $\mathbf{X}\phi$ est 1 en exécutant l'action λ_2. Dans l'état s_4 $\mathbf{X}\phi$ est valide en exécutant γ avec probabilité $\frac{3}{4}$. Dans le reste des états (s_0 et s_3) la probabilité correspondant à $\mathbf{X}\phi$ est 0 car la formule de chemin est invalide dans ces états quelque soit l'action exécutée. Les seuls états du modèle où la probabilité est inférieure ou égale à $\frac{1}{4}$ sont s_0 et s_3. Nous pouvons alors conclure que la formule $\mathcal{P}_{\leq \frac{1}{4}}(\mathbf{X}(a \vee b))$ est valide uniquement dans ces deux états.

Le calcul peut s'effectuer aussi simplement par multiplication de matrice/vecteur. Nous construisons d'abord le vecteur V_ϕ de taille $|S|$ tel que $\forall i = 0\,..\,4$, $\mathrm{V}_\phi(i) = 1$ si $s_i \models \phi$ et 0 sinon. La probabilité de satisfaction de $\mathbf{X}\phi$ pour chaque état s_i est déterminée par la variable x_i. Ces probabilités de satisfaction sont ainsi calculées en effectuant la multiplication de la matrice des probabilités de transition par le vecteur V_ϕ puis en prenant le maximum dans chaque état car l'inégalité dans la formule est "$\leq$". En effet :

$$\begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \left(\begin{array}{ccccc} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ \hline 0 & \frac{1}{3} & \frac{2}{3} & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ \hline 0 & 0 & 0 & 1 & 0 \\ \hline 0 & 0 & 0 & 0 & 1 \\ \hline 0 & \frac{1}{4} & 0 & \frac{3}{4} & 0 \end{array}\right) \cdot \begin{pmatrix} 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix} = \left(\begin{array}{c} 0 \\ 0 \\ \hline \frac{2}{3} \\ 1 \\ \hline 1 \\ \hline 0 \\ \hline \frac{3}{4} \end{array}\right) \overset{\mathbf{Max}}{\Longrightarrow} \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \\ \frac{3}{4} \end{pmatrix}$$

Le résultat des opérations montre que la formule est vérifiée uniquement dans les états s_0 et s_3 du modèle où la probabilité maximale est inférieure ou égale à $\frac{1}{4}$. Pour plus détails sur les

algorithmes de *model checking* PCTL et leur implémentation dans PRISM voir Hansson et Jonsson (1994); Bianco et de Alfaro (1995); Baier (1998); Parker (2002).

2.2.4 Composition parallèle dans les MDPs

Dans nombreux des cas, les systèmes résultent de la composition d'un ensemble de processus communicant et/ou évoluant indépendamment. Leur spécification et vérification nécessitent de définir un ensemble de règles de composition à partir des différents processus impliqués. Ces règles sont généralement définies sous-forme d'algèbres de processus (par exemple CCS de Milner (1989), CSP (*Communicating Sequential Processes*) de Hoare (1985), etc.) où les termes représentent les processus et les opérateurs la manière dont ils sont combinés. Dans cette section, nous discutons brièvement de la composition parallèle style CSP où les systèmes de transitions se synchronisent uniquement sur les actions identiques et évoluant indépendamment sur les autres actions. Nous couvrons uniquement la composition parallèle entre MDPs. Sachant qu'une DTMC peut être considérée comme un MDP où dans chaque état une seule action est disponible, les mêmes règles peuvent être appliquées.

La composition parallèle entre deux processus de décision markoviens $\mathrm{M}p_1 = (S_1, s_{init}, \Delta_1, L_1, Act_1)$ et $\mathrm{M}p_2 = (S_2, t_{init}, \Delta_2, L_2, Act_2)$ notée $\mathrm{M}p_1 \parallel \mathrm{M}p_2$ est un MDP $\mathrm{M}p = (S_1 \times S_2, (s_{init}, t_{init}), \Delta, L)$ avec les fonctions Δ et L définies ci-dessous.

En effet, $\forall s \in S_1$, $\forall t \in S_2$, $\forall \alpha \in Act$, nous avons les règles suivantes :

1. si $\exists \mu_1 \in \mathfrak{D}(S_1)$ tel que $(s, \alpha, \mu_1) \in \Delta_1$ et si $\nexists \mu_2 \in \mathfrak{D}(S_2)$ tel que $(t, \alpha, \mu_2) \in \Delta_2$, alors $((s,t), \alpha, \mu) \in \Delta$ avec $\mu = \mu_1$;
2. si $\nexists \mu_1 \in \mathfrak{D}(S_1)$ tel que $(s, \alpha, \mu_1) \in \Delta_1$ et si $\exists \mu_2 \in \mathfrak{D}(S_2)$ tel que $(t, \alpha, \mu_2) \in \Delta_2$, alors $((s,t), \alpha, \mu) \in \Delta$ avec $\mu = \mu_2$;
3. si $\exists \mu_1 \in \mathfrak{D}(S_1)$ tel que $(s, \alpha, \mu_1) \in \Delta_1$ et si $\exists \mu_2 \in \mathfrak{D}(S_2)$ tel que $(t, \alpha, \mu_2) \in \Delta_2$, alors $((s,t), \alpha, \mu) \in \Delta$ avec $\mu = \mu_1 \otimes \mu_2$;
4. $L((s,t)) = L_1(s) \cup L_2(t)$.

Les règles (1), (2) et (3) concernent la fonction de transitions tandis que la règle (4) concerne la fonction d'étiquetage du MDP Mp. Les règles (1) et (2) montrent que deux MDPs évoluent indépendamment sur des actions différentes et la règle (3) correspond à la synchronisation de deux processus de décision markoviens sur des actions identiques. Enfin, avec la règle (4), l'ensemble des propositions atomiques valides dans chaque état du MDP résultant correspond

à la réunion des ensembles des propositions valides dans les états partiels impliqués dans la composition de cet état.

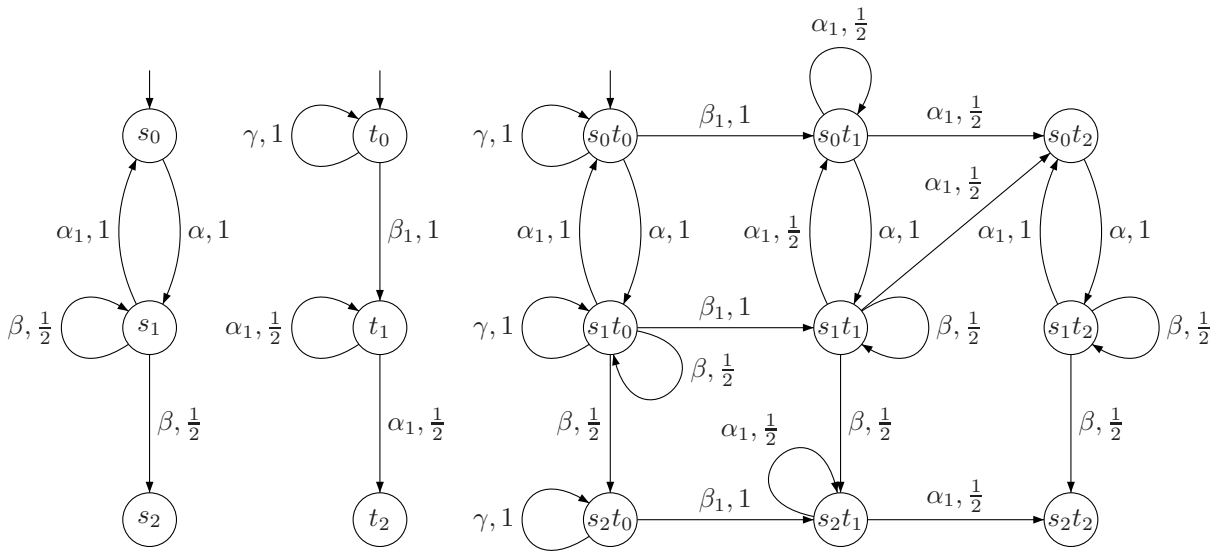


Figure 2.4 Deux MDPs en (a) et (b) et leur composition parallèle en (c)

La Figure 2.4 représente le résultat de la composition parallèle entre deux processus de décision markoviens Mp_1 et Mp_2. Toutes les transitions de la composition parallèle sont obtenues avec les règles (1) et (2) exceptée la transition étiquetée par α_1 sortant de l'état s_1 qui se synchronise avec la transition etiquetée par α_1 sortant de l'état t_1. La transition résultante est étiquétée par l'action α_1 et correspond à la distribution $\{\frac{1}{2},\ \frac{1}{2}\}$ qui est le produit des distributions des actions impliquées dans la synchronisation c.-à-d. $\{1\}$ pour α_1 de Mp_1 et $\{\frac{1}{2},\ \frac{1}{2}\}$ pour α_1 de Mp_2.

2.3 PRISM et formalismes de spécification

PRISM Kwiatkowska *et al.* (2002) supporte trois types de modèles probabilistes dont les chaînes de markov à temps discret (DTMCs) et les processus de décision markoviens (MDPs) présentés dans les sections précedentes ainsi que les chaînes de markov à temps continu (CTMCs) qui ne font pas l'objet de notre mémoire. Cet outil nécessite deux types d'entrées, la description du système à analyser et celle de l'ensemble des propriétés à vérifier relatives au système. Le comportement du système à analyser est spécifié en utilisant un langage inspiré

du langage des modules réactifs de Alur et Henzinger (1996). Les composants de base sont les *modules* et les *variables*. Un système peut être composé d'un nombre quelconque de modules pouvant intéragir entre-eux. Le système est alors construit sous-forme d'une composition parallèle entre les différents modules. Un module doit être spécifié comme suit :

$$\begin{array}{l} module < nom\ du\ module > \\ \ldots \\ endmodule \end{array}$$

Chaque module comporte un ensemble de variables dont les valeurs déterminent l'état du module à chaque instant. Les valeurs des variables de tous les modules déterminent ainsi l'état de tout le système à chaque instant. PRISM supporte deux types de variables, les variables *entières* et *booléennes*. L'exemple suivant montre comment est déclarée et initialisée une variable booléenne *pay* à *false* :

$$pay\ :\ bool\ init\ false\,;$$

Pour une variable entière, la déclaration indique la plage de valeurs que peut prendre la variable ainsi que sa valeur initiale. L'exemple ci-dessous

$$longueur\ :\ [\,0..10\,]\ init\ 0\,;$$

déclare une variable *longueur* pouvant prendre une des 11 valeurs que représente l'intervalle $[\,0..10\,]$ et initialisée à 0. PRISM permet également de déclarer des constantes qui peuvent être des entiers, des doubles ou des booléens avec la syntaxe suivante :

$$const\ <type>\ nom\_const\ =\ val\,;$$

où *type* désigne le type de la constante, *nom_const* le nom attribué à la constante et *val* la valeur qui lui est assignée. Chaque module comporte également un ensemble de *commandes gardées* qui définissent les différentes transitions du système. Chaque commande gardée correspond à la forme suivante :

$$[]\ <garde>\ ->\ <commande>\,;$$

où *garde* est un prédicat sur les variables du système et *commande* une transition effectuée par le système lorsque la garde est évaluée à la valeur *true*. Intuitivement, la commande correspond à une mise à jour des valeurs des variables du module. Les variables qui ne sont

pas affectées par la commande gardent les anciennes valeurs. Dans le cas où une transition doit être choisie probabilistiquement, la syntaxe est la suivante :

$$\begin{aligned} [] < garde > \; -> \; & < prob_1 > : < commande_1 > \; + \\ & \ldots + \\ & < prob_N > : < commande_N > ; \end{aligned}$$

où la transition relative à la $commande_i$ est effectuée suivant la probabilité $prob_i$, avec $\sum_{i=1}^{N} prob_i = 1$.

Les crochets peuvent contenir une chaîne de caractères indiquant l'action correspondant à la commande gardée. Ces actions jouent un rôle majeur dans le modèle de restriction des ordonnanceurs que nous présentons à la Section 3.1. Chaque action étiquette alors toutes les transitions engendrées par la commande gardée dans laquelle elle est présente. Durant la construction du système, PRISM assume la synchronisation style CSP de Hoare (1985) pour les commandes en provenance des différents modules et comportant les mêmes actions.

Les propriétés que doit satisfaire le système modélisé avec le langage PRISM sont spécifiées en logique PCTL comme décrit dans les Sections 2.2.1 et 2.2.2 suivant qu'il s'agit d'une DTMC ou d'un MDP. Les propriétés sur les CTMCs sont spécifiées avec la logique CSL de Aziz *et al.* (1996) que nous n'abordons pas dans ce mémoire. Les spécifications ci-dessous représentent deux exemples de propriétés exprimées en PCTL dans PRISM.

$$\begin{aligned} & P =? \, [true \, U \, Fail] \\ & Pmin =? \, [true \, U \, leader] \end{aligned}$$

La première correspond au système décrit comme une DTMC et permet de déterminer la probabilité avec laquelle un système atteint un état de disfonctionnement. Quant à la seconde, elle détermine la probabilité minimale avec la quelle un leader est élu sur un protocole distribué d'élection de leader décrit sous-forme d'un MDP. Nous utilisons les formalismes présentés dans cette section lors de l'analyse des études de cas de protocoles de sécurité que nous discutons dans le Chapitre 4.

2.4 Diagrammes de décision

La vérification automatique des propriétés sur de grands systèmes exige une représentation compacte et efficace des systèmes de transitions, des ensembles d'états et des formules exprimées en logique temporelle. Dans cette section nous présentons deux types de structures

de données symboliques appelées diagrammes de décision binaires (BDDs) et diagrammes de décision binaires multi-terminaux (MTBDDs). Ces structures sont utilisées par le vérificateur des systèmes probabilistes PRISM pour représenter de façon compacte en mémoire les systèmes décrits avec le langage PRISM. Tous nos algorithmes présentés dans la Section 3.3 opèrent essentiellement sur les BDDs et MTBDDs.

2.4.1 Diagramme de décision binaire

Les diagrammes de décision binaires originalement créés par Akers (1978) puis nettement améliorés par Bryant (1986) offrent un moyen efficace permettant de représenter des fonctions booléennes. Les fonctions booléennes peuvent être utilisées pour représenter un ensemble d'états ou une fonction de transitions relatifs à un système de transitions.

Soient Var un ensemble de variables $x_1, x_2, \ldots, x_n$ et Val un ensemble des évaluations possibles pour $x_1, x_2, \ldots, x_n$ dans l'ensemble $\{0, 1\}$. Une fonction booléenne sur les variables $x_1, x_2, \ldots, x_n \in Var$ peut être définie comme étant une fonction $f : Val \longrightarrow \{0, 1\}$. La représentation d'une fonction booléenne sous-forme d'un arbre de décision binaire découle de l'expansion de Shannon (1938).

Définition et représentation

L'expansion de Shannon décompose récursivement une fonction booléenne en termes de cofacteurs obtenus par des évaluations successives de la fonction f. Une fonction booléenne f sur des variables booléennes $x_1, x_2, \ldots, x_n$ possède deux cofacteurs par rapport à la variable x_i, $i = 1 \ldots n$, le cofacteur positif $f_{|x_i=1}$ et le cofacteur négatif $f_{|x_i=0}$.

Théorème 2.4.1 (Expansion de Shannon) *Si f est une fonction booléenne sur l'ensemble Var, alors $\forall x_i \in Var :\ f = (\neg x_i \wedge f_{|x_i=0}) \vee (x_i \wedge f_{|x_i=1})$.*

La preuve de ce théorème peut être trouvée dans Baier et Katoen (2008). En terme de diagramme de décision binaire, chaque noeud interne étiqueté par la variable booléenne x_i représente une fonction booléenne quelconque g et les deux branches les cofacteurs $g_{|x_i=0}$ et $g_{|x_i=1}$. La Figure 2.5 représente un *arbre de décision binaire* appelé aussi *arbre de Shannon* pour la fonction booléenne $f = x_1 \wedge (\neg x_2 \vee x_3)$ dont les valeurs possibles sont représentées dans le Tableau 2.2. La représentation suppose l'énumération des variables dans l'ordre x_1, x_2, x_3. La racine et les noeuds intermédiaires sont étiquetés par les variables x_i, $i = 1 \ldots 3$. Les noeuds terminaux représentent les valeurs possibles de la fonction (0 et 1). Pour chaque noeud non terminal correspondant à la variable x_i, deux branches représentent les deux évaluations possibles de la variable booléenne, la branche Pos pour $x_i = 1$ et la branche Neg pour $x_i = 0$.

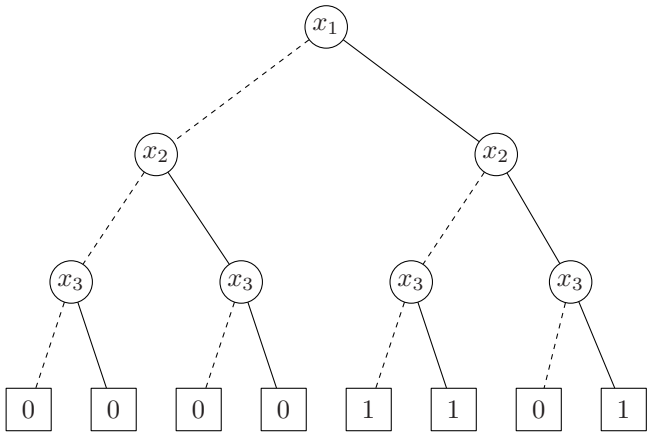


Figure 2.5 arbre de décision binaire pour la fonction booléenne $x_1 \wedge (\neg x_2 \vee x_3)$

La branche Pos est représentée en traits continus et celle de Neg en pointiés.

Formellement, nous pouvons définir un diagramme de décision binaire (BDD) comme un tuple $(V, V_I, V_T, var, val, v_0)$ où :

- V est l'ensemble fini de noeuds représentant le BDD ;
- V_I est le sous-ensemble de V contenant les noeuds non terminaux. Chaque noeud non terminal v possède un argument $indice(v) \in \{1, \ldots, n\}$ représentant l'indice de la variable booléenne et deux branches $Pos(v)$, $Neg(v) \in V$;
- V_T est le sous-ensemble de V contenant les noeuds terminaux. Chaque noeud terminal v possède un argument $value(v) \in \{0, 1\}$;
- var est une fonction de $V_I \longrightarrow Var$ assignant à chaque noeud non terminal v une variable $var(v) \in Var$;
- val est une fonction de $V_T \longrightarrow \{0, 1\}$ assignant à chaque noeud terminal v une valeur $val(v) \in \{0, 1\}$;
- $v_0 \in V_I$ est le noeud racine.

Chaque représentation d'une fonction booléenne sous-forme d'un BDD nécessite de définir un certain ordre "$<_{var}$" entre les variables $x_1, x_2, \ldots, x_n$ tel que $x_i <_{var} x_j$ ssi $indice(v_{x_i}) < indice(v_{x_j})$ avec v_{x_i} et v_{x_j} désignant les noeuds étiquetés respectivement par les variables booléennes x_i et x_j. Cet ordre induit la *canonicité* des BDDs Bryant (1986) après application des règles de réduction présentées dans la section suivante, c.-à-d. que chaque fonction booléenne

Tableau 2.2 Table de vérité pour la fonction booléenne $x_1 \wedge (\neg x_2 \vee x_3)$

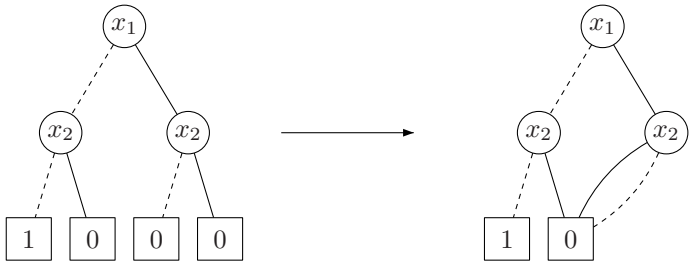
$\mathbf{x_1}$	$\mathbf{x_2}$	$\mathbf{x_3}$	$\mathbf{x_1 \wedge (\neg x_2 \vee x_3)}$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

possède une représentation unique sous-forme d'un BDD pour un ordre des variables donné. Les chemins partant de la racine aux noeuds terminaux représentent toutes les évaluations possibles que peut prendre une fonction représentée sous-forme d'un BDD. Chaque noeud du BDD exceptée la racine a un prédécesseur, ce qui assure que chaque noeud soit accessible depuis la racine.

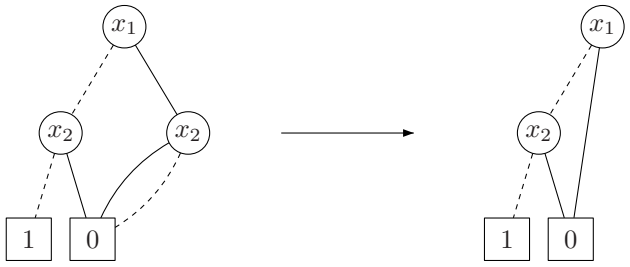
Reduction des BDD

L'objectif de représenter des fonctions booléennes sous-forme de BDDs est de tirer profit de la compacité que ces structures de données peuvent offrir. Bryant (1986) propose trois règles de réduction exploitant la *redondance* des noeuds dans l'arbre de décision binaire afin d'aboutir à une structure plus compacte. La Figure 2.6 représente en (a) la première règle éliminant les terminaux redondants, chaque BDD comportant au final deux noeuds terminaux en l'occurence les terminaux 0 et 1. La deuxième règle en (b) élimine les tests redondants. En effet si v est un noeud non terminal avec $Pos(v) = Neg(v) = w$ où $w \in V$, la règle enlève le noeud v et redirige toutes les branches entrant sur le noeud v directement sur le noeud w. Enfin la troisième règle en (c) est une règle éliminant la redondance de sous-BDDs. En effet si v, w sont des noeuds non terminaux de V avec $v \neq w$ tels que : $\begin{cases} var(v) = var(w) \\ Pos(v) = Pos(w) \\ Neg(v) = Neg(w) \end{cases}$ alors enlever le noeud v et rediriger toutes les branches entrant sur le noeud v directement sur le noeud w. La Figure 2.7 représente le BDD compacte obtenu après l'application des règles de réduction sur l'arbre de décision binaire de la Figure 2.5.

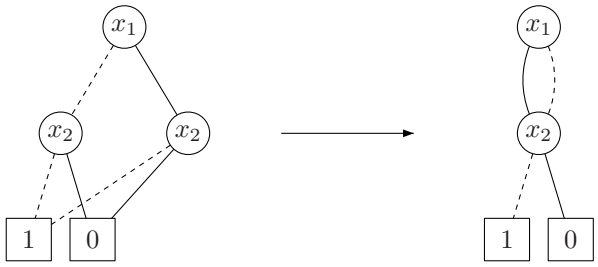
Le degré de compacité obtenu par application des règles de réduction dépend de l'ordre des variables choisi. Le problème de choisir un ordre des variables minimisant la taille du BDD est *NP-complet* Bolling et Wegener (1996). Pour contourner cette difficulté, des algorithmes



(a) Elimination des terminaux redondants



(b) Elimination des tests redondants



(c) Elimination de la redondance de sous − arbres isomorphiques

Figure 2.6 Règles de réduction d'un diagramme de décision binaire

heuristiques sont utilisés pour améliorer un ordre des variables donné. Une bonne compréhension du domaine étudié peut également permettre de choisir facilement un ordre des variables adéquat.

2.4.2 Diagramme de décision binaire multi-terminaux

Les diagrammes de décision binaires multi-terminaux Fujita *et al.* (1997); Bahar *et al.* (1997), sont une généralisation des diagrammes de décision binaires de Bryant (1986) non pas à des seuls terminaux 0 et 1 mais à des terminaux d'un ensemble quelconque fini. Ces structures représentent des fonctions de l'ensemble $\{0,1\}^n$ vers $\mathbb{D}$ où $\mathbb{D}$ constitue un do-

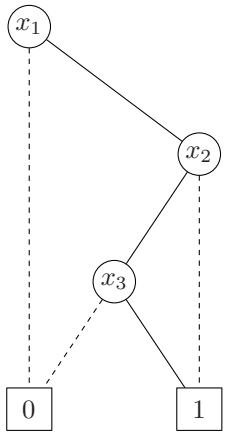
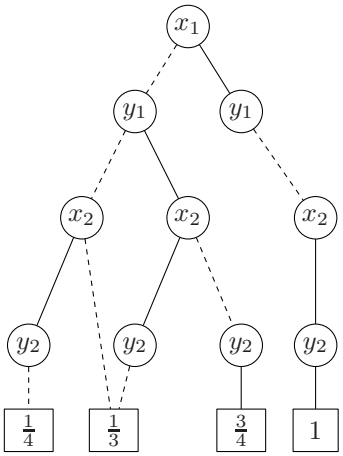


Figure 2.7 Diagramme de décision binaire réduit de $f = x_1 \wedge (\neg x_2 \vee x_3)$

maine quelconque de valeurs réelles. Les MTBDDs possèdent la même structure que les BDDs et leur représentation sous-forme de graphe acyclique découle également de l'expansion de Shannon. L'ordre "$<_{var}$" entre les variables booléennes ainsi que l'application des règles de réduction similaires à celles présentées à la Section 2.4.1 leur assure également la *canonicité*. L'idée principale derrière cette extension est d'avoir une représentation compacte des matrices et des vecteurs. Un vecteur V de taille n peut être considéré comme une fonction $f_{\mathrm{V}} : \{0, \ldots, n-1\} \longrightarrow \mathbb{D}$. Les valeurs $0, \ldots, n-1$ désignent les indices du vecteur et $\mathbb{D}$ le domaine des valeurs stockées dans le vecteur (entiers, réels, ou tout autre ensemble dérivé). Chaque indice du vecteur est représenté sur $m = \lceil \log_2(n) \rceil$ variables booléennes $x_1, x_2, \ldots, x_m$ dans le MTBDD en tenant compte de la représentation binaire des entiers. Une matrice M de taille $n \times n$ est représentée sous-forme d'une fonction $f_{\mathrm{M}} : \{0, \ldots, n-1\} \times \{0, \ldots, n-1\} \longrightarrow \mathbb{D}$. Un MTBDD correspondant à la fonction f_{M} est représenté avec les variables booléennes $x_1, x_2, \ldots, x_m$ encodant les indices de lignes et les variables booléennes $y_1, y_2, \ldots, y_m$ encodant les indices de colonnes. La Figure 2.8 montre un MTBDD représentant une fonction $f_{\mathrm{M}} : \{0, \ldots, 3\} \times \{0, \ldots, 3\} \longrightarrow [0, 1]$ relative à une matrice M. Les lignes sont encodées avec les variables booléennes x_1, x_2 et les colonnes avec les variables booléennes y_1, y_2. Pour des raisons de simplicité et de clarté, nous avons omis de représenter les entrées nulles de la matrice sur le MTBDD de la Figure 2.8. Ces entrées constituent le reste des combinaisons booléennes représentées par des traits "–" dans la table comportant les entrées de la matrice M. Les variables booléennes représentant les lignes et les colonnes sont entrelacées ce qui confère l'ordre $x_1 <_{var} y_1 <_{var} x_2 <_{var} y_2$ aux variables booléennes. Cette façon d'ordonner les variables sur une matrice assure une meilleure compacité du MTBDD Fujita *et al.* (1997). Chaque entrée de la matrice est obtenue en parcourant le MTBDD de la racine aux terminaux. Les indices



$\mathbf{x_1}$	$\mathbf{y_1}$	$\mathbf{x_2}$	$\mathbf{y_2}$	f_{M}
0	0	0	0	$\mathrm{M}[0,0] = 1/3$
0	0	0	1	$\mathrm{M}[0,1] = 1/3$
0	1	0	0	$\mathrm{M}[0,2] = 1/3$
0	0	1	0	$\mathrm{M}[1,0] = 1/4$
0	1	1	1	$\mathrm{M}[1,3] = 3/4$
1	0	1	1	$\mathrm{M}[3,1] = 1$
–	–	–	–	$\mathrm{M}[i,j] = 0$

Figure 2.8 MTBDD encodant une matrice représentée par la fonction f_{M}

lignes et colonnes s'obtiennent en reconstituant les valeurs entières correspondant aux valeurs des variables booléennes constituant le chemin. Pour notre exemple de la Figure 2.8, $f_{\mathrm{M}}[x_1 = 0, y_1 = 1, x_2 = 1, y_2 = 1] = \frac{3}{4}$ correspond à l'entrée $\mathrm{M}[1,3]$ de la matrice M.

PRISM Kwiatkowska *et al.* (2002) utilise les MTBDDs pour représenter les systèmes probabilistes en mémoire. La matrice des probabilités de transition pour les DTMCs est définie comme une fonction $f_\Delta : \{0,\ldots,2^n-1\} \times \{0,\ldots,2^n-1\} \longrightarrow [0,1]$ sur les variables booléennes $x_1,\ldots,x_n,y_1,\ldots,y_n$. Les variables $x_1,x_2,\ldots,x_n$ encodent les états de départ et $y_1,y_2,\ldots,y_n$ les états d'arrivée pour les transitions. Le nombre de variables booléennes nécessaires pour représenter un état dépend du nombre de variables PRISM dont les différentes évaluations possibles déterminent les états possibles du système. Les valeurs possibles de chaque variable PRISM déterminent le nombre de variables booléennes nécessaires pour son encodage. Chaque état du sytème est ainsi encodé sur la réunion des variables booléennes correspondant à toutes les variables PRISM utilisées dans le modèle. Quant aux probabilités de transition pour les MDPs, la représentation est un peu différente des DTMCs du fait de la présence du non déterminisme qu'il faut aussi encoder. Le nombre maximal de distributions de probabilités possibles dans chaque état détermine le nombre de variables booléennes nécessaires pour encoder le non déterminisme. En effet les probabilités de transitions sur le MDP sont représentées avec une fonction $f_\Delta : \{0,\ldots,2^n-1\} \times \{0,\ldots,2^{nd}-1\} \times \{0,\ldots,2^n-1\} \longrightarrow [0,1]$ sur les variables booléennes $x_1,\ldots,x_n,z_1,\ldots,z_{nd},y_1,\ldots,y_n$. Les variables $x_1,\ldots,x_n,y_1,\ldots,y_n$ encodent les états de départ et d'arrivée comme dans les DTMCs et les variables $z_1,z_2,\ldots,z_{nd}$ encodent le non déterminisme (c.-à-d. les différentes distributions de probabilités disponibles dans chaque état du MDP).

Comme à chaque distribution de probabilités correspond une action de l'ensemble *Act*, il faut trouver un moyen de représenter les actions étiquetant les transitions dans les modèles. Pour les DTMCs, la représentation est simple par le fait que dans chaque état une seule action est possible. A chaque action α de l'ensemble *Act* de la DTMC est associée une fonction $f_\alpha : \{0, \ldots, 2^n - 1\} \longrightarrow \{0, 1\}$ sur les variables booléennes $x_1, \ldots, x_n$. Intuitivement, la fonction f_α encode tous les états du modèle où il est possible d'effectuer l'action α. Pour les MDPs, la possibilité d'avoir plus d'une action possible est prise en compte. En effet, à chaque action β de l'ensemble *Act* du MDP est associée une fonction f_β encodant toutes les distributions de probabilités liées à l'action β dans tous les états du modèle où il est possible d'effectuer l'action β. La fonction $f_\beta : \{0, \ldots, 2^n - 1\} \times \{0, \ldots, 2^{nd} - 1\} \longrightarrow \{0, 1\}$ est alors définie sur les variables booléennes $x_1, \ldots, x_n$ encodant les états disposant de l'action β et sur les variables booléennes $z_1, \ldots, z_{nd}$ indiquant quelle distribution associée à β parmi toutes les distributions de probabilités disponibles dans l'état concerné.

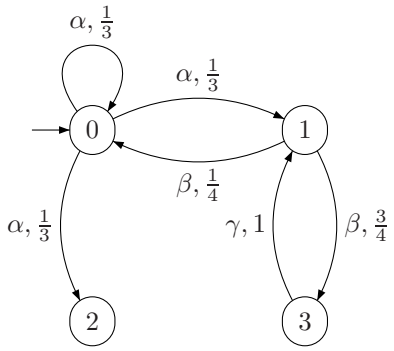
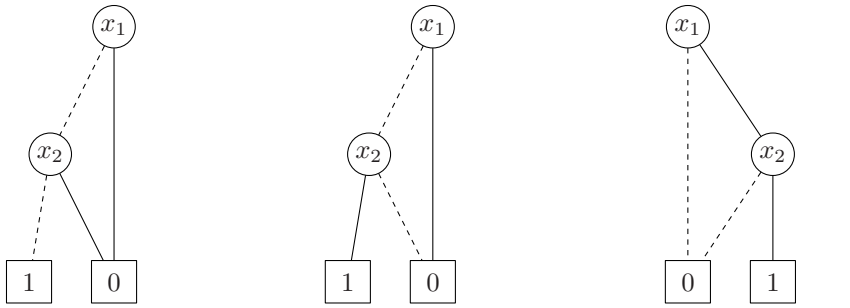


Figure 2.9 DTMC dont la matrice des transitions probabilistes Δ est représentée avec le MTBDD de la Figure 2.8

Dans la représentation de la fonction f_Δ, PRISM utilise $x_1 <_{var} y_1 <_{var} \cdots <_{var} x_n <_{var} y_n$ comme ordre des variables pour les DTMCs. En plus de la compacité qu'offre ce choix, il procure aussi de l'efficacité aux algorithmes de *model checking* qui se traduisent par des appels récursifs sur les sous-matrices relatives aux cofacteurs de f_Δ. Pour les MDPs, l'ordre des variables se fait ainsi $z_1 <_{var} \cdots <_{var} z_{nd} <_{var} x_1 <_{var} y_1 <_{var} \cdots <_{var} x_n <_{var} y_n$, en plaçant en tête d'abord les variables non déterministes suivies d'un entrelacement des variables lignes et colonnes. L'exemple de la Figure 2.8 correspond au MTBDD représentant la fonction f_Δ de la DTMC de la Figure 2.9. Le modèle comporte 4 états et chaque état est encodé sur 2 variables booléennes. Les variables x_1, x_2 encodent les états de départ et y_1, y_2 les états de destination des transitions. Toutes les transitions non nulles correspondent à tous les chemins allant de la racine du MTBDD aux terminaux non nuls. En effet le chemin



(a) BDD encodant f_α (b) BDD encodant f_β (c) BDD encodant f_γ

Figure 2.10 BDDs encodant les actions de la DTMC de la Figure 2.9

$[x_1 = 0, y_1 = 1, x_2 = 1, y_2 = 1]$ représente la transition de l'état 1 à l'état 3 avec probabilité $\frac{3}{4}$. Les actions sur les transitions de la DTMC de la Figure 2.9 sont représentées à l'aide des BDDs de la Figure 2.10. A chacune des trois actions α, β, γ correspond un BDD qui encode tous les états de la DTMC où l'action est disponible. L'action α est disponible uniquement à l'état 0 représenté en (a) avec le chemin $[x_1 = 0, x_2 = 0]$. Enfin, les actions β, γ sont disponibles respectivement aux états 1 et 3 représentés en (b) et (c) par les chemins $[x_1 = 0, x_2 = 1]$ et $[x_1 = 1, x_2 = 1]$.

2.4.3 Quelques opérations sur les diagrammes de décision

Dans cette section, nous présentons certaines opérations sur les MTBDDs et BDDs utilisées dans nos algorithmes. Ces opérations se répartissent en deux grandes catégories : les opérations de création des noeuds et les opérations induites par celles pouvant être effectuées sur leurs fonctions représentatives. Dans ce qui suit, nous assumons des MTBDDs (BDDs) M, M_1, M_2 définis sur une liste de variables booléennes $x_1, x_2, \ldots, x_n$. Toutes ces opérations sont implémentées dans le package CUDD de Somenzi (1997) ou dans le package DD de PRISM.

- **Const**(*valeur*) crée un nouveau MTBDD (BDD) constitué d'un seul noeud étiqueté par la valeur *valeur* $\in \mathbb{R}$. Pour les BDDs, la fonction *Const* ne peut prendre que de paramètres 0 et 1.
- **Var**(i) crée un nouveau noeud MTBDD (BDD) interne ayant l'indice i s'il n'existe pas ou renvoie la référence du noeud s'il existe déjà.
- **Compl**(M), où M est un BDD, représente le BDD correspondant au complément de

la fonction f_{M} représentative de M.

- **EtLog**(M_1, M_2), où M_1 et M_2 sont des BDDs, retourne le BDD correspondant à la fonction $f_{\mathrm{M}_1} \wedge f_{\mathrm{M}_2}$.

- **Abstraire**$(op, (x_1, x_2, \ldots, x_n), M)$, où M est un MTBDD (BDD) et op un opérateur binaire (logique ou arithmétique), retourne le MTBDD (BDD) résultant de l'application de l'opérateur op sur toutes les abstractions en considérant toutes les valeurs possibles que peuvent prendre les variables $x_1, \ldots, x_n$. En effet, par exemple Abstraire$(+, (x_1, x_2), M)$ correspond au MTBDD représentant la fonction $f_{\mathrm{M}|(x_1=0,x_2=0)} + f_{\mathrm{M}|(x_1=0,x_2=1)} + f_{\mathrm{M}|(x_1=1,x_2=0)} + f_{\mathrm{M}|(x_1=1,x_2=1)}$ et quant à Abstraire$(\vee, (x_1, x_2), M)$, le BDD résultant représente la fonction $f_{\mathrm{M}|(x_1=0,x_2=0)} \vee f_{\mathrm{M}|(x_1=0,x_2=1)} \vee f_{\mathrm{M}|(x_1=1,x_2=0)} \vee f_{\mathrm{M}|(x_1=1,x_2=1)}$.

- **Appliquer**(op, M_1, M_2), où M_1 et M_2 sont des MTBDDs et op est une opération binaire sur les réels ($+$, $-$, $\times$, $\div$, etc.), représente le MTBDD correspondant à la fonction $f_{\mathrm{M}_1}\, op\, f_{\mathrm{M}_2}$.

- **PlusGrandQue**$(M, valeur)$, où M est un MTBDD et $valeur$ une valeur double, retourne un BDD dont les terminaux valent 1 pour toutes les valeurs supérieures à $valeur$, le reste des terminaux étant égale à 0.

- **MultiMatrice**$(M_1, M_2, varsZ)$, où M_1 et M_2 sont des MTBDDs et $varsZ$ la liste des variables booléennes de sommation, calcule le produit de deux matrices représentées respectivement par les fonctions f_{M_1} et f_{M_2}.

- **InsElementMatrice**$(M, lVars, cVars, ligne, colonne, valeur)$, où M est un MTBDD représentant une matrice encodée sur les variables lignes $lVars$ et colonnes $cVars$, insère la valeur $valeur$ dans M à la position indicée par $ligne$ et $colonne$.

- **RecElementMatrice**$(M, lVars, cVars, ligne, colonne)$, où M est un MTBDD représentant une matrice encodée sur les variables lignes $lVars$ et colonnes $cVars$, retourne une valeur stockée dans la position indicée par $ligne$ et $colonne$.

CHAPITRE 3

MODÈLE DE RESTRICTION DE L'ADVERSAIRE ET IMPLÉMENTATION DANS PRISM

Ce chapitre est consacré à la définition du modèle de restriction de l'adversaire basé sur la réduction de la puissance des ordonnanceurs appliquée aux systèmes de sécurité modélisés sous-forme de *processus de décision markoviens*. Nous présentons également l'intégration du modèle dans l'architecture du *model checker* PRISM **version 3.2.r935** ainsi que la description des principaux algorithmes utilisés dans son implémentation.

3.1 Modèle de sécurité pour les MDPs

Comme nous l'avons montré dans la Section 2.2.2, l'évaluation des propriétés probabilistes dans un MDP passe par la résolution du non déterminisme. Ce non déterminisme est résolu par un ordonnanceur qui choisit une seule action parmi plusieurs actions possibles dans chaque état. Dans le cas où le MDP représente le système de transitions d'un protocole de sécurité, les choix de certaines transitions peuvent résulter des calculs internes du protocole.

D'une part, le formalisme du MDP ne fait pas la différence entre les choix internes et les choix externes (ne dépendant pas de calculs internes) et d'autre part, l'évaluation des protocoles de sécurité suppose l'exécution dans un environnement hostile où l'adversaire contrôle le réseau de communications et par conséquent contrôle tout l'ordonnancement du protocole. Cela a pour conséquence l'octroi du contrôle des choix internes du protocole à l'adversaire lors de l'analyse formelle, ce qui lui confère une puissance trop forte impossible à atteindre dans les conditions réelles d'utilisation du protocole.

Dans cette section nous présentons un modèle de restriction de la puissance des ordonnanceurs sur les MDPs inspiré des travaux de Hamadou et Mullins (2010) et de Canetti *et al.* (2006). Ce modèle décompose l'ordonnancement en deux niveaux d'ordonnanceurs : Les ordonnanceurs externes sous le contrôle de l'adversaire et l'ordonnanceur interne hors du contrôle de l'adversaire. Ce découpage en deux niveaux est guidé par la définition d'une relation d'équivalence observationnelle sur les actions du modèle.

3.1.1 Actions équivalentes

L'idée de la définition des actions observationnellement équivalentes vient de la nécessité de faire la différence entre les actions du MDP dont le choix dépend des calculs internes (c.-à-d. invisibles à l'environnement) et les autres actions pouvant être contrôlées par l'adversaire lors de l'analyse du protocole modélisé sous-forme d'un MDP.

Soient $\mathrm{M}p = (S, s_{init}, \Delta, L, Act)$ un MDP, Obs un ensemble d'observables et $\mathcal{O} : Act \longrightarrow Obs$ une fonction d'observation. Nous définissons une relation $\Re$ telle que $\forall \alpha, \beta \in Act : (\alpha, \beta) \in \Re$ ssi $\mathcal{O}(\alpha) = \mathcal{O}(\beta)$.

Nous pouvons facilement montrer que $\Re$ est une relation d'équivalence sur l'ensemble Act appelée *relation d'équivalence observationnelle*. Nous dénotons par $Obs(s)$ l'ensemble des observables disponibles dans l'état s du modèle.

Un modèle de sécurité est ainsi considéré comme une paire $\mathcal{S} = (\mathrm{M}p, \Re)$ où $\mathrm{M}p = (S, s_{init}, \Delta, L, Act)$ est un MDP et $\Re$ est une relation d'équivalence observationnelle définie sur Act.

Une fois les actions équivalentes établies, les ordonnanceurs externes (sous le contrôle de l'adversaire) choisissent au niveau de l'état courant un sous-ensemble d'actions T appelé *tâche* et regroupant les actions observationnellement équivalentes. L'ensemble des tâches obtenues sur le MDP $\mathrm{M}p$ suivant la relation d'équivalence observationnelle $\Re$ est noté $Tasks_{[\mathrm{M}p/\Re]}$.

Considérons les deux exemples ci-dessous de relations d'équivalence observationnelle définies sur un protocole modélisé sous-forme d'un MDP :

1. $\Re_{tot}$ dénotant une relation d'équivalence sur Act telle que $\forall \alpha, \beta \in Act, \alpha \, \Re_{tot} \, \beta$.
2. $\Re_{id}$ dénotant une relation d'équivalence identité sur Act telle que chaque classe d'équivalence est réduite à une seule action parmi les actions du modèle.

Dans la première situation, $\Re_{tot}$ confère une puissance très faible à l'adversaire puisque pour l'ordonnanceur externe, il n'y a qu'une seule tâche possible dans chaque état du fait que toutes les actions sont observationnellement équivalentes. Quant à la seconde situation, la relation $\Re_{id}$ fournit à l'attaquant une puissance trop forte par le fait que toutes les actions du protocole sont distinguables et par conséquent, il y a autant de tâches qu'il y a d'actions au niveau de chaque état.

Les deux exemples de situations montrent bien que la relation d'équivalence observationnelle sur les actions du modèle est un moyen permettant de contrôler la puissance des ordonnan-

ceurs externes et par conséquent la puissance attribuée à l'attaquant lors de la phase de vérification du modèle. Dans la section suivante, nous allons voir comment est résolu le non déterminisme restant dans une *tâche* une fois qu'elle est choisie par l'ordonnanceur externe.

3.1.2 Ordonnanceur interne

Le résultat de l'ordonnancement effectué par un ordonnanceur externe ζ dans chaque état est un observable correspondant à une tâche $T \in Tasks_{[\mathrm{M}p/\Re]}$. Et comme une tâche peut comporter plusieurs actions, chacune correspondant à une distribution de probabilités, il devient alors indispensable de résoudre le non déterminisme entre les différentes actions de la tâche afin d'évaluer les propriétés probabilistes. Le non déterminisme à l'intérieur d'une tâche est résolu par un ordonnanceur statique hors-contrôle de l'adversaire appelé *ordonnanceur interne*.

Nous définissons un ordonnanceur interne comme une fonction

$$\begin{aligned} \xi : Tasks_{[\mathrm{M}p/\Re]} &\longrightarrow \mathfrak{D}(Act) \\ T &\longmapsto \mu_T \end{aligned}$$

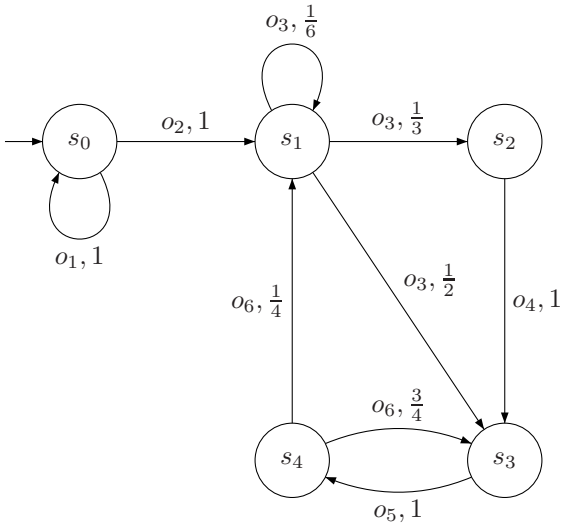
telle que $\forall T \in Tasks_{[\mathrm{M}p/\Re]}$ et $\forall \alpha \in T$, $\mu_T(\alpha) = \frac{1}{|T|}$ où $|T|$ désigne le nombre d'actions disponibles dans la tâche T.

Cette fonction retourne, pour une tâche donnée, une distribution de probabilité uniforme sur les actions disponibles dans la tâche. L'ordonnanceur interne engendre ainsi sur l'observable correspondant à la tâche une nouvelle distribution résultant de la combinaison linéaire convexe des distributions de probabilités associées aux différentes actions constituant la tâche. Pour des raisons d'implémentation dans PRISM, nous choisissons de résoudre à l'avance tout le non déterminisme interne dans les tâches.

Nous définissons ainsi le nouveau MDP issu de la résolution du non déterminisme interne dans les tâches comme un tuple $\mathrm{M}p_\xi = (S, s_{init}, \Delta', L, Obs)$ tel que :

- S, s_{init} et L sont les mêmes que dans le MDP $\mathrm{M}p$;
- Obs est l'ensemble des observables définies sur $\mathrm{M}p$;
- $\Delta' : S \times Obs \longrightarrow \mathfrak{D}(S) \cup \{\surd\}$ est une fonction de transitions telle que $\forall o \in Obs, \forall s \in S$:
  $\Delta'(s,o) = \begin{cases} \surd & \text{si } o \notin Obs(s) \\ \sum_{a \in Act(s) \cap o} \xi(Act(s) \cap o)(a) \cdot \Delta(s,a) & \text{sinon} \end{cases}$

En effet, l'exécution du MDP Mp_ξ sous un ordonnanceur externe ζ de l'ensemble *Scheds* est complètement probabiliste et rend donc possible l'évaluation des propriétés probabilistes. Considérons le MDP Mp et sa fonction Δ représentés par la Figure 2.2 de la Section 2.2.2.



$$\Delta' = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ \hline 0 & \frac{1}{6} & \frac{1}{3} & \frac{1}{2} & 0 \\ \hline 0 & 0 & 0 & 1 & 0 \\ \hline 0 & 0 & 0 & 0 & 1 \\ \hline 0 & \frac{1}{4} & 0 & \frac{3}{4} & 0 \end{pmatrix} \begin{matrix} o_1 \\ o_2 \\ o_3 \\ o_4 \\ o_5 \\ o_6 \end{matrix}$$

Figure 3.1 MDP de la Figure 2.2 tel que observé par l'environnement ne distinguant pas β de β_1

Nous définissons ensuite une relation d'équivalence observationnelle $\Re$ sur l'ensemble $Act = \{\alpha,\ \alpha_1,\ \beta,\ \beta_1,\ \gamma,\ \gamma_1, \lambda,\ \lambda_1\}$ telle que $\beta\ \Re\ \beta_1$. La Figure 3.1 représente le nouveau MDP Mp_ξ et sa fonction Δ' (telle que vue par l'environnement externe) résultant du regroupement des actions $\{\beta,\ \beta_1\}$ en une tâche vue par l'ordonnanceur externe comme l'observable o_3 (le reste des actions forment chacune une tâche correspondant à un observable distinct). Le non déterminisme existant entre les actions β et β_1 est ensuite résolu en répartissant une distribution uniforme sur les distributions $\{\frac{1}{3},\ \frac{2}{3}\}$ et $\{1\}$ associées respectivement aux actions β et β_1. La nouvelle distribution sur o_3 combine donc les deux distributions et correspond à $\{\frac{1}{6},\ \frac{1}{3},\ \frac{1}{2}\}$. Les distributions associées au reste des observables sont les mêmes que celles des actions correspondantes car les tâches correspondant à ces observables forment chacune une seule action. L'évaluation des propriétés probabilistes sur le nouveau MDP Mp_ξ issu de la restriction des ordonnanceurs se fait en utilisant les mêmes techniques vues dans le Chapitre 2.

3.2 Intégration du modèle dans l'architecture de PRISM

Pour cibler le meilleur endroit à intégrer notre modèle de restriction des ordonnanceurs dans l'architecture du *model checker* PRISM, il est indispensable de connaître sa structure interne contenant en totalité près de 30 packages et utilisant 3 langages différents (C/C++, Java et PEPA de Hillston (1996)).

La Figure 3.2 représente l'architecture de PRISM intégrant notre modèle de restriction des

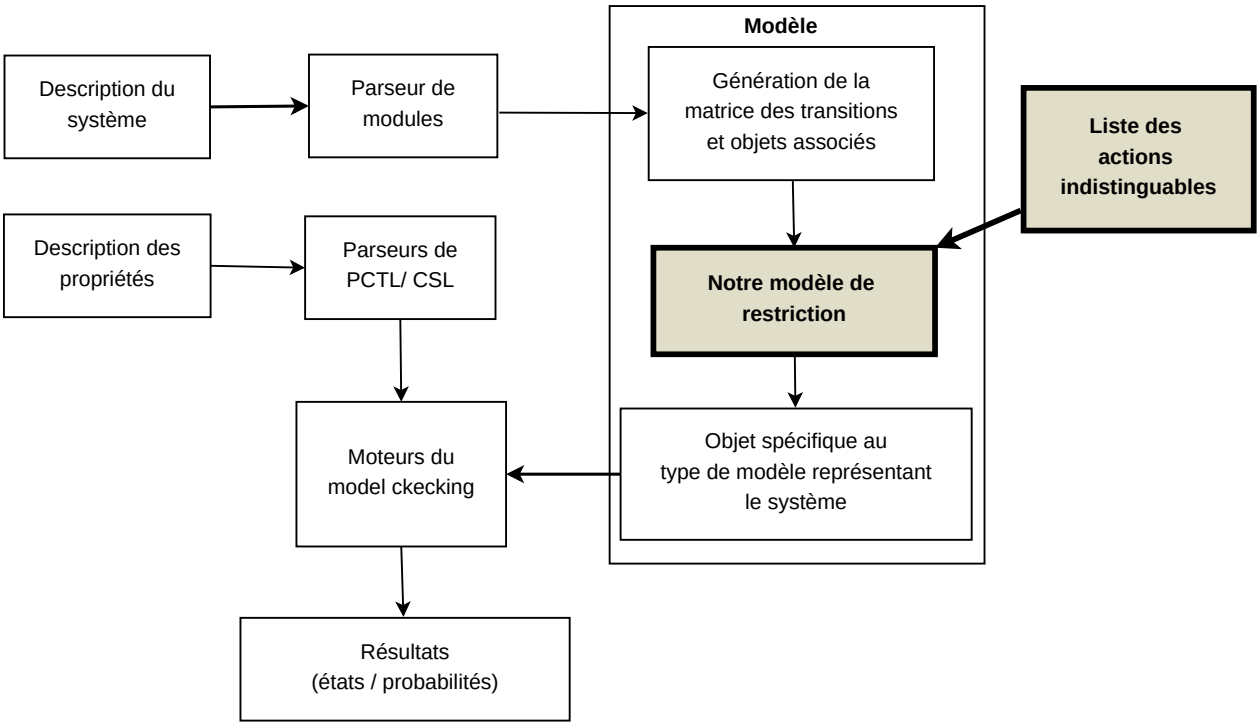


Figure 3.2 L'architecture de PRISM intégrant le modèle de restriction des ordonnanceurs

ordonnanceurs. Originalement (sans le modèle de restriction), lorsque les utilisateurs lui fournissent les descriptions du système en utilisant les formalismes présentés dans la Section 2.3 du Chapitre 2, la première étape de PRISM consiste à parser ces fichiers et à générer un ensemble de structures internes contenant la liste des variables, des valeurs, des noms de modules, des constantes, des commandes, etc.. Ces structures sont pour la plupart des tableaux dynamiques. La seconde étape concerne la transformation de ces structures explicites en structures symboliques suivie de leur encapsulation en objet spécifique au type de modèle représentant le système (non déterministe pour MDP, probabiliste pour DTMC ou stochastique pour CTMC). PRISM utilise les MTBDDs pour la représentation du système en mémoire mais deux autres approches peuvent être utilisées pour optimiser les opérations de *model*

checking : l'approche matrices parsemées dans but d'accroître les performances en terme de rapidité de calcul lorsque l'espace des états accessibles est modeste et l'approche hybride combinant les MTBDDs et les matrices parsemées pour tirer le meilleur des deux afin d'optimiser la performance pour les systèmes larges. Une fois le modèle construit en mémoire, les propriétés spécifiées en logique PCTL ou CSL sont transmises après le *parsing* aux algorithmes de *model checking* qui retournent les résultats de vérification en termes d'états ou des probabilités. Du fait que l'approche de base exploitant les MTBDDs utilise le package CUDD de Fabio Somenzi (1997) écrit en C, PRISM s'appuie sur le JNI (*Java Native Interface*) afin de mapper toutes les fonctions dans une classe Java appelée JDD. Cela a pour avantage de pouvoir appeler les fonctions de manipulation des MTBDDs à partir de n'importe quel package de PRISM écrit en Java.

Comme le montre la Figure 3.2, nous avons choisi d'intégrer notre modèle de restriction des ordonnanceurs après la génération de la matrice des probabilités de transition et des objets associés sous-forme symbolique et avant leur encapsulation dans un objet spécifique au type de modèle et prêt à être exploité par les méthodes de *model checking*. Ce choix a été motivé par deux raisons principales. La première assure que les modifications en rapport avec le modèle de restriction des ordonnanceurs s'opèrent directement sur les MTBDDs représentant la matrice des probabilités de transition, ce qui nous permet d'analyser notre système avec les mêmes méthodes de *model checking* implémentées dans PRISM sans qu'elles soient modifiées. La seconde raison de cette approche est la volonté d'intégrer cette fonctionnalité à PRISM de façon optionnelle. Autrement dit, il s'agit d'une fonctionnalité activable et désactivable depuis les interfaces utilisateurs. Une fois la fonctionnalité désactivée, PRISM garde son comportement original. Si elle est activée, PRISM transforme les MTBDDs représentant la fonction de transitions en se basant sur des actions indistinguables listées dans un fichier qu'il doit préalablement charger.

3.3 Implémentation de la restriction des ordonnanceurs

Dans cette section, nous présentons en détail différentes étapes effectuées par notre modèle de restriction intégré dans PRISM afin de résoudre certains non déterminismes que nous voulons ôter du contrôle de l'adversaire lors de l'évaluation des protocoles de sécurité. Ces non déterminismes, comme nous le discutons dans la Section 3.1, résultent du choix des actions internes aux participants ou composants du protocole et devant rester invisibles à l'adversaire. Nous définissons ces actions dans un fichier qui doit être chargé par PRISM lors du processus de restriction des ordonnanceurs. Chaque ligne du fichier indique les actions appartenant à la même classe d'équivalence. Le non déterminisme étant susceptible d'exister

dans n'importe quel état du système de transitions, un parcours efficace de tout les états accessibles est nécessaire pour examiner s'il y a lieu de résoudre le non déterminisme entre les transitions étiquetées par les actions de la même classe d'équivalence.

3.3.1 Parcours des états accessibles du système

L'espace des états est exprimé en termes des évaluations possibles des variables PRISM utilisées pour décrire le modèle. Nous implémentons l'algorithme de parcours des états accessibles du système sous-forme d'un itérateur. A chaque itération, l'algorithme traverse le BDD représentant l'ensemble d'états accessibles jusqu'à attendre un terminal non nul. Une fois le terminal atteint, l'algorithme renvoie la liste des valeurs des variables PRISM décrivant l'état correspondant au chemin parcouru depuis la racine du BDD. Les structures ci-dessous comportent des informations essentielles aux calculs des états accessibles par l'algorithme de parcours :

- **taille** : Le nombre d'états accessibles du système.
- **numVars** : Le nombre total des variables booléennes utilisées pour représenter le BDD de l'ensemble des états accessibles.
- **varValeurs** : Un vecteur d'entiers dont la taille correspond au nombre de variables PRISM utilisées pour décrire le comportement du système. Il sert à stocker les valeurs des variables PRISM correspondant à l'état courant. Ce vecteur est initialisé à 0.
- **varTaille** : Un vecteur d'entiers dont la taille correspond aussi au nombre de variables PRISM utilisées. Ce vecteur stocke le nombre de bits nécessaires pour représenter toutes les valeurs possibles de chaque variable PRISM.
- **niveau** : Un entier initialisé à 0 indiquant le niveau du noeud du BDD en cours de parcours. Le niveau 0 indique le noeud racine.
- **varCourant** : Un entier initialisé à 0 indiquant la variable PRISM en cours de calcul. La valeur 0 indique la première variable de la liste des variables PRISM utilisées dans la description du système.
- **niveauVarCourant** : Un entier initialisé à 0 indiquant le niveau de la variable booléenne dans la liste des variables booléennes encodant la variable PRISM *varCourant*. La valeur 0 indique la première des variable booléennes.
- **compteur** : Un entier initialisé à 0 et servant à compter le nombre d'états déjà parcourus.

- **visite** : Un vecteur de booléens dont la taille correspond au nombre total de variables booléennes utilisées dans le BDD représentant l'ensemble des états accessibles. Il sert à indiquer si un noeud a déjà été visité ou non. Ce vecteur est initialisé à *false*.

```
parcours(dd)

1.  Begin
2.  bool continuer, bool descente = true ;
3.  while (compteur < taille) begin
4.          do begin
5.                  if (descente)begin
6.                          if (dd=Const(0)) begin
7.                                  descente = false ;
8.                                  continuer = true ;
9.                          endif ;
10.                         else begin
11.                         if (niveau=numVars) begin
12.                                 continuer = false ;
13.                                 descente=false ;
14.                                 compteur++ ;
15.                                 listeValeurs() ;
16.                         endif ;
17.                         else begin
18.                                 continuer=true ;
19.                                 descendre() ;
20.                         endelse ;
21.                         endelse ;
22.                 endif ;
23.                 else begin
24.                         remonter() ;
25.                         continuer = true ;
26.                 endelse ;
27.         enddo ; while (continuer) ;
28. endwhile ;
29. End.
```

Figure 3.3 Algorithme principal de parcours du BDD représentant l'ensemble des états accessibles

La Figure 3.3 décrit l'algorithme principal de parcours du BDD nous permettant de faire ressortir tous les états accessibles. Le parcours démarre depuis la racine du BDD. L'arrêt de l'algorithme est conditionné par l'atteinte du nombre d'états accessibles déterminé par la valeur *taille*. A Chaque noeud du BDD, la procédure *parcours* opère différement selon qu'il s'agit d'un terminal nul, d'un terminal non nul ou d'un noeud intermédiaire non terminal. Le terminal nul exprime le cas où la combinaison des variables booléennes depuis la racine jusqu'au terminal ne représente pas un état accessible. Dans cette situation la procédure remonte dans le graphe pour explorer d'autre chemins. Si le terminal est non nul, la combinaison des variables booléennes depuis la racine jusqu'à ce terminal constitue un état accessible du système. Les valeurs des variables PRISM représentant cet état sont alors retournées par la fonction *listeValeurs*. Cette fonction ne fait que lire les valeurs actuelles dans le vecteur *varValeurs*. Le calcul de ces valeurs se fait progressivement pendant le parcours par les procédures *descendre* et *remonter*. La procédure *parcours* remonte ensuite dans le graphe à la recherche d'autres états éventuels. Le dernier cas constitue un noeud non terminal. Dans cette situation, le parcours continue avec l'un des noeuds fils du noeud en cours. Le choix du noeud avec lequel il faut poursuivre le parcours se fait avec la procédure *descendre* comme le montre l'algorithme de la Figure 3.4.

La procédure *descendre* commence par examiner s'il y aurait pas de noeud éliminé durant le processus de réduction du BDD afin de déterminer correctement les noeuds correspondant aux fils gauche (cofacteur négatif) et fils droit (cofacteur positif). Puis elle détermine avec quel noeud continuer le parcours. Si le noeud du niveau courant a déjà été visité, le parcours continue avec la branche *Pos* du noeud courant *dd*. Sinon une mise à jour du niveau de la variable PRISM *varCourant* est éventuellement effectuée ainsi que le niveau de la variable booléenne *niveauVarCourant* parmi celles encodant cette variable PRISM, puis l'exploration continue avec la branche *Neg* du noeud courant *dd*.

La Figure 3.5 montre la procédure de remontée dans le graphe chaque fois que le parcours du BDD nécessite de faire l'ascension vers les noeuds parents du noeud en cours (une fois qu'un terminal est atteint ou lorsque la remontée mène vers un parent marqué comme déjà visité). Chaque appel de la procédure *remonter* déplace le pointeur du noeud en cours d'exploration vers le noeud parent. Puis la procédure examine si le noeud atteint a déjà été visité. S'il l'a déjà été, la procédure met à jour le niveau de la variable booléenne en cours *niveauVarCourant* et éventuellement le niveau de la variable PRISM *varCourant*. Avant de repasser au noeud parent, la procédure *remonter* met à jour la valeur de la variable PRISM correspondant au niveau *varCourant* dans le vecteur *varValeurs*. Si le noeud n'a pas déjà été visité, la procédure met à jour le niveau de la variable booléenne en cours et éventuellement le niveau

```
descendre()

1.  Begin
2.  BDD e,t ;
3.  if (indice(dd) > indice(Var(niveau)) e=t=dd ;
4.  else begin
5.              e=Neg(dd) ;
6.              t=Pos(dd) ;
7.       endelse ;
8.  if ( !visite[niveau]) begin
9.                              niveauVarCourant++ ;
10.                             if (niveauVarCourant = varTaille[varCourant]) begin
11.                                                      varCourant++ ;
12.                                                      niveauVarCourant=0 ;
13.                                                  endif ;
14.                             niveau++ ;
15.                             visite[niveau] = false ;
16.                             dd = e ;
17.                     endif ;
18.  else begin
19.             niveau++ ;
20.             dd = t ;
21.       endelse ;
22.  End.
```

Figure 3.4 Procédure de descente dans les noeuds du BDD représentant l'ensemble des états accessibles

de la variable PRISM comme dans le cas du noeud déjà visité puis met à jour la valeur de la variable PRISM en cours dans le vecteur *varValeurs*. Une fois ces mises à jour effectuées, le contrôle est repassé à la procédure *descendre* à la recherche des chemins non encore explorés menant vers un noeud terminal.

L'algorithme de parcours est essentiel pour la construction des *directives de fusion* des transitions non déterministes qui indiquent, pour chaque état accessible du système, les transitions à fusionner afin de résoudre le non déterminisme existant entre elles. Dans la section suivante, nous décrivons le processus de construction de ces directives de fusion qui sont représentées de façon compacte dans un MTBDD.

```
remonter()

1.  Begin
2.  niveau−− ;
3.  dd=Parent(dd) ;
4.  if ( !visite[niveau] ) begin
5.                              descente=true ; niveauVarCourant−− ;
6.                              visite[niveau]=true ;
7.                              if (niveauVarCourant= −1) begin
8.                              varCourant−− ;
9.                              niveauVarCourant=varTaille[varCourant]−1 ;
10.                             endif ;
11. varValeurs[varCourant]+=(1 ≪ a (varTaille[varCourant]−1−niveauVarCourant)) ;
12. niveauVarCourant++ ;
13. if (niveauVarCourant = varTaille[varCourant]) begin
14.                                     varCourant++ ;
15.                                     niveauVarCourant=0 ;
16.                                     endif ;
17.                             endif ;
18. else begin
19.         descente=false ;
20.         niveauVarCourant−− ;
21.         if (niveauVarCourant = −1) begin
22.                                      varCourant−− ;
23.                                      niveauVarCourant = varTaille[varCourant]−1 ;
24.                                      endif ;
25. varValeurs[varCourant]−=(1 ≪ (varTaille[varCourant]−1−niveauVarCourant)) ;
26.         endelse ;
27. End.
```

a. Opération de décalage de bits à gauche

Figure 3.5 Procédure de remontée dans les noeuds du BDD représentant l'ensemble des états accessibles

3.3.2 Construction des directives de fusion des transitions

Les *directives de fusion* des transitions non déterministes constituent un moyen d'indiquer à l'algorithme de fusion quelles sont les différentes distributions de probabilités à fusionner pour chaque état accessible du système. L'espace d'états accessibles étant potentiellement

grand, nous représentons les informations de fusion dans MTBDD afin d'avoir une structure compacte. La première étape consiste à examiner pour chaque état les actions étiquetant chaque classe de transitions (distribution de probabilités).

Détermination et encodage des actions pour les classes de transitions

PRISM supporte deux types d'actions, les actions *indéterminées* matérialisées par les crochets vides au niveau des commandes du modèle décrit avec le langage PRISM ainsi que les actions explicites représentées par des chaînes de caractères dans les crochets des commandes du modèle. Les actions explicites sont appelées aussi des actions de *synchronisation*. Les transitions étiquetées par les actions indéterminées sont encodées dans un BDD appelé *transInd*. Quant aux actions de synchronisation, un vecteur de BDDs *transSynch* est utilisé pour encoder les transitions relatives à ces actions. Chaque BDD du vecteur *transSynch* encode toutes les transitions étiquetées par l'une des actions de synchronisation du modèle. La taille du vecteur *transSynch* dépend alors du nombre d'actions distinctes du modèle. Nous nous intéressons uniquement au cas des MDPs supportant le non déterminisme. Les BDDs *transInd* et *transSynch*$[i]$, où i est l'indice variant de zéro à la taille de *transSynch* moins 1, sont alors représentés avec les variables booléennes $x_1, x_2, \ldots, x_n$ encodant les états du système ainsi que les variables booléennes $z_1, z_2, \ldots, z_{nd}$ encodant les classes de transitions. Les variables booléennes sont ordonnées ainsi $z_1 <_{var} \cdots <_{var} z_{nd} <_{var} x_1 <_{var} \cdots <_{var} x_n$, les variables non déterministes encodant les distributions de probabilités en tête suivies des variables lignes encodant les états. Ces structures nous permettent de pouvoir déterminer pour chaque état du système l'action étiquetant chaque classe de transitions représentant une distribution de probabilités.

Algorithme de construction des directives de fusion

L'algorithme de génération des directives de fusion des transitions non déterministes encode des informations dans un MTBDD que l'algorithme de fusion exploite pour savoir dans chaque état quelles sont les distributions de probabilités à fusionner durant la phase de modification du modèle. Nous déterminons le nombre de variables booléennes encodant ce MTBDD en fonction du nombre total d'états accessibles et du nombre maximal de classes non déterministes qui peuvent exister dans un état. Si *tailleSt* et *tailleCl* désignent respectivement le nombre d'états accessibles et le nombre maximal de classes de transitions que peut contenir un état, le nombre de variables booléennes encodant les lignes est $n = \lceil \log_2(tailleSt) \rceil$ et celui des variables booléennes encodant les colonnes est $m = \lceil \log_2(tailleCl) \rceil$. Nous créons le vecteur *lVars* contenant les variables booléennes $x'_1, x'_2, \ldots, x'_n$ encodant les indices lignes

ainsi que le vecteur *cVars* contenant les variables booléennes $y'_1, y'_2, \ldots, y'_m$ encodant les indices colonnes de la matrice de fusion. Comme généralement $tailleSt \ggg {}^1tailleCl$, nous choisissons de ne pas entrelacer les variables lignes et colonnes pour avoir une grande compacité du MTBDD. Les variables booléennes du MTBDD sont ainsi ordonnées $x'_1 <_{var} x'_2 <_{var} \cdots <_{var} x'_n <_{var} y'_1 <_{var} y'_2 <_{var} \cdots <_{var} y'_m$.

***genererFus*(mdp, transInd, transSynch, actions)**

1. **Begin**
2. Array_String etiquette ;
3. Array_Int tmp ;
4. MTBDD res = $Const(0)$;
5. BDD trans01 = $PlusGrandQue(\text{mdp}, 0)$;
6. BDD listEtats = $Abstraire(\vee, \text{colVars}, Abstraire(\vee, \text{trans01}, \text{ndetVars}))$;
7. BDD listClasses = $Abstraire(\vee, \text{linVars}, Abstraire(\vee, \text{trans01}, \text{colVars}))$;
8. $parcours(\text{listEtats})$;
9. **For** $(i = 0 \ldots nbEtats - 1)$ **Begin**
10. Array_Int etat = $listeValeurs()$;
11. $parcours(\text{listClasses})$;
12. **For** $(j = 0 \ldots nbClasses - 1)$ **Begin**
13. Array_Int classe = $listeValeurs()$;
14. etiquette(j) = $rechercher(\text{transInd, transSynch, etat, classe})$;
15. **endFor** ;
16. tmp = $fusionTab(\text{etiquette, actions})$;
17. **For** $(k = 0 \ldots nbClasses - 1)$ **Begin**
18. res = $InsElementMatrice(\text{res, lVars, cVars, } i, k, \text{tmp}(k))$;
19. **endFor** ;
20. **endFor** ;
21. return res ;
22. **End**.

Figure 3.6 Algorithme de génération du MTBDD encodant les directives de fusion des transitions non déterministes

La Figure 3.6 représente le *pseudo-code* de l'algorithme de construction des informations de fusion. L'algorithme reçoit en paramètres le MTBDD *mdp* représentant la matrice des probabilités de transition f_Δ, le BDD *transInd* encodant les transitions étiquetées par l'action indéterminée, le vecteur *transSynch* de BDDs encodant les transitions étiquetées par les actions

1. Largement supérieur

de synchronisation ainsi que le fichier *actions* contenant les listes des actions équivalentes. L'algorithme commence par initialiser le MTBDD résultat puis détermine le BDD encodant les états accessibles ainsi le BDD encodant la liste des classes de transitions. La détermination du BDD représentant les états accessibles se fait par abstraction des variables colonnes $colVars = \{y_1, y_2, \ldots, y_n\}$ et des variables non déterministes $ndetVars = \{z_1, z_2, \ldots, z_{nd}\}$ sur la version BDD du MTBDD encodant la fonction f_Δ. La liste des classes de transitions se calcule par abstraction des variables lignes $linVars = \{x_1, x_2, \ldots, x_n\}$ et des variables colonnes $colVars = \{y_1, y_2, \ldots, y_n\}$.

L'étape suivante consiste à déterminer et à encoder dans le MTBDD résultat, des informations de fusion pour chaque état accessible du système. Chaque état est déterminé par l'algorithme de parcours qui, à chaque itération correspondant à l'atteinte d'un terminal non nul du BDD *listEtats* représentant les états accessibles, retourne les valeurs des variables PRISM représentant l'état. A chaque état, le parcours du BDD *listClasses* encodant les classes de transitions est effectué afin de retrouver la valeur entière représentant chaque classe de transitions. Nous utilisons l'algorithme *parcours* même si chaque liste retournée est constituée d'un seul élément pour tirer profit de l'efficacité de cet algorithme. Chaque valeur représentant une classe de transitions est alors combinée avec les valeurs des variables PRISM de l'état correspondant afin de déterminer par la fonction *rechercher* l'action étiquetant la classe de transitions. La fonction *rechercher* opère de façon simple. Elle détermine dans quel BDD parmi *transInd* et les BDDs du vecteur *transSynch* est encodée la combinaison des valeurs des variables PRISM et de la valeur représentant la classe de transitions. Si le parcours relatif à cette combinaison depuis la racine du BDD jusqu'à un terminal aboutit à un, l'action correspondante est insérée dans le vecteur *etiquette* à l'indice correspondante. Une classe de transitions n'existe pas si la combinaison aboutit au terminal zéro dans tous les BDDs. Ceci s'explique par le fait que nous considérons pour chaque état, le nombre maximal de classes de transitions pouvant ne pas être atteint. Dans cette situation, il est important de marquer cette absence de classe de transitions avec une information spéciale. Dans notre algorithme, nous marquons une telle absence par le caractère spécial "$*$". Une fois déterminé toutes les actions correspondant aux classes de transitions de l'état, nous construisons un vecteur de fusion *tmp* pour l'état qui va ensuite être encodé dans le MTBDD résultat *res* en insérant ses éléments avec l'opération *InsElementMatrice*. Ce vecteur est construit avec la fonction *fusionTab* à partir du vecteur contenant les actions étiquetant les classes de transitions comme le montre le schéma de la Figure 3.7.

En effet, le vecteur V représente les actions étiquetant six classes de transitions pour un état e quelconque d'un exemple de système de transitions représenté sous-forme d'un MDP. En

$$\mathbf{Cl_1} = \{a, c\}$$
$$\mathbf{Cl_2} = \{b, d\}$$

$$\mathbf{V} = \begin{pmatrix} a \\ b \\ * \\ c \\ * \\ d \end{pmatrix} \longrightarrow \mathbf{V_F} = \begin{pmatrix} 0 \\ 1 \\ -1 \\ 0 \\ -1 \\ 1 \end{pmatrix}$$

Figure 3.7 Construction d'un vecteur de fusion pour un état.

réalité le MDP contient quatre distributions de probabilités à l'état e. Deux des six classes de transitions représentées dans le vecteur V sont fictives et sont désignées par "$*$". Les ensembles $Cl_1 = \{a, c\}$ et $Cl_2 = \{b, d\}$ représentent des classes d'équivalence sur l'ensemble $Act = \{a, b, c, d\}$ des actions du modèle et sont définies dans un fichier. Le vecteur V_F est obtenu en appliquant la fonction *fusionTab* sur le vecteur V. Cette fonction assigne le même numéro aux classes de transitions étiquetées par des actions issues de la même classe d'équivalence. Ainsi les classes de transitions correspondant aux actions a et c vont être fusionnées en une seule classe par l'algorithme de fusion que nous présentons dans la prochaine section, de même que pour les classes de transitions relatives aux actions b et d. Les classes de transitions correspondant aux distributions de probabilités devant être fusionnées doivent contenir le même numéro indiquant la valeur encodant la future nouvelle classe de transitions après fusion. Les valeurs "-1" de V_F indiquent les classes de transitions fictives qui sont marquées par le caractère "$*$" dans le vecteur V.

Une fois terminé l'encodage des vecteurs de fusion pour tous les états accessibles du système, l'algorithme de fusion peut commencer à combiner les classes de transitions relatives aux actions équivalentes conformément aux indications du MTBDD encodant les directives de fusion pour les états.

3.3.3 Processus de fusion des classes non déterministes

Dans cette section, nous décrivons en détail le processus de fusion de certaines transitions non déterministes en tenant compte des indications encodées dans le MTBDD représentant les directives de fusion. Le non déterminisme interne réside entre les classes de transitions correspondant, pour un état donné, à une même valeur (différent de "-1") dans le MTBDD représentant les directives de fusion. Nous résolvons un tel non déterminisme entre les classes avec une distribution uniforme. La Figure 3.8 représente l'algorithme principal de modification du mo-

dèle en résolvant le non déterminisme entre les classes de transitions équivalentes (étiquetées par des actions de la même classe d'équivalence). L'algorithme prend en paramètre le MTBDD *mdp* représentant la fonction f_Δ, les listes des variables booléennes $linVars = \{x_1, x_2, \ldots, x_n\}$ encodant les états de départ, $colVars = \{y_1, y_2, \ldots, y_n\}$ encodant les états de destination des transitions, $ndetVars = \{z_1, z_2, \ldots, z_{nd}\}$ encodant le non déterminisme ainsi que le MTBDD *dirFus* représentant les directives de fusion des transitions non déterministes.

L'algorithme commence par désassembler *mdp* suivant les classes de transitions. La procédure de décomposition est inspirée de l'algorithme existant dans PRISM utilisé dans la transformation des MTBDDs en matrices parsemées. Le principe récursif de la procédure est décrit à la Figure 3.9. Chaque appel de la procédure *decomposRec* termine si le MTBDD passé en paramètre correspond à la fonction constante nulle ou si le niveau *niveau* de la variable booléenne racine du MTBDD correspond à la variable booléenne ligne x_1 (*nbNdetVars* indique le nombre de variables booléennes non déterministes du MTBDD représentant la fonction f_Δ). Si c'est le deuxième cas, le MTBDD dont la racine correspond au niveau *niveau* est inséré dans le vecteur *arr*. Dans la mesure où aucun des deux cas triviaux n'est vérifié, deux appels récursifs sont respectivement effectués sur les cofacteurs négatif et positif de la racine du MTBDD de la procédure appelante. Cet algorithme de décomposition a pour but de dissocier les transitions entre états suivant les classes de transitions dans l'optique de pouvoir effectuer facilement des fusions de certaines classes de transitions. Nous créons ensuite le vecteur de MTBDDs *nouvArr* dont la taille correspond au nouveau nombre maximal de classes de transitions *nbNouvCl* que peut avoir un état une fois fusionné toutes les classes de transitions équivalentes. La valeur de *nbNouvCl* peut être facilement déterminée en ajoutant un à la valeur maximale stockée dans le MTBDD $dirFus$. Le vecteur *nouvArr* sert à reconstruire un nouveau MTBDD représentant la nouvelle fonction f'_Δ issue de la fusion des classes de transitions équivalentes. Afin de commencer à construire chaque MTBDD de *nouvArr*, nous déterminons le MTBDD représentant la liste des états accessibles $listEtats$ de *mdp* en effectuant une suite d'abstractions sur certains ensembles de variables booléennes comme dans l'algorithme présenté à la Figure 3.6.

L'étape suivante consiste à construire pour chaque état atteignable du système, les nouvelles distributions de probabilités issues de la résolution du non déterminisme entre certaines classes de transitions. Pour chaque état atteignable, nous effectuons la création du MTBDD *selecL* afin de permettre l'extraction des distributions de probabilités des transitions de l'état à partir des MTBDDs issus de la décomposition de *mdp* et rangés dans le vecteur *arr*. Le MTBDD *selecL* est construit avec l'opération $InsTabElementMatrice$ qui diffère un peu de l'opération $InsElementMatrice$ de la Section 2.4.3. Au lieu de considérer les indices de la

```
fusTrans(mdp, linVars, colVars, ndetVars, dirFus)

1.  Begin
2.  Array_MTBDD arr, nouvArr ;
3.  MTBDD tmp, selecL, res ;
4.  decomposRec (mdp, ndetVars, 0, arr, 0) ;
5.  For (i = 0 ... nbNouvCl − 1) nouvArr(i)= Const(0) ;
6.  BDD trans01 = PlusGrandQue(mdp, 0) ;
7.  BDD listEtats = Abstraire(∨, colVars, Abstraire(∨, trans01, ndetVars)) ;
8.  parcours(listEtats) ;
9.  For (i = 0 ... nbEtats − 1) Begin
10.     Tb_Int etat = listeValeurs() ;
11.     selecL = InsTabElementMatrice(const(0), linVars, varsZ, etat, etat, 1) ;
12.     For (j = 0 ... nbClasses − 1) Begin
13.         Int val = RecElementMatrice(dirFus, lVars, cVars, i, j) ;
14.         if(val ≠ −1) Begin
15.         tmp = MultiMatrice(selecL, arr(j), varsZ) ;
16.         nouvArr(val) = Appliquer(+, tmp, nouvArr(val)) ;
17.         endif ;
18.     endFor ;
19. endFor ;
20. For (i = 0 ... nbNouvCl − 1) Begin
21.     tmp = Abstraire(+, nouvArr(i), colVars) ;
22.     nouvArr(i) = Appliquer(÷, nouvArr(i),tmp) ;
23. endFor ;
24. res = Const(0) ;
25. For (i = 0 ... nbNouvCl − 1) Begin
26.     tmp = genererMTBDD(i, nouvNdetVars) ;
27.     tmp = Appliquer(×, tmp, nouvArr(i)) ;
28.     res = Appliquer(+, res, tmp) ;
29. endFor ;
30. return res ;
31. End.
```

Figure 3.8 Algorithme de fusion des transitions conformément aux directives de fusion des transitions non déterministes

matrice comme des valeurs entières uniques, chaque indice est considéré comme une suite de valeurs entières relatives aux variables PRISM utilisées pour décrire le système. L'extraction et l'encapsulation des valeurs de probabilités des transitions équivalentes en une nouvelle classe de transitions se font en examinant les valeurs représentant les nouvelles classes de

```
decomposRec (mdp, ndetVars, niveau, arr, cpt)

1.  Begin
2.  MTBDD e,t ;
3.  if(mdp = Const(0)) return ;
4.  if(niveau = nbNdetVars) Begin
5.                              arr(cpt) = mdp ;
6.                              cpt = cpt+1 ;
7.                              return ;
8.                              endif ;
9.  if(indice(mdp) > indice(Var(niveau)) Begin
10.                                       e = mdp ;
11.                                       t = mdp ;
12.                                       endif ;
13. else Begin
14.       e = Neg(mdp) ;
15.       t = Pos(mdp) ;
16.       endelse ;
17. decomposRec (e, ndetVars, niveau+1, arr, cpt) ;
18. decomposRec (t, ndetVars, niveau+1, arr, cpt) ;
19. End.
```

Figure 3.9 Décomposition récursive du MTBDD de f_Δ suivant les variables non déterministes

transitions extraites du MTBDD $dirFus$ avec la fonction $RecElementMatrice$ en fonction des numéros d'état et d'anciennes classes de transitions. Le MTBDD encodant les valeurs probabilistes de la transition dans l'état est extrait par multiplication matricielle du MTBDD $selecL$ et $arr[j]$ encodant toutes les transitions relatives à la classe j. Le MTBDD extrait est alors ajouté au vecteur $nouvArr$ stockant les transitions de nouvelles classes à la position indicée par la valeur extraite dans $dirFus$. Rapelons aussi que si la valeur extraite dans $dirFus$ est -1, cela indique qu'il s'agit d'une transition fictive dont il ne faut pas tenir compte. Les variables booléennes $varsZ = \{x''_1, x''_2, \ldots, x''_n\}$ sont utilisées comme variables de sommation dans la multiplication matricielle. Une fois terminé le processus d'extraction et de fusion des transitions équivalentes pour tous les états accessibles du système, nous effectuons la normalisation des MTBDDs de $nouvArr$ avant de les réassembler pour construire le nouveau MTBDD représentant la nouvelle fonction f'_Δ. Le processus de normalisation a pour objectif de répartir la probabilité $\frac{1}{k}$ aux distributions de probabilités relatives aux k classes de transitions équivalentes dans chaque état. Après la normalisation, le processus de

réassemblage peut se mettre en oeuvre une fois déterminé la nouvelle liste de variables booléennes $nouvNdetVars = \{z'_1, z'_2, \ldots, z'_{ndt}\}$, où $ndt = \lceil \log_2(nbNouvCl) \rceil$, encodant le non déterminisme dans la nouvelle fonction f'_Δ. Chaque valeur de nouvelle classe de transitions est encodée dans un MTBDD *tmp* sur les *ndt* variables booléennes en utilisant la procédure détaillée à la Figure 3.10. Cette procédure prend en paramètres une valeur entière *val* et un

***genererMTBDD* (val, vars)**

1. **Begin**
2. MTBDD tmp, res ;
3. res = $Const(1)$;
4. **For** ($i = 0 \ldots nbVars - 1$) **Begin**
5. tmp = $Var(i)$;
6. **if** ((val &[a] (1≪[b]($nbvars - i - 1$))) = 0) tmp = $EtLog$(res, $Compl$(tmp)) ;
7. **else** tmp = $EtLog$(res, tmp) ;
8. **endFor** ;
9. return res ;
10. **End**.

a. Opération de ET logique bit à bit
b. Opération de décalage de bits à gauche

Figure 3.10 Génération d'un MTBDD encodant une valeur entière sur un ensemble de variables booléennes

vecteur *vars* de *nbVars* variables booléennes. Elle entreprend ensuite un ensemble d'opérations *EtLog* effectuées successivement sur les noeuds MTBDDs créés avec l'opération *Var* ou leurs compléments, chacun correspondant à un niveau de variable booléenne. L'opération *EtLog* s'effectue sur le complément du noeud MTBDD créé si la valeur de la variable booléenne correspondant au noeud vaut 0. Chaque MTBDD encodant le non déterminisme relatif à la nouvelle classe est ensuite combiné au MTBDD de *nouvArr* encodant les nouvelles distributions de probabilités de la classe correspondante puis ajouté au MTBDD représentant la nouvelle fonction f'_Δ.

Le modèle nouvellement créé représente le système de transitions du protocole ôté du non déterminisme entre actions de la même tâche et devant rester indistinguables par l'adversaire qui a le contrôle sur l'environnement dans lequel évolue le protocole. Les propriétés probabilistes peuvent dès lors être spécifiées en logique PCTL et évaluées sur le modèle.

CHAPITRE 4

ÉTUDES DE CAS ET RÉSULTATS

Dans ce chapitre, nous passons à l'évaluation du modèle de restriction des ordonnanceurs implémenté dans PRISM sur des études de cas de protocoles de sécurité. Nous présentons une analyse de deux protocoles de sécurité probabilistes dont le *protocole de dîner des cryptographes* de Chaum (1988) et le *protocole de signature de contrat* de Rabin (1983). Pour chaque protocole, nous effectuons une étude comparative en considérant un adversaire relativement à deux situations : le cas où toutes les actions du protocole sont distinguables par l'environnement externe (adversaire trop fort et irréaliste) et le cas où les actions de choix internes aux participants du protocole sont indistinguables par l'environnement externe (adversaire admissible). Les résultats que nous présentons dans ce chapitre proviennent des analyses réalisées sur un ordinateur **Intel Core 2 Duo à 1,73GHz avec 2 Go de RAM**.

4.1 Protocole de dîner des cryptographes (DCP)

Le problème de dîner des cryptographes est un problème introduit par David Chaum (1988) dont l'idée principale est d'assurer des communications anonymes entre plusieurs parties. Chaum pose le problème comme suit :

> *Trois cryptographes sont assis autour d'une table circulaire dans leur restaurant favori pour dîner. Leur hôte les informe qu'un arrangement a été fait avec le maître d'hôtel pour que la facture du dîner soit réglée de façon anonyme. Il se pourrait que ce soit leur employeur, l'agence de sécurité nationale (NSA), qui paye le dîner ou l'un trois cryptographes. Les trois cryptographes respectent le droit de chacun d'entre-eux de payer de façon anonyme mais veulent savoir si la NSA leur paye le dîner ou pas.*

Pour résoudre ce problème des cryptographes, Chaum propose le protocole suivant comme solution : trois pièces de monnaie sont utilisées dont chacune est disposée entre deux cryptographes de manière à ce qu'elle ne soit uniquement visible que par les deux qui lui sont adjacents. Chaque cryptographe jette une pièce se trouvant entre lui et son voisin de droite. Ensuite, chacun annonce à haute voix si les deux pièces qu'il voit (à sa gauche et à sa droite) sont tombées en exposant la même face ou non. Un cryptographe non payeur doit annoncer

exactement ce qu'il voit ("*égales*" si les pièces exposent la même face ou "*différentes*" sinon) tandis que un cryptographe payeur (s'il y en a) annonce le contraire de ce qu'il voit. L'idée derrière ce protocole de Chaum est que si les pièces ne sont pas biaisées et que les cryptographes suivent correctement le protocole, alors un nombre pair d'annonces de "*différentes*" indique que c'est la NSA qui paye tandis que le nombre impair indique que l'un des cryptographes est payeur sans que les deux autres apprennent quoi que ce soit sur l'identité du payeur.

4.1.1 Modélisation du DCP avec le langage PRISM

Le modèle du protocole de dîner des cryptographes est décrit sous-forme d'un processus de décision markovien (MDP) car à priori n'importe quel cryptographe ou la *NSA* peut être le payeur. Nous établissons premièrement les différentes constantes dont `nombre` indiquant le nombre de cryptographes dans le modèle, les constantes `crypto`i, $i \in \{1, 2, 3\}$ indiquant les identités des cryptographes dans le modèle et la constante `nsa` indiquant l'identité de la *NSA*. En effet, les instructions déclarant ces constantes sont les suivantes :

$$\texttt{const int nombre} = 3\,;$$
$$\texttt{const int crypto1} = 1\,;$$
$$\texttt{const int crypto2} = 2\,;$$
$$\texttt{const int crypto3} = 3\,;$$
$$\texttt{const int nsa} = 0\,;$$

Une variable globale `payeur` indiquant l'identité du payeur est déclarée comme suit :

$$\texttt{global payeur} : [0\,..\,\texttt{nombre}]\,;$$

La valeur 0 de la variable globale `payeur` spécifie la *NSA* comme payeur du dîner tandis que la valeur i, $i \in \{1, 2, 3\}$ signale que le payeur est le cryptographe dont l'identité est `crypto`i.

Le comportement de chaque cryptographe est décrit avec un module PRISM. Comme les actions possibles sont similaires pour tous les cryptographes, nous présentons la spécification du module `cryptographe`i, $i \in \{1, 2, 3\}$ que doit correspondre chaque processus du système représentant un cryptographe du protocole ayant l'identité `crypto`i. Les variables locales au module `cryptographe`i sont déclarées comme suit :

$$\texttt{piece}i : [0\,..\,2]\,;$$
$$\texttt{egale}i : [0\,..\,1]\,;$$
$$\texttt{ordre}i : [0\,..\,3]\,;$$

$\texttt{statut}i : [0\,..\,1]\,;$

La variable `piece`i sert à exprimer le résultat du jet de pièce effectué par le cryptographe `crypto`i. La variable `egale`i indique le résultat de l'annonce de `crypto`i (0 s'il annonce que les pièces affichent les faces différentes et 1 s'il annonce que les pièces exhibent les faces identiques). La variable `ordre`i exprime la position d'annoncement du résultat. Quant à la variable `statut`i, elle indique si le cryptographe `crypto`i a effectué ou non toutes les actions qu'il doit accomplir durant l'évolution du protocole.

Une fois définies les variables du module, nous passons à la description des actions de chaque cryptographe dans le module `cryptographe`i. Chaque cryptographe jette une pièce de monnaie, ceci est spécifié par la commande PRISM suivante :

$[\texttt{x}i]\ \texttt{piece}i{=}0 \dashrightarrow 1/2 : (\texttt{piece}i'{=}1) + 1/2 : (\texttt{piece}i'{=}2)\,;$

La valeur 0 indique que la pièce n'est pas encore jetée. Une fois la pièce lancée, les valeurs 1 et 2 modélisent les deux sorties possibles de la face de la pièce. La probabilité uniforme entre les deux indique le fait qu'il s'agit d'une pièce équitable (non biaisée). Nous modélisons ensuite les différentes actions entreprises par chaque cryptographe dépendamment de l'affichage des deux pièces qui lui sont visibles et de son statut de payeur ou non payeur.
Quatre situations s'imposent :

1**.** Les pièces vues par le cryptographe `crypto`i affichent les mêmes faces et il se trouve que `crypto`i ne soit pas le payeur. Dans ce cas il choisit une position parmi celles disponibles dans l'ordre d'annoncement afin de publier le résultat "*égales*". Les trois commandes ci-dessous résument cette situation :

$[\texttt{a1}i]\ \texttt{statut}i = 0\ \&\ \texttt{piece}i > 0\ \&\ \texttt{piece}(i+1)\texttt{mod}\ 3{>}0\ \&\ \texttt{piece}i{=}\texttt{piece}(i+1)\texttt{mod}\ 3\ \&\ (\texttt{payeur}\,!{=}\texttt{crypto}i)\ \&\ (\texttt{ordre}(i+1)\texttt{mod}\ 3\,!{=}1)\ \&\ (\texttt{ordre}(i+2)\texttt{mod}\ 3\,!{=}1) \dashrightarrow (\texttt{statut}i'{=}1)\ \&\ (\texttt{egale}i'{=}1)\ \&\ (\texttt{ordre}i'{=}1)\,;$

$[\texttt{a2}i]\ \texttt{statut}i = 0\ \&\ \texttt{piece}i > 0\ \&\ \texttt{piece}(i+1)\texttt{mod}\ 3{>}0\ \&\ \texttt{piece}i{=}\texttt{piece}(i+1)\texttt{mod}\ 3\ \&\ (\texttt{payeur}\,!{=}\texttt{crypto}i)\ \&\ (\texttt{ordre}(i+1)\texttt{mod}\ 3\,!{=}2)\ \&\ (\texttt{ordre}(i+2)\texttt{mod}\ 3\,!{=}2) \dashrightarrow (\texttt{statut}i'{=}1)\ \&\ (\texttt{egale}i'{=}1)\ \&\ (\texttt{ordre}i'{=}2)\,;$

$[\texttt{a3}i]\ \texttt{statut}i = 0\ \&\ \texttt{piece}i > 0\ \&\ \texttt{piece}(i+1)\texttt{mod}\ 3{>}0\ \&\ \texttt{piece}i{=}\texttt{piece}(i+1)\texttt{mod}\ 3\ \&\ (\texttt{payeur}\,!{=}\texttt{crypto}i)\ \&\ (\texttt{ordre}(i+1)\texttt{mod}\ 3\,!{=}3)\ \&\ (\texttt{ordre}(i+2)\texttt{mod}\ 3\,!{=}3) \dashrightarrow (\texttt{statut}i'{=}1)\ \&\ (\texttt{egale}i'{=}1)\ \&\ (\texttt{ordre}i'{=}3)\,;$

2**.** Les pièces vues par le cryptographe `crypto`i affichent les faces différentes et il se trouve que `crypto`i ne soit pas le payeur. Dans ce cas, il choisit une position dans l'ordre d'annonce-

ment afin de publier le résultat "*différentes*". Les trois commandes ci-dessous résument cette deuxième situation :

[b1i] statut$i$ = 0 & piece$i$ > 0 & piece$(i+1)$mod 3>0 & piece$i$!=piece$(i+1)$mod 3 & (payeur!=crypto$i$) & (ordre$(i+1)$mod 3!=1) & (ordre$(i+2)$mod 3!=1) –> (statut$i$'=1) & (ordre$i$'=1) ;

[b2i] statut$i$ = 0 & piece$i$ > 0 & piece$(i+1)$mod 3>0 & piece$i$!=piece$(i+1)$mod 3 & (payeur!=crypto$i$) & (ordre$(i+1)$mod 3!=2) & (ordre$(i+2)$mod 3!=2) –> (statut$i$'=1) & (ordre$i$'=2) ;

[b3i] statut$i$ = 0 & piece$i$ > 0 & piece$(i+1)$mod 3>0 & piece$i$!=piece$(i+1)$mod 3 & (payeur!=crypto$i$) & (ordre$(i+1)$mod 3!=3) & (ordre$(i+2)$mod 3!=3) –> (statut$i$'=1) & (ordre$i$'=3) ;

3**.** Les pièces vues par le cryptographe crypto$i$ affichent les faces identiques et crypto$i$ est le payeur. Dans ce cas il annonce le résultat "*différentes*" dans une position choisie dans l'ordre d'annoncement. Les trois commandes ci-dessous résument cette situation :

[c1i] statut$i$ = 0 & piece$i$ > 0 & piece$(i+1)$mod 3>0 & piece$i$=piece$(i+1)$mod 3 & (payeur=crypto$i$)&(ordre$(i+1)$mod 3!=1) & (ordre$(i+2)$mod 3!=1) –> (statut$i$'=1) & (ordre$i$'=1) ;

[c2i] statut$i$ = 0 & piece$i$ > 0 & piece$(i+1)$mod 3>0 & piece$i$=piece$(i+1)$mod 3 & (payeur!=crypto$i$) & (ordre$(i+1)$mod 3!=2) & (ordre$(i+2)$mod 3!=2) –> (statut$i$'=1) & (ordre$i$'=2) ;

[c3i] statut$i$ = 0 & piece$i$ > 0 & piece$(i+1)$mod 3>0 & piece$i$=piece$(i+1)$mod 3 & (payeur=crypto$i$) & (ordre$(i+1)$mod 3!=3) & (ordre$(i+2)$mod 3!=3) –> (statut$i$'=1) & (ordre$i$'=3) ;

4**.** Enfin, cette dernière situation résume le cas où le cryptographe crypto$i$ est le payeur et où les pièces vues par crypto$i$ affichent les faces différentes. Dans ce cas, il annonce le résultat "*égales*". Cette situation est décrite par les commandes ci-dessous :

[d1i] statut$i$ = 0 & piece$i$ > 0 & piece$(i+1)$mod 3>0 & piece$i$!=piece$(i+1)$mod 3 & (payeur=crypto$i$) & (ordre$(i+1)$mod 3!=1) & (ordre$(i+2)$mod 3!=1) –> (statut$i$'=1) & (egale$i$'=1) & (ordre$i$'=1) ;

[d2i] statut$i$ = 0 & piece$i$ > 0 & piece$(i+1)$mod 3>0 & piece$i$!=piece$(i+1)$mod 3 & (payeur=crypto$i$) & (ordre$(i+1)$mod 3!=2) & (ordre$(i+2)$mod 3!=2) –> (statut$i$'=1) & (egale$i$'=1) & (ordre$i$'=2) ;

[d3i] statut$i$ = 0 & piece$i$ > 0 & piece$(i+1)$mod 3>0 & piece$i$!=piece$(i+1)$mod 3 & (payeur=crypto$i$) & (ordre$(i+1)$mod 3 !=3) & (ordre$(i+2)$mod 3 !=3) –> (statut$i$'=1) & (egale$i$'=1) & (ordre$i$'=3) ;

Dans la deuxième et troisième situation, la variable PRISM egale$i$ indiquant le résultat annoncé n'est pas mis à jour, elle garde la même valeur qui est initialisée à zéro car le cryptographe annonce "*différentes*". La dernière commande du module cryptographe$i$ est présentée ci-dessous :

[fin] statut$i$=1 –> true ;

Cette commande spécifie la terminaison des actions de chaque cryptographe. Elle permet également d'ajouter des boucles avec probabilité 1 sur les états terminaux afin d'éviter les états accessibles sans transitions dans le système de transitions construit après application de l'opérateur de composition parallèle entre les trois modules décrivant les actions des trois cryptographes du protocole. Les trois modules se synchronisent alors sur cette commande et c'est pour cette raison que la chaîne de caractères dans les crochets est la même pour tous les modules cryptographe$i$, $i \in \{1, 2, 3\}$. Par contre, les autres commandes comportent une chaîne de caractères distincte entre les crochets.

Dans tout système de transitions il est indispensable de définir l'état initial du système. Pour notre protocole, l'état initial symbolise le moment où aucun cryptographe n'a encore entrepris aucune action parmi toutes les actions décrites dans les différents modules. En langage PRISM l'instruction ci-dessous initialise notre système et est placée n'importe où dans le modèle à l'extérieur de la spécification des modules cryptographe$i$, $i \in \{1, 2, 3\}$:

init piece1=0 & statut1=0 & egale1=0 & piece2=0 & statut2=0 & egale2=0 & piece3=0 & statut3=0 & egale3=0 & ordre1=0 & ordre2=0 & ordre3=0 endinit

Le modèle en langage PRISM étant spécifié, nous passons à l'analyse de la sécurité du protocole sur le système de transitions construit par le vérificateur PRISM à partir de cette description.

4.1.2 Modèle de sécurité et analyse du DCP avec PRISM

Modèle de l'adversaire

L'analyse formelle d'un système de transitions représentant un protocole de sécurité s'effectue en supposant un environnement hostile où l'attaquant contrôle tout le réseau de communications Dolev et Yao (1983). Le contrôle du réseau de communications par l'adversaire

lui donne la capacité de contrôler l'ordonnanceur qui résout le non déterminisme durant l'évolution du système. Notre modèle de l'adversaire pour le DCP se réduit aux ordonnanceurs qui choisissent les actions du MDP représentant le protocole. Toutesfois, nous restreignons cet adversaire pour l'empêcher de contrôler les choix internes aux cryptographes en définissant une relation d'équivalence.

Pour le protocole à trois cryptographes, nous définissons une relation $\Re_{pos}$ sur l'ensemble $Act = \{\mathrm{x}i, \mathrm{a1}i, \mathrm{a2}i, \mathrm{a3}i, \mathrm{b1}i, \mathrm{b2}i, \mathrm{b3}i, \mathrm{c1}i, \mathrm{c2}i, \mathrm{c3}i, \mathrm{d1}i, \mathrm{d2}i, \mathrm{d3}i, \mathrm{fin}\}$, $i \in \{1, 2, 3\}$ qui rend indistinguables par l'adversaire, les actions de choix de la position d'annoncement du résultat par le cryptographe. Intuitivement, cette relation empêche l'adversaire de forcer le cryptographe à communiquer le résultat dans une position qu'il a soigneusement choisie pour des fins d'attaques du protocole. Le choix de position d'annoncement du résultat relève de la décision exclusive de chaque cryptographe. La relation $\Re_{pos}$ induit alors 16 classes d'équivalence sur l'ensemble Act. Quatre classes d'équivalence comportent chacune une seule action dont {x1}, {x2}, {x3} et {fin}. Les 12 autres classes d'équivalence contiennent chacune trois actions de choix de position d'annoncement du cryptographe une fois consulté les affichages des deux pièces qui lui sont visibles. La liste de ces classes est la suivante : $\{\mathrm{a11}, \mathrm{a21}, \mathrm{a31}\}$, $\{\mathrm{b11}, \mathrm{b21}, \mathrm{b31}\}$, $\{\mathrm{c11}, \mathrm{c21}, \mathrm{c31}\}$, $\{\mathrm{d11}, \mathrm{d21}, \mathrm{d31}\}$, $\{\mathrm{a12}, \mathrm{a22}, \mathrm{a32}\}$, $\{\mathrm{b12}, \mathrm{b22}, \mathrm{b32}\}$, $\{\mathrm{c12}, \mathrm{c22}, \mathrm{c32}\}$, $\{\mathrm{d12}, \mathrm{d22}, \mathrm{d32}\}$, $\{\mathrm{a13}, \mathrm{a23}, \mathrm{a33}\}$, $\{\mathrm{b13}, \mathrm{b23}, \mathrm{b33}\}$, $\{\mathrm{c13}, \mathrm{c23}, \mathrm{c33}\}$ et $\{\mathrm{d13}, \mathrm{d23}, \mathrm{d33}\}$.

La spécification du protocole avec PRISM ainsi que le modèle de l'adversaire peuvent se généraliser pour un nombre fini de cryptographes en utilisant les mêmes principes que pour le protocole à trois cryptographes.

Construction du modèle avec PRISM

Le Tableau 4.1 représente les informations du modèle construit par PRISM en termes du nombre d'états accessibles, du nombre de transitions, du nombre de choix (distributions de probabilités) ainsi que du temps de modification du système de transitions en tenant compte de la relation $\Re_{pos}$. Les valeurs entre parenthèses indiquent le nombre d'états initiaux pour chaque espace d'états accessibles relativement au nombre de cryptographes définis dans le modèle. En comparant les choix avant et après restriction de l'adversaire, nous observons une diminution du nombre de choix après restriction peu importe le nombre de cryptographes dans le modèle. Cela est dû au fait que le modèle de restriction de l'adversaire combine, dans tous les états accessibles, toutes les distributions de probabilités associées aux actions issues de la même classe d'équivalence en une seule distribution de probabilités.

Les Tableaux 4.2 et 4.3 représentent les différentes informations sur les principaux MTBDDs

Tableau 4.1 Les informations du DCP pour 3–6 cryptographes et le temps de construction du modèle de sécurité dérivant de la restriction des ordonnanceurs

Crypto	États	Transitions	Choix		Tps(sec.)
			Av. Restr.	Ap. Restr.	
3	1,884(4)	4,200	3,948	2,316	5
4	33,365(5)	96,120	91,420	45,100	3.324×10^2
5	667,098(6)	2,391,540	2,290,350	990,030	1.634×10^4
6	14,853,279(7)	64,276,716	61,859,406	28,453,239	1.402×10^5

Tableau 4.2 Le nombre de noeuds des principaux MTBDDs et le nombre de classes de transitions encodées dans dans le MTBDD de f_Δ pour 3–6 cryptographes

Crypto	MTBDD f_Δ		MTBDD dirFus	Classes de trans.
	Av. Restr.	Ap. Restr.		
3	4,277(3)	2,379(4)	2,280(17)	40
4	21,858(3)	11,146(5)	18,977(22)	69
5	90,667(3)	41,688(6)	151,352(27)	106
6	385,715(3)	170,918(7)	1,754,740(32)	151

Tableau 4.3 Le nombre de variables booléennes utilisées pour représenter les MTBDDs des fonctions f_Δ et f'_Δ pour 3–6 cryptographes

Crypto	Var. lignes	Var. colonnes	Var. non déterministes	
			Av. Restr.	Ap. Restr.
3	20	20	15	6
4	31	31	20	7
5	38	38	25	7
6	45	45	30	8

utilisés dans l'encodage des fonctions de transitions avant et après restriction des ordonnanceurs. Dans le Tableau 4.2, chaque ligne indique pour un nombre donné de cryptographes dans le protocole le nombre de noeuds du MTBDD représentant la fonction f_Δ sans restriction

de l'adversaire, le nombre de noeuds du MTBDD encodant la nouvelle fonction f'_Δ issue du modèle de restriction de l'adversaire ainsi que le nombre des noeuds du MTBDD encodant les directives de fusion suivi du nombre de classes de transitions encodées dans le MTBDD de f_Δ. Les valeurs entre parenthèses représentent le nombre de noeuds terminaux. Particulièrement dans le cas de $dirFus$, le nombre de terminaux donne une idée sur le nombre maximal de classes de transitions de la fonction f'_Δ résultant de la restriction de l'adversaire (la valeur entre parenthèses à laquelle est retirée une unité représentant le terminal "-1" relatif aux classes de transitions fictives). Dans le Tableau 4.3, chaque ligne indique pour un nombre donné de cryptographes le nombre de variables booléennes utilisées dans la représentation des MTBDDs du Tableau 4.2. Nous omettons le nombre de variables booléennes utilisées par *dirFus*, elles sont directement calculables sachant le nombre d'états accessibles et le nombre de classes de transitions pour un nombre donné de cryptographes dans le modèle (voir Section 3.3.2). Nous observons également une nette augmentation du temps de modification du modèle représenté dans le Tableau 4.1 qui varie de cinq secondes pour un protocole de trois cryptographes à environ 39 heures pour un protocole de six cryptographes. Comme l'algorithme de modification visite pour chaque état accessible toutes les classes de transitions, le temps de modification du modèle augmente avec le nombre de cryptographes qui engendrent plus d'états et de classes de transitions dans le modèle.

L'espace mémoire occupé par les principaux MTBDDs est illustré par la Figure 4.1. Pour

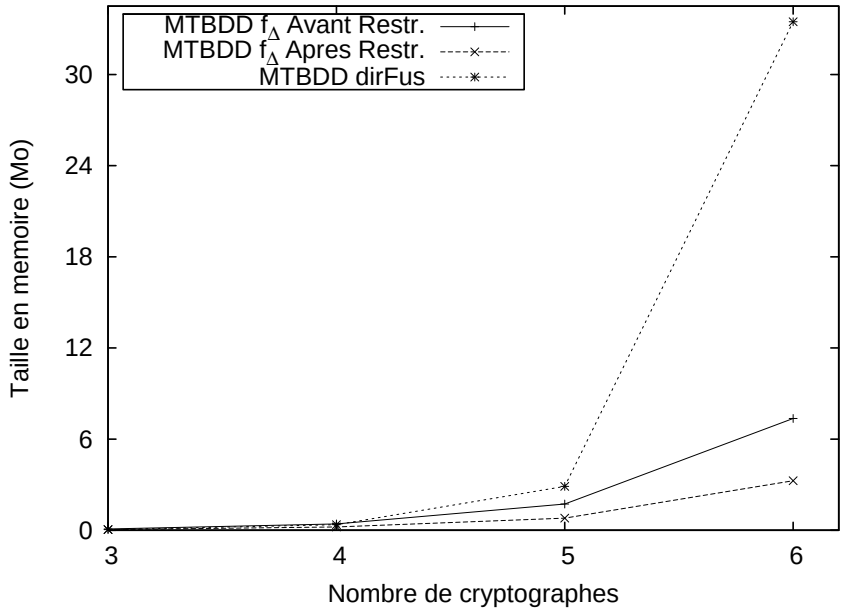


Figure 4.1 Espace mémoire occupée par les principaux MTBDDs pour 3–6 cryptographes

chacun des trois principaux MTBDDs, nous avons la taille mémoire variant de quelques Kilo-octets à quelques Méga-octets en fonction du nombre de cryptographes participant dans le protocole. Nous assumons que chaque noeud MTBDD occupe 20 octets : trois pointeurs et deux entiers, dont chacun sur 32 bits, sont utilisés pour représenter un noeud non terminal dans le package CUDD de Somenzi (1997). Deux des trois pointeurs référencent les noeuds fils (branche *Then* et *Else*) et le troisième maintient le noeud dans une liste chaînée gérée par CUDD. Les deux entiers indiquent respectivement le niveau de la variable booléenne étiquetant le noeud ainsi que le compteur de référence servant au ramasse-miettes de libérer l'espace occupé par le noeud une fois qu'il n'est plus utilisé. Les noeuds terminaux sont représentés différemment mais occupent à peu près la même taille mémoire. Pour un protocole de six cryptographes, nous observons un grand écart entre la taille occupée par le MTBDD *dirFus* (33, 46 Méga-octets) et celle occupée par les MTBDDs représentant f_Δ (3, 26 Méga-octets) et f'_Δ (7, 35 Méga-octets). Cet écart s'explique par l'existence d'un nombre élevé de terminaux qui diminue les possibilités de réduction des noeuds sur un grand nombre de chemins générés par un grand nombre d'états et de classes de transitions encodés dans le MTBDD *dirFus*. Notons également que quelque soit le nombre de cryptographes, la taille mémoire occupée par le MTBDD encodant f_Δ est toujours inférieure à celle occupée par le MTBDD encodant f'_Δ. Cette différence est due à la disparition de certaines variables booléennes relatives au non déterminisme résolu dans le modèle conformément à la relation $\Re_{pos}$.

Spécification des exigences sécuritaires et analyse

Cette section est consacrée à l'évaluation de la sécurité sur le DCP. Avant de formuler la propriété, nous commençons par vérifier si notre protocole respecte les principes de Chaum (1988) pour un protocole à trois cryptographes : si le nombre d'annonces "*différentes*" est pair alors c'est la *NSA* qui paye. Dans le cas contraire c'est l'un des cryptographes qui est le payeur du dîner. Nous spécifions respectivement les deux affirmations avec PCTL comme suit :

`Pmax`=? [`true U statut1`=1 & `statut2`=1 & `statut3`=1 & `payeur`=`nsa` & `func`(`mod`, `egale1`+`egale2`+`egale3`, 2)=1 {"`init`"}{`max`}]

`Pmax`=? [`true U statut1`=1 & `statut2`=1 & `statut3`=1 & `payeur`>0 & `func`(`mod`, `egale1`+ `egale2`+`egale3`, 2)=0 {"`init`"}{`max`}]

La présence de *init* dans les propriétés permet de considérer tous les états initiaux dans le calcul de la probabilité maximale car notre modèle de protocole contient plus d'un état initial (voir Tableau 4.1). Peu importe le modèle de l'adversaire (adversaire contrôlant tous les or-

donnanceurs ou adversaire restreint via la relation $\Re_{pos}$), elles sont toutes évaluées à la valeur 1 confirmant ainsi la correspondance du modèle construit à la description du protocole.

Comme il est très difficile de spécifier la propriété d'anonymat en termes de logique temporelle, nous cherchons à vérifier une propriété indirectement reliée à l'anonymat du protocole. En effet, nous évaluons la probabilité maximale avec laquelle un cryptographe payeur annonce toujours son résultat en dernière position. En PCTL pour un protocole de trois cryptographes, elle s'exprime comme suit :

`Pmax` = ? [`true U` (`statut1`=1 & `statut2`=1 & `statut3`=1) & ((`payeur`=`crypto1` & `ordre1` =3) | (`payeur`=`crypto2` & `ordre2`=3) | (`payeur`=`crypto3` & `ordre3`=3)){"`init`"}{`max`}]

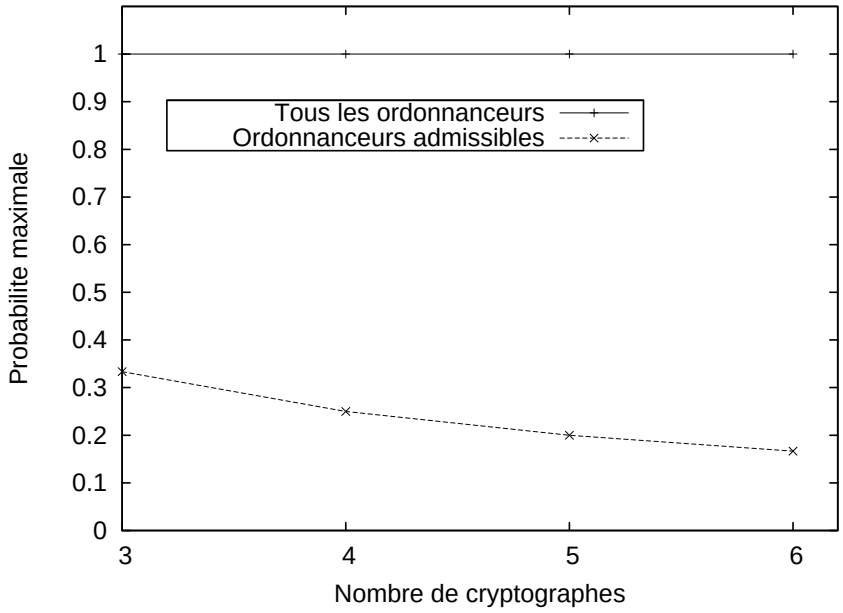


Figure 4.2 Probabilité d'annonce du payeur en dernière position pour 3–6 cryptographes

La Figure 4.2 montre les valeurs probabilistes de cette propriété sur un protocole de trois à six cryptographes en tenant compte de deux situations : la présence d'un adversaire trop fort contrôlant tous les ordonnanceurs et la présence d'un adversaire admissible restreint via la relation $\Re_{pos}$. Dans le cas d'un adversaire trop fort (tous les ordonnanceurs), la probabilité est toujours égale à 1 peu importe le nombre de cryptographes dans le protocole. Intuitivement, cela donne la possibilité à l'adversaire de forcer un cryptographe payeur de communiquer le résultat toujours en dernière position, ce qui entrave au principe anonymat du protocole (Même problème que Garcia *et al.* (2007); Bhargava et Palamidessi (2005) ont relevé sur le DCP mais qu'ils ont traité différemment). Dans la deuxième situation, nous supposons un

adversaire admissible ne contrôlant pas le choix de la position d'annoncement (choix interne de chaque cryptographe). Tous les ordonnanceurs choisissant ces actions internes (ordonnanceurs inadmissibles) sont mis hors de contrôle de l'adversaire, ce qui fait que la probabilité d'annoncencement en dernière position (et plus généralement n'importe quelle position) est la même pour tous les cryptographes peu importe le payeur, soit $\frac{1}{n}$ pour un protocole de n cryptographes avec $3 \leq n \leq 6$, préservant ainsi la propriété d'anonymat sur le DCP.

4.2 Protocole de signature de contrat

Le protocole de signature de contrat de Rabin (1983) permet l'échange de signatures sur un contrat $CONTR$ entre deux personnes ou parties ne se faisant pas confiance à l'aide d'une tierce partie de confiance. La tierce partie de confiance émet à des intervalles de temps réguliers un entier choisi dans un intervalle $[1\,..\,k]$. Le protocole assume que les messages émis par la tierce partie de confiance comportant l'entier choisi i et la date d'émission t sont signés avec sa clé publique. Alice et Bob, les personnes ou parties impliquées, s'échangent des préliminaires tout en se mettant d'accord sur une limite de temps t à accorder à l'échange de leurs préliminaires, date de la prochaine émission de la partie de confiance. Alice envoie alors à Bob le préliminaire comportant le message suivant signé par sa clé publique : "*Moi Alice, j'accepte être liée au contrat $CONTR$ si l'entier j est choisi à la date t*". Une fois reçu le message, Bob répond comme suit : "*Moi Bob, j'accepte être lié au contrat $CONTR$ si l'entier j est choisi à la date t*". D'après la spécification du protocole, l'entier j contenu dans chaque paire de préliminaires est le même et chaque participant calcule cet entier en fonction des entiers n_A et n_B choisis respectivement par Alice et Bob, soit $j = (n_A + n_B) mod\ k$. En assumant que les échanges suivent correctement le protocole, Rabin stipule que Alice et Bob sont liés au contrat à l'intérieur de la période constituée de k intervalles de temps après avoir échangé au maximum $2k$ messages et que l'un des partenaires ne peut arnaquer l'autre avec une probabilité excédant $\frac{1}{k}$.

4.2.1 Modélisation du protocole avec le langage PRISM

Dans la modélisation du protocole en langage PRISM nous adoptons deux modules PRISM, le module spécifiant l'échange des préliminaires entre Alice et Bob ainsi que le module décrivant le rôle joué par la tierce partie de confiance dans le protocole. Nous spécifions le modèle sous-forme d'un processus de décision markovien (MDP) pour deux raisons : La première c'est que nous assumons que le choix d'un entier par la partie de confiance dans l'intervalle $[1\,..\,k]$ se fait de façon non déterministe. La seconde raison vient du fait que pour explorer le comportement du protocole pour un nombre quelconque de préliminaires échan-

gés, nous faisons en sorte que le choix de la date d'échéance pour les échanges se fait de façon non déterministe.

Pour avoir un modèle fini, nous fixons à l'avance l'intervalle dans lequel la tierce partie de confiance choisit l'entier à émettre à la date d'échéance. Nous déclarons une constante k en lui assignant la borne supérieure de l'intervalle. Les autres constantes du modèle concernent la date d'échéance des échanges de préliminaires entre Alice et Bob. La constante `ON` indiquant l'arrivée de la date d'échéance et `OFF` le contraire. En effet, les instructions déclarant ces différentes constantes sont les suivantes en considérant l'intervalle $[1\,..\,10]$:

```
const int k = 10;
const int ON = 1;
const int OFF = 0;
```

Les échanges entre Alice et Bob sont décrits dans le module PRISM nommé `AliceBob`. Deux variables PRISM sont utilisées pour contrôler les échanges à tour de rôle. La variable `alice` indique si elle est à 1 que c'est le tour d'Alice d'envoyer le message à Bob. Quant à la variable `bob`, elle contrôle les envoies de messages de Bob à Alice. De plus, deux autres variables PRISM indiquent les entiers choisis par Alice et Bob dans les messages échangés. Les déclarations de variables du module `AliceBob` se présentent comme suit :

```
jA : [0..10] init 0;
jB : [0..10] init 0;
alice : [0..1] init 1;
bob : [0..1] init 0;
```

L'initialisation de la variable `alice` à 1 indique que c'est Alice qui initie les échanges des préliminaires avec Bob. Le module `AliceBob` comporte deux commandes chacune correspondant au message envoyé par chacun des deux partenaires :

```
[x] alice=1 & jA<k & date=OFF -> (alice'=0) & (bob'=1) & (jA'=jA+1);
```

```
[y] bob=1 & jA<k & date=OFF -> (bob'=0) & (alice'=1) & (jB'=jA);
```

La variable `date` est déclarée dans le module spécifiant la tierce partie de confiance et contrôle l'arrivée de la date d'échéance. Le module nommé `Beacon` modélise le comportement de la partie de confiance. Deux variables indiquant respectivement la date d'échéance et l'entier choisi sont déclarées ci-dessous :

```
date : [OFF..ON] init OFF;
```

$$\texttt{i} : [0\,..\,k]\ \texttt{init}\ 0\,;$$

En fixant la constante k à 10, le module `Beacon` comporte 11 commandes décrivant le choix d'un entier quelconque dans l'intervalle $[1\,..\,k]$ une fois la date d'échéance arrivée. Ces commandes se résument comme suit :

[z] date=OFF –> (date'=ON);

[a1] date=ON & i=0 –> (i'=1);

[a2] date=ON & i=0 –> (i'=2);

⋮

[a10] date=ON & i=0 –> (i'=10);

Le modèle du protocole en langage PRISM étant spécifié pour k fixé à 10, nous passons à l'étape suivante d'analyse et de vérification du système de transitions construit par l'outil PRISM.

4.2.2 Modèle de sécurité et analyse du protocole avec PRISM

Modèle de l'adversaire

L'analyse formelle du MDP représentant le protocole de signature de contrat de Rabin (1983) suppose également l'évolution du protocole dans un environnement hostile. Dans notre analyse, nous considérons un adversaire contrôlant l'ordonnancement des différentes actions du protocole et aussi pouvant être n'importe quel participant entre Alice et Bob essayant d'obtenir la signature de l'autre partenaire sans poser la sienne. Par exemple Alice voulant arnaquer Bob, elle peut arrêter prématurement les échanges avec une possibilité qu'à la date de l'échéance, sans donner sa signature, elle obtient celle de Bob. Cependant, nous restreignons l'adversaire de sorte que l'ordonnanceur choisissant l'entier à émettre par la tierce partie de confiance ne soit pas considéré sous son contrôle durant l'analyse formelle du protocole.

En considérant le protocole où k est fixé à 10, nous définissons une relation d'équivalence $\Re_{choice}$ sur l'ensemble $Act = \{\mathrm{x}, \mathrm{y}, \mathrm{z}, \mathrm{a1}, \mathrm{a2}, \mathrm{a3}, \mathrm{a4}, \mathrm{a5}, \mathrm{a6}, \mathrm{a7}, \mathrm{a8}, \mathrm{a9}, \mathrm{a10}\}$ qui rend indistinguables par l'adversaire, les actions de choix de l'entier émis par la tierce partie de confiance à la date d'échéance. La relation $\Re_{choice}$ induit quatre classes d'équivalence dont $\{\mathrm{x}\}$, $\{\mathrm{y}\}$, $\{\mathrm{z}\}$ et $\{\mathrm{a1}, \mathrm{a2}, \mathrm{a3}, \mathrm{a4}, \mathrm{a5}, \mathrm{a6}, \mathrm{a7}, \mathrm{a8}, \mathrm{a9}, \mathrm{a10}\}$.

Comme pour le cas du DCP, la spécification du protocole de signature de contrat en langage PRISM et le modèle de l'adversaire peuvent se généraliser pour un intervalle quelconque de choix des entiers émis en fixant la constate k à un entier quelconque fini.

Construction du modèle avec PRISM

Les informations du modèle construit par PRISM représentant le protocole de signature de contrat de Rabin (1983) sont représentées dans le Tableau 4.4. Elles font référence en fonction du paramètre k au nombre d'états accessibles, de transitions, de choix (distributions de probabilités) avant et après restriction des ordonnanceurs ainsi qu'au temps de construction du modèle relativement à la relation d'équivalence $\Re_{choice}$. Le nombre de choix du modèle

Tableau 4.4 Les informations du système de transitions et le temps de construction du modèle restreint en fonction du paramètre k

K	**États**	**Transitions**	**Choix**		**Tps(sec.)**
			Av. Restr.	**Ap. Restr.**	
10	240(1)	439	439	259	1.490
50	5,200(1)	10,199	10,199	5,299	62.030
100	20,400(1)	40,399	40,399	20,599	498.274
150	45,600(1)	90,599	90,599	45,899	1.418×10^3
200	80,800(1)	160,799	160,799	81,199	2.157×10^3
250	126,000(1)	250,999	250,999	126,499	4.603×10^3

avant la restriction des ordonnanceurs correspond exactement au nombre de transitions car elles s'effectuent toutes avec probabilité 1. Après restriction des ordonnanceurs, le nombre de choix est largement inférieur au nombre de transitions du fait que, dans tous les états accessibles, toutes les distributions de probabilités étiquetées par les actions appartenant à la même classe d'équivalence sont combinées. Le temps de construction du modèle issu de la restriction des ordonnanceurs augmente proportionnellement au nombre d'états du modèle par le nombre de classes de transitions construits par PRISM. Notons également que quelque soit la valeur du paramètre k, le modèle comporte toujours un seul état initial. Les Tableaux 4.5 et 4.6 représentent les différentes informations sur les principales structures de données représentant le modèle en fonction du paramètre k. Le Tableau 4.6 montre le nombre de variables booléennes utilisées dans la représentation des fonctions f_Δ et f'_Δ. Dans le Tableau 4.5, chaque ligne indique respectivement dans la $2^{ème}$, $3^{ème}$ et $4^{ème}$ colonne le nombre

Tableau 4.5 Le nombre de noeuds des principaux MTBDDs et le nombre de classes de transitions encodées dans le MTBDD de f_Δ en fonction du paramètre k

K	**MTBDD f_Δ**		**MTBDD dirFus**	**Classes de trans.**
	Av. Restr.	**Ap. Restr.**		
10	1,443(2)	459(3)	66(4)	12
50	22,159(2)	1,674(3)	260(4)	52
100	82,493(2)	3,174(3)	766(4)	102
150	201,145(2)	5,185(3)	933(4)	152
200	317,039(2)	6,132(3)	1,498(6)	202
250	457,732(2)	9,445(3)	1,790(6)	252

Tableau 4.6 Le nombre de variables booléennes utilisées pour représenter les MTBDDs des fonctions f_Δ et f'_Δ en fonction du paramètre k

K	**Var. lignes**	**Var. colonnes**	**Var. non déterministes**	
			Av. Restr.	**Ap. Restr.**
10	15	15	5	2
50	21	21	7	2
100	24	24	8	2
150	27	27	9	3
200	27	27	9	3
250	27	27	9	3

de noeuds MTBDD des structures encodant la fonction f_Δ (avant restriction des ordonnanceurs), la fonction f'_Δ (après restriction des ordonnanceurs) ainsi que la matrice représentant les directives de fusion. La dernière colonne représente le nombre de classes de transitions encodées dans le MTBDD de f_Δ. Les valeurs entre parenthèses indiquent le nombre de noeuds terminaux. Une grande différence de noeuds s'observe entre les MTBDDs représentant les fonctions f_Δ et f'_Δ. En effet, cette grande différence s'explique par le fait que dans tout le système, toutes les k classes de transitions représentant le choix d'un entier émis par la tierce partie de confiance sont combinées en une seule classe de transitions après restriction des ordonnanceurs. Le nombre de variables booléennes non déterministes diminue ainsi (voir Tableau 4.6) tout en entrainant du même coup l'élimination de noeuds MTBDD associés. De

plus, la taille de *dirFus* en terme de noeuds MTBDD reste également moindre par rapport aux MTBDDs de f_Δ et f'_Δ du fait que, d'une part le nombre de terminaux reste relativement bas, et d'autre part le nombre total de variables booléennes utilisées pour son encodage est inférieur à ceux de f_Δ et f'_Δ.

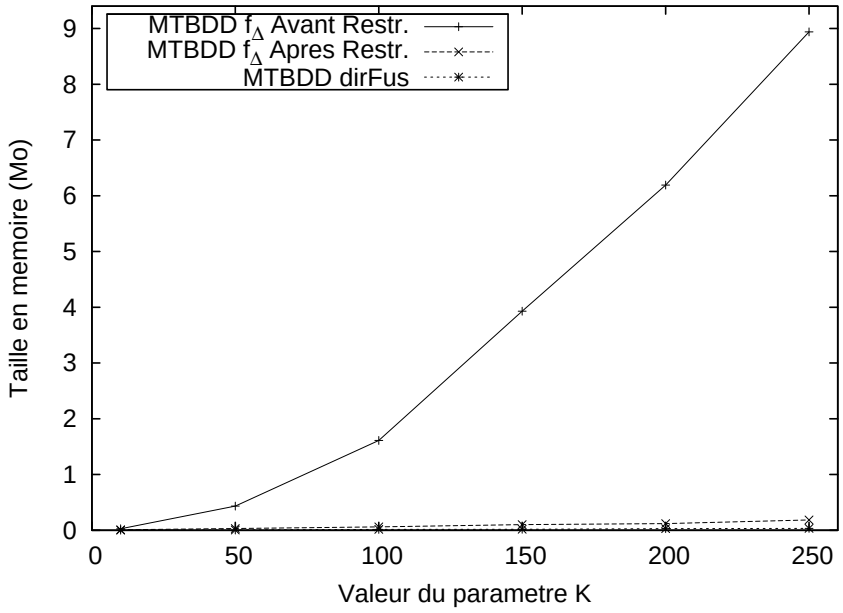


Figure 4.3 Espace mémoire occupé par les principaux MTBDDs en fonction du paramètre k

En assumant les 20 octets de chaque noeud MTBDD du package CUDD de Somenzi (1997), la Figure 4.3 représente les tailles de mémoires (en Méga-octets) occupées par les trois principales structure de données.

Spécification des exigences sécuritaires et analyse

Dans cette section nous évaluons le degré d'équité du protocole de signature de contrat en tenant compte de deux situations : le cas où l'adversaire contrôle tous les ordonnanceurs et le cas où l'adversaire contrôle uniquement l'ensemble des ordonnnanceurs admissibles. Rappelons que cet ensemble d'ordonnanceurs admissibles est déterminé par la relation d'équivalence $\Re_{choice}$. L'inéquité peut survenir uniquement lorsque l'initiateur est le dernier à envoyer le message à la date d'émission de la partie de confiance. Dans le cas contraire, les deux parties sont liés au contrat avec la même probabilité (c.-à-d. soit tous sont liés au contrat soit aucun ne l'est). Examinons le cas où c'est Alice qui a envoyé le dernier message avant la date d'émission de la partie de confiance. Nous exprimons en PCTL la probabilité maximale avec laquelle Alice arnaque Bob (c.-à-d. Bob est lié au contrat sans que Alice ne le soit pas)

comme suit :

`Pmax` = ? [`true U i>0 & jA>=i & jB<i & bob=1`]

Quant à la probabilité avec laquelle dans les mêmes conditions (c.-à-d. Alice est le dernier à envoyer le message) un contrat est lié à Alice et Bob, elle se calcule comme suit :

`Pmax` = ? [`true U i>0 & jA<i & jB<i & bob=1`]

La Figure 4.4 montre les résultats probabilistes de ces deux situations en fonction du para-

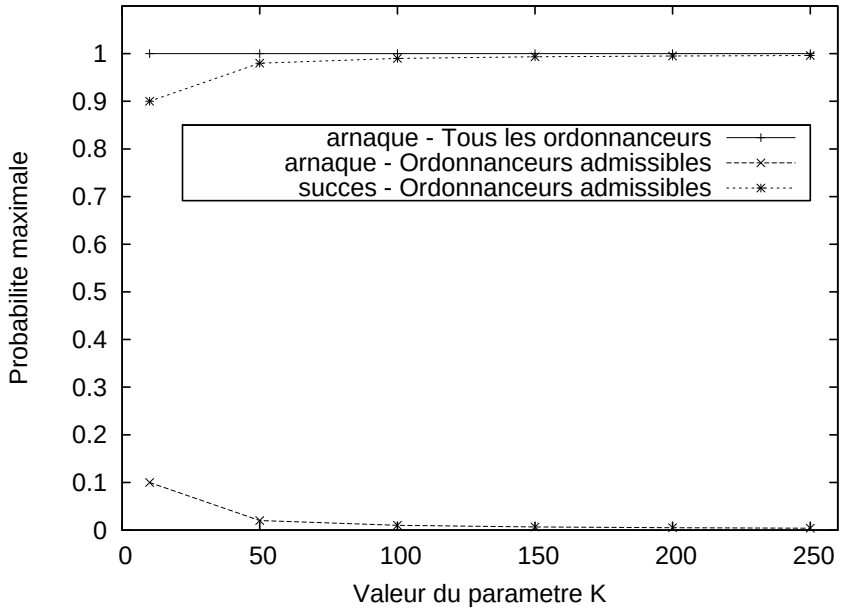


Figure 4.4 Probabilités d'arnaque et de succès dans la signature du contrat en fonction du paramètre k

mètre k et de la puissance de l'adversaire. En considérant tous les ordonnanceurs, la probabilité correspondant à l'arnaque est évaluée à 1. Intuitivement cela correspondrait à la situation où Alice travaillant en concert avec l'adversaire interromp la communication et forçant la partie de confiance à choisir un entier qui le dissocie du contrat tout en obtenant la signature de bob. La probabilité que tous les deux soient liés au contrat en considérant tous les ordonnanceurs serait alors égale à 0 et nous l'avons pas représenté sur la figure car elle est confondue avec l'axe des abcisses. Mais en rendant indistinguables par l'adversaire les actions internes de la tierce partie de confiance via la relation $\Re_{choice}$, pour $10 \leq k \leq 250$, la probabilité maximale que Alice arnaque Bob est $\frac{1}{k}$ tandis que la probabilité que les deux soient liés au contrat est de $1 - \frac{1}{k}$. Ces valeurs probabilistes issues du modèle de restriction de l'adversaire viennent confirmer le théorème de Rabin (1983) qui démontre que Alice ne

peut pas arnaquer Bob avec une probabilité supérieure à $\frac{1}{k}$ en considérant un intervalle de k entiers.

CHAPITRE 5

CONCLUSION ET TRAVAUX FUTURS

5.1 Bilan de la recherche réalisée

Le problème que nous avons abordé dans ce mémoire est un problème bien connu dans la communauté de vérification formelle et concerne la puissance des ordonnanceurs pour l'analyse des systèmes de sécurité basés sur des mécanismes aléatoires. L'objectif de notre mémoire était d'adapter aux modèles de processus de décision markoviens une méthode de restriction des ordonnanceurs définie initialement sur les algèbres de processus probabilistes et de l'intégrer dans un outil automatisé de vérification des systèmes probabilistes. Notre choix a porté sur le *model checker* PRISM pour deux raisons principales : d'une part l'outil supporte les systèmes à la fois probabilistes et non déterministes et d'autre part il utilise des structures de données symboliques nécessaires pour traiter de très grands systèmes.

Nous avons défini une méthode de réduction de la puissance des ordonnanceurs sur des protocoles probabilistes modélisés sous-forme de processus de décision markoviens. Cette méthode utilise une relation d'équivalence observationnelle qui regroupe des ensembles d'actions indistinguables par l'environnement en classes d'équivalences. Deux niveaux de non determinisme sont alors définis sur le modèle, le non déterminisme externe entre les ensembles d'actions équivalentes regroupées dans les tâches et le non déterminisme interne entre les actions de la même tâche. Le non déterminisme interne est résolu statiquement à l'avance par un ordonnanceur interne qui répartit entre les actions de chaque tâche une distribution de probabilités uniforme. Le reste du non déterminisme, sous le contrôle de l'adversaire, est résolu durant la phase d'analyse du protocole. Nous avons ensuite implémenté cette méthode dans l'outil PRISM sous-forme d'une fonctionnalité optionnelle et contrôlable depuis les interfaces utilisateurs. Les résultats obtenus lors de l'évaluation de notre méthode sur des études de cas de protocoles de sécurité ont montré que la puissance excessive des ordonnanceurs pouvait compromettre la vérification en révélant des fausse failles.

Les principales limitations de notre approche sont de deux natures. La première limitation concerne le temps de construction du nouveau modèle issu de la restriction des ordonnanceurs qui augmente énormément avec la taille du système de transitions. Cela est dû au fait que l'algorithme de modification examine tous les états car ils sont tous susceptibles de comporter des tâches ayant plus d'une action et entre lesquelles il faut résoudre le non déterminisme. La

seconde limitation est liée à l'expressivité de la méthode de vérification. En effet, certaines propriétés de sécurité comme l'anonymat sont basées sur les traces observables et sont alors très difficiles à spécifier directement avec la logique temporelle basée sur les états du système. Néanmoins, nous avons contourné cette difficulté en vérifiant avec la logique temporelle une situation indirectement liée à l'anonymat du protocole. Les contributions de notre mémoire se résument ainsi comme suit :

- Définition d'un modèle de sécurité intégrant le modèle du système et le modèle de restriction des ordonnanceurs ;
- Implémentation d'une méthode de construction du modèle de sécurité dans PRISM afin que le *model checking* s'effectue sur le modèle de sécurité et non sur le modèle du système comme le fait habituellement l'outil.

5.2 Améliorations envisageables et travaux futurs

Les principales limitations relevées dans ce travail peuvent faire objet de quelques améliorations. Nous pensons que pour améliorer le temps de construction du modèle de sécurité issu de la restriction des ordonnanceurs, la meilleure voie serait d'étendre la sémantique de construction du modèle définie dans PRISM en y incluant la résolution des transitions non déterministes issues des actions équivalentes. Dans ce cas, la fusion de telles transitions non déterministes se ferait à la volée progressivement depuis le bas niveau c.-à-d. depuis la translation de chaque commande en MTBDD jusqu'à la construction du MTBDD représentant le système de transitions global. Malgré la complexité d'une telle approche, nous pourrions aussi gagner en terme d'espace car la structure intermédiaire encodant les directives de fusion ne serait plus nécessaire. La seconde limitation de notre approche est très difficile à contenir car la plupart des preuves de sécurité s'effectuent par équivalence entre le système réel et le système idéal (sure) par construction, en démontrant que tout comportement possible d'un système réel peut être simulé par un système abstrait contenant une fonctionalité idéale. Néanmoins, nos travaux futurs pourraient investiguer la possibilité d'associer la méthode de *model checking* probabiliste à d'autre méthodes de vérification comme l'analyse des traces d'actions observables issues de l'évolution du système, la simulation ou la bisimulation. D'autres voies de recherche pourraient s'orienter vers la généralisation des stratégies en adoptant des ordonnanceurs internes choisissant dynamiquement une distribution de probabilité dans chaque tâche (par exemple en fonction de l'état, de l'historique dans l'état, etc.).

RÉFÉRENCES

AKERS, S. B. (1978). Binary Decision Diagrams. *IEEE Transactions on Computers*, vol. C-27, pp. 509–516.

ALUR, R. et HENZINGER, T. (1996). Reactive Modules. *Proceedings. 11th Annual IEEE Symposium on Logic in Computer Science (LICS)*. pp. 207–218.

AUDEBAUD, P. et PAULIN-MOHRING, C. (2006). Proofs of Randomized Algorithms in Coq. *Mathematics of Program Construction, volume 4014 of LNCS*. pp. 49–68.

AZIZ, A., SANWAL, K., SINGHAL, V. et BRAYTON, R. (1996). Verifying Continuous Time Markov Chains. *Computer Aided Verification. 8th International Conference, CAV '96. Proceedings*. pp. 269–276.

BAGACHER, V. I. (2007). *Measure Theory*, vol. vol. 1. Springer.

BAHAR, R. I., FROHM, E. A., GAONA, C. M., HACHTEL, G. D., MACII, E., PARDO, A. et SOMENZI, F. (1997). Algebraic Decision Diagrams and their Applications. *Formal Methods in System Design*, vol. 10, pp. 171–206.

BAIER, C. (1998). *On algorithmic Verification Methods for Probabilistic Systems*. Thèse de doctorat, University of Mannheim.

BAIER, C. et KATOEN, J. P. (2008). *Principles of Model Checking*, The MIT Press, chapitre Probabilistic Systems. pp. 745–899.

BAIER, C. et KWIATKOWSKA, M. (1998). Model Checking for a Probabilistic Branching Time Logic with Fairness. *Distributed Computing*, vol. 11, pp. 125–155.

BHARGAVA, M. et PALAMIDESSI, C. (2005). Probabilistic Anonymity. *CONCUR 2005 - Concurrency Theory. 16th International Conference, CONCUR 2005. Proceedings (Lecture Notes in Computer Science Vol. 3653)*. pp. 171–185.

BIALAS, J. (1990). The σ-Additive Measure Theory. *Journal of Formalized Mathematics*, vol. 2.

BIANCO, A. et DE ALFARO, L. (1995). Model Checking of Probabilistic and Nondeteministic Systems. *Foundation of Software Technology and Theorical Computer Science. 15th Conference. Proceedings*. pp. 499–513.

BOLLING, B. et WEGENER, I. (1996). Improving the Variable Ordering of OBDDs is NP-Complete. *IEEE Transactions on Computers*, vol. 45, pp. 993–1002.

BRYANT, R. E. (1986). Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, vol. C-35, pp. 677–691.

BURROWS, M., ABADI, M. et NEEDHAM, R. (1990). A Logic of Authentication. *ACM Transactions on Computer Systems*, vol. 8, pp. 18–36.

CANETTI, R., LING, C., KAYNAR, D., LISKOV, M., LYNCH, N., PAREIRA, O. et SEGALA, R. (2006). Time-Bounded Task-PIOAs : A Framework for Analyszing Security Protocols. *Distributed Computing. 20th International Symposium, DISC 2006. Proceedings (Lecture Notes in Computer Science Vol.4167)*. pp. 238–253.

CHATZIKOKOLAKIS, K. et PALAMIDESSI, C. (2007). Making Random Choices Invisible to the Scheduler. *CONCUR 2007 - Concurrency Theory. Proceedings 18th International Conference, CONCUR 2007*. pp. 42–58.

CHAUM, D. (1988). The Dining Cryptographers Problem : Unconditional Sender and Recipient Untraceability. *Journal of Cryptology*, vol. 1, pp. 65–75.

CHRISTOFF, I. (1990). Testing Equivalences and Fully Abstract Models for Probabilistic Processes. *CONCUR '90. Theories of Concurrency : Unification and Extension*. pp. 126–140.

CLARKE, E. et EMERSON, E. (1984). Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic. *Advanced NATO Study Institute on Logics and Models for Verification and Specification of Concurrent Systems*. pp. 173–239.

CLARKE, E., EMERSON, E. et SISTLA, A. (1986). Automatic Verification of Finite-State Concurrent Systems Using Temporal Logics. *ACM Transactions on Programming Languages and Systems*, vol. 8, pp. 244–263.

DE NICOLA, R. et HENNESSY, M. C. B. (1983). Testing Equivalences for Processes. *Theoretical Computer Science*, vol. 34, pp. 83–133.

DOLEV, D. et YAO, A. C. (1983). On the Security of Public Key Protocols. *IEEE Transactions on Information Theory*, vol. IT–30, pp. 198–208.

ETESSAMI, K., RAJAMANI, S. K. et YOUNES, H. L. S. (2005). Ymer : A Statistical Model Checker. *Computer Aided Verification, Lecture Notes in Computer Science*. pp. 429–433.

FUJITA, M., MCGEER, P. C. et YANG, J. C. Y. (1997). Multi-Terminal Binary Decision Diagrams : An Efficient Data Structure for Matrix Representation. *Formal Methods in System Design*, vol. 10, pp. 149–169.

GARCIA, F. D., ROSSUM, P. V. et SOKOLOVA, A. (2007). Probabilistic Anonymity and Admissible Schedulers. arXiv :0706.1019v1.

GAWANMEH, A. (2008). *On the Formal Verification of Group Key Security Protocols*. Thèse de doctorat, Concordia University.

GIAMPAOLO, B. (2000). *Inductive Verification of Cryptographic Protocols*. Thèse de doctorat, University of Cambridge.

GLABBEEK, V., SMOLKA, R., SCOTT, A., STEFFEN, B. et TOFTS, C. M. N. (1990). Reactive, Generative, and Stratified Models of Probabilistic Processes. *Proceedings - Symposium on Logic in Computer Science*. pp. 130–141.

GOLDWASSER, S., MICALI, S. et RIVEST, R. L. (1988). A digital Signature Scheme Secure Against Adaptive Chosen-Message Attacks. *SIAM Journal on Computing*, vol. 17, pp. 281–308.

GORDON, M. J. C. et MELHAM, T. F. (1993). *Introduction to HOL : A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press.

HAMADOU, S. et MULLINS, J. (2010). Calibrating the Power of Schedulers for Probabilistic Polynomial-Time Calculus. *Journal of Computer Security*, vol. 18, pp. 265–316.

HANSSON, H. et JONSSON, B. (1994). A Logic for Reasonning About Time and Probability. *Formal Aspects of Computing*, vol. 6, pp. 512–535.

HASAN, O. (2008). *Formal Probabilistic Analysis using Theorem Proving*. Thèse de doctorat, Concordia University.

HERMANNS, H., KATOEN, J.-P., MEYER-KAYSER, J. et SIEGLE, M. (2000). A Markov Chain Model Checker. *Proc. 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 00)*. pp. 347–362.

HILLSTON, J. (1996). Performance Evaluation Process Algebra (PEPA). `http://www.dcs.ed.ac.uk/pepa/`.

HOARE, C. (1985). *Communicating Sequential Processes*. Prentice Hall.

HOLZMANN, G. J. (1997). The Model Checker SPIN. *IEEE Transactions on Software Engineering*, vol. 23, pp. 279–295.

HURD, J., MCIVER, A. et MORGAN, C. (2005). Probabilistic Guarded Commands Mechanizedin HOL. *Theoretical Computer Science*, vol. 346, pp. 96–112.

KATOEN, J., KHATTRI, M. M. et ZAPREEV, I. (2005). A Markov Reward Model Checker. *Second International Conference on the Quantitative Evaluation of Systems, 2005*. pp. 243–345.

KELLER, R. (1976). Formal Verification of Parallel Programs. *Communications of the ACM*. pp. 561–572.

KEMENY, J. G., SNELL, J. L. et KNAPP, A. W. (1966). *Denumerable Markov Chains*. Springer-Verlag.

KWIATKOWSKA, M., NORMAN, G. et PARKER, D. (2002). PRISM : Probabilistic Symbolic Model Checker. *Computer Performance Evaluation. Modelling Techniques and Tools. 12th International Conference, TOOLS 2002. Proceedings (Lecture Notes in Computer Science Vol.2324)*. pp. 200–204.

LARSEN, K. G. et SKOU, A. (1989). Bisimulation Through Probabilistic Testing. *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*. pp. 344–352.

LYNCH, N. A. et TUTTLE, M. R. (1989). An Introduction to Input/Output Automata. *CWI Quarterly*, vol. 2, pp. 219–246.

MILNER, R. (1989). *Communication and Concurrency*. Prentice Hall.

MITCHELL, J. C., MITCHELL, M. et STERN, U. (1997). Automated Analysis of Cryptographic Protocols Using Murφ. *Proceedings. 1997 IEEE Symposium on Security and Privacy (Cat. No. 97CB36097)*. pp. 141–151.

OLDENKAMP, H. A. (2007). *Probabilistic Model Checking : A Comparison of tools*. Mémoire de maîtrise, University of Twente.

PARK, D. (1981). Concurrency and Automata on Infinite Sequences. *Proceedings of the 5th GI-Conference on Theoretical Computer Science*. pp. 167–183.

PARKER, D. A. (2002). *Implementation of Symbolic Model Checking for Probabilistic Systems*. Thèse de doctorat, University of Birmingham.

PAULSON, L. C. (1989). The Foundation of a Generic Theorem Prover. *Journal of Automated Reasoning*, vol. 5, pp. 363–397.

PAULSON, L. C. (1994). *Isabelle : A Generic Theorem Prover, volume 828 of Lecture Notes in Computer Science*. Springer-Verlag.

RABIN, M. O. (1983). Transaction Protection by Beacons. *Journal of Computer and System Sciences*, vol. 27, pp. 256–267.

REISCHER, G., LEBLANC, R., RÉMILLARD, B. et LAROQUE, D. (2002). *Théories des Probabilités : Problèmes et Solutions*. Presses de l'Université du Québec.

SCEDROV, A., MITCHELL, J. C., RAMANATHAN, A. et TEAGUE, V. (2006). A Probabilistic Polynomial-Time Process Calculus for the Analysis of Cryptographic Protocols. *Theoretical Computer Science*, vol. 353, pp. 118–164.

SEGALA, R. (1995). *Modeling and Verification of Randomized Distributed Real-Time Systems*. Thèse de doctorat, Massachusetts Institute of Technology.

SEGALA, R. et TURRINI, A. (2007). Approximated Computationally Bounded Simulation Relations for Probabilistic Automata. *20th IEEE Computer Security Foundation Symposium (CSF'07)*. pp. 140–156.

SEN, K., VISWANATHAN, M. et AGHA, G. (2005). VESTA : A Statistical Model-Checker and Analyzer for Probabilistic Systems. *Proc. IEEE International Conference on the Quantitative Evaluation of Systems*. pp. 251–252.

SHANNON, C. E. (1938). Symbolic Analysis of Relay and Switching Circuit. *Transactions of the American Institute of Electrical Engineers*, vol. 57, pp. 713–723.

SOMENZI, F. (1997). CU Decision Diagram Package (CUDD). `http://vlsi.colorado.edu/~fabio/CUDD/`.

STALLINGS, W. (2002). *Cryptography and Network Security : Principles and Practice*. Pearson Education.

VARDI, M. Y. (1985). Automatic Verification of Probabilistic Concurrent Finite-Sate Programs. *Annual Symposium on Foundations of Computer Science (Proceedings)*. pp. 327–338.