Analyse de l'erreur dans la vérification probabiliste

Joël Muhimpundu

Maîtrise en informatique

Québec, Canada

Résumé

Nous nous intéressons aux erreurs qui apparaissent lors de la vérification de systèmes probabilistes en utilisant la technique d'*évaluation de modèle* avec l'outil PRISM. L'évaluation de modèle probabiliste est une technique de vérification qui consiste à déterminer si un modèle probabiliste M vérifie une propriété donnée. Les modèles sont décrits par des systèmes de transitions tandis que la logique temporelle est utilisée comme langage de spécification des propriétés. L'algorithme d'évaluation de modèle qui est appliqué consiste essentiellement à résoudre un système d'équations linéaires $Ax = b$. Nous montrons à quelles étapes du processus d'évaluation de modèle les erreurs apparaissent. Nous distinguons essentiellement deux types d'erreurs à savoir les erreurs d'arrondi (arithmétique à point flottant, model-checking symbolique), et les erreurs de troncature qui proviennent du fait qu'on a remplacé une méthode de calcul direct par des opérations mettant en jeu un nombre fini d'étapes. Nous utilisons la notion de bisimulation approchée pour comparer le modèle M à l'étude et celui réellement encodé par PRISM. Nous faisons aussi une analyse numérique de l'écart entre la solution x du système linéaire $Ax = b$ et celle calculée par PRISM suite aux effets de l'erreur d'arrondi.

Abstract

We are interested in errors that occur during the verification of probabilistic systems using the technique of *model-checking* with PRISM tool. Probabilistic model-checking is a technique for verification that aims at determining whether a probabilistic model satisfies a given property. The models are described by transition systems while temporal logic is used as specification language for properties. The model-checking algorithm under study essentially involves solving a system of linear equations $Ax = b$. We show at what stages of the model-checking process the errors appear. We basically distinguish two types of errors, namely rounding errors (floating-point arithmetic, symbolic model checking), and truncation errors, which arise because a direct method of calculation is replaced by operations involving a finite number of steps. We use a notion of approximate bisimulation to compare the model under study and the one actually encoded by PRISM. We also carry a numerical analysis of the difference between the solution of the linear system $Ax = b$ and the one calculated by PRISM, due to the effects of rounding errors.

Table des matières

Liste des figures

À mon défunt père
À ma mère
À toute ma famille

No amount of experimentation
can ever prove me right ; a single
experiment can prove me wrong.

Albert Einstein

Remerciements

Mes sincères remerciements s'adressent particulièrement à ma directrice de recherche, Mme Josée Desharnais, professeure au Département d'informatique de l'Université Laval, pour son aide et sa disponibilité tout au long de ma maîtrise. C'est grâce à ses conseils, ses éclaircissements ainsi que ses nombreuses lectures que ce travail a pu être réalisé. Sous sa supervision, j'ai appris la rigueur et le souci du travail bien fait. Merci Josée.

Je tiens également à remercier mon co-directeur de recherche, M. François Laviolette, professeur au Département d'informatique de l'Université Laval, pour son apport dans la rédaction de ce mémoire. J'ai été particulièrement impressionné par la pertinence de ses interventions scientifiques.

Mes remerciements vont également à Pierre et Rachel Marchand pour leur appui aux étudiants issus du projet Université Virtuelle Africaine à l'Université Laval. Vous avez été une seconde famille pour toute la cohorte. Merci à vous.

Je voudrais également remercier mes amis et collègues qui m'ont aidé de différentes façons, de par nos nombreuses discussions et leurs encouragements entre autres.

Enfin, je tiens à remercier du fond de mon cœur, ma mère ainsi que le reste de toute ma famille pour leur soutien tout au long de mes études. Ce travail est aussi le résultat de vos efforts.

Chapitre 1

Introduction

Notre dépendance envers les systèmes issus des technologies de l'information et de la communication n'a cessé de croître au cours de ces dernières années. Ces composants informatiques sont de plus en plus complexes et font partie de notre vie quotidienne via Internet ou via toutes sortes de systèmes intégrés comme les cartes de crédits, les téléphones portables, les téléviseurs intelligents, etc. Des services comme les opérations bancaires en ligne ou l'achat en ligne font partie de notre réalité quotidienne.

La fiabilité des systèmes informatiques est devenue un enjeu d'une importance sociale capitale. Plus que les performances en termes de temps de réponse et de capacité de traitement, la fiabilité est une qualité recherchée et utilisée pour l'évaluation d'un système informatique. La fiabilité d'un système informatique est l'aptitude de ce dernier à accomplir une fonction requise dans des conditions données pour une période de temps donnée [24]. En d'autres termes, la capacité à fournir des réponses telles que prévues par la spécification ou encore l'absence d'erreurs.

La fiabilité d'un système est avant tout une question de sécurité surtout quand il s'agit de systèmes critiques. Nous appelons un système critique, tout système dont les défaillances peuvent avoir des conséquences dramatiques et catastrophiques en termes humains ou économiques.

Un des exemples célèbres de système critique aux conséquences néfastes est la machine de radiothérapie *Therac-25* développée conjointement par l'*Atomic Energy of Canada Limited* (**AECL**) et par la société française **CGR MeV**. Entre 1985 et 1987, la machine Therac-25 fut impliquée dans six accidents durant lesquels des patients reçurent des doses massives de radiation. Au moins cinq patients décédèrent des suites de l'irradiation. Une erreur dans la programmation du logiciel conduisait dans certaines situations à utiliser erronément la machine dans un mode Rayon-X au lieu du mode faisceau d'électrons, ce qui envoyait 125 fois plus de radiations que la dose prescrite par le médecin [6].

Les coûts peuvent être catastrophiques en termes de vies humaines mais aussi en termes de coût économique et d'image de la marque. On pense ainsi à de grands échecs comme le vol 501 inaugural de la fusée **Ariane-5** qui a eu lieu le 4 juin 1996 et s'est soldé par un échec. La fusée a explosé 37 secondes après le décollage à la suite d'une panne du système de navigation. L'incident n'a fait aucune victime car la fusée n'était pas habitée. Les enquêtes ont montré que l'incident était dû à une erreur dans les appareils informatiques utilisés par le pilote automatique qui a provoqué la destruction de la fusée ainsi que la charge utile c'est-à-dire 4 sondes de la mission Cluster [6].

Un autre exemple de bogue est le rappel de 2.5 millions de véhicules par le constructeur automobile Honda en octobre 2011. Le rappel était dû à un problème informatique et électronique au niveau de la commande permettant d'actionner les vitres. La saleté qui s'accumulait sur les vitres endommageait les circuits et provoquait un incendie. L'erreur a coûté cher car il a fallu rappeler les véhicules défectueux. L'erreur a ainsi endommagé la réputation de la marque Honda [22].

Ce ne sont pas que les systèmes critiques qui sont seulement concernés par la fiabilité. Pour des raisons de coûts de fabrication et de maintenance, tous les logiciels sont concernés, critiques ou non. Pour détecter les erreurs, le slogan est : « *Plus tôt c'est, mieux c'est* ». Les études ont montré que la correction d'une erreur détectée durant la phase de maintenance coûte 500 fois plus cher qu'une erreur détectée durant les phases de programmation [6]. En effet, la correction des erreurs remet souvent en cause une grande partie du développement effectué, une détection tardive nécessite un retour en arrière dans le développement du logiciel, comme c'est le cas des erreurs découvertes dans la phase de test. De plus, la taille et par voie de conséquence la complexité des systèmes informatiques augmentent. Ils sont intégrés dans de larges systèmes, connectés avec d'autres composantes. Ce faisant, ils deviennent plus vulnérables aux erreurs car le nombre de défectuosités croît proportionnellement au nombre de composantes en interaction.

Il y a ainsi la nécessité de détecter les erreurs plus tôt tout en sachant que les systèmes à vérifier sont complexes, avec des comportements qui sont souvent difficiles à prédire pour l'humain. Des techniques et des outils ont donc été conçus pour permettre aux concepteurs de faire la vérification et la validation des logiciels de façon automatique.

1.1 Techniques de vérification

Les techniques de vérification sont utilisées pour vérifier si le système à l'étude vérifie certaines propriétés. Les propriétés à valider sont obtenues de la spécification du système. Cette dernière décrit ce que le système doit faire ou pas. Un système est considéré *correct* s'il satisfait toutes les propriétés de la spécification. Il apparaît dès lors que l'absence d'erreurs est toujours relative à la spécification et ne peut être donc une propriété absolue d'un système.

Parmi les techniques de vérification du logiciel, l'**évaluation par les pairs** et les **tests logiciels** sont les principales techniques utilisées.

L'*évaluation par les pairs* ou la *revue de code* est une vérification du logiciel qui est effectuée par une équipe de concepteurs logiciels, de préférence qui n'ont pas été impliqués dans le développement du logiciel à l'étude. Le code non compilé n'est pas exécuté mais analysé complètement de façon analytique. Quoique la plupart des fois appliquées de façon ad hoc, des procédures plus spécifiques d'évaluation par les pairs peuvent être faites, notamment celle dédiée à la détection d'erreurs spécifiques.

L'évaluation par les pairs est utilisée dans 80% des projets logiciels en dépit du fait qu'elle soit exécutée souvent manuellement. Puisque elle est faite de façon statique, il est évident que certaines erreurs subtiles comme celles dues à la concurrence peuvent être difficiles à détecter.

Les *tests logiciels* constituent quant à eux, une part significative de tout projet de génie logiciel. Environ 30% à 50% du coût total d'un projet sont dédiés aux tests. A l'opposé de l'évaluation par les pairs où le code est analysé statiquement sans être exécuté, les tests logiciels sont des procédures dynamiques qui exécutent le code. Une partie du logiciel à l'étude est compilée avec des valeurs d'entrées appelés *tests*. Le comportement du logiciel est alors déterminé en forçant le logiciel à traverser un ensemble de chemins d'exécution. L'ensemble de tous les chemins d'exécution est constitué des séquences d'états du code. En se basant sur les observations durant l'exécution des tests, la sortie obtenue est alors comparée avec la sortie désirée tel que documenté dans la spécification. Même si la génération du test ainsi que l'exécution peuvent être effectuées de manière automatique, la comparaison entre la sortie obtenue et celle souhaitée est souvent effectuée par les humains. Le principal avantage des tests logiciels est qu'ils peuvent être appliqués lors de la conception de différentes types de logiciels comme les applications de bureau, les applications web, les systèmes d'exploitation, etc.

Il est à noter qu'une exécution exhaustive de tous les chemins d'exécution est dans la pratique infaisable, seulement une petite partie de cet ensemble est traitée. Les tests logiciels ne peuvent donc pas être complets. Ceci pour dire que "*les tests peuvent prouver la présence d'erreurs, mais pas l'absence d'erreurs*".

Un autre problème avec les tests logiciels est de savoir quand s'arrêter. En pratique, il est difficile, voire impossible, d'indiquer le degré à atteindre pour avoir un certain niveau d'erreurs, par exemple pour atteindre un taux donné de lignes de code sans erreurs.

Les études [6] ont montré que l'évaluation par les pairs et les tests logiciels permettent de détecter différentes classes d'erreurs et à différents moments du cycle de développement. Ils sont donc utilisés ensemble. Pour un résultat meilleur, ils sont précédés par des techniques d'amélioration du logiciel. Il s'agit par exemple de conception structurée et de spécification

comme le langage UML (*Unified Modeling Language*) ainsi que des systèmes de gestion de version.

Une autre technique utilisée pour prévenir les erreurs logicielles est la *preuve formelle* (ou démonstration mathématique) du fonctionnement du programme. L'ensemble des techniques de vérification formelles est appelé *méthodes formelles*. Elles regroupent un ensemble de techniques de vérification qui ont pour but d'établir l'absence d'erreurs de fonctionnement avec la rigueur mathématique.

Les recherches dans le domaine des méthodes formelles ont abouti à différentes techniques de vérification. Elles sont désormais implémentées sous forme d'outils logiciels qui peuvent être utilisés pour la vérification automatique d'une partie des étapes du processus de vérification.

Parmi les techniques développées dans le cadre des méthodes formelles, nous distinguons les *techniques basées sur les modèles*. Les modèles utilisés décrivent de manière précise et non ambiguë le comportement du système à l'étude. Ces techniques sont complétées par des algorithmes qui explorent systématiquement tous les états du modèle. L'exploration peut être exhaustive (cas du model-checking), restreinte à un ensemble de scénarios sur le modèle (cas de la simulation) ou sur l'objet à l'étude (cas des tests). Dans ce travail, nous nous intéressons à une technique de vérification formelle basée sur le modèle appelée **model-checking** [1]. Nous nous intéressons plus précisément aux erreurs qui apparaissent lorsqu'on utilise cette technique pour la vérification de systèmes probabilistes.

1.2 Model-checking

Le model-checking [6] se base sur un parcours exhaustif de tous les états du système. Le model-checker, l'outil logiciel qui exécute le model-checking, examine tous les scénarios possibles du système de façon systématique. De cette façon, il est possible de démontrer qu'un modèle de système vérifie une propriété donnée.

Définition 1.2.1. *Le model-checking est une technique automatisée qui, étant donné un modèle d'un système et une propriété formelle, vérifie systématiquement si la propriété est vérifiée par le modèle (ou par un état du modèle).*

Les modèles sont décrits par des *automates à états finis* [6, 37] qui consistent en un ensemble d'états et un ensemble de transitions. Les états contiennent l'information sur la valeur courante des variables, sur l'état de l'exécution précédente, etc. Les transitions décrivent comment le système évolue d'un état à un autre.

1. Nous conservons l'expression consacrée en anglais par conformité aux auteurs cités. L'expression équivalente en français pourrait être évaluation de modèle ou vérification de modèle.

Pour rendre possible une vérification rigoureuse, les propriétés doivent être décrites de façon précise et non-ambiguë. Ceci est fait en utilisant un langage de spécification de propriétés. Le model-checking a recours à la *logique temporelle* [39] comme langage de spécification de propriétés.

La logique temporelle est une forme de logique modale qui est appropriée pour spécifier des propriétés pertinentes des systèmes informatiques. C'est une extension de la logique propositionnelle avec des opérateurs qui réfèrent au comportement du système au cours du temps [39, 37, 35]. Elle permet la spécification d'une large bande de propriétés pertinentes comme l'*exactitude fonctionnelle* (est-ce que le système fait ce qu'il est censé faire ?), l'*atteignabilité* (est-il possible que le système finisse dans un état de blocage ?), la *sûreté* (un comportement indésirable n'arrivera jamais), la *vivacité* (un comportement désiré arrivera éventuellement), l'*équité* (est ce que sous certaines conditions, un événement apparaît de manière répétée ?) et des *propriétés temporelles* (est-ce que le système est fonctionnel à un certain moment ?).

1.2.1 Phases du model-checking

Comme nous le verrons en détail dans le chapitre suivant, le processus de model-checking comporte les phases suivantes :

- Phase de *modélisation* :
  - modéliser le modèle à l'étude en utilisant le langage de description de modèle du model-checker disponible ;
  - pour faire une première vérification et une rapide évaluation du modèle construit, effectuer quelques simulations ;
  - formaliser les propriétés à vérifier en utilisant le langage de spécification.

- Phase d'*exécution* : exécuter le model-checker pour vérifier la validité de la propriété dans le modèle du système à l'étude.

- Phase d'*analyse* :
  - propriété vérifiée ? → vérifier la suivante (s'il y'en a) ;
  - propriété violée ? →
    1. analyser les contre-exemples générés par simulation ;
    2. raffiner la description, le modèle ou la propriété ;
    3. répéter la procédure entière.
  - défaut de mémoire ? → essayer de réduire le modèle ou essayer encore.

Durant la phase d'exécution du model-checker, celui-ci doit être initialisé avec la configuration adéquate qui permet une vérification exhaustive. Le processus de model-checking qui est alors effectué par le model-checker est un processus algorithmique où la validité de la propriété à l'étude est vérifiée dans tous les états du modèle.

Il s'ensuit la phase d'analyse des résultats. Trois résultats peuvent être obtenus : soit la propriété est vérifiée, soit elle ne l'est pas ou soit le modèle est trop grand pour être analysé avec les limites physiques (mémoire) de l'ordinateur. Dans le cas où la propriété est vérifiée, on conclut que le système vérifie la propriété souhaitée. Si ce n'est pas le cas, avant de conclure que le système à l'étude ne satisfait pas la propriété, il faut s'assurer si des erreurs de nature à influencer le résultat n'auraient pas pu être commises durant le processus de vérification : *erreur de modélisation* (le modèle testé ne correspond au système à vérifier), *erreur de conception* (l'algorithme de model-checking n'est pas correctement implémenté par le model-checker), *erreur de propriété* (la propriété a été mal formulée). C'est seulement après cette vérification que nous pouvons conclure que la propriété n'est pas vérifiée par le système. Toutefois, il faut mentionner qu'il est difficile dans la pratique d'authentifier que le problème formalisé à l'étude (modèle + propriétés) est une description adéquate du problème de vérification en cours [6]. La complexité des systèmes à l'étude ainsi que les failles dans la spécification des fonctionnalités du système rendent cette tâche ardue.

Dans le cas où le modèle est trop grand, il existe des techniques qui permettent de réduire la taille du modèle comme le recours à une abstraction qui regroupe ensemble les états qui ont un comportement identique [31, 43]. Le modèle obtenu est un modèle réduit où chaque état correspond à un sous ensemble d'états de l'ancien modèle à l'étude.

Dans les premiers outils de model-checking, les propriétés qui étaient vérifiées étaient de nature qualitative : est-ce que le résultat généré est correct ? Le système peut-il atteindre un état de blocage, par exemple quand deux programmes concurrents sont en attente l'un de l'autre ? Actuellement, des propriétés de nature quantitative peuvent aussi être vérifiées : la probabilité de délivrer un message durant les t prochaines unités de temps doit être inférieure à 0.98, la probabilité qu'un blocage puisse apparaître dans la prochaine heure est nulle, etc.

Le model-checking probabiliste est effectué sur des modèles qui intègrent de l'information sur la probabilité qu'une transition entre états du modèle se produise. L'exemple qui suit 1.2.2 décrit un système probabiliste et donne un exemple de propriété quantitative que le model-checking probabiliste permet de vérifier.

Exemple 1.2.2. (Protocole zeroconf) [6] *Le protocole zeroconf pour IPv4 a été conçu pour des appareils dans un réseau local (ordinateurs portables, lecteur DVD, etc..) qui sont dotés d'une interface réseau pour une communication mutuelle. De tels réseaux doivent permettre l'ajout ou retrait d'un nouvel appareil automatiquement. Cela veut dire entre autres que, dès*

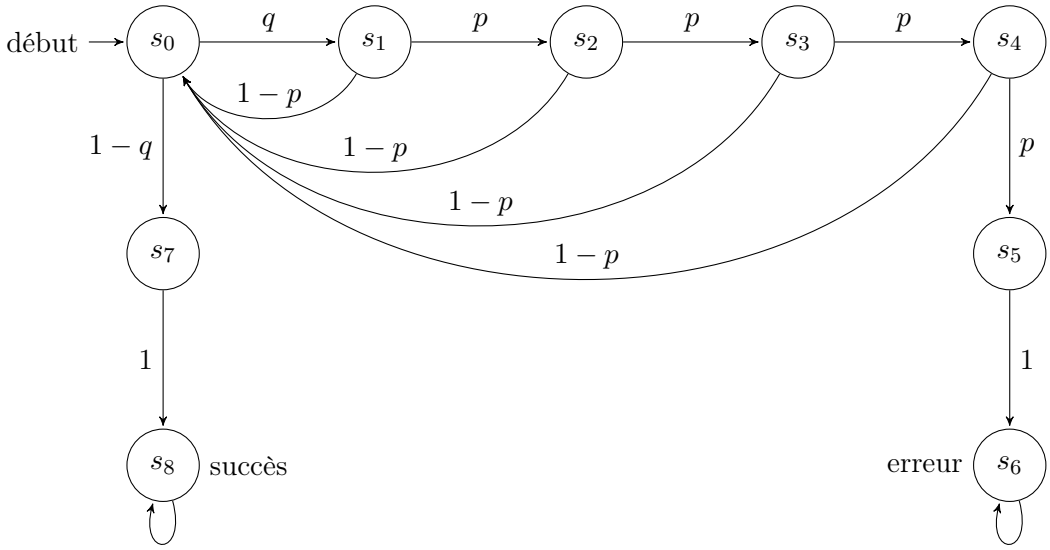


FIGURE 1.1: Le protocole zeroconf (pour $n = 4$ sondes) [6].

*qu'un nouvel appareil est connecté sur le réseau, il doit obtenir une adresse IP unique automatiquement. Le protocole zeroconf est responsable de cette tâche. Un appareil se connecte au réseau et suit le protocole en effectuant les actions suivantes : il choisit aléatoirement une adresse IP, que nous désignons par U, parmi les 65024 adresses disponibles. Il diffuse ensuite à tous les appareils sur le réseau un message appelé "*sonde*" qui demande "*à qui appartient l'adresse U ?*" Si la sonde est reçue par un appareil qui utilise déjà l'adresse U, il répond que U est utilisée. Après avoir reçu le message, l'appareil qui doit être configuré redémarre, il choisit aléatoirement une adresse IP, diffuse une sonde, etc.*

*À cause d'éventuelles pertes de message ou du fait qu'un appareil peut être occupé (i.e. incapable de répondre), une sonde ou une réponse peuvent ne pas parvenir aux autres appareils. Pour augmenter la fiabilité du protocole, ce dernier exige que l'appareil qui se connecte au réseau envoie n sondes, suivies chacune d'une période d'attente de r unités de temps. Ainsi, un appareil qui se connecte peut commencer à utiliser une adresse IP après avoir envoyé n sondes et si aucune réponse n'a été reçue durant $n \times r$ unités de temps. Cependant, malgré toutes ces précautions, un appareil peut se connecter sur le réseau avec une adresse IP utilisée, par exemple si toutes les sondes ont été perdues. Cette situation appelée "*collision d'adresses*" est indésirable puisqu'elle force un appareil à s'arrêter et redémarrer les connections TCP/IP.*

Le comportement d'une machine hôte sur le réseau est décrit par un automate à $n+5$ états où n est le nombre maximal de sondes autorisées. La figure 1.1 montre cet automate pour $n = 4$. L'état initial est étiqueté par début*. Dans l'état s_{n+4} (étiqueté* succès*), l'hôte termine avec une adresse non utilisée ; dans l'état s_{n+2} (étiqueté* erreur*), l'hôte termine avec une adresse déjà utilisée c'est-à-dire qu'il y a collision d'adresses. Un état s_i ($0 < i \leq n$) est atteint*

après l'émission de la ième sonde. Dans l'état initial, l'hôte choisit aléatoirement une adresse IP. Avec la probabilité $q = m/65024$, où m est le nombre de machines hôtes connectées sur le réseau, cette adresse est déjà utilisée. Avec une probabilité $1 - q$, l'hôte choisit une adresse inutilisée et termine dans l'état s_{n+3}. Après cela, l'hôte envoie $n-1$ sondes et attend $n \cdot r$ unités de temps avant d'utiliser cette adresse. Si l'adresse IP est déjà utilisée, l'état s_1 est atteint. Deux situations sont alors possibles. Avec la probabilité p, aucune réponse n'est reçue durant r unités de temps (soit la sonde ou le message de réponse ont été perdus), et la prochaine sonde est envoyée, ce qui nous amène à l'état s_2. Si par contre, une réponse est arrivée à temps, l'hôte retourne à l'état initial et redémarre le protocole. Le comportement dans un état s_i $(2 \leq i < n)$ est similaire. Cependant, si dans un état s_n, aucune réponse n'a été reçue dans r unités de temps après avoir envoyé la nième sonde, une collision d'adresse se produit.

Pour le protocole zeroconf, un exemple de propriété quantitative à vérifier par le model-checking probabiliste serait de déterminer la probabilité qu'une réponse soit obtenue après l'envoi de n' sondes $(n' \leq n)$. Un exemple typique de propriété quantitative serait de savoir si la probabilité de recevoir une réponse après l'émission de n' sondes est comprise dans un intervalle I donné.

1.2.2 Forces et faiblesses

Parmi les avantages du model-checking, nous pouvons citer :

- Il se base sur des fondements mathématiques solides comme la théorie algorithmique des graphes, les structures de données et la logique formelle.
- Il est facilement intégrable durant les cycles de développement d'un logiciel (phase d'analyse, de développement ou de test) et il permet aussi la réduction du temps de développement.
- Il fournit une *information de diagnostic* dans le cas où une propriété n'est pas vérifiée ; ce qui peut être utile pour déboguer.
- Il jouit d'un intérêt croissant du milieu industriel. Plusieurs compagnies dans le domaine du matériel informatique, du logiciel et des services informatiques comme la compagnie Bell [20] et IBM [1, 41] ont désormais des laboratoires où le model-checking est utilisé.

Parmi les faiblesses du model-checking, nous pouvons citer :

- Le model-checking vérifie un modèle de système et non pas le système réel ou son prototype. Le résultat dépend par conséquent de la qualité du modèle.
- Le model-checking souffre du problème d'*explosion combinatoire* : le nombre de chemins à explorer pour analyser avec précision le comportement d'un système peut devenir rapidement grand et par conséquent impossible à parcourir.
- Comme nous le verrons dans la suite de ce travail, le model-checking ne garantit pas un résultat exact dans le cas probabiliste. Dans le cas classique (model-checking non

probabiliste), on a un résultat exact sauf s'il y a des fautes de programmation du model-checker. Dans le cas probabiliste, nous devons représenter les probabilités sous forme de nombres réels en utilisant l'arithmétique à point flottant utilisée par les ordinateurs actuels, telle que décrite par la norme IEEE 754 [25]. L'arithmétique à virgule flottante ne permettant de représenter les nombres réels que jusqu'à une certaine précision (15 chiffres après la virgule), un nombre qui ne peut être représenté avec cette précision est arrondi. La conversion d'un réel en un nombre représentable génère de l'instabilité numérique et peut avoir un impact sur les probabilités calculées par le model-checker.

1.2.3 Applications

Les techniques de vérification par model-checking ont connu un essor ces dernières années. Elles ont été utilisées avec succès pour la détection des erreurs pendant le développement de logiciels dans des domaines variés comme la santé, le commerce électronique, les protocoles de communications, etc.

Parmi les model-checkers existants, nous pouvons citer à titre d'exemple *Spin* [21], *CISMO* [40] et *PRISM* [29].

Spin est un model-checker développé par le centre de recherche en informatique des laboratoires Bell au début des années 80. Il est essentiellement utilisé pour la vérification des systèmes distribués tels que les systèmes d'exploitation, les protocoles de communication de données, les algorithmes concourants, les protocoles de signalisation ferroviaire, etc.

CISMO est un model-checker développé à l'Université Laval par Richard [40]. Il diffère des autres model-checkers en ceci qu'il est un outil de vérification de systèmes probabilistes à espace d'états continu alors que pour la plupart, l'ensemble des états est fini. Bien que cela puisse suffire pour certains systèmes, d'autres dépendent de facteurs continus comme le poids d'un ascenseur, la température d'un radiateur, etc.

PRISM est un model-checker probabiliste développé initialement à l'Université de Birmingham et actuellement à l'Université d'Oxford. Il utilise les modèles probabilistes avec lesquels nous travaillons dans ce document. Nafa [35] a rapporté un cas intéressant d'utilisation de *PRISM* où ce dernier a été utilisé pour vérifier les performances d'un protocole de gestion des canaux. Il s'agissait d'analyser dans quelles circonstances une station mobile (un téléphone portable par exemple) qui se déplace, peut perdre une communication déjà établie. *PRISM* a été utilisé pour vérifier et analyser la performance de ce protocole. Il a permis de réduire la probabilité de perte de communication à moins de 10^{-6}.

Les différents outils cités ci-haut ont permis de découvrir des erreurs au sein des applications industrielles et de fournir un cadre rigoureux de développement [29, 21]. Ils permettent ainsi de garantir les exigences de fonctionnement des systèmes logiciels et d'accroître la confiance

dans ces systèmes.

1.3 But de la recherche

Nous nous intéressons aux erreurs qui apparaissent lors de la vérification de systèmes probabilistes en utilisant la technique du model-checking avec l'outil PRISM. Comme la plupart d'autres model-checkers probabilistes, PRISM fait état dans certaines situations de résultats inexacts lors du calcul des probabilités, sans que la source de telles erreurs soit une erreur de l'utilisateur (erreur de modélisation ou de spécification) ou une erreur dans l'implémentation de l'algorithme de model-checking [9].

Une des causes de ces erreurs est l'usage d'une arithmétique approximative à point flottant durant la phase de modélisation du système à l'étude. L'usage de l'arithmétique à point flottant fait que le système encodé par le model-checker correspond en réalité à un système *approximatif* du modèle à vérifier. Comme le model-checking est alors effectué sur un système approximatif, le résultat peut varier. Pour pouvoir analyser l'écart entre le résultat attendu et celui retourné par PRISM, il nous est apparu important de comparer le système à vérifier et celui encodé. Pour cela, nous avons choisi d'utiliser la notion de bisimulation approximative [14] pour faire la comparaison entre les systèmes probabilistes car la bisimulation classique n'est pas appropriée dans le contexte probabiliste. En effet, la bisimulation classique nous fournit une réponse binaire, i.e., elle nous apprend si deux systèmes sont équivalents (bisimilaires) ou pas. Elle n'est donc pas assez précise puisqu'elle identifie deux systèmes proches comme non-bisimilaires, de même que deux systèmes très différents. Or nous savons déjà que nos systèmes à analyser ne sont pas équivalents, nous voulons plutôt une mesure de leur différence.

Partant de cette comparaison entre le système à vérifier et celui réellement encodé par PRISM, nous voulons faire une comparaison entre le résultat attendu et celui retourné. Nous voulons vérifier si, après l'introduction de l'erreur lors de l'encodage du système à vérifier, il n'y a pas propagation d'erreurs au cours du processus de model-checking. Pour cela, nous allons montrer à quels moments du processus de model-checking, les erreurs sont introduites ainsi que l'impact de ces erreurs, seules ou combinées.

1.4 Plan du mémoire

La suite de ce mémoire est organisée comme suit : dans le chapitre suivant, nous présentons les notions de base associées au model-checking probabiliste. Nous décrivons les différents modèles probabilistes utilisés pour représenter les systèmes à l'étude, la logique formelle utilisée pour spécifier les propriétés ainsi que l'algorithme de model-checking. Nous décrivons aussi comment la technique du model-checking est implémentée en pratique par le model-checker PRISM. Au chapitre 3, nous montrons à quels moments du processus de model-checking avec PRISM

les erreurs apparaissent. Au chapitre 4, nous utilisons la notion de bisimulation approximative pour comparer le système réellement encodé par PRISM et celui à l'étude tandis qu'au chapitre 5, nous faisons une analyse numérique de l'écart entre la probabilité retournée par PRISM et celle attendue. Le chapitre 6 est consacré à la conclusion et à une discussion sur les travaux futurs.

Chapitre 2

Model-checking probabiliste

Dans ce chapitre, nous introduisons les notions de base associées au model-checking probabiliste. Le model-checking consiste à faire une représentation du modèle à l'étude, la spécification des propriétés à vérifier dans un formalisme non ambigu (la plupart des fois à l'aide d'une logique), et des techniques (algorithmes) pour vérifier si le modèle décrit satisfait les propriétés spécifiées.

La section 2.1 décrit différents modèles probabilistes, à savoir : les chaînes de Markov à temps discret (*Discrete-Time Markov Chains* ou DTMC), les processus de décision de Markov (*Markov Decision Processes* ou MDP) ainsi que les chaînes de Markov à temps continu (*Continuous-Time Markov Chains* ou CTMC). Bien que pour le reste du chapitre nous allons présenter le model-checking pour les DTMC, nous décrivons aussi les autres modèles probabilistes pour mieux expliquer les similarités avec les DTMC. La section 2.2 introduit la logique PCTL utilisée pour la spécification des propriétés pour les DTMC et les MDP. La section 2.3 décrit l'algorithme de model-checking pour les DTMC avec la logique PCTL. La section 2.4 montre comment le model-checking est implémenté en pratique.

2.1 Modèles probabilistes

Le model-checking est une technique de vérification automatique de systèmes informatiques. Il consiste à vérifier algorithmiquement si un modèle donné, le système lui-même ou une abstraction du système satisfait une spécification souvent formulée en termes de logique temporelle.

Dans le cas du model-checking probabiliste, nous utilisons des modèles qui intègrent l'information sur la probabilité qu'une transition entre états se produise. Dans cette section, nous décrivons les modèles utilisés pour décrire le comportement d'un système probabiliste.

2.1.1 Chaînes de Markov à temps discret

Les *chaînes de Markov à temps discret* [6, 37, 49] ou DTMC (de l'anglais *Discrete-Time Markov Chain*) sont le modèle le plus simple représentant des systèmes probabilistes. Ce sont des systèmes de transitions où les transitions sont étiquetées par la probabilité de passer d'un état à un autre.

Définition 2.1.1. Soit AP (de l'anglais *Atomic Proposition*) un ensemble fixe de propositions atomiques. Une *chaîne de Markov à temps-discret* est un tuple $M = (S, s_0, P, L)$ tel que *:*

- S est un ensemble fini non vide d'états*,*
- s_0 est l'état initial*,*
- $P : S \times S \to [0, 1]$ est une fonction qui assigne une probabilité à chaque transition de telle sorte que pour tout $s \in S$, on a $\sum_{s' \in S} P(s, s') \in \{0, 1\}$*,*
- $L : S \to 2^{AP}$*,* est une fonction qui assigne à chaque état $s \in S$*,* un ensemble de propositions atomiques $L(s)$ valides dans s*.*

Exemple 2.1.2. *Considérons le DTMC représenté par le graphe de la figure 2.1. L'ensemble des états est* $S = \{s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7\}$*, l'état initial* s_0 *est marqué par une flèche entrante, les transitions sont illustrées par des flèches sur la figure et la probabilité* P *d'une transition est donnée par l'étiquette sur cette flèche. Par exemple,* $P(s_4, s_6) = 2/3$*. On a que* $L(s_0) = L(s_2) = L(s_7) = \{a\}, L(s_1) = \{a, b\}, L(s_3) = L(s_5) = \{a, c\}, L(s_4) = L(s_6) = \{b, c\}$*. Nous disons que la proposition atomique* a *est satisfaite dans les états* $s_0, s_1, s_2, s_3, s_5, s_7$*, la proposition atomique* b *est satisfaite dans les états* s_1, s_4, s_6 *et la proposition atomique* c *est satisfaite dans les états* s_3, s_4, s_5, s_6*.*

Définition 2.1.3. Une *exécution* ou *chemin* du système est une suite non vide finie ou infinie d'états $\pi = s_0 s_1 s_2 s_3 ... s_k$ avec $k \geq 0$, où $s_0, s_1, s_2, s_3 ... s_k \in S$ et $P(s_i, s_i + 1) > 0$ pour tout $i = 0, 1, 2, ...k$. On note par $|\pi| = k$, la longueur de π et par $\pi(i) = s_i$, le (i+1)ème état de π. Un état t est atteint à partir de s_0 s'il existe un chemin fini π dont le premier état est s_0 et le dernier état est t. La probabilité de ce chemin notée $Prob(\pi)$, est calculée de la façon suivante :

$$Prob(s_0 s_1 s_2 s_3 ..... s_t) = P(s_0, s_1) \cdot P(s_1, s_2) \cdot P(s_2, s_3) \cdot \ldots \cdot P(s_{t-1}, s_t).$$

On note par $Paths(s)$ [1], l'ensemble des chemins infinis qui débutent par s.

Exemple 2.1.4. *Considérons le système probabiliste fini de la figure 2.1.*
La suite $\pi = s_0 s_2 s_4 s_7$ *est une exécution de ce système. L'exécution* π *débute à l'état* s_0 *et se*

1. Le terme anglais est conservé par conformité avec tous les auteurs cités.

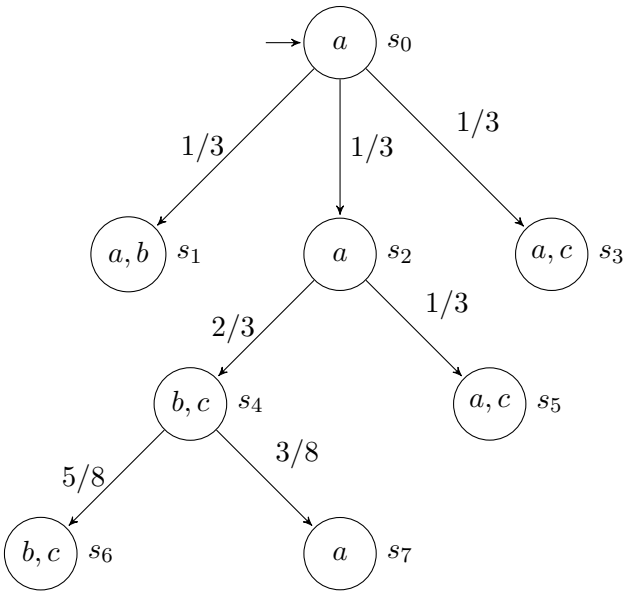


FIGURE 2.1: Une chaîne de Markov à temps discret.

termine à l'état s_7, et a comme longueur $|\pi| = 3$. La probabilité de cette exécution se calcule de la façon suivante :

$$Prob(\pi) = P(s_0, s_2) \cdot P(s_2, s_4) \cdot P(s_4, s_7) = 1/3 \cdot 2/3 \cdot 3/8 = 1/12$$

Afin d'analyser le comportement probabiliste d'un système, nous avons aussi besoin de déterminer la probabilité d'un ensemble de chemins qui satisfont certaines propriétés. Dans ce qui suit, nous introduisons les concepts mathématiques qui vont nous aider à définir le calcul de probabilité d'un ensemble de chemins.

Définition 2.1.5. Une *σ-algèbre* sur un ensemble S est une famille $\Sigma = \{S_1, S_2, \cdots, S_n\}$ de sous-ensembles de S qui contient l'ensemble S lui-même, le complémentaire dans S de chaque sous-ensemble ainsi que l'union dénombrable des sous-ensembles.
Plus formellement :

1. $S \in \Sigma$,
2. $\forall S_i \in \Sigma$, alors $S \backslash S_i \in \Sigma$,
3. $\forall i \in \mathbb{N}$, $S_i \in \Sigma$ alors $\bigcup_{i \in \mathbb{N}} S_i \in \Sigma$.

De cette définition, il ressort que $\emptyset \in \Sigma$ puisque son complément dans S à savoir l'ensemble S lui-même appartient à Σ.

Exemple 2.1.6. *Pour tout ensemble S, $\Sigma = P(S)$, où $P(S)$ est l'ensemble puissance de S et $\Sigma = \{\emptyset, S\}$ sont des σ-algèbres sur S.*

Exemple 2.1.7. *Si* $S = \{a, b, c, d\}$*, alors* $\Sigma = \{\emptyset, \{a\}, \{b, c, d\}, S\}$ *est une* σ*-algèbre sur* S*.*

Étant donnée une collection $\mathcal{C}$ de sous-ensembles d'un ensemble S, la plus petite σ-algèbre qui la contient existe toujours et est appelée la σ-algèbre engendrée par $\mathcal{C}$.

Définition 2.1.8. On appelle *espace de mesure* ou *espace mesurable*, le couple (S, Σ), où S est un ensemble et Σ, une σ-algèbre sur S.

Définition 2.1.9. Soit un espace de mesure (S, Σ). Une *mesure de probabilité* sur S est une fonction μ de Σ dans $\mathbb{R}^+$ telle que :

- $\mu(\emptyset) = 0$,
- si $\{S_i | i \in \mathbb{N}\}$ est une collection d'ensembles deux à deux disjoints de Σ, alors $\mu(\bigcup_{i \in \mathbb{N}} S_i) = \sum_{i \in \mathbb{N}} \mu(S_i)$,
- $\mu(S) = 1$.

Un espace de probabilité (S, Σ, μ) est un espace de mesure doté d'une mesure de probabilité. Nous avons déjà défini la probabilité d'un chemin fini. Nous voulons étendre cette définition à un ensemble de chemins. Dans le cas du model-checking probabiliste, nous aurons à calculer la probabilité d'emprunter un ensemble chemin $B(\pi)$ à partir d'un état s d'un système probabiliste $M = (S, s_0, P, L)$. L'ensemble de tous les chemins à partir de s noté $Paths(s)$ correspond à une σ-algèbre Σ_s sur l'ensemble des états S du système probabiliste. Pour trouver la probabilité d'emprunter un ensemble de chemin $B(\pi) \in \Sigma_s$, nous définissons la mesure de probabilité $Prob$ sur l'espace de mesure (S, Σ_s).

Définition 2.1.10. Soit un système probabiliste $M = (S, s_0, P, L)$, $s \in S$, $\pi = s_0 s_1 s_2 s_3 \cdots s_k$ une exécution finie dans S, et Σ_s la plus petite σ-algèbre sur $Paths(s)$ engendrée par les ensembles de la forme :

$$B(\pi) = \{\rho \in Paths(s) : \pi \textit{ est un préfixe de } \rho\}.$$

On définit la probabilité de l'ensemble $B(\pi)$ comme suit :

$$Prob(B(\pi)) = Prob(\pi) = P(s_0, s_1) \cdot P(s_1, s_2) \cdot P(s_2, s_3) \ \cdots \ P(s_{k-1}, s_k).$$

Le théorème d'extension unique de la mesure [8] nous assure qu'il existe une extension unique de $Prob$ sur Σ_s.

2.1.2 Processus de décision de Markov

Les DTMC ne permettent pas de modéliser le non-déterminisme. Les processus de décision de Markov [7, 37] sont une généralisation des DTMC qui permettent à la fois un choix probabiliste et non déterministe.

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

Définition 2.1.11. Soit AP un ensemble fixe de propositions atomiques. Un *processus de décision de Markov* ou MDP (de l'anglais *Markov Decision Process*) est un tuple $M = (S, \iota_{init}, Act, P, L)$ tel que :

- S est un ensemble fini non vide d'états,
- $\iota_{init} : S \to [0, 1]$ est la distribution initiale telle que $\sum_{s \in S} \iota_{init}(s) = 1$,
- Act est un ensemble d'actions,
- $P : S \times Act \times S \to [0, 1]$ est la fonction des transitions de probabilité telle que pour tout état $s \in S$ et action $\alpha \in Act$: $\sum_{s' \in S} P(s, \alpha, s') \in \{0, 1\}$,
- $L : S \to 2^{AP}$, une fonction d'étiquetage.

Une action α est permise dans un état s si et seulement si $\sum_{s' \in S} P(s, \alpha, s') = 1$. On dénote par $Act(s)$, l'ensemble des actions permises dans s. Pour tout état s, il est requis que $Act(s) \neq \emptyset$. Chaque état s' pour lequel $P(s, \alpha, s') > 0$ est appelé un α-successeur de s. Un MDP est fini si l'ensemble des états S, l'ensemble d'actions Act et l'ensemble AP des propositions atomiques sont finis. Un MDP a une distribution initiale ι_{init} unique.

Le comportement opérationnel dans un MDP M est le suivant. Selon la distribution initiale ι_{init}, une expérience stochastique débute dans un état initial s_0 tel que $\iota_{init}(s) > 0$. En choisissant un état s disons, on doit résoudre un choix non déterministe entre les actions permises. C'est-à-dire, on doit déterminer quelle action parmi $Act(s)$ doit être exécutée. En l'absence d'une quelconque information sur la fréquence des actions, étant donné des actions α et β dans $Act(s)$, il est impossible de déterminer combien de fois α doit être choisi. Ce choix est purement non déterministe et dépend généralement d'un utilisateur externe. Supposons que l'action α est choisie. En exécutant α, un des α-successeurs de s est aléatoirement choisi selon la distribution $P(s, \alpha, .)$. Avec la probabilité $P(s, \alpha, t)$, l'état suivant est t. Si t est l'unique α-successeur de s, alors nécessairement t est l'état suivant après avoir sélectionné α, i.e., $P(s, \alpha, t) = 1$. Dans ce cas, $P(s, \alpha, u) = 0$, pour tout état $u \neq t$.

Un DTMC est un MDP dans lequel pour chaque état s, $Act(s)$ est un singleton. Inversement, tout MDP avec cette propriété est un DTMC. Les étiquettes des actions sont dans ce cas sans importance et peuvent être omises. Les DTMC sont donc un sous-ensemble propre des MDP.

Exemple 2.1.12. *Soit le MDP de la figure 2.2.*

L'état s est le seul état initial, i.e., $\iota_{init}(s) = 1$ et $\iota_{init}(t) = \iota_{init}(u) = 0$. Les ensembles des actions permises sont :

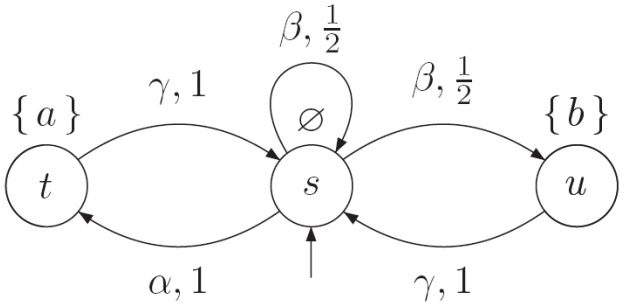


FIGURE 2.2: Un processus de décision de Markov

- $Act(s) = \{\alpha, \beta\}$ *avec* $P(s, \alpha, t) = 1, P(s, \beta, u) = P(s, \beta, s) = 1/2$ *et,*
- $Act(t) = Act(u) = \{\gamma\}$ *avec* $P(t, \gamma, s) = P(u, \gamma, s) = 1$.

Dans l'état s, un choix non déterministe existe entre les actions α et β. En sélectionnant l'action α, le prochain état est t ; en sélectionnant l'action β, les états successeurs s et u sont équiprobables.

Comme pour les DTMC, un chemin dans un MDP est une séquence non finie de la forme :

$$\pi = s_0 \xrightarrow{\alpha 1} s_1 \xrightarrow{\alpha 2} s_2 \xrightarrow{\alpha 3} \ ...,$$

tel que $P(s_i, \alpha_{i+1}, s_{i+1}) > 0$ pour tout $i \geq 0$. $Paths(s)$ dénote l'ensemble des chemins infinis partant de l'état s.

Pour les DTMC, l'ensemble des chemins est doté d'une σ-algèbre et d'une mesure de probabilité qui reflète la notion intuitive de probabilités pour ensembles mesurables de chemins. Pour les MDP, cela n'est pas possible puisqu'aucune contrainte n'est imposée sur la résolution du choix non déterministe.

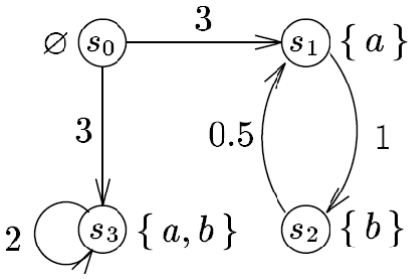
Les MDP ne sont pas donc dotés d'une mesure de probabilité unique. A la place, déterminer la probabilité d'un ensemble de chemins implique la résolution du non-déterminisme. Cette résolution est effectuée par un ordonnanceur. Un ordonnanceur choisi dans chaque état s, une des actions permises $\alpha \in Act(s)$ ($Act(s)$ est non vide pour tout état s). L'ordonnanceur n'impose aucune contrainte sur le choix probabiliste qui est résolu une fois que α est choisi. Pour plus de compréhension des ordonnanceurs sur les MDP, nous référons le lecteur à [7, 37].

2.1.3 Chaînes de Markov à temps continu

Le dernier modèle que nous décrivons, les chaînes de Markov à temps continu (*Continuous-Time Markov Chains* ou CTMC)[7, 4, 37, 44], étend aussi les DTMC mais d'une manière différente. Pour les DTMC, une transition correspond à une étape discrète d'un point de vue temporel. Dans un CTMC, les transitions peuvent être exécutées en temps réel. Chaque transition est étiquetée d'un coût, définissant le délai qui s'écoule avant qu'elle ne soit exécutée.

Définition 2.1.13. Soit AP un ensemble de propositions atomiques. Une *chaîne de Markov à temps continu* est un tuple $M = (S, s_0, R, L)$ où *:*

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



$$\mathbf{R} = \begin{pmatrix} 0 & 0 & 0 & 3 \\ 0 & 0 & 1 & 0 \\ 0 & 0.5 & 0 & 0 \\ 0 & 0 & 0 & 2 \end{pmatrix} \qquad \mathbf{E} = \begin{pmatrix} 6 \\ 1 \\ 0.5 \\ 2 \end{pmatrix}$$

FIGURE 2.3: Chaîne de Markov à temps continu

Intuitivement, $R(s, s')$ spécifie que la probabilité de passer de l'état s à l'état s' en t unités de temps (pour un t positif) est $1 - \exp^{-R(s,s').t}$, une distribution exponentielle avec un coût $R(s, s')$.

Si $R(s, s') > 0$ pour plus d'un état s, on suppose qu'une compétition entre les transitions existe. On désigne cette situation couramment par le terme *race condition*.

Soit $E(s) = \sum_{s' \in S} R(s, s')$, le coût total des transitions partant d'un état s. Ce coût correspond au séjour total dans l'état s. Plus précisément, $E(s)$ (la probabilité de quitter s après t unités de temps) est $1 - \exp^{-E(s).t}$ à cause du fait que le minimum d'une distribution exponentielle (en compétition dans une *race condition*) est caractérisé par la somme de leurs coûts. Par conséquent, la probabilité de passer d'un état s à un état s' par une seule transition $P(s, s')$ est déterminé par le délai d'aller de s à s' avant les délais des autres noeuds sortants de s. Plus formellement, $P(s, s') = R(s, s')/E(s)$ sauf si s est un état absorbant, i.e., si $E(s) = 0$. Dans ce cas, $P(s, s') = 0$. La matrice P décrit une chaîne de Markov discrète dérivée de la chaîne de Markov à temps continu à l'étude.

Exemple 2.1.14. *Soit la chaîne de Markov à temps continu de la figure 2.3. Quelques probabilités de transitions sont* $P(s_0, s_3) = P(s_0, s_1) = 1/2$ *et* $P(s_1, s_2) = 1$.

A l'instar des DTMC, on peut définir pour les CTMC, une σ-algèbre sur $Paths(s)$ et une mesure de probabilité unique $Prob$ sur la plus petite σ-algèbre sur $Paths(s)$. Pour plus de détails sur les CTMC, nous référons le lecteur à [37, 44].

2.2 Spécification des propriétés

Dans cette section, nous décrivons les formalismes que nous utilisons pour spécifier les propriétés des modèles probabilistes. Dans le cas du model-checking, ces formalismes sont des logiques temporelles, en l'occurrence la logique PCTL qui est utilisée pour décrire le comportement des DTMC et des MDP, et la logique CSL qui est utilisée pour décrire le comportement des CTMC. Comme on l'a déjà souligné au début de ce chapitre, nous développons en détail la logique PCTL appliquée aux DTMC, puisque la structure globale de l'algorithme de model-checking est assez similaire dans les trois cas (PCTL appliquée aux DTMC, PCTL appliquée au MDP et CSL appliquée aux CTMC). Nous décrivons la syntaxe de PCTL ainsi que la sémantique de PCTL interprétée pour le cas des DTMC.

Parmi les logiques temporelles, on distingue deux catégories : les logiques temporelles linéaires et les logiques temporelles arborescentes.

Les logiques temporelles linéaires, comme LTL (Linear Temporal Logic) [39] introduite en 1977 par Pnueli, permettent d'exprimer des propriétés sur les chemins d'un système. Elles sont linéaires car leurs formules ne permettent pas d'exprimer l'existence de chemins qui se greffent à celui qui est considéré.

Une propriété comme "*à un état donné sur le chemin, il existe des chemins sortants de celui-ci qui satisfont telle propriété*" ne peut pas être exprimée en LTL. En effet, une fois que le chemin est fixé, nous ne pouvons pas exprimer la possibilité de branchement dans le temps sur ce chemin.

Les logiques temporelles arborescentes comme CTL et PCTL expriment des propriétés relatives aux états. Nous les interprétons sur des systèmes de transitions dont les exécutions sont représentées comme des arbres, la notion de séquence étant alors remplacée par la notion de branchement [35].

Les logiques temporelles se distinguent selon leur puissance d'expressivité, le type de propriété qu'elles expriment et selon le type de systèmes sur lesquels on peut les interpréter. Dans ce travail, nous ne faisons pas une comparaison entre les logiques temporelles linéaires et arborescentes ou entre PCTL et CTL. Nous référons le lecteur à [6, 35] pour une meilleure

comparaison des logiques utilisées.

2.2.1 La logique PCTL

PCTL (*Probabilistic Computational Tree Logic*) introduite par H. Hansson et B. Jonsson [19] est une extension de la logique temporelle CTL [6].

Définition 2.2.1. Syntaxe de PCTL
Soit AP un ensemble fini de propositions atomiques.

Formules d'états : $\Phi ::= true \mid a \mid \neg\Phi \mid (\Phi_1 \wedge \Phi_2) \mid P_{\sim p}(\varphi)$
Formules de chemins : $\varphi ::= \mathcal{X}\Phi \mid \Phi_1 \,\mathcal{U}\, \Phi_2 \mid \Phi_1 \,\mathcal{U}^{\leq k} \Phi_2$

où $a \in AP$, $\sim \in \{<, \leq, \geq, >\}$ et $p \in [0, 1]$.

Cette présentation s'appelle la présentation Backus-Naur [6]. Elle signifie que les formules de la logique PCTL sont construites récursivement à partir de cette syntaxe de la manière suivante :

1. Formules d'états
   a) a est une formule d'état.
   b) Si Φ est une formule d'état, alors $\neg\Phi$ est une formule d'état.
   c) Si Φ_1 et Φ_2 sont des formules d'états, alors $\Phi_1 \wedge \Phi_2$ est une formule d'état.
   d) Si φ est une formule de chemin, $P_{\sim p}(\varphi)$ est une formule d'état.
   e) Rien d'autre n'est une formule d'état.
2. Formules de chemin
   a) Si Φ est une formule d'état, alors $\mathcal{X}\Phi$ est une formule de chemin.
   b) Si Φ_1 et Φ_2 sont des formules d'états, alors $\Phi_1 \,\mathcal{U}\, \Phi_2$ est une formule de chemin.
   c) Si Φ_1 et Φ_2 sont des formules d'états et $k \in \mathbb{N}$, alors $\Phi_1 \,\mathcal{U}^{\leq k} \Phi_2$ est une formule de chemin.
   d) Rien d'autre n'est une formule de chemin.

Dans cette syntaxe, nous avons distingué les formules d'états Φ, des formules de chemins φ qui sont évaluées sur les états et les chemins respectivement. Une propriété du modèle sera toujours exprimée avec une formule d'état. Les formules de chemins apparaissent seulement comme des paramètres de l'opérateur probabiliste $P_{\sim p}(\varphi)$. Pour construire les formules d'états à partir des formules de chemins, PCTL utilise cet opérateur. Intuitivement, un état s satisfait la formule $P_{\sim p}(\varphi)$ si la probabilité de prendre un chemin à partir de s qui satisfait φ est comprise dans l'intervalle défini par $\sim p$.

Les opérateurs booléens *false* et $\vee$ sont déduits à partir des opérateurs de la négation et de la conjonction. L'opérateur $\mathcal{X}$ (à lire *next*) exprime la notion de "*prochain état*" et l'opérateur $\mathcal{U}$

(à lire *until*) exprime la notion de "*jusqu'à ce que*". Ainsi, une formule de chemin de la forme $\mathcal{X}\Phi$ est satisfaite sur un chemin si au 2è état du chemin, Φ est satisfaite.

Une formule de chemin de la forme $\Phi_1 \mathcal{U} \Phi_2$ est satisfaite sur un chemin si, à partir d'un certain état s, Φ_2 est satisfaite et tous les états précédant s satisfont Φ_1. Plus intuitivement, cela signifie que l'on satisfait Φ_1 jusqu'à ce que l'on atteigne un état à partir duquel, on satisfait Φ_2. Une formule de chemin de la forme $\Phi_1 \mathcal{U}^{\leq k} \Phi_2$ est une généralisation de la forme $\Phi_1 \mathcal{U} \Phi_2$ à part que l'on exige que Φ_2 soit satisfaite après k-étapes.

Exemple 2.2.2. *Soit $a, b \in AP$, φ_1, φ_2, des formules de chemins.*

- *Les formules $P_{\geq 1/2}(\mathcal{X} P_{\geq 1/3}(\mathcal{X} a))$, $P_{\geq 1/2}(true \mathcal{U} a)$ et $a \wedge b$ sont des formules correctes en PCTL car elles respectent la syntaxe PCTL.*
- *Les formules $P_{\geq 1/2}(\varphi_1 \mathcal{U} \varphi_2)$, $P_{\geq 1/2}(\mathcal{X} \varphi_1)$ et $\varphi_1 \wedge \varphi_2$ ne sont pas conformes à la syntaxe de PCTL car elles appliquent les opérateurs $\mathcal{X}$,$\mathcal{U}$ et $\wedge$ à des formules de chemins à la place de formules d'états.*

La logique PCTL dérive de CTL. Bien que nous n'allons pas l'utiliser par la suite, nous décrivons la syntaxe de CTL afin de pouvoir faire une comparaison rapide entre les deux logiques.

Définition 2.2.3. Syntaxe de CTL
Soit AP un ensemble fini de propositions atomiques.

Formules d'états : $\Phi ::= true \mid a \mid \neg\Phi \mid (\Phi_1 \wedge \Phi_2) \mid \exists\varphi \mid \forall\varphi$
Formules de chemins : $\varphi ::= \mathcal{X}\Phi \mid \Phi_1 \mathcal{U} \Phi_2$

PCTL et CTL diffèrent par leurs opérateurs sur les formules de chemins. Pour construire les formules d'états à partir de formules de chemin, la logique PCTL utilise l'opérateur probabiliste $P_{\sim p}(\phi)$, alors que la logique CTL utilise les quantificateurs de chemin $\forall$ (pour toute exécution) et $\exists$ (il existe une exécution). Il existe donc des formules PCTL qui ne sont pas dans CTL et inversement. Par exemple, la formule CTL $\forall\phi$ qui signifie que tous les chemins partant d'un état donné satisfont ϕ ne peut être exprimée en PCTL. Inversement, les formules probabilistes PCTL de la forme $P_{\sim p}(\phi)$ ne peuvent pas être exprimées en CTL. Ces deux logiques sont donc incomparables au niveau de l'expressivité [6, 35].

2.2.2 Sémantique de PCTL pour les DTMC

Soient un DTMC $M = (S, s_0, P, L)$, un état $s \in S$ et une formule PCTL Φ. On écrit $s \models \Phi$ pour indiquer que Φ est satisfaite dans s, c'est à dire que Φ est vraie dans s. Un modèle M satisfait une formule Φ si Φ est satisfaite dans son état initiale. On note par $Sat(\Phi)$, l'ensemble

$\{s \in S | s \models \Phi\}$, qui est l'ensemble des états qui satisfont Φ. Par similarité, pour une formule de chemin φ, on écrit par $s \models \varphi$.

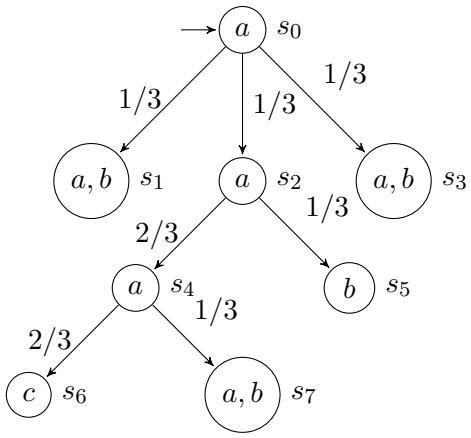


FIGURE 2.4: Exemple d'un modèle probabiliste.

Définition 2.2.4. Sémantique de PCTL

Sémantique pour les formules d'états :

$s \models true$	*pour tout* s
$s \models a$	*ssi* $a \in L(s)$
$s \models \neg\Phi$	*ssi* $s \not\models \Phi$
$s \models (\Phi_1 \wedge \Phi_2)$	*ssi* $s \models \Phi_1$ *et* $s \models \Phi_2$
$s \models P_{\sim p}(\varphi)$	*ssi* $Prob(\{\pi \in Paths(s) \mid \pi \models \varphi\}) \sim p$

Sémantique pour les formules de chemins :

$\pi \models \mathcal{X}\Phi$	*ssi* $\pi = s_0 s_1 s_2 ...$ *et* $s_1 \models \Phi$
$\pi \models \Phi_1 \mathcal{U} \Phi_2$	*ssi* $\pi = s_0 s_1 s_2 ...$ *et* $\exists k$ *tq.* $s_k \models \Phi_2$ *et* $\forall j < k, s_j \models \Phi_1$
$\pi \models \Phi_1 \mathcal{U} \leq^k \Phi_2$	*ssi* $\exists i \leq k$ *tq.* $s_i \models \Phi_2 \wedge s_j \models \Phi_1, \forall j < i$

Les opérateurs booléens *true*, $\wedge$ et la négation sont interprétés de la façon habituelle. Par exemple, la formule $\Phi_1 \wedge \Phi_2$ est valide dans un état s'il satisfait à la fois Φ_1 et Φ_2.

La formule $P_{\sim p}(\varphi)$ est satisfaite dans un état s, si et seulement si $Prob(\{\pi \in Paths(s) \mid \pi \models \varphi\})$ est comprise dans l'intervalle défini par $\sim p$. Par exemple $P_{\geq p}(\mathcal{X}\Phi)$ est satisfaite par le modèle M si la probabilité de l'ensemble des chemins partant de l'état initial s_0 et dont le prochain état s_1 satisfait Φ, est supérieure ou égale à p. Le calcul de la probabilité d'un ensemble de chemins partant d'un état s a été développée à la section 2.1.

Exemple 2.2.5. *Soit le modèle probabiliste de la figure 2.4. Nous voulons savoir si ce modèle satisfait la formule* $P_{\geq 2/3}(\mathcal{X}(a \wedge b))$*, autrement dit si l'état initial* $s_0 \models P_{\geq 2/3}(\mathcal{X}(a \wedge b))$*. Cela revient à vérifier si il y'a des transitions partant de* s_0 *pour lesquelles le successeur de* s_0

satisfait $a \wedge b$. On vérifie que la probabilité de l'ensemble de ces chemins est supérieure ou égale à 2/3. Parmi les successeurs de s_0 (s_1, s_2, s_3), seuls $s1$ et s_3 satisfont $a \wedge b$.

La probabilité de l'ensemble des chemins partant de s_0 et incluant $s1$ et s_3 est donnée par :

$$\begin{aligned}
Prob\{\pi \in Paths(s_0) | \pi \models \mathcal{X}(a \wedge b) &= Prob\{\pi \in Paths(s_0) | \pi = s_0 t_1 t_2 ... \text{ et } t_1 \models (a \wedge b)\} \\
&= Prob\{\pi \in Paths(s_0) | \pi = s_0 t_1 t_2 ... \text{ et } t_1 \in \{s_1, s_3\}\} \\
&= Prob(B(s_0 s_1) \cup B(s_0 s_3)) \\
&\quad < \text{comme } B(s_0 s_1) \text{ et } B(s_0 s_3) \text{ sont disjoints} > \\
&= Prob(B(s_0 s_1)) + Prob(B(s_0 s_3)) \\
&= P(B(s_0 s_1)) + P(B(s_0 s_3)) \\
&= 1/3 + 1/3 = 2/3.
\end{aligned}$$

On conclut que la formule $P_{\geq 2/3}(\mathcal{X}(a \wedge b))$ est satisfaite par le modèle M. Plus généralement, la probabilité $Pr(s, \mathcal{X}\Phi)$ pour un état s d'emprunter un chemin qui satisfait la formule $\mathcal{X}\Phi$ est donnée par :

$$Pr(s, \mathcal{X}\Phi) = \sum_{s_i \in Sat(\Phi)} P(s, s_i).$$

où $Sat(\Phi)$ désigne l'ensemble des états qui satisfont la formule Φ. Déterminer si un état satisfait une formule de la forme $P_{\sim p}(\Phi_1 \mathcal{U} \Phi_2)$ est beaucoup plus complexe. Nous détaillons dans la section qui suit, l'algorithme utilisé pour déterminer si un état s satisfait une formule d'état, et ce, pour n'importe quelle formule d'état.

2.3 Algorithme de model-checking

Dans les sections précédentes, nous avons choisi les chaînes de Markov à temps discret comme notre modèle de système probabiliste et la logique PCTL pour la description des propriétés désirées. Dans cette section, nous décrivons comment calculer l'ensemble des états qui satisfont une formule PCTL donnée.

Un algorithme de model-checking pour la logique PCTL ou CSL prend en entrée un modèle probabiliste M approprié (DTMC ou MDP pour PCTL et CTMC pour CSL), une formule Φ de la logique et retourne l'ensemble $Sat(\Phi)$ contenant l'ensemble des états qui satisfont Φ.

La structure globale de l'algorithme de model-checking PCTL pour DTMC (de même que PCTL pour MDP et CSL pour CTMC) dérive de l'algorithme de model-checking CTL présenté en [6]. L'algorithme consiste à :

1. Construire l'arbre syntaxique de la formule Φ,

2. Traverser récursivement l'arbre syntaxique de bas vers le haut (des feuilles vers la racine), en déterminant l'ensemble $Sat(\phi) = \{s \in S | s \models \phi\}$, pour chaque sous-formule ϕ de Φ.

L'arbre syntaxique d'une formule Φ est un arbre dont chaque noeud est une sous-formule de Φ et la racine de l'arbre, la formule Φ elle même. Les feuilles de l'arbre sont des propositions atomiques ou l'opérateur booléen *true* pour le cas de PCTL. Nous donnons un exemple de l'exécution de l'algorithme sur une formule simple (qui ne contient pas l'opérateur probabiliste) avant de décrire en détail son implémentation.

Exemple 2.3.1. *Soient le modèle probabiliste de la figure 2.5 qui est le même que celui de la figure 2.4 et l'arbre syntaxique de* $\Phi = a \vee (b \wedge \neg c)$ *décrit par la figure 2.6.*

Nous voulons déterminer l'ensemble $Sat(\Phi) = \{s \in S | s \models \Phi\}$.

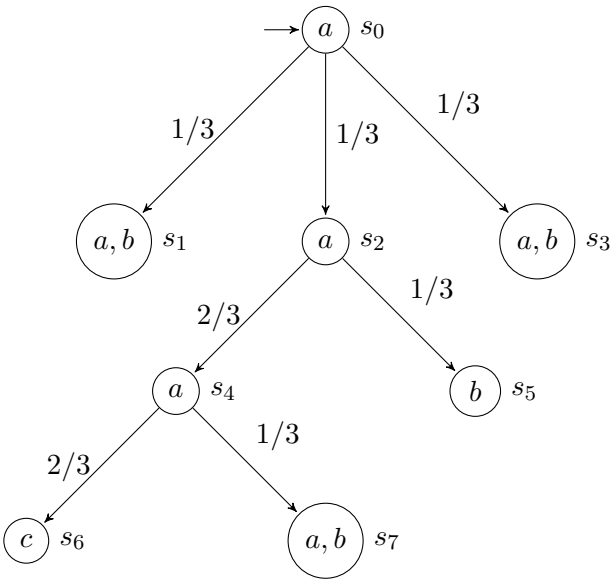


FIGURE 2.5: Un modèle probabiliste.

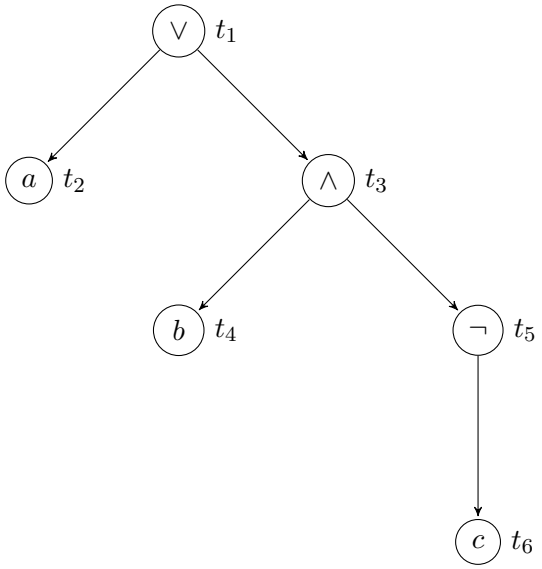


FIGURE 2.6: Arbre syntaxique de $\Phi = a \vee (b \wedge \neg c)$.

– 1ère étape : Nous débutons le parcours de l'arbre syntaxique par la feuille t_6. Désignons par ϕ_1 la sous-formule représentée à ce niveau ($\phi_1 = c$). A cette étape, $Sat(\phi_1) = Sat(c) = \{s_6\}$.

– 2ème étape : Nous remontons au noeud t_5 et désignons par ϕ_2 la sous-formule représentée par ce noeud. L'ensemble $Sat(\phi_2)$ des états qui satisfont ϕ_2 est donné par : $Sat(\phi_2) = Sat(\neg\phi_1) = S \backslash Sat(\phi_1) = S \backslash \{s_6\}$.

– 3ème étape : Nous remontons à la feuille t_4 et désignons par ϕ_3 la sous-formule représentée par ce noeud. Comme pour t_6, $Sat(\phi_3) = Sat(b) = \{s_1, s_3, s_5, s_7\}$.

– 4ème étape : Nous remontons au noeud t_3 et désignons par ϕ_4 la sous-formule représentée par ce noeud. Il s'agit d'une conjonction des sous-formules correspondant au noeud t_5 et de la feuille t_4. Cet ensemble que nous désignons par $Sat(\phi_4)$ est donné par :

$$\begin{aligned} Sat(\phi_4) &= Sat(\phi_2 \wedge \phi_3) \\ &= Sat(\phi_2) \cap Sat(\phi_3) \\ &= ((S \backslash \{s_6\}) \cap (\{s_1, s_3, s_5, s_7\})) \\ &= \{s_1, s_3, s_5, s_7\}. \end{aligned}$$

– 5ème étape : Nous remontons à la feuille t_2 et désignons par ϕ_5 la sous-formule représentée par ce noeud. $Sat(\phi_5) = Sat(a) = \{s_0, s_1, s_2, s_3, s_4, s_7\}$.

– 6ème étape : Nous remontons à la racine t_1 de notre arbre et désignons par ϕ_6 la sous-formule représentée par ce noeud. Il s'agit d'une disjonction entre les sous-formules de la feuille t_2 et le noeud t_3.

$$\begin{aligned} Sat(\phi_6) &= Sat(\Phi) \\ &= Sat(\phi_4 \vee \phi_5) \\ &= Sat(\phi_4) \cup Sat(\phi_5) \\ &= (\{s_1, s_3, s_5, s_7\}) \cup (\{s_0, s_1, s_2, s_3, s_4, s_7\})) \\ &= S \backslash \{s_6\}. \end{aligned}$$

Quand nous atteignons la racine de notre arbre, l'algorithme s'arrête. Nous concluons que pour notre modèle, $Sat(\Phi = a \vee (b \wedge \neg c)) = S \backslash \{s_6\}$.

En général, après avoir construit l'arbre syntaxique d'une formule PCTL, calculer l'ensemble $Sat(\phi) = \{s \in S | s \models \phi\}$, pour chaque sous formule ϕ se fait de la manière suivante : a est une proposition atomique ; Φ, Φ_1 et Φ_2, des formules PCTL d'états ; φ, une formule PCTL de chemin ; et $p \in [0, 1]$.

$$\begin{aligned} Sat(true) &= S \\ Sat(a) &= \{s \in S | a \in L(s)\} \\ Sat(\neg \Phi) &= S \backslash Sat(\Phi) \\ Sat((\Phi_1 \wedge \Phi_2)) &= Sat(\Phi_1) \cap Sat(\Phi_2) \\ Sat(P_{\sim p}(\varphi)) &= \{s \in S | Pr(s, \varphi) \sim p\}. \end{aligned}$$

Avec $Pr(s, \varphi)$ qui représente $Prob(\{\pi \in Paths(s) | \pi \models \varphi\})$.

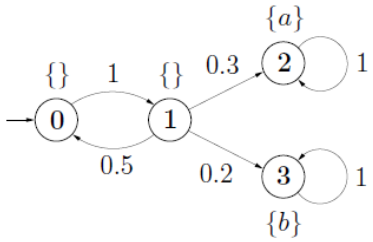
Comme nous l'avons montré dans l'exemple précédent 2.8, trouver $Sat(\Phi)$ pour le cas des opérateurs booléens est assez facile. Par conséquent, la principale tâche du model-checking probabiliste consiste à déterminer $Pr(s, \varphi)$, dépendamment du quantificateur de chemins($\mathcal{X}$,$\mathcal{U}^{\leq k}$,$\mathcal{U}$). Dans ce qui suit, nous distinguons les trois cas.

Le quantificateur *next* ($\mathcal{X}$)

On rappelle que pour un DTMC $M = (S, s_0, P, L)$ donné, la fonction P peut être considérée comme une matrice $|S| \times |S|$ des transitions de probabilité où chaque élément $P(s, t)$ représente la probabilité de faire la transition de l'état s vers t. Il est facile de voir que $Pr(s, \mathcal{X}\Phi) = \sum_{s_i \in Sat(\Phi)} P(s, s_i)$. Posons le vecteur indexé b_Φ des états de S, qui est tel que $b_\Phi(s) = 1$, si $s \models \Phi$ et 0 sinon. On peut calculer le vecteur $p(\mathcal{X}\Phi)$ qui représente la probabilité que chaque état satisfasse Φ comme suit :

$$p(\mathcal{X}\Phi) = P \cdot b_\Phi.$$

Ce qui nécessite une seule multiplication matrice-vecteur.



$$\mathbf{P} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0.5 & 0 & 0.3 & 0.2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad b_\Phi = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 1 \end{pmatrix}$$

FIGURE 2.7: Un DTMC et sa matrice des transitions de probabilité. [37]

Exemple 2.3.2. *Considérons le DTMC de la figure 2.7, sa matrice des transitions de probabilité P, et le vecteur-indexé b_Φ pour la formule $\Phi = a \vee b$. On peut voir par exemple dans b_Φ, que seul les états 2 et 3 de notre DTMC satisfont Φ.*

Le vecteur indexé $p(\mathcal{X}\Phi)$ des états représentant la probabilité avec laquelle les états de M satisfont la formule $\mathcal{X}(a \vee b)$ est donné par :

$$\begin{aligned} p(\mathcal{X}\Phi) &= P \cdot b_\Phi \\ &= \begin{pmatrix} 0 \\ 0.5 \\ 1 \\ 1 \end{pmatrix} \end{aligned}$$

On peut en conclure que dans notre modèle, l'état 0 satisfait $\mathcal{X}(a \vee b)$, avec une probabilité nulle ; l'état 1, avec une probabilité égale à 0.5 ; l'état 2 et 3 avec une probabilité égale à 1.

Le quantificateur *until k-steps* ($\mathcal{U}^{\leq k}$)

Pour calculer $Pr(s, \varphi = \Phi_1 \mathcal{U}^{\leq k} \Phi_2)$, on divise l'ensemble de tous les états S en trois ensembles disjoints S^0, S^1 et $S^?$:

$$\begin{aligned} S^0 &= S \backslash (Sat(\Phi_1) \cup Sat(\Phi_2)) \\ S^1 &= Sat(\Phi_2) \\ S^? &= S \backslash (S^{no} \cup S^{yes}) \end{aligned}$$

Les ensembles S^0, S^1 contiennent respectivement l'ensemble des états pour lesquels $Pr(s, \varphi)$ est égal à 0 ou 1. La tâche principale consiste à déterminer la probabilité $Pr(s, \varphi)$ pour les états dans $S^?$.

Pour ces états, on a que :

$$Pr(s, \Phi_1 \mathcal{U}^{\leq k} \Phi_2) = \begin{cases} 0 & si\ k = 0 \\ \sum_{s' \in S} P(s, s') \cdot Pr(s, \Phi_1 \mathcal{U}^{\leq k-1} \Phi_2) & si\ k \geq 1 \end{cases}$$

Si on définit la matrice P' comme suit :

$$P'(s, s') = \begin{cases} P(s, s') & si\ s \in S^? \\ 1 & si\ s \in S^1\ \wedge\ s = s' \\ 0 & autrement \end{cases}$$

L'idée derrière la construction de la matrice P' est détaillée dans la section suivante pour le cas de l'opérateur $\mathcal{U}$ à la différence près que pour le cas présent de l'opérateur $\mathcal{U}^{\leq k}$, si $s \in S^1$, on ignore les transitions sortant de s car on n'a pas à calculer les états qui sont atteints à partir de s.

Nous voulons calculer le vecteur des probabilités x_k où chaque entrée $x_k(s) = Pr(s, \Phi_1 \mathcal{U}^{\leq k} \Phi_2)$.

Pour $k = 0$, le vecteur x_0 est donné par :

$$x_0(s) = \begin{cases} 1 & si\ s \in S^1 \\ 0 & autrement \end{cases}$$

Pour $k > 0$, le vecteur x_k est calculé itérativement : $x_k = P' \cdot x_{k-1}$.

Au total, calculer la probabilité avec laquelle chaque état de notre modèle satisfait une formule de la forme $\Phi_1 \mathcal{U}^{\leq k} \Phi_2$ nécessite k multiplications matrice-vecteur.

Le quantificateur *until* ($\mathcal{U}$)

Comme pour le cas de l'opérateur précédent $\mathcal{U}^{\leq k}$, on commence par déterminer les ensembles S^0 et S^1. Les ensembles S^0 et S^1 sont calculés par deux algorithmes PROB0 et PROB1 présentés par Baier et al. [6] et Parker [37]. Ces algorithmes, qui constituent la partie pré-calcul de l'algorithme de model-checking, sont d'autant plus intéressants qu'ils permettent de déterminer la probabilité exacte des états dans S^0 et S^1. L'ensemble $S^?$ est déterminé comme précédemment, $S^? = S \backslash (S^0 \cup S^1)$. Il reste à calculer $Pr(s, \Phi_1 \,\mathcal{U}\, \Phi_2)$ pour les éléments de $S^?$. Ce qui peut être fait en résolvant le système d'équations linéaires avec les variables $x(s)$ pour $s \in S$:

$$x(s) \quad = \quad \begin{cases} 0 & si\ s \in S^0 \\ 1 & si\ s \in S^1 \\ \sum_{s' \in S} P(s,s') \cdot x(s') & si\ s \in S^? \end{cases}$$

La variable $x(s)$ correspond à la probabilité $Pr(s, \Phi_1 \,\mathcal{U}\, \Phi_2)$ que nous recherchons. Pour chaque $s \in S^?$, le système linéaire précédent stipule que $x(s) = \sum_{s' \in S} P(s,s') \cdot x(s')$ qui s'écrit sous forme vectorielle $x = Px$. Pour tirer profit du fait que pour les états dans S^0 et S^1, $x(s)$ est égal à 0 et 1 respectivement, on transforme la matrice de transitions P en extrayant les états qui satisfont Φ_2 et on obtient $x = P'x + b$ où P' est une matrice tirée de P comme suit :

$$P'(s,s') \quad = \quad \begin{cases} P(s,s') & si\ s \in S^? \\ 0 & autrement \end{cases}$$

et b, un vecteur-colonne sur les états avec $b(s) = 1$ si $s \in S^1$ et 0 autrement. On note que l'équation $x = P'x + b$ est équivalente à $x = Px$ puisque :

$$\begin{aligned} x = Px \Leftrightarrow x(s) \quad &= \quad \sum_{s' \in S^?} P(s,s') \cdot x(s') + \sum_{s' \in S^1} P(s,s') \cdot 1 + \sum_{s' \in S^0} P(s,s') \cdot 0 \\ x(s) \quad &= \quad \sum_{s' \in S^?} P(s,s') \cdot x(s') + \sum_{s' \in S^1} P(s,s') \end{aligned}$$

Comme $x = P'x + b$, on a que :

$$\begin{aligned} x \quad &= \quad P'x + b \\ x - P'x \quad &= \quad b \\ (I - P')x \quad &= \quad b \end{aligned}$$

Pour réécrire notre système d'équations dans la forme traditionnelle $Ax = b$, on pose $A = I - P'$ où I est la matrice identité. Ce système d'équations $Ax = b$ peut être résolu par n'importe quelle approche standard comme l'élimination gaussienne, ou par une méthode itérative comme la méthode de Jacobi ou la méthode de Gauss-Seidel. Puisqu'il s'agit de faire le model-checking pour de larges systèmes, les méthodes itératives sont le plus souvent utilisées. Nous reviendrons dans les sections qui suivent sur les raisons qui motivent ce choix. Nous décrivons aussi en détail certaines méthodes itératives utilisées dans le chapitre suivant, dans la section 3.2.

Voyons maintenant un exemple pour illustrer l'algorithme qui détermine si un état s vérifie une formule $\Phi_1 \mathcal{U} \Phi_2$.

Exemple 2.3.3. *Soit le DTMC de la figure 2.8. Nous voulons déterminer l'ensemble des états qui satisfont la formule PCTL* $P_{\geq 1/2}(\neg b \mathcal{U} a)$. *Pour cela, nous devons calculer* $Pr(s, \neg b \mathcal{U} a)$ *pour chaque* $s \in \{0, 1, 2, 3\}$. *Puisque* a *est vrai dans l'état* 2, *et que* a *et* $\neg b$ *ne sont pas vrais dans l'état* 3, *nous déduisons que* $S^{no} = \{3\}, S^{yes} = \{2\}$ *et* $S^{?} = \{0, 1\}$. *Dans le cas présent, la matrice* P' *correspond à :*

$$P' = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0.5 & 0 & 0.3 & 0.2 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

tandis que la matrice A *et le vecteur* b *correspondent à :*

$$A = I - P' = \begin{pmatrix} 1 & -1 & 0 & 0 \\ -0.5 & 1 & -0.3 & -0.2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad \textit{et} \qquad b = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}$$

Les probabilités dans $S^{?}$ *peuvent être calculées en résolvant le système d'équations linéaires suivant :*

$$\begin{aligned} x_0 &= x_1 \\ x_1 &= 0.5x_0 + 0.3x_2 + 0.2x_3 \\ x_2 &= 1 \\ x_3 &= 0 \end{aligned}$$

Ce qui donne le résultat $(x_0, x_1, x_2, x_3) = (0.6, 0.6, 1, 0)$. Nous en déduisons que la formule $P_{\geq 1/2}(\neg b \vee a)$ est satisfaite seulement dans les états 0, 1, et 2.

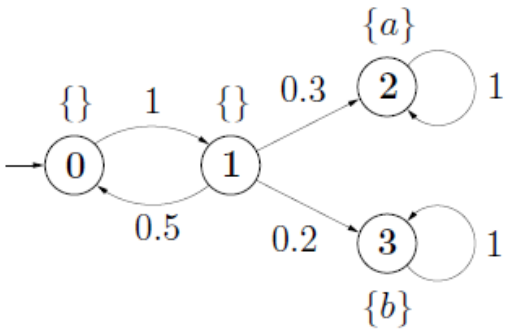


FIGURE 2.8: Un modèle probabiliste.

2.4 Model-checking avec PRISM

Le model-checking des systèmes probabilistes fait donc référence à un ensemble de techniques visant à déterminer la probabilité d'atteindre un certain ensemble d'états pour des systèmes présentant un comportement probabiliste.

L'ensemble des états pourrait représenter certains "*mauvais*" états qui devraient être visités avec une faible probabilité, ou inversement un ensemble de "*bons*" états qui devraient être visités plus fréquemment. Plusieurs outils ont été développés pour permettre de faire le model-checking probabiliste automatique, comme MRMC [26], CISMO [36, 40] et PRISM [31, 37, 29] entre autres.

PRISM, qui est l'outil qui nous intéresse dans le cadre de ce travail, est un model-checker probabiliste développé initialement à l'université de Birmingham et actuellement maintenu par l'université d'Oxford. PRISM supporte le model-checking pour les trois types de modèles probabilistes décrits dans la section 2.1 à savoir les DTMC, les MDP et les CTMC. En entrée, l'outil prend deux paramètres, une description du modèle dans le langage PRISM et une liste de spécification du modèle soit en PCTL (pour les DTMC et MDP), soit en CSL (pour les CTMC). [2] Le langage PRISM est une variante du formalisme pour systèmes réactifs de Alur et Henzinger [3]. Pour une meilleure compréhension de ce langage, nous référons le lecteur à [37] et [29].

La figure 2.9 décrit le mode de fonctionnement de PRISM. PRISM construit d'abord le modèle suivant la description reçue (soit DTMC, MDP et CTMC), calcule l'ensemble d'états atteignables et identifie les possibilités de blocage. Il réalise ensuite un model-checking du système

2. PRISM supporte aussi d'autres modèles probabilistes à savoir les automates probabilistes PA (de l'anglais Probabilistic Automata) et les automates temporisés probabilistes (Probabilistic Timed Automata) plus une extension des modèles susmentionnés avec coûts et récompenses.

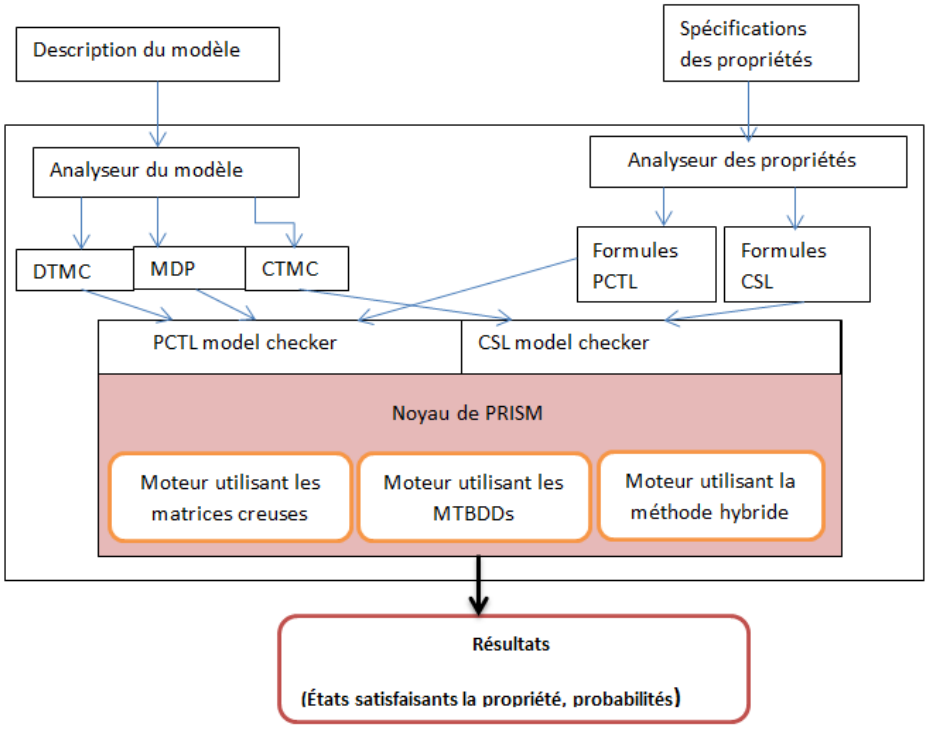


FIGURE 2.9: Architecture du système de PRISM [31]

pour déterminer quels états du modèle satisfont chaque spécification donnée.

Comme décrit à la section 2.3, le model-checking probabiliste se réduit à deux types de calculs : *l'analyse qualitative* qui consiste à déterminer si un état satisfait ou non une propriété, c'est-à-dire quand il s'agit de vérifier une propriété qui ne fait pas référence à l'opérateur probabiliste $P_{\sim p}$ (ou si p égal à 1 ou est égal à 0), et l'*analyse quantitative* qui sert à déterminer la probabilité d'un état s de vérifier une propriété φ donnée ($Pr(s, \varphi)$).

La principale tâche pour le model-checking probabiliste correspond à l'analyse quantitative. Si φ est une formule de chemin de la forme $\mathcal{X}\Phi$, nous avons vu que cela nécessite une multiplication matrice-vecteur. Si φ est de la forme $\Phi_1 \mathcal{U}^{\leq k} \Phi_2$, cela nécessite k multiplications matrice-vecteur. Et si φ est de la forme $\Phi_1 \mathcal{U} \Phi_2$, cela se réduit à résoudre un système d'équations linéaire de la forme $Ax = b$.

Ce système d'équations peut être résolu en utilisant les méthodes dites directes ou standards, comme l'élimination gaussienne, ou une méthode itérative comme la méthode de Jacobi ou la méthode de Gauss-Seidel [37]. Dans le cas du model-checking avec PRISM, puisqu'on fait le model-checking des systèmes qui peuvent avoir un très grand nombre d'états, on utilise les méthodes itératives, puisque les méthodes directes sont coûteuses en terme d'espace et de

temps.

Sur la figure 2.9, on peut voir que PRISM offre trois outils qui peuvent être utilisés indépendamment pour les calculs numériques à effectuer pour le model-checking des systèmes probabilistes, l'un utilisant les matrices creuses, l'autre utilisant les MTBDD qui sont une variante des BDD et enfin l'outil utilisant la méthode hybride qui combine les deux techniques. Dans ce qui suit, nous présentons chaque technique, sa méthode d'implantation du modèle-checking, ses forces et faiblesses, et une comparaison de leur efficacité.

2.4.1 Systèmes probabilistes à matrice de transition creuse

Deux des modèles probabilistes que nous considérons sont décrits par des matrices à valeurs, à savoir les DTMC et CTMC, réelles. Ces matrices sont souvent très larges, mais elles contiennent un nombre relativement faible d'éléments non nuls. De telles matrices avec peu d'entrées non nulles sont dites *creuses*. Pour exploiter la particularité que ces matrices contiennent peu d'entrées non nulles (et ainsi sauver de l'espace), seules les entrées non nulles de la matrice sont explicitement conservées.

Supposons que l'on veut conserver une matrice P représentant les transitions de probabilité d'un système probabiliste. Si S est l'ensemble d'états, P est de taille $|S| \times |S|$, et les indices de P vont de 0 à $|S| - 1$. On conserve seulement pour les éléments non nuls, l'indice de la ligne, l'indice de la colonne ainsi que la valeur de l'élément. Ces informations sont conservées dans 3 tableaux, val, row et col. Les tableaux val et col conservent les éléments non nuls de la matrice ainsi que leurs indices de colonne respectifs. Le tableau row nous indique à quelles rangées appartiennent les éléments non nuls conservés : $row[n]$ correspond à l'indice de colonne dans val et col de la première valeur non nulle sur la rangée n.

La figure 2.10 montre une matrice 4×4 et sa représentation à l'aide des tableaux val, col et row. La valeur d'une entrée arbitraire (r, c) dans la matrice peut être déterminée en vérifiant les

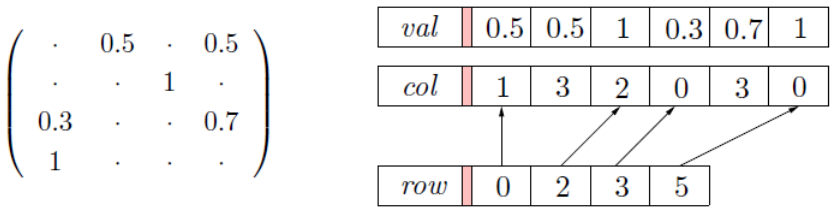


FIGURE 2.10: Matrice de transition et sa représentation par les matrices creuses. [37]

r-ième et $(r+1)$-ième entrées dans row et on peut utiliser ces valeurs dans col pour déterminer si la valeur c existe entre $col[row[r]]$ et $col[row[r+1]-1]$ inclusivement. Si c n'existe pas, alors $(r, c) = 0$. Sinon, l'entrée (r, c) est non nulle et on peut retrouver sa valeur dans le tableau val.

Dans ce cas, si i est l'indice de c dans le tableau col avec $col[row[r]] \leq i \leq col[row[r+1]-1]$, alors $(r, c) = val[i]$.

Exemple 2.4.1. *Soit la matrice de transitions de la figure 2.10 que nous désignons par P et sa représentation à l'aide des tableaux val, row et col. Nous voulons déterminer la valeur des entrées $P(2,1)$ et $P(2,3)$. Pour l'entrée $P(2,1)$, $col[row[2]] = col[3]$ et $col[row[3]] = col[5]$. Comme la valeur 1 n'apparaît pas entre $col[3]$ et $col[5-1]$ ($col[3] = 0$ et $col[4] = 3$), nous déduisons que l'entrée $P(2,1) = 0$. Pour l'entrée $P(2,3)$, nous faisons la même procédure et trouvons que la valeur 3 apparaît entre $col[3]$ et $col[4]$ en l'occurrence $col[4] = 3$. La valeur de l'entrée $P(2,3)$ correspond donc à la valeur de $val[4]$, c'est-à-dire $P(2,3) = val[4] = 0.7$.*

L'exemple de la figure 2.10 ne montre pas clairement le gain de mémoire puisque les tableaux *val*, *col* et *row* totalisent en tout 16 entrées soit autant que la matrice que l'on veut sauvegarder. Cependant, cette technique peut permettre de réduire significativement le nombre d'éléments à sauvegarder. En effet, si e est le nombre d'entrées non nulles dans la matrice de taille $n \times n$, la technique conserve $2e + n$ éléments. La technique est donc rentable si $2e + n < n^2$. Ce qui est le cas si e est relativement petit c'est-à-dire si notre matrice comporte peu d'entrées non nulles.

```
SparseMVMult(row, col, val, b)
1.  for (i := 0 ... |S|−1)
2.      res[i] := 0
3.      l := row[i]
4.      h := row[i + 1]−1
5.      for (j := l ... h)
6.          res[i] := res[i] + val[j] × b[col[j]]
7.      endfor
8.  endfor
9.  return res
```

FIGURE 2.11: Multiplication d'une matrice creuse par un vecteur. [37]

Model checking des systèmes avec une matrice de transition creuse

Pour pouvoir faire le model checking des systèmes ayant une matrice de transition creuse, nous devons pouvoir faire une multiplication d'une matrice A avec un vecteur b (quand la matrice A est représentée avec les 3 tableaux *val*, *col* et *row*).

Même si nous venons de montrer qu'on peut obtenir une entrée arbitraire de la matrice de transition, nous devons pouvoir le faire pour toutes les entrées, du moins les entrées non nulles.

L'algorithme **SparseMVMult** de la figure 2.11 permet de faire une telle multiplication. La matrice de transition A est stockée dans les tableaux val, col et row, le vecteur b ainsi que le vecteur résultat de $A \cdot b$ sont stockés respectivement dans b et res. L'algorithme prend en entrée les tableaux val, col, row et b et retourne le vecteur res.

2.4.2 Approche basée sur les MTBDD

Diagrammes de décision binaire à terminaux multiples (MTBDD)

Les diagrammes de décision binaire à terminaux multiples [31, 36, 43] ou MTBDD (de l'anglais *Multi-Terminal Binary Decision Diagrams*) sont une généralisation des diagrammes de décision binaire ordonnés [31, 36, 43] ou OBDD (de l'anglais *Ordered Binary Decision Diagram*) où des valeurs différentes de 0 ou 1 sont permises pour les feuilles. Nous présentons une brève description des OBDD avant de décrire les MTBDD.

Définition 2.4.2. *Soient S un ensemble d'états, $V = \{x_1, x_2, \cdots, x_n\}$ un ensemble non vide de variables booléennes, $<$ un ordre total sur les variables de V tel que $x_1 < x_2 < \cdots < x_n$, $var : S \to V$ une fonction qui associe à un sommet s une variable booléenne dans V et $val : S \to \mathbb{B}$ une fonction qui associe à un sommet s une valeur booléenne.*

Un OBDD sur $\langle V, < \rangle$ est un graphe acyclique orienté avec un ensemble non vide S de sommets qui sont de deux types :

- *des sommets terminaux s ou feuilles, étiquetés par une valeur booléenne $val(s) \in \mathbb{B}$,*
- *des sommets non terminaux s, étiquetés par une variable booléenne, $var(s) \in V$, qui ont deux enfants $left(s)$ et $right(s)$ tels que pour tout sommet t :*

$$t \in \{left(s), right(s)\} \Rightarrow (var(s) < var(t)) \text{ ou } (t \text{ est terminal}).$$

La deuxième condition exige que chaque variable soit rencontrée au plus une seule fois et dans le même ordre pour tous les chemins qui commencent à la racine et qui se terminent à une feuille. C'est pour cette raison qu'on dit que le BDD est ordonné (OBDD).

Un OBDD sur un ensemble ordonné de variables booléennes $x_1 < \cdots < x_n$ représente en fait une fonction $f(x_1, \cdots, x_n) : \mathbb{B}_n \to \mathbb{B}$ définie sur ces variables. La figure 2.12 représente un OBDD qui correspond à la fonction :

$$f(x_1, x_2, x_3) = \overline{x_1 x_2 x_3} + x_1 x_2 + x_2 x_3$$

En 1986, Bryant [10] a ajouté trois contraintes à la définition des OBDD. Ces trois contraintes sont appelées les règles de réduction. Un OBDD réduit est noté ROBDD, de l'anglais *Reduced* OBDD. Grâce aux règles de réduction, les OBDD possèdent des propriétés qui en font une structure de donnée efficace quand il s'agit de stocker et manipuler les données. En effet, grâce

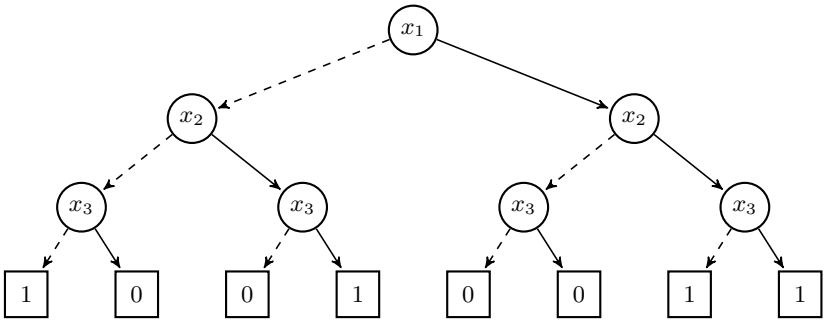


FIGURE 2.12: Un diagramme de décision binaire ordonné.

à ces règles, ils sont stockés de manière réduite, mais cela crée d'autres avantages. Pour un ordre de variables donné $x_1 < \cdots < x_n$, les OBDD réduits sont canoniques. Cela signifie qu'un OBDD correspond à une seule fonction et vice versa, ce qui peut être utile quand il s'agit de comparer deux OBDD. Puisqu'elles vont nous servir pour analyser l'impact de l'erreur sur le résultat du model-checking, c'est au chapitre 4 que nous présentons en détail les règles de réduction des OBDD ainsi que les propriétés qui en découle. Pour le moment, nous nous contentons de les définir et de décrire comment ils peuvent être utilisés pour représenter des systèmes probabilistes.

Définition 2.4.3. *Soient S un ensemble d'états, $V = \{x_1, x_2, \cdots, x_n\}$ un ensemble non vide de variables booléennes, $<$ un ordre total sur les variables de V tel que $x_1 < x_2 < \cdots < x_n$, $var : S \to V$ une fonction qui associe à un sommet s une variable booléenne dans V et $val : S \to \mathbb{R}$ une fonction qui associe à un sommet s une valeur réelle.*

Un MTBDD sur $\langle V, < \rangle$ est un graphe acyclique orienté avec un ensemble non vide S de sommets qui sont de deux types :

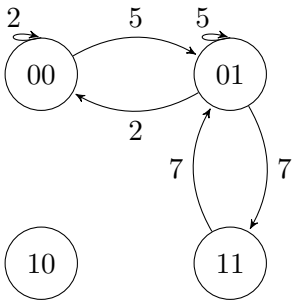
- *des sommets terminaux s ou feuilles, étiquetés par une valeur réelle $val(s) \in \mathbb{R}$,*
- *des sommets non terminaux s, étiquetés par une variable booléenne $var(s) \in V$, qui ont deux enfants $left(s)$ et $right(s)$ tels que pour tout sommet t :*

$$t \in \{left(s), right(s)\} \Rightarrow (var(s) < var(t)) \text{ ou } (t \text{ est terminal}).$$

Les MTBDD ne sont autres que des OBDD avec des feuilles qui peuvent prendre des valeurs réelles. Ils représentent des fonctions $f(x_1, \cdots, x_n) : \mathbb{B}_n \to \mathbb{R}$ au lieu de $f(x_1, \cdots, x_n) : \mathbb{B}_n \to \mathbb{B}$ pour les OBDD. Ils possèdent donc les mêmes propriétés et avantages.

Représentation des systèmes probabilistes à l'aide des MTBDD

Les MTBDD peuvent être utilisés pour représenter les matrices de transitions d'un système probabiliste. Si on considère la matrice de transition comme une fonction de $S \times S \to \mathbb{R}$,



$$R = \begin{pmatrix} 2 & 5 & - & 0 \\ 2 & 5 & - & 7 \\ - & - & - & - \\ 0 & 7 & - & 0 \end{pmatrix}$$

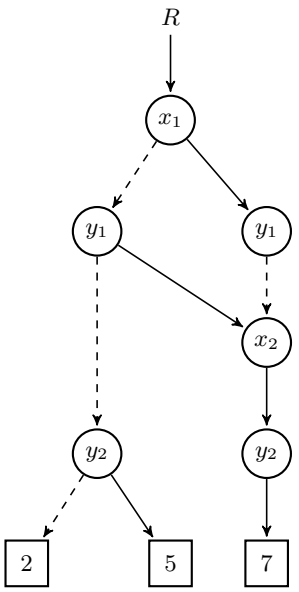
FIGURE 2.13: Un CTMC et sa matrice des transitions.

il suffit d'énumérer tous les états de S pour considérer la matrice de transition comme une relation définie ainsi : $\{0, \ldots, |S| - 1\} \times \{0, \ldots, |S| - 1\} \to \mathbb{R}$.

Et si chaque état est encodé en binaire, la matrice de transition peut être vue comme une relation définie de $\mathbb{B}_n \to \mathbb{R}$. Si on considère alors les variables booléennes $x_1, x_2, \cdots, x_n$ pour encoder les rangées de la matrice de transition d'un système et les variables booléennes $y_1, y_2, \cdots, y_n$ pour les colonnes, nous pouvons représenter notre matrice des transitions comme un MTBDD sur l'ensemble des variables booléennes $\{x_1, x_2, \cdots x_n, y_1, y_2, \cdots, y_n\}$ où les variables x_i correspondent aux sommets de départ d'une transition tandis que les variables y_i correspondent aux sommets d'arrivées d'une transition.

Exemple 2.4.4. *La figure 2.13 montre un CTMC et sa matrice des transitions tandis que la figure 2.14 montre un MTBDD réduit R qui représente ce CTMC et une table qui explique l'encodage du MTBDD R. Le MTBDD R est défini sur 4 variables booléennes, x_1, y_1, x_2, y_2 (dans cet ordre !). Les lignes pointillées représentent une affectation de la valeur 0 à la variable booléenne considérée, et les lignes pleines, une affectation de la valeur 1. Les variables x_i correspondent aux sommets de départ d'une transition tandis que les variables y_i correspondent aux sommets d'arrivées d'une transition. Par exemple la transition $01 \to 00$ est encodée par $(0, 0, 1, 0)$ car l'encodage de l'état de départ est $(x_1, x_2) = (0, 1)$ et l'encodage de l'état d'arrivée est $(y_1, y_2) = (0, 0)$. On peut ainsi voir que le parcours $(x_1, y_1, x_2, y_2) = (0, 0, 1, 0)$ sur le MTBDD R retourne la valeur 2, qui est celle de la transition $01 \to 00$.*

Il faut noter que les variables x_i et y_i sont alternés dans l'ordre choisi. Cela est dû au fait que la taille d'un MTBDD sur $\langle V, < \rangle$ dépend de l'ordre $<$ sur l'ensemble des variables booléennes V. En choisissant l'ordre (x_1, y_1, x_2, y_2) au lieu de (x_1, x_2, y_1, y_2), on utilise une heuristique (efficace) d'ordonnancement des variables pour réduire la taille d'un MTBDD.



Transition	x_1	x_2	y_1	y_2	x_1	y_1	x_2	y_2	R
$0 \xrightarrow{2} 0$	0	0	0	0	0	0	0	0	2
$0 \xrightarrow{5} 1$	0	0	0	1	0	0	0	1	5
$1 \xrightarrow{2} 0$	0	1	0	0	0	0	1	0	2
$1 \xrightarrow{5} 1$	0	1	0	1	0	0	1	1	5
$1 \xrightarrow{7} 3$	0	1	1	1	0	1	1	1	7
$3 \xrightarrow{7} 1$	1	1	0	1	1	0	1	1	7

FIGURE 2.14: Un MTBDD représentant le CTMC de la figure 2.13 [31].

Cet exemple montre qu'on peut représenter un système probabiliste à l'aide d'un MTBDD.

Model checking des systèmes représentés avec les MTBDD

On vient de montrer que l'on peut représenter un système probabiliste à l'aide d'un MTBDD. Si on utilise la même technique d'encodage, on peut aussi représenter un vecteur d'un ensemble d'états indexés (le vecteur b dans la multiplication $A \cdot b$). On peut aussi représenter un ensemble d'états S', si on utilise sa fonction caractéristique (i.e. la fonction f_s pour laquelle $f_s = 1$ si $s \in S'$ ou $f_s = 0$ dans le cas contraire) par un MTBDD. Un tel ensemble représente l'ensemble des états qui satisfont une propriété donnée Φ (l'ensemble $Sat(\Phi)$). Ceci nous donne tous les éléments nécessaires pour pouvoir faire le model checking probabiliste à l'aide des MTBDD en utilisant des opérations qui ont été définies sur les OBDD.

Prenons l'exemple du model checking d'une formule de la forme $P_{\sim p}(\Phi_1 \,\mathcal{U}\, \Phi_2)$. Nous pouvons calculer l'ensemble S^{no} (l'ensemble des états qui vérifient $\Phi_1 \,\mathcal{U}\, \Phi_2$ avec une probabilité nulle) à l'aide des OBDD comme le montre l'algorithme PROB0 de la figure 2.15. L'algorithme prend en entrée deux OBDD représentant les ensembles $Sat(\Phi_1)$ et $Sat(\Phi_2)$ (nous assumons que ces deux ensembles ont déjà été déterminés) et à l'aide d'opérations sur les OBDD retourne l'OBDD représentant l'ensemble S^{no}. Nous pouvons de même construire l'ensemble S^{yes} (avec un algorithme que l'on nommera PROB1) et en déduire ainsi l'ensemble $S^{?} = S \backslash (S^{no} \cup S^{yes})$.

Pour les calculs numériques, comme nous pouvons faire une multiplication matrice-vecteur à l'aide des OBDD, nous pouvons utiliser l'une ou l'autre méthode itérative. Dans l'algorithme

```
Prob0(phi_1, phi_2)
1.  sol := phi_2
2.  done := false
3.  while (done = false)
4.      sol' := sol ∨ (phi_1 ∧ ThereExists(y, T ∧ ReplaceVars(sol, x, y))
5.      if (sol' = sol) then done := true
6.      sol := sol'
7.  endwhile
8.  return ¬ sol
```

FIGURE 2.15: L'algorithme PROB0 implémenté avec les MTBDD. [37]

qui suit, on a choisi la méthode de Jacobi. Un algorithme pour le model checking probabiliste d'une formule du genre la forme $P_{\sim p}(\Phi_1 \mathcal{U} \Phi_2)$, pourrait être :

```
PctlUntil(phi_1, phi_2)
1.  s_no := Prob0(phi_1, phi_2)
2.  s_yes := Prob1(phi_1, phi_2, s_no)
3.  s_? := ¬ (s_no ∨ s_yes)
4.  P' := s_? × P
5.  A := Identity(x, y) − P'
6.  b := s_yes
7.  probs := SolveJacobi(A, b, b)
8.  return probs
```

FIGURE 2.16: L'algorithme PCTL UNTIL. [37]

Nous déterminons d'abord les ensembles S^{no}, S^{yes} et $S^{?}$. La matrice $A = I - P'$ est représenté par **A** et le vecteur b par **b**. Les calculs des probabilités sont alors effectués à part par un autre algorithme itératif, l'algorithme de Jacobi.

Comparaison des méthodes basées sur les MTBDD et les matrices creuses

Les MTBDD se sont montrés efficaces dans la gestion de l'espace. Dans certains cas [28, 30], ils ont permis de faire le model-checking de systèmes présentant un très grand nombre d'états, là où, pour les matrices creuses, il n'aurait même pas été possible de représenter l'ensemble des états. Par contre, les techniques basées sur les MTBDD sont moins performants en terme de

rapidité, cela en grande partie due à une perte de régularité lors des calculs itératifs [31]. En effet, avec les techniques basées sur les matrices creuses, les vecteurs solutions sont stockées dans un tableau où l'accès et la modification de leur contenu est plus rapide.

En résumé, les MTBDD sont plus performants en terme de gestion de l'espace tandis que les matrices creuses sont plus performants en terme de rapidité. Ces deux méthodes présentent des avantages complémentaires d'où l'apparition d'une méthode combinant ces avantages, il s'agit de la méthode hybride.

2.4.3 Approche hybride

La méthode hybride est une combinaison de la matrice utilisant les matrices creuses ainsi que la méthode utilisant les MTBDD. Avec cette méthode, on stocke la matrice des transitions dans un MTBDD (pour pouvoir représenter des systèmes avec un très grand nombre d'états) et le vecteur d'itération b dans un tableau (pour pouvoir effectuer les opérations aussi rapidement qu'avec les matrices creuses).

La matrice A est alors un MTBDD et le vecteur b est un tableau. Pour pouvoir faire le model checking, il nous faut pouvoir faire l'opération de base pour le model checking probabiliste, à savoir la multiplication matrice-vecteur $A \cdot b$ avec A stockée sous forme d'un MTBDD et b sous forme de tableau.

Pour cela, on exécute l'algorithme de la figure 2.11 (conçu initialement pour les matrices creuses) mis à part que cette fois-ci la matrice A est stockée sous forme d'un MTBDD. Dans ce cas, il faut noter que l'on ne peut pas avoir les entrées de la matrice A ligne par ligne ou colonne par colonne. Ce problème n'en est pas un, puisque l'ordre dans lequel on tire les entrées de la matrice A *n'influe pas sur le résultat final* de $A \cdot b$. L'important est d'extraire toutes les entrées. En effet dans une multiplication matrice-vecteur, que l'on exécute la multiplication progressivement d'une ligne de la matrice à une autre, ou que l'on prenne les lignes de la matrice dans un ordre différent, le résultat sera valable si vous tenez en considération à la fin toutes les lignes.

Un parcours de tout le MTBDD représentant la matrice A, par exemple un parcours en profondeur, nous permet d'obtenir les valeurs de toutes les entrées de la matrice A. Ce qu'il nous faut pour effectuer l'opération $\text{res}[i] := \text{res}[i] + \text{val} \times b[j]]$, avec $\text{res} = Ab$, c'est les indices de la ligne et colonne i et j de val, qui est la valeur d'une feuille non nulle dans notre MTBDD A. En gardant une trace du parcours dans le MTBDD, on peut retrouver facilement ces indices. Par exemple, dans le MTBDD de la figure 2.14, avec un parcours correspondant à l'évaluation $(x_1, y_1, x_2, y_2) = (0, 1, 1, 1) = 7$, on peut voir que $x_1x_2 = 01$ et $y_1y_2 = 11$. D'où $i = 01$ en binaire, c'est-à-dire égale à 1 et en faisant de même $j = 11_2 = 3$. La valeur de l'entrée $(1, 3)$ dans A est donc 7.

Puisqu'on peut alors obtenir, par un parcours total du MTBDD A, et pour toutes les entrées de la matrice A, la valeur de l'entrée, ainsi que les indices i, j de la ligne et de la colonne de l'entrée, on peut simuler l'algorithme de la figure 2.11 dans le cas de la méthode hybride, ce qui nous permet de faire le model-checking probabiliste avec cette méthode.

La méthode hybride a été optimisée [31]. Dans cette version optimisée de la méthode hybride, on ne tient pas compte des états atteignables (comme l'état 2 da la figure 2.13), et on utilise un système de cache où les noeuds partagés, puisqu'ils sont visités souvent, sont stockés et réutilisés au besoin. En effet deux ou plusieurs sous-matrices distinctes de A peuvent être identiques, surtout si il s'agit d'une matrice dont les éléments sont 0 ou 1. Dans ce cas, on ne conserve qu'une seule copie, ce qui permet un gain significatif en rapidité.

En conclusion, l'implémentation de ces trois méthodes a confirmé que l'approche basée sur les matrices creuses est certes rapide, mais se révèle inefficace pour représenter (et traiter donc) de très grands systèmes (10^6 états et plus). A l'inverse, la méthode basée sur les MTBDD permet de représenter des systèmes avec un nombre d'états élevé mais elle est moins rapide. En combinant les deux méthodes, on obtient une méthode hybride qui peut représenter des systèmes avec un nombre d'états élevé (autant qu'avec les MTBDD) mais pas aussi rapide que la méthode qui se base sur les matrices creuses. C'est en optimisant l'approche hybride que l'on a pu avoir une méthode aussi rapide que celle qui se base sur les matrices creuses et pouvant représenter des systèmes aussi grands qu'avec les MTBDD.

Chapitre 3

Sources d'erreur

Dans le chapitre précédent, nous avons introduit les notions de base associées au model-checking. Nous avons montré qu'il implique une représentation du modèle à l'étude, la spécification à l'aide d'une logique formelle des propriétés à vérifier et des techniques (algorithmes) pour vérifier si le modèle décrit satisfait les propriétés spécifiées. Dans le cas du model-checking probabiliste, il consiste essentiellement à déterminer la probabilité d'un état du modèle de vérifier une propriété donnée.

Nous avons décrit la manière dont les algorithmes de model-checking probabilistes sont implémentés en pratique par le model-checker PRISM. Pour un outil comme PRISM dont le but est de fournir une garantie fonctionnelle d'un système, il est primordial que les réponses fournies par l'outil soient fiables. Or PRISM, comme la plupart d'autres model-checkers probabilistes, fait état de résultats inexacts lors du calcul des probabilités dans certaines situations. La source de telles erreurs n'est pas une implémentation incorrecte de l'algorithme de model-checking, mais l'usage entre autres, d'une arithmétique approximative à point flottant.

Dans ce chapitre, nous nous intéressons à l'impact de calculs inexacts sur le résultat fourni par PRISM. Nous montrons à quelles étapes du processus de model-checking les erreurs apparaissent. Nous répertorions et décrivons les sources d'erreurs éventuelles.

Nous avons répertorié dans la littérature [9, 49] quatre sources d'erreurs, à savoir :

1. l'**arithmétique à virgule flottante** utilisée par PRISM pour représenter les probabilités,
2. les **méthodes itératives** utilisées pour résoudre les systèmes d'équations linéaires $Ax = b$,
3. la **librairie CUDD** [43] utilisée pour implémenter les opérations sur les MTBDD lors du model-checking symbolique,
4. la **méthode d'uniformisation** (ou méthode de Jensen)[5] utilisée lors du model-checking

pour CTMC.

Comme nous avons choisi de décrire le model-checking pour les DTMC, nous décrivons seulement les trois premières sources d'erreurs répertoriées. Nous ne décrivons pas la dernière source d'erreurs qui concerne le model-checking pour CTMC.

3.1 L'arithmétique à virgule flottante

Durant le processus de model-checking, les erreurs peuvent survenir à différentes phases du processus. Un des faits souvent oublié est que les ordinateurs utilisent l'arithmétique à virgule flottante pour représenter les nombres réels. L'arithmétique à virgule flottante ne peut représenter les nombres réels que jusqu'à une certaine précision. Actuellement pour la représentation des nombres à virgule flottante, la norme IEEE 754 [25] est la plus couramment utilisée. Elle définit le format de représentation pour les nombres à simple précision (32 bits), double précision (64 bits) et double précision étendue (79 bits). Le format double précision étendue est utilisé pour les calculs internes aux processeurs et n'est pas employé pour le stockage des données. Seuls les formats simple et double précision sont actuellement supportés par la majorité des processeurs du côté matériel, et par les langages de programmation du côté logiciel. La norme ne couvre pas seulement la représentation des nombres à virgule flottante, mais gère aussi les cas d'exception comme le débordement (nombre impossible à représenter avec la précision disponible) et autres conditions d'exception (division par zéro, nombre infini, etc...).

Le format double précision (64 bits) est celui utilisé pour représenter les nombres réels (de même pour les probabilités). Un nombre x flottant à double précision est représenté sous la forme $x = (-1)^S \cdot 2^{E-1023} \cdot (1 \cdot M)$, où S est le bit de signe (1 bit), E est l'exposant (11 bits) et M est la mantisse (52 bits).

La précision offerte par un format dépend du nombre de bits réservé à la mantisse. La précision de la machine est le plus petit nombre qui, ajouté à 1.0, donne un résultat différent de 1.0. La norme IEEE 754 induit une précision de $\epsilon_d = 2^{-52} = 2.22 \cdot 10^{-16}$ pour le format double précision et $\epsilon_s = 2^{-23} = 1.19 \cdot 10^{-7}$ pour le format simple précision. Ainsi donc, le format double précision n'offre que 15 chiffres significatifs pour la représentation d'un nombre alors que le format simple précision ne permet que 6 chiffres.

Il est à noter que la précision de la machine ne correspond pas au nombre le plus petit qui peut être représenté en utilisant un format donné, étant donné que ce nombre dépend de la taille de l'exposant.

Si un nombre ne peut être représenté comme un nombre flottant avec cette précision, le nombre représentable le plus proche est choisi à la place. C'est en tronquant ainsi le résultat d'une

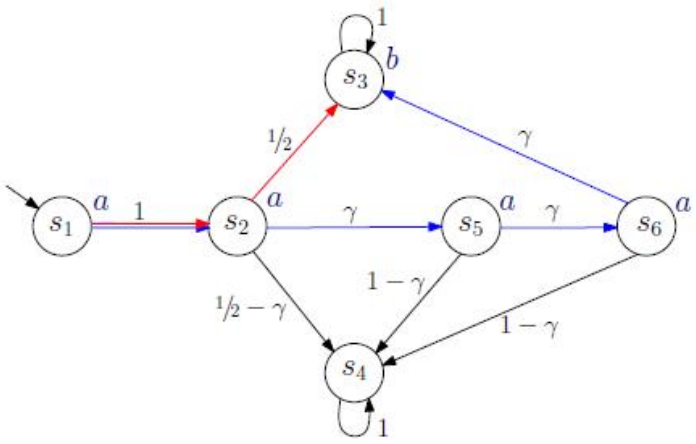


FIGURE 3.1: Une chaîne de Markov. [49]

opération en un nombre représentable, que la norme peut être à l'origine de résultats inexacts retournés par les model-checkers comme décrit à l'exemple ci-après. En effet, déterminer si un état s vérifie une propriété $P_{\sim p}(\varphi)$ consiste à vérifier si $Pr(s,\varphi) \sim p$. Si $Pr(s,\varphi) = p + \delta$ avec δ inférieur à la précision machine, $Pr(s,\varphi)$ pourrait être arrondi à p. Dans ce cas, l'état s peut être considéré comme un état qui vérifie la propriété $P_{\sim_p}(\varphi)$ alors que ce n'est pas le cas ou inversement, un état qui ne vérifie pas la propriété donnée alors que c'est le cas. L'exemple 3.1.1 qui suit montre une situation ou le model-checker PRISM répond qu'un système satisfait une propriété donnée alors que ce n'est pas le cas.

Exemple 3.1.1. *Considérons le DTMC décrit par la figure 3.1. Soit γ une constante de petite valeur $< 1/2$. Nous voulons calculer, pour un état donné, la probabilité de satisfaire la formule PCTL :*

$$\Phi = P_{\leq 1/2}(a \,\mathcal{U}\, b)$$

qui signifie que nous devons emprunter un chemin sur lequel a est toujours satisfait avant d'atteindre un état qui satisfait b et cela avec une probabilité inférieure ou égale à $1/2$. Le tableau de la figure 3.2 montre la probabilité pour chaque état de satisfaire $P_{\leq 1/2}(a \,\mathcal{U}\, b)$ et souligne, pour chaque état, s'il satisfait la formule susmentionnée. Pour l'état initial s_1, comme la probabilité de satisfaire $a \,\mathcal{U}\, b$ est de $1/2 + \gamma^3$, l'état initial ne satisfait pas la formule $P_{\leq 1/2}(a \,\mathcal{U}\, b)$.

Si nous prenons $\gamma = 10^{-6}$ et que nous simulons l'exemple 3.1.1 avec PRISM, l'outil nous répond malheureusement que l'état s_1 satisfait la formule Φ comme le montre la figure 3.3.

D'où vient cette erreur ? Pour déterminer si un état satisfait la formule $P_{\leq 1/2}(a \,\mathcal{U}\, b)$, PRISM doit représenter la valeur $1/2 + \gamma^3$. Pour $\gamma = 10^{-6}$, cette valeur est égale à :

$$1/2 + \gamma^{-18} = 0.500000000000000001$$

État	$Pr(a \mathcal{U} b)$	$P_{\leq 1/2}(a \mathcal{U} b)$
s_1	$1/2 + \gamma^3$	non
s_2	$1/2 + \gamma^3$	non
s_3	1	non
s_4	0	oui
s_5	γ^2	oui
s_6	γ	oui

FIGURE 3.2: Probabilités des états de S de satisfaire $a \mathcal{U} b$.

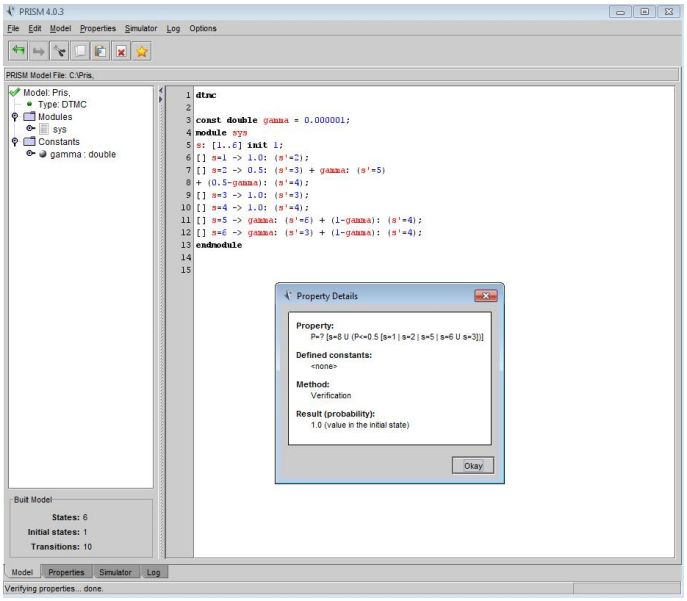


FIGURE 3.3: Analyse de l'exemple 3.1.1 avec PRISM.

Comme nous l'expliquions précédemment, la représentation des nombres réels à point flottants utilisée avec PRISM est une représentation sur 64 bits, qui ne peut représenter que 15 chiffres après la virgule. Le résultat de la représentation est arrondi à 0.5. En changeant la valeur de $1/2 + \gamma^3$ à $1/2$, l'état s_1 devient un état qui satisfait la formule alors que ce n'est pas le cas. Nous décrivons dans ce qui suit la représentation d'un nombre à virgule flottante à l'aide de la norme IEEE 754 pour une meilleure compréhension de ce standard.

Représentation d'un nombre selon la norme IEEE 754

Avec le standard IEEE 754, un nombre binaire est un mot de longueur W, les bits sont indexés de 0 à $W-1$ inclus. Le bit 0 est placé à droite, et il représente le bit de poids faible (c'est-à-dire celui qui provoque la plus petite variation s'il est modifié).

Dans un format général, un nombre flottant est formé de trois éléments : la **mantisse**, l'**exposant** et le **signe**. Le bit de poids fort est le bit de signe. Cela signifie que si ce bit

est à 1, le nombre est négatif, et s'il est à 0, le nombre est positif. Les "e" bits suivants représentent l'exposant décalé, et les m bits suivants (m bits de poids faible) représentent la mantisse.

Signe	Exposant décalé	Mantisse
(1 bit)	(e bits)	(m bits)

L'interprétation d'un nombre est donc :

$$valeur = signe \times 1, mantisse \times 2^{(exposant - decalage)}$$

Avec :

- $signe = \pm 1$
- $decalage = 2^{e-1} - 1$

Le standard définit trois formats pour représenter des nombres à virgule flottante :

- **simple précision** (32 bits : 1 bit de signe, 8 bits d'exposant (-126 à 127), 23 bits de mantisse),
- **double précision** (64 bits : 1 bit de signe, 11 bits d'exposant (-1022 à 1023), 52 bits de mantisse),
- **double précision étendue** (80 bits : 1 bit de signe, 15 bits d'exposant (-16382 à 16383), 64 bits de mantisse).

Le format double précision est identique au format simple précision, mis à part que les champs sont plus grands (52 bits de mantisse au lieu de 23 et 11 bits d'exposant au lieu de 8). Pour cela, bien que le format double précision soit celui qui nous intéresse, nous décrivons le format simple précision au lieu du format double précision, pour les besoins de simplification.

Pour pouvoir représenter des exposants négatifs, le standard utilise un *décalage* pour l'exposant. Si e est le nombre de bits réservés pour représenter l'exposant, le décalage est égal à $2^{e-1} - 1$, ce qui donne un décalage égal à 127 pour le format simple précision (car $e = 8$) et un décalage égal à 1023 pour le format double précision.

C'est la valeur v de l'exposant après lui avoir appliqué le décalage qui est stockée par le standard. La valeur réelle de l'exposant est donc donnée par : $exposant = v - decalage$. Pour le format simple précision ($e = 8$), cela permet des exposants allant de -127 à 128 et pour le format double précision ($e = 11$), des exposants allant de -1023 à 1024.

Exposant avant décalage	-127	0	128
Exposant stocké en simple précision	0	127	255

Appliquer un décalage d'exposant consiste donc à diviser l'ensemble des exposants en deux : la moitié (de 0 à $2^{e-1}-1$) est utilisée pour représenter les exposants négatifs tandis que l'autre moitié (de 2^{e-1} à 2^e-1) est utilisée pour représenter les exposants positifs.

Exemple 3.1.2. *Soit le nombre binaire* 0100 0000 1011 1000 0000 0000 0000 0000 *en format simple précision.*

Signe	Exposant	Mantisse
0	1000 0001	011 1000 0000 0000 0000 0000

- *Le signe est* 0*, le nombre est donc positif.*
- *Le champ exposant est* 1000 0001*, soit* 129 *en décimal. La valeur réelle de l'exposant est donc* $129-127$*, ce qui donne* 2*.*
- *La mantisse est* 011 1000 0000 0000 0000 0000*.*

La représentation finale du nombre en notation scientifique binaire est donc :
$(-1)^0 \cdot 1,0111 \cdot 2^2$

Mathématiquement, cela veut dire :

- $1 \cdot (1 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdot 2^{-3} + 1 \cdot 2^{-4}) \cdot 2^2$
- $(2^0 + 2^{-2} + 2^{-3} + 2^{-4}) \cdot 2^2$
- $2^2 + 2^0 + 2^{-1} + 2^{-2}$
- $4 + 1 + 0.5 + 0.25$

La valeur du nombre à virgule flottante est donc de 5.75*.*

Puisque la mantisse a une taille limitée, le format double précision utilisé pour représenter les réels (probabilités) ne permet que de conserver des nombres sur 15 chiffres après la virgule. Si le résultat d'une opération ne peut être représenté avec cette précision, le nombre est arrondi. La norme définit 4 modes d'arrondis :

1. Au plus proche : mode utilisé par défaut, on arrondit à la valeur la plus proche. S'il arrive que le nombre soit entre-deux, le nombre pair est préféré (celui dont le bit faible est à zéro)
2. Vers moins l'infini
3. Vers plus l'infini
4. Vers zéro : on tronque vers le zéro

La norme ne se contente pas de décrire la représentation des nombres à virgule flottante, elle définit aussi des règles pour traiter des situations exceptionnelles comme le dépassement de capacité, le résultat d'une opération arithmétique invalide, etc. En effet, le bit de poids fort de la mantisse est déterminé par la valeur de l'exposant décalé. Si l'exposant décalé est différent de 0 et de 2^{e-1}, le bit de poids fort de la mantisse est 1 et le nombre est dit "*normalisé*". Si l'exposant décalé est nul, le bit de poids fort de la mantisse est nul et le nombre est "*dénormalisé*".

Il y a trois cas particuliers :

- si l'exposant décalé et la mantisse sont tous deux nuls, le nombre est ± 0 (selon le bit de signe),
- si l'exposant décalé est égal à 2^{e-1}, et si la mantisse est nulle, le nombre est infini (selon le bit de signe),
- si l'exposant décalé est égal à 2^{e-1} mais que la mantisse n'est pas nulle, le nombre est **NaN** (not a number : pas un nombre).

Nous pouvons le résumer ainsi :

Type	Exposant décalé	Mantisse
Zéros	0	0
Nombres dénormalisés	0	différente de 0
Nombres normalisés	1 à $2^e - 2$	quelconque
Infinis	$2^e - 1$	0
NaNs	$2^e - 1$	différente de 0

Un **NaN**("*Not a Number*", en français "pas un nombre") est une valeur ou un symbole produit par le résultat d'une opération arithmétique invalide comme déterminer la racine carrée de nombres négatifs, diviser par zéro, etc. Comme le résultat de telles opérations est invalide, elles donneront alors un résultat équivalent à un NaN.

3.2 Les méthodes itératives

A la section 2.3, nous avons montré que le model-checking pour certains opérateurs PCTL consiste à résoudre un système d'équations linéaires. Nous pouvons représenter un tel système sous la forme traditionnelle $Ax = b$ où A est une matrice de réels, b un vecteur de réels et x, le vecteur contenant la solution devant être calculée. Puisque pour les systèmes à résoudre, les matrices et les vecteurs sont indexés selon un ensemble d'état S, nous pouvons affirmer que la matrice A est de taille $|S \times S|$ et les vecteurs x et b de taille $|S|$. Nous utiliserons aussi l'indexation $0, \cdots, |S-1|$ pour représenter les états du système à l'étude.

Résoudre un système d'équations linéaires est un problème largement documenté. Les méthodes pour résoudre un tel problème se divisent en 2 catégories : les *méthodes directes* et les *méthodes itératives*. Les méthodes directes calculent la solution exacte (dans les limites de la précision machine) après un nombre fixe d'étapes. Dans cette catégorie, on peut citer notamment l'élimination gaussienne ou méthode de Gauss et la méthode de décomposition L/U. Les méthodes itératives calculent des approximations successives de la solution, l'itération s'arrête lorsque la séquence de solution converge vers une précision donnée. Dans ces méthodes, on peut citer la méthode de Jacobi, la méthode des puissances ainsi que la méthode de Gauss-Seidel [37].

Comme le model-checking produit des systèmes où la matrice A est large et creuse, les méthodes directes ne sont pas appropriées pour résoudre de tels systèmes à cause du phénomène dit de *fill-in* (remplissage) : les modifications faites à la matrice A au fur et à mesure que l'on effectue les calculs augmentent le nombre d'entrées non nulles dans la matrice.

Pour de très grands systèmes, l'accroissement de l'espace de stockage résultant peut rendre les calculs insolubles. Cela n'est pas le cas avec les méthodes itératives. La matrice n'est pas modifiée au cours du processus de calculs et il n'y a pas ainsi d'accroissement dans la consommation de mémoire.

Une autre raison qui fait que les model-checkers utilisent les méthodes itératives est le choix de la structure de données à utiliser pour le stockage de la matrice A. Les modifications effectuées à la matrice lors des calculs étant sans incidence, nous pouvons choisir la structure de données la mieux apte à représenter notre matrice de manière compacte et facile d'accès aux éléments. C'est ce qui est à l'origine du choix des matrices creuses comme structure de données pour le stockage de données et même les MTBDD.

Nous décrivons dans ce qui suit une des méthodes itératives utilisée, à savoir la méthode de Jacobi [37]. Cependant, la procédure générale pour toutes les autres méthodes itératives est identique : en partant d'une estimation initiale de la solution du système d'équations, chaque itération produit une estimation plus précise de la solution du système d'équations linéaires. La solution produite à la k-ème itération est notée $x^{(k)}$. La solution initiale est notée $x^{(0)}$.

Chaque solution approximative $x^{(k)}$ est calculée à partir de la solution précédente $x^{(k-1)}$. Le processus d'itération s'arrête quand le vecteur solution est jugé avoir suffisamment convergé. Les critères de convergence varient. Un des critères utilisé est d'arrêter l'itération si la différence entre éléments de vecteurs consécutifs est inférieure à une certaine précision ϵ :

$$\max_i |x^{(k)}(i) - x^{(k-1)}(i)| < \epsilon.$$

Ce critère d'arrêt n'est pas optimal puisque si le vecteur contient des petites valeurs inférieures à ϵ, le processus d'itération pourrait s'arrêter prématurément. Une meilleure approche qui est

celle utilisée par PRISM est de considérer la différence relative entre éléments :

$$\max_i \left(\frac{|x^{(k)}(i) - x^{(k-1)}(i)|}{|x^{(k)}(i)|} \right) < \epsilon.$$

PRISM fixe ϵ par défaut (paramètre qui peut être changé par l'utilisateur) à 10^{-6}.

La méthode de Jacobi

La méthode de Jacobi est basée sur l'observation que la i^e équation du système d'équations linéaires $Ax = b$ qui s'écrit comme suit :

$$\sum_{j=0}^{|S|-1} A(i,j) \cdot x(j) = b(i)$$

peut être réécrite de la façon suivante :

$$x(i) = \left(b(i) - \sum_{\substack{j=0 \\ j \neq i}}^{|S|-1} A(i,j) \cdot x(j) \right) / A(i,i).$$

Le i^e élément de la k^e itération est déterminé à partir du $(k-1)^e$ vecteur comme suit :

$$x^k(i) = \left(b(i) - \sum_{\substack{j=0 \\ j \neq i}}^{|S|-1} A(i,j) \cdot x^{k-1}(j) \right) / A(i,i).$$

Pour les systèmes à vérifier pour PCTL, il est à signaler que les entrées $A(i,i)$ ne sont pas nulles.

Soit $\mathbf{D} - (\mathbf{L} + \mathbf{U})$, une partition de la matrice A avec $\mathbf{D}$, une matrice contenant les éléments diagonaux, $\mathbf{L}$ et $\mathbf{U}$ sont respectivement la matrice triangulaire inférieure et la matrice triangulaire supérieure dérivées de A, i.e., $\mathbf{L} + \mathbf{U}$ contient les valeurs opposées de toutes les entrées non diagonales.

Comme $\mathbf{D}(i,i) = A(i,i)$, nous avons que $\mathbf{D}^{-1}(i,i) = 1/A(i,i)$. La précédente équation peut être réécrite comme suit :

$$x^k(i) = \mathbf{D}^{-1}(i,i) \cdot \left(b(i) - \sum_{\substack{j=0 \\ j \neq i}}^{|S|-1} A(i,j) \cdot x^{k-1}(j) \right)$$

Sous forme matricielle, la solution approximative $x^{(k)}$ calculée à la k^i itération est donnée par :

$$x^{(k)} = \mathbf{D}^{-1} \cdot ((\mathbf{L} + \mathbf{U}) \cdot x^{(k-1)} + b).$$

Il apparaît clairement qu'une itération nécessite une seule multiplication matrice-vecteur. Si k représente le nombre d'itérations nécessaires pour déterminer la solution, l'ensemble du processus nécessite donc k multiplications matrice-vecteur.

Les méthodes itératives sont sources de calculs inexacts car elles calculent une approximation de la solution à ϵ près. De plus, il apparaît que la précision par défaut de 10^{-6} est assez grande pour de larges systèmes, surtout que l'on travaille sur des probabilités comprises entre 0 et 1.

La problématique est la même que pour le cas de l'arithmétique à virgule flottante. On souhaite vérifier si un état s donné vérifie une propriété de la forme $P_{\sim p}(\varphi)$. Il s'agit de déterminer si $Pr(s, \varphi) \sim p$.

Désignons par $x(s)$, la probabilité de s calculée à la fin du processus d'itération. En théorie :

$$x(s) = Pr(s, \varphi).$$

En pratique :

$$Pr(s, \varphi) - \epsilon \leq x(s) \leq Pr(s, \varphi) + \epsilon.$$

$x(s)$ est donc égal à : $x(s) = Pr(s, \varphi) \pm \delta$, avec $0 < \delta < \epsilon$. Comme pour l'exemple 3.1.1, si $Pr(s, \varphi)$ est proche ou égal à p, en fournissant une réponse approximative de $Pr(s, \varphi)$, il y'a risque de transformer l'état s en un état qui vérifie (ou pas) la propriété énoncée alors que ce n'est pas le cas.

Les méthodes itératives font passer la précision des réponses fournies de 10^{-15} (précision machine) à 10^{-6} qui est la valeur par défaut utilisée par PRISM pour les critères de terminaison. PRISM désigne par ϵ *de terminaison*, la valeur de la précision désirée pour les méthodes itératives.

Si PRISM choisit par défaut une valeur a priori élevée, c'est par souci de gagner en espace mémoire et temps de calcul utilisés. Il s'agit de résoudre le dilemme suivant : si on choisit ϵ de terminaison plus petit, on augmente la chance d'obtenir une réponse plus précise, mais on augmente aussi le nombre d'itérations effectuées et par conséquent l'espace mémoire et le temps de calcul utilisés. Si par contre on choisit une valeur plus grande, on prend le risque d'avoir une réponse moins précise, mais on réduit le nombre d'itérations à faire. Il s'agit de trouver une valeur médiane qui tient compte de ces contraintes et du système à l'étude.

De plus, PRISM fixe (paramètre modifiable aussi) un nombre maximal k d'itérations (par défaut $k = 10000$). Si l'une des 2 conditions est vérifiée (précision obtenue ou nombre d'ité-

rations atteint), l'itération s'arrête. Il est évident qu'en augmentant la précision recherchée, il y'a risque d'atteindre le nombre maximal d'itérations avant la fin des itérations.

L'exemple 3.2.1 qui suit met en évidence le critère de terminaison des méthodes itératives comme source d'erreurs pour les résultats retournés par PRISM. Il s'agit de calculer une probabilité représentable avec la précision machine, mais pas avec la précision du critère de terminaison. Nous montrons que dans ce cas, le model-checker fournit une réponse incorrecte tandis qu'avec une valeur de probabilité représentable avec la précision du critère de terminaison, le model-checker fournit une réponse correcte.

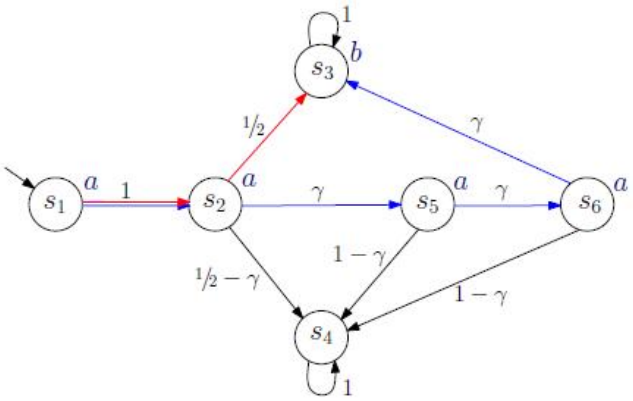


FIGURE 3.4: Une chaîne de Markov.

Exemple 3.2.1. *Soit le DTMC de la figure 3.1 que nous reprenons à la figure 3.4. Nous voulons savoir si l'état initial s_1 satisfait la formule PCTL $P_{\leq 1/2}(a\,\mathcal{U}\,b)$. Comme $Pr(s_1, a\,\mathcal{U}\,b) = 1 = 2 + \gamma^3$ avec $\gamma > 0$, l'état initial ne satisfait pas la formule PCTL $P_{\leq 1/2}(a\,\mathcal{U}\,b)$.*

Le tableau de la figure 3.5 montre les résultats obtenus avec PRISM pour différentes valeurs de γ et ϵ de terminaison. Pour $\epsilon = 10^{-6}$ et $\gamma = 10^{-5}$ ou $\gamma = 10^{-3}$, PRISM fournit une réponse incorrecte parce que la probabilité à calculer ($1/2+\gamma^3 = 10^{-15}$ ou $1/2+\gamma^3 = 10^{-9}$) est tronquée avec la précision choisie. Par contre, pour la même valeur de ϵ (10^{-6}) et $\gamma = 10^{-2}$, la réponse obtenue est correcte. Il en est de même pour les autres simulations effectuées. L'outil fournit une réponse correcte si la probabilité à calculer est représentable avec la précision choisie. De plus, il est à noter que lorsque l'outil fournit une réponse correcte, le nombre d'itérations est plus grand (5 au lieu de 3), i.e., la réponse obtenue est plus précise.

3.3 Librairie CUDD pour le model-checking symbolique

Dans la section précédente, nous avons exposé les raisons qui font que les model-checkers aient recours aux méthodes itératives pour résoudre les systèmes d'équations linéaires $Ax = b$. Une des raisons était que l'on peut se focaliser sur le choix de la structure de donnée la plus efficace

γ	$Pr(s_1, a\,\mathcal{U}\,b)$	ϵ de terminaison	itérations	$s_1 \models P_{\leq 1/2}(a\,\mathcal{U}\,b)$?
10^{-5}	$1/2 + 10^{-15}$	10^{-6}	3	vrai
10^{-3}	$1/2 + 10^{-9}$	10^{-6}	3	vrai
10^{-2}	$1/2 + 10^{-6}$	10^{-6}	5	faux
10^{-3}	$1/2 + 10^{-9}$	10^{-8}	3	vrai
10^{-3}	$1/2 + 10^{-9}$	10^{-9}	5	faux
10^{-5}	$1/2 + 10^{-15}$	10^{-15}	5	faux

FIGURE 3.5: Simulation avec PRISM de l'exemple 3.2.1

pour encoder la matrice A sans se soucier de la perte de régularité au cours du processus de calcul. L'argument est valable autant pour les matrices creuses que pour les MTBDD.

Si nous choisissons de représenter la matrice A sous forme de matrices creuses, on parle de *model-checking explicite*. Par contre, si ce choix porte sur les MTBDD, on parle de *model-checking symbolique* [31]. PRISM est un model-checker symbolique même s'il offre la possibilité d'utiliser les matrices creuses ainsi que la méthode hybride combinant les deux méthodes.

Pour implémenter les opérations sur les MTBDD, PRISM utilise CUDD [43] (Colorado University Decision Diagram), une librairie publique développée par Fabio Somenzi et son équipe à l'université de Colorado. CUDD à l'opposé des autres librairies sur les OBDD, présente l'avantage de supporter aussi les MTBDD et plusieurs autres variantes des OBDD.

Les règles de réduction sur les OBDD que l'on décrit en détail au chapitre 4 exigent que si 2 feuilles d'un MTBDD donné ont la même valeur, les 2 feuilles soient fusionnées pour ne garder qu'une seule copie. Dans CUDD, cette règle est accentuée : lors de la construction d'un MTBDD, il existe une constante ϵ_c telle qu'une nouvelle feuille v est générée s'il n'existe pas une autre feuille v' avec une valeur $|v - v'| \leq \epsilon_c$. Deux feuilles sont donc considérées équivalentes à ϵ_c près.

Cette règle de réduction modifiée est aussi source de résultats inexacts retournés par PRISM. Comme décrit à la section 2.4, les MTBDD sont utilisés pour représenter un système probabiliste, plus précisément sa matrice de transition. Une feuille v correspond à la valeur $Pr(s,t)$ d'une transition entre un état s et un état t. En changeant la valeur de v en une valeur v' qui est telle que $v - \epsilon_c \leq v' \leq v + \epsilon_c$, cela revient à changer $Pr(s,t)$ en une nouvelle valeur $Pr'(s,t)$ qui est telle que :

$$Pr(s,t) - \epsilon_c \leq Pr'(s,t) \leq Pr(s,t) + \epsilon_c.$$

Il apparaît clairement qu'un tel changement peut avoir des répercussions quant à la probabilité retournée par le model-checker. Nous décrivons en détail au chapitre 4 l'impact de ce changement sur le résultat obtenu.

Pour le moment, nous nous contentons de fournir un exemple où nous mettons en évidence cette règle comme étant à l'origine du calcul inexact effectué.

Considérons toujours le DTMC de la figure 3.4 que nous reprenons à la figure 3.6. Nous avons déjà montré que $s_1 \not\models P_{\leq 1/2}(a \,\mathcal{U}\, b)$ puisque $Pr(s_1, a \,\mathcal{U}\, b) = 1/2 + \gamma^3$.

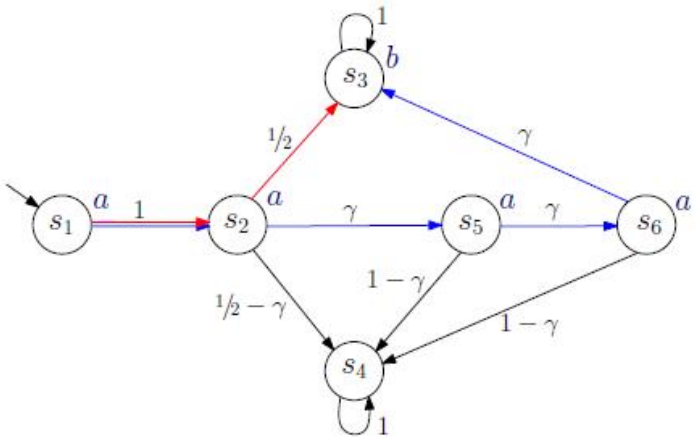


FIGURE 3.6: Une chaîne de Markov.

Nous avons aussi déjà montré que pour déterminer si un état donné satisfait une formule PCTL, cela revient à résoudre un système d'équations linéaires $Ax = b$. Dans notre cas, la matrice A (voir section suivante 3.4) est égale à :

$$A = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1/2 & 1/2-\gamma & \gamma & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1-\gamma & 0 & \gamma \\ 0 & 0 & \gamma & 1-\gamma & 0 & 0 \end{pmatrix}$$

En représentant la matrice A sous forme de MTBDD pour un ϵ_c donné et $\gamma < \epsilon_c$, le MTBDD généré *pourrait*[1] correspondre à la matrice suivante A' :

1. Nous montrons au chapitre 4 que cela dépend de l'ordre d'apparition des feuilles.

$$A' = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1/2 & 1/2 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

Le MTBDD correspondant à la matrice A possède les feuilles suivantes :

$$0, 1, 1/2, 1/2 - \gamma, \gamma, 1 - \gamma$$

Dans le MTBDD correspondant à A', les feuilles $1/2 - \gamma$,γ,$1 - \gamma$ ont été fusionnées respectivement avec $1/2$, 0 et 1. En effet, la feuille de valeur $1/2 - \gamma$ est considérée équivalente à la feuille de valeur 1, puisque $|(1/2 - \gamma) - 1/2| = \gamma < \epsilon_c$. Il en est de même pour les feuilles γ et $1 - \gamma$.

La matrice A' correspond en réalité au DTMC de la figure 3.7. Pour le DTMC de la figure 3.7, $Pr(s_1, a \mathcal{U} b) = 1/2$. L'état initial satisfait donc la formule PCTL $P_{\leq 1/2}(a \mathcal{U} b)$, ce qui n'est pas le cas pour le DTMC de départ. La réponse obtenue est inexacte à cause du paramètre ϵ_c. CUDD fixe par défaut ϵ_c à 10^{-12} tandis que PRISM fixe ce paramètre à 10^{-15} de l'ordre de la précision machine.

Une grande valeur de ϵ_c crée des MTBDD de petite taille donc faciles à stocker, mais avec une précision moins fine. Une petite valeur de ϵ_c produit à l'opposé de très grands MTBDD mais avec une précision plus fine. Cependant, dans tous les cas, le nombre de feuilles du MTBDD généré (la taille) est borné par le nombre de probabilités différentes dans le DTMC, donc par le nombre d'arêtes.

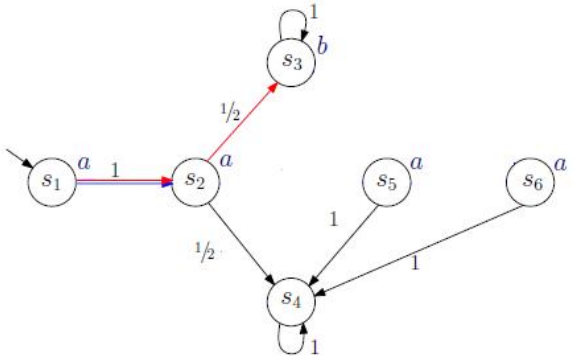


FIGURE 3.7: Exemple d'une chaîne de Markov à temps discret.

3.4 Expérimentation avec PRISM

Dans cette section nous décrivons les expérimentations faites sous PRISM pour mettre en évidence les sources d'erreurs décrites dans les sections précédentes. Pour cela, nous avons dû simuler l'exemple 3.1.2 décrit dans les sections précédentes pour différentes valeurs de γ, ϵ_t (utilisé par le critère de terminaison pour les méthodes itératives) et ϵ_c (utilisé par la librairie CUDD).

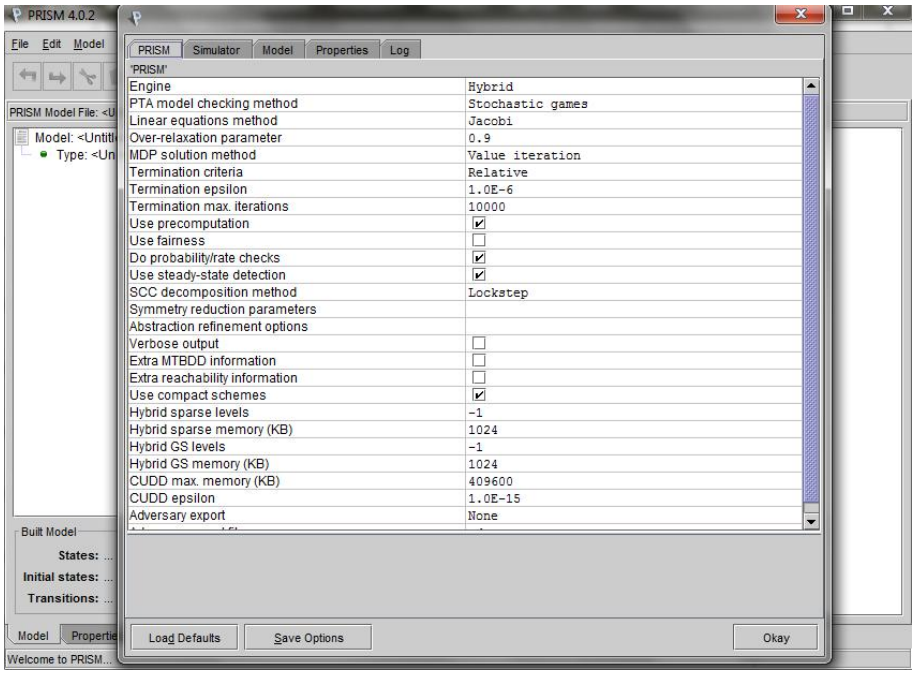


FIGURE 3.8: Configuration des options de PRISM.

Comme on peut le constater avec la figure 3.8, PRISM présente l'avantage de pouvoir fixer des valeurs autres que les valeurs par défaut pour ces paramètres, mais aussi pour de nombreux autres paramètres. Nous décrivons d'abord à quoi correspondent quelques paramètres qui nous intéressent :

- **Engine** : correspond à la méthode de model-checking souhaité : matrices creuses, MTBDD ou hybride. C'est la méthode hybride qui est choisie par défaut.

- **Linear equations method** : La méthode itérative utilisée pour résoudre les systèmes d'équations linéaires. Elle est fixée par défaut à la méthode de Jacobi décrite précédemment. Les autres options sont : Gauss-Seidel, JOR, SOR ainsi que des variantes de ces 3 méthodes.

– **Termination criteria** : Le critère de terminaison pour les méthodes itératives. Les options disponibles sont : relatif (par défaut) ou absolu. Voir 3.2 pour la différence.

– **Termination epsilon** : Epsilon pour les méthodes itératives, correspond à ϵ_t. La valeur par défaut est 10^{-6}.

– **Termination max. iterations** : le nombre maximal d'itérations. Fixé par défaut à 10000.

– **CUDD epsilon** : Epsilon pour la librairie CUDD, correspond à ϵ_c. La valeur par défaut est 10^{-15} avec PRISM. Elle est de 10^{-12} par défaut dans la librairie CUDD.

Nous reprenons à la figure 3.9 le modèle probabiliste présenté dans les sections précédentes. Nous voulons déterminer si l'état initial satisfait la formule PCTL $P_{\leq 1/2}(a\,\mathcal{U}\,b)$. Pour notre cas, les ensembles S^{yes}, S^{no}, et $S^{?}$ correspondent à :

$$S^{yes} = \{s_3\}, S^{no} = \{s_4\}, S^{?} = \{s_1, s_2, s_5, s_6\}.$$

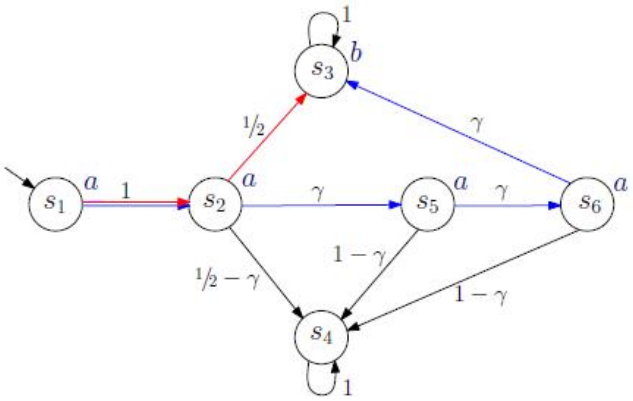


FIGURE 3.9: Exemple d'une chaîne de Markov à temps discret

Le système d'équation $Ax = b$ correspondant (voir l'opérateur $\mathcal{U}^{\leq k}$ à la section 2.3) se résout itérativement comme suit :
$x^{(k)} = Ax^{(k-1)}, \forall k > 0$, avec A et $x^{(0)}$ correspondant à :

$$\mathrm{A} = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1/2 & 1/2-\gamma & \gamma & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1-\gamma & 0 & \gamma \\ 0 & 0 & \gamma & 1-\gamma & 0 & 0 \end{pmatrix} \quad \text{et} \quad x^{(0)} = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

État	$Pr(a\,\mathcal{U}\,b)$	$P_{\leq 1/2}(a\,\mathcal{U}\,b)$?
s_1	$1/2+\gamma^3$	non
s_2	$1/2+\gamma^3$	non
s_3	1	non
s_4	0	oui
s_5	γ^2	oui
s_6	γ	oui

FIGURE 3.10: Probabilités des états de S de satisfaire $a\,\mathcal{U}\,b$

Les vecteurs solutions obtenus après chaque itération sont :

$$x^{(1)} = \begin{pmatrix} 0 \\ 1/2 \\ 1 \\ 0 \\ 0 \\ \gamma \end{pmatrix} \quad x^{(2)} = \begin{pmatrix} 1/2 \\ 1/2 \\ 1 \\ 0 \\ \gamma^2 \\ \gamma \end{pmatrix} \quad x^{(3)} = \begin{pmatrix} 1/2 \\ 1/2+\gamma^3 \\ 1 \\ 0 \\ \gamma^2 \\ \gamma \end{pmatrix} \quad x^{(4)} = \begin{pmatrix} 1/2+\gamma^3 \\ 1/2+\gamma^3 \\ 1 \\ 0 \\ \gamma^2 \\ \gamma \end{pmatrix}$$

$$x^{(5)} = \begin{pmatrix} 1/2+\gamma^3 \\ 1/2+\gamma^3 \\ 1 \\ 0 \\ \gamma^2 \\ \gamma \end{pmatrix}$$

Après la 5^e itération, le processus d'itération s'arrête puisqu'il a convergé. Chaque entrée du vecteur-solution obtenu correspond à la probabilité pour un état s du modèle de satisfaire $a\,\mathcal{U}\,b$. Avec cette probabilité, nous pouvons répondre à la question de savoir si $s \models P_{\leq 1/2}(a\,\mathcal{U}\,b)$.

Pour $\delta > 0, s_1 \not\models P_{\leq 1/2}(a\,\mathcal{U}\,b)$ puisque $Pr(s1, a\,\mathcal{U}\,b) = 1/2+\gamma^3$.

Le tableau de la figure 3.11 montre les résultats retournés par PRISM pour différentes valeurs de γ, epsilon de terminaison (ϵ_t) et CUDD epsilon (ϵ_c). Nous avons testé l'exemple avec la méthode hybride. Dans ce tableau, la 1ère colonne montre la source d'erreurs mise en évidence, ϵ_t et ϵ_c correspondent respectivement à epsilon de terminaison et CUDD epsilon. Les colonnes "Noeuds" et "itération" correspondent à la taille du MTBDD et au nombre d'itérations effectuées. Φ correspond à la formule PCTL $P_{\leq 1/2}(a\,\mathcal{U}\,b)$.

Nous constatons que lorsque le model-checker fournit une réponse inexacte (vrai au lieu de faux), le processus de model-checking s'arrête après 3 itérations au lieu de 5, mais pour des raisons différentes selon la source d'erreur :

	γ	$Pr(s_1, a\,\mathcal{U}\,b)$	ϵ_t	ϵ_c	Noeuds	itérations	$s_1 \models \Phi$?
Arithmétique à point flottant	10^{-6}	$1/2 + 10^{-18}$	10^{-6}	10^{-15}	31	3	vrai
	10^{-6}	$1/2 + 10^{-18}$	10^{-15}	10^{-15}	31	3	vrai
Méthodes itératives	10^{-5}	$1/2 + 10^{-15}$	10^{-6}	10^{-15}	31	3	vrai
	10^{-3}	$1/2 + 10^{-9}$	10^{-6}	10^{-15}	31	3	vrai
	10^{-2}	$1/2 + 10^{-6}$	10^{-6}	10^{-15}	31	5	faux
	10^{-3}	$1/2 + 10^{-9}$	10^{-8}	10^{-15}	31	3	vrai
	10^{-3}	$1/2 + 10^{-9}$	10^{-9}	10^{-15}	31	5	faux
	10^{-5}	$1/2 + 10^{-15}$	10^{-15}	10^{-15}	31	5	faux
Librairie CUDD	10^{-5}	$1/2 + 10^{-15}$	10^{-15}	10^{-15}	31	5	faux
	10^{-5}	$1/2 + 10^{-15}$	10^{-15}	10^{-6}	31	5	faux
	10^{-5}	$1/2 + 10^{-15}$	10^{-15}	10^{-3}	13	3	vrai

FIGURE 3.11: Simulation avec PRISM de l'exemple 3.2.1.

- **Arithmétique à point flottant** : la valeur $1/2+\gamma^3$ du vecteur $x^{(3)}$ est arrondie à $1/2$ car non représentable en point flottant. Le vecteur $x^{(3)}$ devient dans ce cas identique au vecteur $x^{(2)}$ et le processus d'itération s'arrête. A ce moment, $Pr(s_1, a\,\mathcal{U}\,b) = 1/2$ d'où le model-checker retourne vrai comme réponse.
- **Méthodes itératives** : pour ce cas, la valeur $1/2+\gamma^3$ du vecteur $x^{(3)}$ est représentable en point flottant. Le processus s'arrête car la différence entre $x^{(3)} - x^{(2)} = \gamma^3 < \epsilon_t$ pour les valeurs tests choisies comme $\gamma = 10^{-3}$ et $\epsilon_t = 10^{-6}$ (dans ce cas $\gamma^3 = 10^{-9} < 10^{-6}$).
- **Librairie CUDD** : pour pouvoir mettre en évidence cette situation, nous avons choisi un $\gamma < \epsilon_c$ ($\gamma = 10^{-5}$ et $\epsilon_c = 10^{-3}$) bien que cette situation soit peu plausible. Le MTBDD généré est un MTBDD simplifié comme on le voit par sa taille (13 noeuds au lieu de 31). Dans ce cas par exemple, les feuilles de valeurs $1/2 + \gamma$ ($1/2 + 10^{-5}$) sont fusionnées avec les feuilles de valeur $1/2$, étant considérées égales à ϵ_c (10^{-3}) près. La fusion des feuilles entraîne l'application à nouveau des règles de réduction. Il en ressort ainsi un MTBDD de taille plus réduite (noeuds internes et feuilles).

Tel que mentionné au début de ce chapitre, nous avons répertorié dans la littérature [9, 49] les sources d'erreurs (arithmétique à virgule flottante, méthodes itératives, model-checking symbolique) lors du processus de model-checking. Les expérimentations faites dans ce chapitre avec l'outil PRISM nous ont permis d'identifier à quels moments du processus de model-checking, ces erreurs apparaissent. Pour ce faire, nous avons dû simuler un cas d'utilisation de PRISM en changeant les valeurs de certains paramètres de configuration (ϵ_t, ϵ_c, etc.). Les différentes combinaisons des valeurs affectées aux paramètres de configuration ont été choisies de telle manière à mettre en évidence une seule source d'erreurs à la fois, même si certaines valeurs choisies sont peu probables dans la vie courante (par exemple $\epsilon_c = 10^{-3}$). Dans la suite de ce travail, nous allons analyser l'impact des sources d'erreurs sur le résultat retourné

par le model-checker.

Chapitre 4

Bisimulation et équivalence approchées

Dans ce chapitre, nous nous intéressons aux erreurs qui apparaissent lors de la vérification de systèmes probabilistes en utilisant la technique du model-checking symbolique. Au chapitre précédent, nous avons vu que pour des raisons pratiques, la transcription du système probabiliste en MTBDD est modifiée : le système encodé correspond en réalité à un système "*approximatif*" du modèle que l'on souhaite représenter. Nous voulons déterminer à quel point les deux modèles, la spécification ainsi que le modèle réellement encodé sont similaires.

Dans le domaine de la vérification automatique, un problème classique est celui de vérifier si un système satisfait à une certaine spécification. Dans notre cas, il s'agit de vérifier (comparer) si le modèle encodé correspond au modèle à l'étude. Pour un système non probabiliste, on procède à la comparaison entre les deux modélisations de la spécification et du modèle en utilisant une relation d'équivalence. L'équivalence choisie est souvent la bisimulation.

Cependant, la comparaison basée sur des relations d'équivalence comme la bisimulation n'est pas appropriée dans le contexte probabiliste [18]. Une relation d'équivalence comme la bisimulation nous fournit une réponse binaire, i.e., elle nous apprend si deux systèmes sont équivalents (bisimilaires) ou pas. Une relation d'équivalence n'est donc pas assez précise puisqu'elle identifie deux systèmes proches comme non bisimilaires, de même que pour deux systèmes très différents. Une autre raison est que la plupart du temps les informations de nature stochastique viennent d'observations, ou d'estimations théoriques. Dans un tel contexte, il est plus utile d'avoir des notions d'équivalence approchée, ou de distance.

Pour répondre au problème de la non-robustesse de la bisimulation, plusieurs distances ont été définies pour la comparaison de systèmes probabilistes [13, 46], et d'autres travaux ont été conduits pour définir des métriques servant à estimer jusqu'à quel degré, des systèmes peuvent être similaires [16].

Dans ce chapitre, nous avons choisi d'utiliser la notion d'ϵ-bisimulation définie par [14] pour faire la comparaison entre le système probabiliste à l'étude et celui encodé par PRISM sous forme de MTBDD.

4.1 Propriétés des MTBDD

Nous avons vu à la section 2.4 que les MTBDD sont une extension des BDD (Binary Decision Diagram). Un BDD est un graphe acyclique orienté qui représente une fonction booléenne de la forme $f : \mathbb{B}_n \to \mathbb{B}$. Les MTBDD permettent de représenter des fonctions qui peuvent prendre des valeurs dans n'importe quel ensemble arbitraire D, i.e., des fonctions de la forme $f : \mathbb{B}_n \to D$. Dans la majeure partie des cas, l'ensemble D est l'ensemble des réels $\mathbb{R}$, ce qui fait des BDD un cas spécial des MTBDD.

Nous rappelons la définition d'un MTBDD énoncée à la section 2.4 pour introduire aisément les propriétés des MTBDD.

Définition 4.1.1. *Soient S un ensemble d'états, $V = \{x_1, x_2, \cdots, x_n\}$ un ensemble non vide de variables booléennes, $<$ un ordre total sur les variables de V tel que $x_1 < x_2 < \cdots < x_n$, $var : S \to V$ une fonction qui associe à un sommet s une variable booléenne dans V et $val : S \to \mathbb{R}$, une fonction qui associe à un sommet s une valeur réelle.*

Un MTBDD sur $\langle V, < \rangle$ est un graphe acyclique orienté avec un ensemble non vide S de sommets qui sont de deux types :

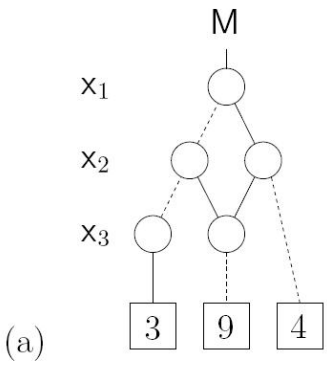
- *des sommets terminaux s, ou feuilles, étiquetés par une valeur réelle $val(s) \in \mathbb{R}$,*
- *des sommets non terminaux s, étiquetés par une variable booléenne $var(s) \in V$, qui ont deux enfants $left(s)$ et $right(s)$ tels que pour tout sommet t :*

$$t \in \{left(s), right(s)\} \Rightarrow (var(s) < var(t)) \text{ ou } (t \text{ est terminal}).$$

Le graphe contient deux types de noeuds : *non terminal* et *terminal*. Un noeud non terminal s est étiqueté par une variable $var(s) \in V = \{x_1, x_2, \cdots, x_n\}$ et a deux enfants, $left(s)$ et $right(s)$. Un noeud terminal s ou feuille est étiqueté par un nombre réel $val(s)$ et n'a pas d'enfants.

L'ordre $<$ sur l'ensemble des variables V sur le graphe exige qu'un enfant s' d'un noeud non terminal s est soit un noeud terminal, soit un noeud non terminal qui satisfait $var(s) < var(s')$.

Exemple 4.1.2. *La figure 4.1 montre un exemple d'un MTBDD. Les noeuds sont organisés sur des niveaux horizontaux, un niveau par variable booléenne. La variable $var(s)$ pour un noeud s est indiquée à gauche du niveau qui la contient. Les deux enfants d'un noeud s sont connectés à ce dernier par des arêtes, une arête sous forme de ligne continue pour l'enfant*



(b)

x_1	x_2	x_3	f_M
0	0	1	3
0	1	0	9
1	1	0	9
1	0	0	4
1	0	1	4
Autrement			0

FIGURE 4.1: Un MTBDD M ainsi que sa fonction correspondante f_M.

right(s) *et une arête sous forme de pointillés pour left*(s)*. Les feuilles sont représentées sous forme de carrés au lieu des cercles pour les non-terminaux avec leur valeur* $val(s)$*. Pour les besoins de clarté, nous omettons les feuilles avec une valeur nulle et toute autre arête qui mène vers une feuille du genre.*

Si $V = \{x_1, x_2, \cdots, x_n\}$ est un ensemble de variables booléennes et $<$ un ordre total sur V, un MTBDD M sur $\langle V, < \rangle$ représente une fonction $f_M(x_1, x_2, \cdots, x_n) : \mathbb{B}_n \to \mathbb{R}$. La valeur de $f_M(x_1, x_2, \cdots, x_n)$, la fonction correspondant au MTBDD, est déterminée en traversant le MTBDD à partir de la racine. A chaque noeud non terminal s, on choisit la transition *right*(s) si $var(s)$ est 1, ou la transition *left*(s) si $var(s)$ est 0. La fonction représentée par le MTBDD de la figure 4.1(a) est montrée dans la figure 4.1(b). Dans cet exemple, la valeur de $f_M(0,1,0) = 9$ et $f_M(0,0,1) = 3$.

On utilise aussi les notations $f_M[x_1 = 0, x_2 = 0, x_3 = 1]$ ou $f_M[V = (0,0,1)]$ si on veut exprimer la valeur d'un MTBDD sans faire une référence explicite à l'ordre sur ses variables booléennes.

4.1.1 Réduction de la taille des MTBDD

La taille d'un MTBDD correspond au nombre de noeuds qui le composent. Les MTBDD sont une structure de données qui permettent de faire un stockage compact des données car ils sont représentés sous forme réduite.

Pour réduire la taille des MTBDD, Bryant[10] a ajouté à la définition des MTBDD trois contraintes supplémentaires connues sous le nom de *règles de réduction* :

1. pour toutes feuilles s et t,
$$val(s) = val(t) \Rightarrow s = t$$

2. pour tous noeuds s et t,
$$((var(s) = var(t)) \wedge (right(s) = right(t)) \wedge (left(s) = left(t))) \Rightarrow s = t$$

3. pour tout noeud s,
$$left(s) \neq right(s)$$

Les deux premières règles permettent la *fusion de noeuds* identiques. Si un noeud non terminal s et un noeud t sont identiques (i.e. $var(s) = var(t)$, $right(s) = right(s)$ et $left(s) = left(s)$), une seule copie du noeud est sauvegardée. On fait de même si 2 feuilles s et t sont étiquetées par la même valeur réelle (i.e. $val(s) = val(t)$).

La troisième règle permet le *saut de niveau*, c'est-à-dire enlever du MTBDD un noeud s si ses deux enfants sont identiques. En effet, si un noeud s est tel que $right(s) = left(s)$, ce noeud est supprimé et toutes les arêtes entrantes vers ce noeud sont redirigées vers son unique enfant.

Exemple 4.1.3. *La figure 4.2 illustre la manière dont le MTBDD de la figure 4.1 a été réduit : (a) montre l'arbre binaire complet correspondant à la fonction. Dans (b), les noeuds identiques ont été fusionnés. Pour les besoins de simplification, les feuilles avec la valeur zéro sont ignorées. Nous verrons dans la section suivante qu'elles représentent des transitions qui n'existent pas (avec une probabilité nulle) entre deux états d'un système probabiliste. Dans (c), les noeuds avec des enfants identiques ont été supprimés introduisant ainsi des niveaux qui ont été enlevés.*

Dans le reste du travail, sauf si autrement spécifié, nous considérons que les MTBDD sont totalement réduits.

Le MTBDD représentant une fonction dépend de l'ordre des variables choisi. Selon l'ordre des variables, il peut y avoir deux MTBDD différents qui représentent la même fonction. Cependant, pour un ordre de variable donné, la représentation par MTBDD est canonique c'est-à-dire qu'il existe une correspondance un à un entre une fonction donnée et le MTBDD qui lui correspond.

Théorème 4.1.4. (**Règle de la canonicité**) [10]
Soit $V = \{x_1, x_2, \cdots, x_n\}$, un ensemble de variables booléennes et $<$ un ordre total sur V. Pour toute paire de MTBDD M et M' sur $\langle V, < \rangle$, on a que :

$$f_M = f'_M \Leftrightarrow M \text{ et } M\text{' sont isomorphes.}$$

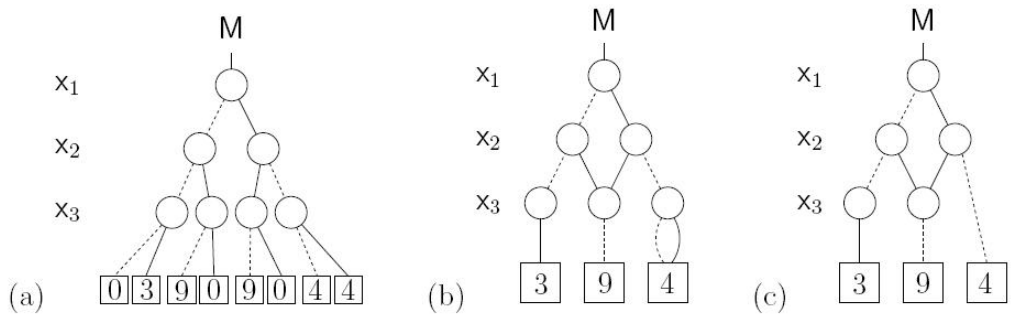


FIGURE 4.2: Réduction de la taille d'un MTBDD [37].

Une conséquence du théorème 4.1.4 est que l'ordre d'application des règles de réduction n'a pas d'importance puisqu'une fois réduit, le MTBDD obtenu sera toujours le même.

Une autre conséquence est que vérifier si deux MTBDD sont égaux se fait en temps constant : typiquement, on regroupe tous les MTBDD calculés dans un seul graphe. Pour vérifier l'égalité, on n'a qu'à regarder si les deux pointent vers le même noeud.

La taille (le nombre de noeuds) d'un MTBDD M sur $\langle V, < \rangle$ représentant une fonction donnée dépend essentiellement de l'ordre $<$ sur l'ensemble des variables booléennes V. Par exemple, le MTBDD (BDD pour être plus précis) représentant la fonction $f(x_1, x_2, x_3, x_4) = x_1x_2 + x_3x_4$ avec l'ordre des variables $x_1 < x_2 < x_3 < x_4$ est représenté par le MTBDD de la figure 4.3(a). Si par contre, pour la même fonction nous choisissons l'ordre $x_1 < x_3 < x_2 < x_4$, nous obtenons un MTBDD de taille plus grande, celui de la figure 4.3(b).

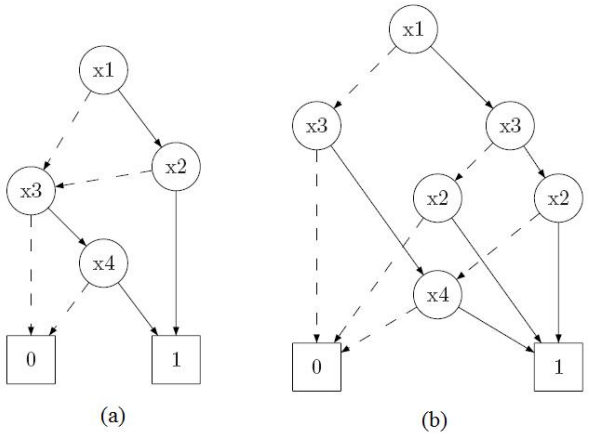


FIGURE 4.3: Variation de l'ordre des variables sur un BDD.

L'ordre des variables joue donc un rôle très important dans la taille d'un MTBDD. Malheureusement, déterminer l'ordre qui minimise la taille d'un MTBDD est un problème NP-complet [45]. On se rabat généralement sur des heuristiques pour déterminer l'ordre optimal approximatif.

Remarque 4.1.5. *Dans le cadre de ce travail, nous ne nous intéressons pas aux choix de l'ordre des variables qui minimise la taille du MTBDD. L'ordre choisi peut être optimal ou pas. Par contre, nous supposons que pour une fonction donnée, l'ordre choisi pour construire le MTBDD est toujours le même, quelle que soit la procédure décrite. En d'autres termes, nous supposons que les procédures décrites dans ce travail disposent de la même politique pour choisir un ordre de variable sur un MTBDD. Cela implique que pour une fonction donnée, deux procédures choisiront le même ordre.*

4.1.2 Représentation de systèmes probabilistes

Nous avons déjà vu au chapitre 2 que les MTBDD permettent d'encoder un système de transition ou plutôt sa matrice de transition. Les MTBDD servent à représenter des fonctions $f : \mathbb{B}_n \to \mathbb{R}$ où n représente le nombre de variables booléennes nécessaires pour représenter l'ensemble des états du système de transition. En effet, nous devons être en mesure de décrire l'ensemble des états et des transitions du système de transition sous forme d'une fonction $f : \mathbb{B}_n \to \mathbb{R}$ pour encoder un système sous forme de MTBDD. Pour déterminer cette fonction, nous numérotons chaque état du système de transition de 0 à $|S| - 1$ avec S qui est l'ensemble des états du système de transition. Cette fonction est donnée par la matrice des transitions.

Pour avoir une fonction de type $f : \mathbb{B}_n \to \mathbb{R}$, nous établissons une bijection entre les ensembles $\{x \in \mathbb{N} : x < 2^{\lceil log_2|S| \rceil}\}$ et $\{0,1\}^{\lceil log_2|S| \rceil}$. La bijection la plus utilisée est la représentation en binaire des nombres. Si S est l'ensemble des états du système de transition, nous avons besoin de $2^{\lceil log_2|S| \rceil}$ variables booléennes (ou bits) pour représenter l'ensemble des états sous forme binaire. Par exemple, si nous avons au plus quatre états, on aura besoin de deux variables booléennes et si nous avons huit états, nous avons besoin trois variables booléennes.

Nous obtenons une fonction :

$$f : \mathbb{B}^{2\lceil log_2|S| \rceil} \to \mathbb{R}.$$

Nous pouvons dès lors considérer la matrice des transitions $n \times n$ comme une fonction :

$$f(x_1, \cdots, x_{\lceil log_2|n| \rceil}, y_1, \cdots, y_{\lceil log_2|n| \rceil}) \to \mathbb{R}.$$

Cette fonction est entièrement définie par la matrice des transitions. Une transition est identifiée par son état de départ et son état d'arrivée. Pour distinguer ces deux types d'états, nous étiquetons les états de départ par les variables x_i et ceux d'arrivée par y_i.

Définition 4.1.6. *Soit S, un système de transition probabiliste.*
On désigne par $MTBDD(S)$, la procédure qui construit le MTBDD correspondant à S.

Cette procédure utilise un encodage binaire pour représenter l'ensemble des états de S à l'aide de variables booléennes. De plus, elle choisit un ordre sur les variables booléennes qui minimise la taille du MTBDD correspondant au système de transition S.

Algorithm 1 Procédure $MTBDD(S)$

Étiqueter tous les états de S de 0 à $|S-1|$;
Déterminer l'ensemble des variables booléennes :

$$V = \{x_1, \cdots, x_{\lceil log_2 |S| \rceil}, y_1, \cdots, y_{\lceil log_2 |S| \rceil}\}$$

Choisir un ordre total $<$ sur l'ensemble V ;
Déterminer $f : \mathbb{B}^{2\lceil log_2 |S| \rceil} \to \mathbb{R}$ correspondant à la matrice de transition de S ;
Construire le MTBDD sur $\langle V, < \rangle$ correspondant à f.

Théorème 4.1.7. (**Règle de la canonicité pour système de transitions**) [10]
Soient S et S' deux systèmes de transitions, on a que :

$$S \equiv S' \Leftrightarrow MTBDD(S) \text{ et } MTBDD(S') \text{ sont isomorphes.}$$

Le théorème précédent est une conséquence du théorème 4.1.4. Il stipule qu'il existe une correspondance un à un entre un système de transition donné et le MTBDD qui lui correspond à l'instar de la correspondance entre un MTBDD M et la fonction f_M qui lui correspond.

4.2 Bisimulation approchée

Nous présentons la notion de bisimulation (simulation) approchée définie par [14] pour faire la comparaison de systèmes probabilistes. Cette notion a été définie pour les LMP (*Labelled Markov Process*). Nous montrons d'abord qu'elle peut être appliquée aux DTMC sans perte de signification.

4.2.1 Bisimulation approchée pour DTMC

Définition 4.2.1. (LMP) *Soit un ensemble d'actions L aussi appelé étiquettes. Un Labelled Markov Process est un tuple $\mathcal{S} = (S, \Sigma, h : L \times S \times \Sigma \to [0;1], i)$ tel que (S, Σ) est un espace mesurable, $i \in S$ est l'état initial et pour tout $a \in L$, $s \in S$, $h(a, s, \cdot)$ est une mesure de sous-probabilité sur (S, Σ).*

Nous prenons pour acquis que l'espace des états S est dénombrable et que $\Sigma = \mathcal{P}(S)$ où $\mathcal{P}(S)$ est l'ensemble puissance de S. Si l'ensemble S n'était pas dénombrable, une condition de mesurabilité de $h(a, \cdot, X)$ devrait être ajoutée à la définition précédente. La valeur $h(a, s, X)$ que nous notons $h_a(s, X)$ dénote la probabilité que le LMP passe de l'état s à l'ensemble des états X en effectuant l'action a.

Un LMP est par conséquent un MDP avec la contrainte que h est maintenant une mesure de probabilité, i.e., $\sum h(a, s, \cdot) \leq 1$ pour les LMP, alors que $\sum h(a, s, \cdot) \in \{0; 1\}$ pour les MDP.

Or un DTMC est un cas particulier d'un MDP avec une seule action autorisée dans l'ensemble des actions L (action qui est omise pour les besoins de simplification).

Pour le cas des CTMC, on remarque aisément qu'une généralisation de la notion de bisimulation approchée est applicable aussi aux CTMC.

Il en ressort que la notion de ϵ-bisimulation, qui a été définie pour les LMP, est applicable pour les DTMC. Nous pouvons dire sans perte de signification de deux DTMC M et M' qu'ils sont ϵ-bisimilaires notée $M \sim_\epsilon M'$.

Nous désignerons dans ce qui suit par *système probabiliste*, les modèles probabilistes auxquels nous pouvons appliqué la notion de (bi)simulation approchée.

4.2.2 ϵ-simulation et ϵ-bisimulation

La notion de (bi)simulation approchée dérive de la notion (bi)simulation probabiliste classique. Intuitivement, un état simule un second s'il peut faire au moins ce que le second peut faire. Ceci peut être formalisé de plusieurs façons. La définition de la simulation suivante est une définition structurelle. Elle se base sur l'existence d'une relation particulière entre les états d'un système probabiliste.

Définition 4.2.2. *Soient $\mathcal{S} = (S, \Sigma, h, i)$ un système probabiliste, $\mathcal{R}$ une relation binaire sur S et X, un sous-ensemble de S. L'ensemble X est dit $\mathcal{R}$-clos si, pour tout $s \in X$, dès que nous avons $s\mathcal{R}t$ alors $t \in X$.*

Définition 4.2.3. *Soit $\mathcal{S} = (S, \Sigma, h, i)$ un système probabiliste. Une relation binaire $\mathcal{R}$ sur S est une simulation si dès que $s\mathcal{R}t$, nous avons que pour tout $a \in L$ et tout ensemble $X \in \Sigma$ qui est $\mathcal{R}$-clos, $h_a(s, X) \leq h_a(t, X)$. Nous dirons qu'un état s est similaire à un état t s'il existe une relation de simulation $\mathcal{R}$ sur S telle que $s\mathcal{R}t$.*

On peut passer directement d'une relation de simulation entre états à une relation de simulation entre systèmes probabilistes, en prenant en compte les états initiaux : deux systèmes probabilistes $\mathcal{S}$ et $\mathcal{T}$ sont en relation de simulation si leurs états initiaux le sont quand on les considère comme états du système probabiliste résultant de l'union de $\mathcal{S}$ et $\mathcal{T}$.

Définition 4.2.4. *Soit $\mathcal{S} = (S, \Sigma, h, i)$ un système probabiliste. Une relation binaire $\mathcal{R}$ sur S est une bisimulation si dès que $s\mathcal{R}t$, nous avons que pour tout $a \in L$ et tout ensemble $X \in \Sigma$ qui est $\mathcal{R}$-clos, $h_a(s, X) = h_a(t, X)$. Nous dirons qu'un état s est bisimilaire à un état t s'il existe une relation de bisimulation $\mathcal{R}$ sur S telle que $s\mathcal{R}t$.*

La bisimulation est équivalente à une simulation dans les deux sens [14, 47]. C'est à dire, deux états s et t sont bisimilaires ssi s est simulé par t et t est simulé par s.

La définition suivante de la ϵ-simulation est une généralisation de la définition structurelle de la simulation. Elle définit des *relations de ϵ-simulation*.

Définition 4.2.5. (Relation de ϵ-simulation) *Une relation $\mathcal{R} \subseteq S \times S$ est une relation de ϵ-simulation si dès que $s\mathcal{R}t$, on a que pour tout $a \in L$, $X \in \Sigma$, $\epsilon \in [-1, 1]$,*

$$h_a(t, \mathcal{R}(X)) \geq h_a(s, X) - \epsilon$$

Deux états s et t sont ϵ-similaires, noté $s \preceq_\epsilon t$, s'il existe une relation de ϵ-simulation $\mathcal{R}$ sur S telle que $s\mathcal{R}t$.

Comme pour la simulation classique, Desharnais et al. [14] ainsi que Tracol [47] montrent que, pour un système avec un nombre dénombrable d'états, $\preceq_\epsilon$ est la plus grande relation de ϵ-simulation et elle peut-être calculée comme le plus grand point fixe d'un opérateur.

Exemple 4.2.6. *Soit $\epsilon > 0$. Considérons les systèmes suivants :*

Pour tout $\epsilon' > \epsilon$, nous avons $s \preceq_{\epsilon'} t$ et $s \preceq_{\epsilon'} t$.

L'exemple précédent reflète typiquement la comparaison que nous souhaitons faire entre systèmes probabilistes. En effet, partant du principe que le modèle d'un système ne peut refléter exactement la réalité, nous essayons de comparer les modèles avec un certain degré de tolérance. Plus précisément, $\mathcal{S}$ peut être vu comme un modèle du système réel $\mathcal{T}$ (et inversement). Nous souhaitons pouvoir les considérer comme équivalents si nous pouvons tolérer une erreur de ϵ.

Définition 4.2.7. (Relation de ϵ-bisimulation) *Une relation $\mathcal{R} \subseteq S \times S$ est une relation de ϵ-bisimulation si c'est une relation de ϵ-simulation symétrique. Deux états s et t sont ϵ-bisimilaires, noté $s \sim_\epsilon t$, s'il existe une relation de ϵ-simulation $\mathcal{R}$ sur S telle que $s\mathcal{R}t$.*

Dans l'exemple 4.2.6, nous avons $s \sim_{\epsilon'} t$ pour tout $\epsilon' > \epsilon$.

Dans la définition précédente de la bisimulation approchée, la symétrie de la relation pourrait être remplacée par l'exigence que l'inverse de $\mathcal{R}$ soit une relation de ϵ-simulation de même que $\mathcal{R}$.

4.3 Equivalence approchée

À l'instar de la notion de bisimulation approchée introduite pour atténuer la comparaison entre systèmes probabilistes, nous introduisons dans cette section, la notion de ϵ-equivalence entre MTBDD pour comparer des MTBDD qui sont quasi isomorphes. Nous voulons déterminer si deux DTMC qui sont ϵ-bisimilaires ont des MTBDD qui sont aussi ϵ-bisimilaires. Pour cela, nous introduisons une notion de similitude directement sur les MTBDD plutôt que de seulement comparer les DTMC correspondants. Cette notion sera définie à partir de la réduction suivante, qui prend une liste d'éléments en entrée et en retourne une version filtrée par la précision ϵ.

Définition 4.3.1. *(ϵ-**fonction de réduction**). Soit $\epsilon \geq 0$, $V[0 \cdots n-1]$, un tableau indexé de réels. Une ϵ-fonction de réduction est une fonction injective $f_\epsilon : V \to V$ qui, à chaque élément $V[j]$ de V, associe un élément $V[i]$ de V tel que :*

1. *$i < j$,*
2. *i est le plus petit indice tel que $|V[i] - V[j]| \leq \epsilon$.*

L'ϵ-fonction de réduction procède comme suit : elle parcourt le tableau V du plus petit indice au plus grand et pour chaque indice j, elle vérifie si elle n'a pas déjà sauvegardé en position $i < j$, un élément ϵ-équivalent à $V[j]$. Si c'est le cas, seul $V[i]$ est conservé, $V[j]$ est supprimé car considéré comme un doublon de $V[i]$.

Il faut noter que les indices sont ici importants. Ils créent un ordre $\prec$ sur les éléments du tableau. Il est à noter que cet ordre ne correspond pas à l'ordre numérique. On peut choisir comme le montre l'exemple qui suit de sauvegarder 9 à la place de 8 car apparu avant. L'ϵ-fonction de réduction consiste donc à éliminer les doublons qui sont ϵ-équivalents et ne conserver qu'une copie (celui avec le plus petit indice).

Exemple 4.3.2. *Soit [4,9,8,6,7] un tableau indexé de réels.*

a. $f_1([4,9,8,6,7]) = [4,9,6]$.

b. $f_2([4,9,8,6,7]) = [4,9]$

Dans l'exemple a, 9 et 8 sont 1-équivalents. La 1-fonction de réduction conserve seulement la valeur 9 car de plus petit indice. 6 et 7 sont aussi 1-équivalents, 6 est seulement conservé pour les mêmes raisons. Dans l'exemple b, 8 et 7 sont 2-équivalents à 9 tandis que 6 est 2-équivalent à 4. Les nombres 8, 6 et 7 ont donc été supprimés par la 2-fonction de réduction.

Définition 4.3.3. (La relation ϵ-équivalence). *Deux MTBDD M et M' sont ϵ-équivalents, noté $M \equiv_\epsilon M'$, s'il existe une ϵ-fonction de réduction f_ϵ telle que M' peut-être obtenu à partir de M (ou vice-versa) en appliquant f_ϵ sur l'ensemble des feuilles de M selon un ordre $\prec$.*

Si $M \equiv_\epsilon M'$ et que V représente le tableau indexé de l'ensemble des feuilles du MTBDD M selon l'ordre $\prec$, le tableau indexé V' de l'ensemble des feuilles du MTBDD M' est donné par : $f_\epsilon(V) = V'$.

Plus clairement, on obtient un MTBDD M' qui est ϵ-équivalent à un MTBDD donné M en procédant comme suit : considérons l'ensemble des feuilles de M selon un ordre total $\prec$ sur cet ensemble. On obtient un tableau indexé. Après avoir appliqué une ϵ-fonction de réduction à ce tableau indexé des feuilles de M (selon l'ordre $\prec$), le nombre de feuilles restant est inférieur ou égal au nombre de feuilles de départ. Le MTBDD résultant peut-être alors non réduit. Les règles de réduction sont encore appliquées pour le rendre réduit. C'est ce MTBDD réduit qui est ϵ-équivalent à M.

Exemple 4.3.4. *La figure 4.4 montre deux MTBDD M et M'' qui sont $\equiv_{0.1}$. Dans le MTBDD M, on considère le tableau indexé $V = [0.1, 0.18, 0.5]$ de l'ensemble des feuilles. En appliquant à V une 0.1-fonction de réduction, on obtient le MTBDD M'. M'' est obtenu en réduisant M'. Le MTBDD M'' est donc $\equiv_{0.1}$ au MTBDD M. On note aussi que M' est égal à M'', seule une réduction les différencie.*

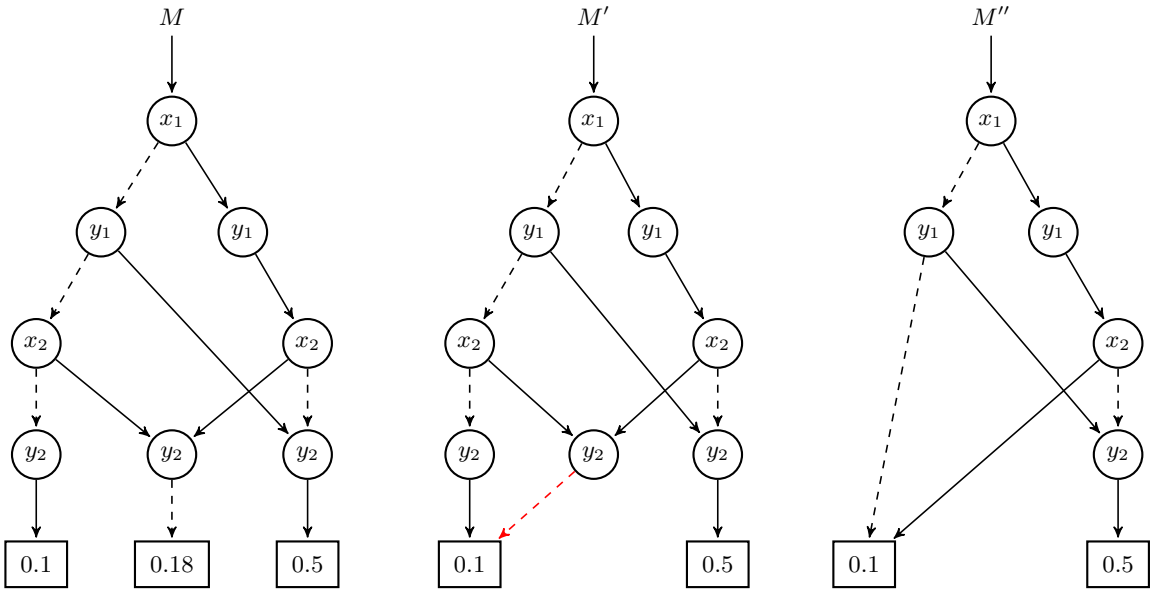


FIGURE 4.4: $M \equiv_{0.1} M''$ en considérant l'ordre des feuilles [0.1,0.18,0.5].

Définition 4.3.5. (Degré sortant). *On appelle degré sortant $d^-(s)$ d'un sommet s d'un système de transition, le nombre de transitions sortant de s.*

Nous appellerons *degré sortant* Δ^- d'un système de transitions, son degré sortant maximal.

Exemple 4.3.6. *Le système de transition de la figure 4.5 est de degré sortant 3. L'état qui a le nombre maximal de transitions sortantes est l'état s_2. Les états s_1, s_3 et s_4 ont une transition sortante tandis que les états s_5 et s_6 en ont deux.*

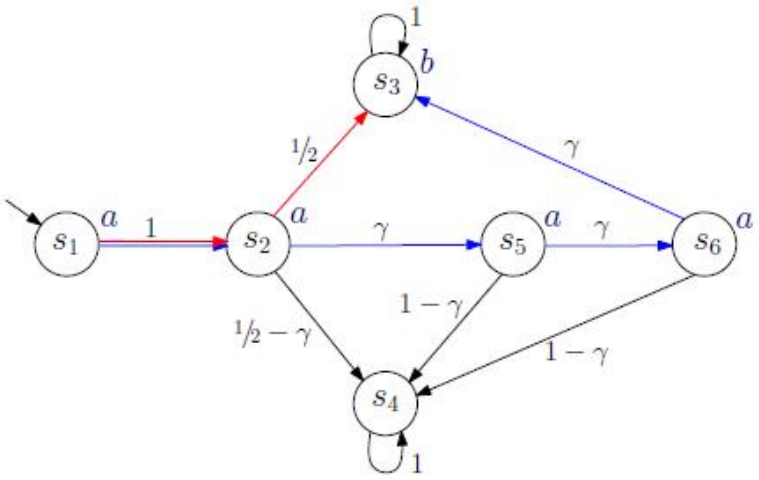


FIGURE 4.5: Un système probabiliste.

Remarque 4.3.7. *On remarque aisément que si $MTBDD(S) \equiv_\epsilon MTBDD(S')$ alors S et S' ont le même nombre d'états et l'ensemble des transitions de l'un est inclus dans l'autre. En effet, appliquer une ϵ-fonction de réduction à $MTBDD(S)$ modifie seulement la valeur de ses feuilles et donc des transitions dans S. Elle ne modifie pas l'ensemble des états du système probabiliste S. Les transitions communes à S et S' peuvent avoir des valeurs différentes ou pas. Les transitions propres à S correspondent à des transitions à valeurs nulles dans S' et sont donc omises. Pour distinguer les deux systèmes de transitions, on désigne par s, un état de S et par s', son correspondant dans S'.*

Le théorème suivant représente l'aboutissement de la comparaison entre systèmes de transition à partir des MTBDD correspondants. Il dit que si 2 MTBDD sont ϵ-équivalents alors les systèmes de transitions correspondants sont $\delta\epsilon$-bisimilaires, avec δ, un entier naturel qui est tel que : $\delta \leq \max(|S|, |S'|)$.

Théorème 4.3.8. *Pour toute paire de systèmes de transitions $\mathcal{S} = (S, \Sigma, h, i)$ et $\mathcal{S}' = (S', \Sigma', h', i')$, on a que :*

$$MTBDD(S) \equiv_\epsilon MTBDD(S') \Rightarrow \mathcal{S} \sim_{\delta\epsilon} \mathcal{S}'$$

où $\delta = \max(\Delta^-(S), \Delta^-(S'))$.

Démonstration : Nous notons que d'après la remarque 4.3.7, nous savons que si $MTBDD(S) \equiv_\epsilon MTBDD(S')$ alors $\mathcal{S}$ et $\mathcal{S}'$ ont le même nombre d'états et l'ensemble des transitions de l'un est inclus dans l'autre tandis que les transitions communes à $\mathcal{S}$ et $\mathcal{S}'$ peuvent avoir des valeurs différentes ou pas.

Soit f, la valeur d'une feuille de $MTBDD(S)$. Elle correspond à la probabilité $h_a(s,t)$ de faire l'action a et de passer de l'état s vers l'état t dans le système de transition $\mathcal{S}$. Dans $MTBDD(S')$, f correspond à une nouvelle feuille f' qui est telle que :

$$f - \epsilon \leq f' \leq f + \epsilon$$

De ce qui précède, il en ressort que dans S', la probabilité $h'_a(s',t')$ de faire l'action a pour passer de l'état s' vers l'état t' est telle que :

$$h_a(s,t) - \epsilon \leq h'_a(s',t') \leq h_a(s,t) + \epsilon \tag{4.1}$$

De 4.1, nous pouvons déduire que :

$$\left(\sum_{t\in X} h_a(s,t)\right) - \delta\epsilon \leq \sum_{t\in X}(h_a(s,t) - \epsilon) \leq \sum_{t'\in X'} h'_a(s',t') \tag{4.2}$$

La première inéquation de 4.2 vient du fait que $|X| < \delta$ (le nombre de transitions sortantes reliant un état s vers l'ensemble des états X est inférieur au nombre maximal $\delta = \Delta^-(S)$ des transitions sortantes d'un état du système). Nous pouvons appliquer le même raisonnement pour la borne supérieure de $h'_a(s',t')$:

$$\sum_{t'\in X'} h'_a(s',t') \leq \sum_{t\in X}(h_a(s,t) + \epsilon) \leq \left(\sum_{t\in X} h_a(s,t)\right) + \delta\epsilon \tag{4.3}$$

De 4.2 et 4.3, on a que :

$$\left(\sum_{t\in X} h_a(s,t)\right) - \delta\epsilon \leq \sum_{t'\in X'} h'_a(s',t') \leq \left(\sum_{t\in X} h_a(s,t)\right) + \delta\epsilon \tag{4.4}$$

De ce qui précède, nous pouvons déduire que $\mathcal{S} \sim_{\delta\epsilon} \mathcal{S}'$.

□

Le théorème 4.3.8 décrit l'impact sur le système de transitions S d'une ϵ-fonction de réduction appliquée sur le $MTBDD(S)$ qui lui correspond. Appliquer une ϵ-fonction de réduction à $MTBDD(S)$ générera $MTBDD(S')$ qui est ϵ-équivalent. Le système de transition S' correspondant à $MTBDD(S')$ est tel que : $S \sim_{\delta\epsilon} S'$.

Comme nous le verrons dans la section suivante, en pratique l'encodage d'un système probabiliste sous forme de MTBDD est modifiée pour des raisons d'optimisation (gain de mémoire et de temps). Le MTBDD est progressivement construit, une nouvelle feuille v n'est conservée que s'il n'existe pas d'autre feuille v' qui soit équivalente à un certain seuil ϵ près. Nous distinguons donc le MTBDD théorique correspondant au système de transition encodé (celui qui devrait être généré) et celui construit en pratique. Le résultat du théorème 4.3.8 nous permettra d'analyser l'effet des perturbations lors de l'encodage en pratique d'un système de transitions sous forme de MTBDD.

Remarque 4.3.9. *La figure 4.6 met en évidence le fait que si deux MTBDD sont ϵ-équivalents, cela n'implique pas que les systèmes de transitions correspondants soient ϵ-bisimilaires (ils peuvent l'être !). Dans cet exemple, $MTBDD(S) \equiv_{0.1} MTBDD(S')$ et $\delta = 2$. Pour $\delta' = 1$, $S \not\sim_{0.1} S'$ puisque par exemple $h(0, \{0, 1\}) = 0.6$ dans S et $h'(0', \{0', 1'\}) = 0.4$ dans S'. Or $h(0, \{0, 1\}) - h'(0', \{0', 1'\}) = 0.2 > 0.1 = \delta'$. Cet exemple montre que pour tout $\delta' < \delta$, on peut trouver S et S' tel que $MTBDD(S) \equiv_\epsilon MTBDD(S')$ mais $S \not\sim_{\delta'\epsilon} S$. En effet, si $MTBDD(S) \equiv_\epsilon MTBDD(S')$ et $\delta = \Delta^-(S)$, alors pour tout $\delta' < \delta$, on n'a pas nécessairement $S \sim_{\delta'\epsilon} S$.*

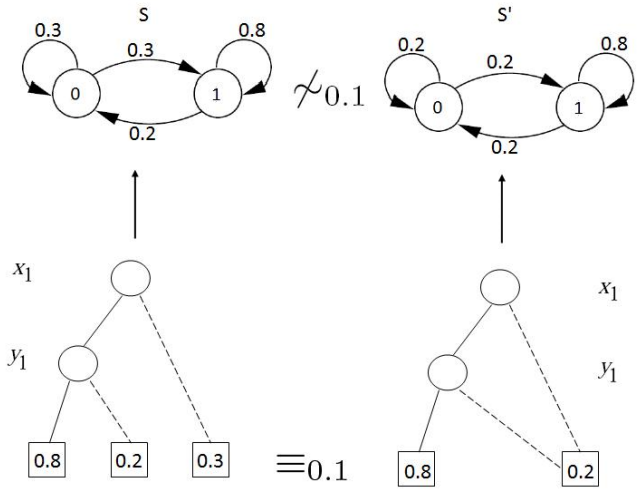


FIGURE 4.6: $MTBDD(S) \equiv_{0.1} MTBDD(S')$ et $S \not\sim_{0.1} S'$

Le théorème suivant met en évidence l'impact des perturbations sur les MTBDD générés et le fait que de telles perturbations s'accumulent. Nous pouvons alors imaginer l'impact de telles accumulations d'erreurs sur le résultat retourné par PRISM, surtout si elles sont combinées avec les autres sources d'erreurs mentionnées au chapitre 3.

Théorème 4.3.10. (Transitivité de l'équivalence approximative)
Soient trois MTBDD, M, M' et M'', on a que :

$$(M \equiv_\epsilon M') \wedge (M' \equiv_\epsilon M'') \Rightarrow M \equiv_{2\epsilon} M''$$

Démonstration : Pour une même valuation sur l'ensemble des variables booléennes, i.e., pour une valeur d'une feuille donnée v dans M, on s'intéresse à la valeur v'' qui lui correspond dans M''.

1. Si $M \equiv_\epsilon M'$, alors la valeur v' correspondant à v dans M' est telle que : $v - \epsilon \leq v' \leq v + \epsilon$.
2. Si $M' \equiv_\epsilon M'$, alors v'' est telle que : $v' - \epsilon \leq v'' \leq v' + \epsilon$.
3. De (1) et (2), nous obtenons que : $v - 2\epsilon \leq v'' \leq v + 2\epsilon$.

□

4.4 Implémentation avec PRISM

Dans cette section, nous décrivons la manière dont les MTBDD sont utilisées en pratique par le model-checker PRISM. Puisqu'il s'agit de développer des structures de données efficientes et d'analyser leurs performances, il est vital d'analyser leur implémentation réelle.

Nous avons déjà souligné dans les chapitres précédents que, pour encoder les systèmes probabilistes sous forme de MTBDD, PRISM utilise le package CUDD qui, à l'opposé des autres packages traitant les BDD, traite également les MTBDD.

Nous examinons la manière dont les MTBDD sont stockés. Dans la section 3.3, nous avons vu que les MTBDD sont conservés sous une forme réduite où les noeuds identiques ne sont pas dupliqués et que dans la pratique, cette règle est accentuée.

Tous les MTBDD en usage à un moment donné sont maintenus dans une large structure de données, en fait un MTBDD très grand avec plusieurs noeuds racines. Cela signifie que des noeuds identiques sur différents MTBDD ne sont pas dupliqués. À tout moment lors d'une opération sur un MTBDD où un nouveau noeud doit être créé, on vérifie d'abord si le noeud existe déjà et si c'est le cas, on le réutilise.

Afin de s'assurer que ces vérifications puissent être effectuées rapidement, les noeuds sont stockés dans un ensemble de tables de hachage, un pour chaque niveau (i.e. par variable du MTBDD). Cette structure est généralement connue sous le nom de la *table unique*.

Les feuilles du MTBDD sont également stockées de la même manière. Leurs valeurs sont stockées sous forme de réels en utilisant le format double précision des nombres à virgule flottante. Nous avons montré à la section 3.3 que dans la pratique, deux feuilles sont considérées comme identiques si la différence entre leurs valeurs est inférieure à un certain seuil minime ϵ.

Dans le package Cudd, la valeur par défaut pour ϵ est de 10^{-12}. PRISM réduit ce paramètre à 10^{-15} de l'ordre de la précision d'un double, afin de minimiser une erreur d'arrondi dans ses calculs numériques.

Dans le fonctionnement normal, des MTBDD sont constamment créés et détruits. Par conséquent, il devient nécessaire de garder une trace des noeuds qui sont nécessaires à un certain moment et ceux qui ne le sont plus. Cette problématique est compliquée par le fait que les noeuds sont stockés dans une structure commune (i.e la table unique) et un noeud donné peut être contenu dans plusieurs MTBDD. Cette contrainte est contournée en conservant un compteur pour chaque référence sur un noeud, indiquant combien de MTBDD en usage à un moment donné, contiennent ce noeud. Si le compteur de références atteint zéro, le noeud est alors considéré comme un noeud mort et il peut être enlevé. De plus, puisque la suppression d'un noeud mort à partir de la table unique peut potentiellement impliquer la restructuration de la structure de données entière, les noeuds morts sont seulement enlevés périodiquement, généralement lorsque leur nombre dépasse un certain seuil prédéfini. Ce processus est appelé la *collecte des ordures*.

Désignons par $MTBDD_\epsilon(S)$, la procédure utilisée par PRISM pour construire un MTBDD correspondant à un système de transition S. Entre autres particularités, $MTBDD_\epsilon(S)$ construit le MTBDD correspondant à S et le maintient réduit en tout temps de son utilisation. Le théorème suivant montre que le MTBDD résultant de la procédure $MTBDD_\epsilon(S)$ est en fait un MTBDD qui est ϵ-équivalent à celui généré par la procédure $MTBDD(S)$.

Théorème 4.4.1. *Pour tout système de transition S, on a que :*

$$MTBDD(S) \equiv_\epsilon MTBDD_\epsilon(S)$$

Démonstration : Soient le $MTBDD(S)$ correspondant au système de transition S et $MTBDD_\epsilon(S)$, le MTBDD généré par la procédure de PRISM.

1. En appliquant une ϵ-fonction de réduction à $MTBDD(S)$ (selon l'ordre d'apparition des feuilles durant la procédure $MTBDD_\epsilon(S)$), on obtient un MTBDD $MTBDD'(S)$ qui est

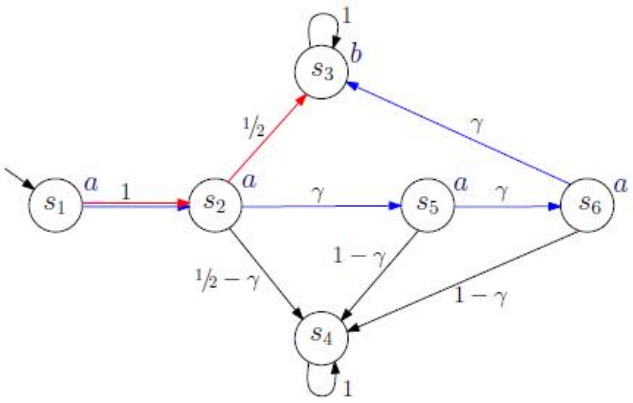


FIGURE 4.7: Un système probabiliste.

tel que :

$$MTBDD(S) \equiv_\epsilon MTBDD'(S)$$

.

2. De par la règle de canonicité des MTBDD, $MTBDD'(S)$ est identique à $MTBDD_\epsilon(S)$, c'est-à-dire que :

   $$MTBDD'(S) \equiv MTBDD_\epsilon(S).$$

   En effet,les deux MTBDD ont les mêmes valeurs pour les feuilles et les mêmes transitions correspondant aux feuilles.

3. De par (1) et (2), on a que : $MTBDD(S) \equiv_\epsilon MTBDD_\epsilon(S)$.

□

En résumé, pour encoder un système probabiliste S, PRISM est supposé recourir à la procédure *MTBDD* pour générer le MTBDD correspondant. Cette dernière correspond plutôt à une procédure théorique. Pour des raisons pratiques, PRISM a recours à une procédure *modifiée* que nous avons nommée $MTBDD_\epsilon$. Nous avons montré que cette procédure génère un MTBDD qui est quasi-isomorphe au MTBDD théorique, mais pas identique ($MTBDD(S) \equiv_\epsilon MTBDD_\epsilon(S)$). De ce fait, nous avons montré dès lors que le modèle S' encodé correspond à un modèle proche du modèle initial S ($S \sim_{\delta\epsilon} S'$). Nous avons utilisé la notion de bisimulation approchée pour comparer les deux modèles, le modèle à l'étude ainsi que le modèle réellement encodé. L'exemple suivant montre une situation où le fait que le MTBDD ϵ-équivalent au MTBDD théorique qui est généré par PRISM (ainsi que le système de transition correspondant) est à l'origine d'une mauvaise réponse retourné par le model-checker.

Exemple 4.4.2. *Nous reprenons à la figure 4.7, le système de transition S du chapitre précédent et nous fixons $\gamma = 10^{-5}$. Nous voulons savoir s'il satisfait la formule PCTL $\Phi =$*

$P_{\leq 1/2}(a \mathcal{U} b)$. *Le MTBDD théorique* $MTBDD(S)$ *correspond à la figure 4.8 (31 noeuds). Le MTBDD généré par PRISM,* $MTBDD_\epsilon(S)$ *avec* $\epsilon = \epsilon_c = 10^{-3}$ *est représenté à la figure 4.9 (13 noeuds) tandis que le système de transition* S' *qui correspond à* $MTBDD_\epsilon(S)$*, est représenté à la figure 4.10. De par les théorèmes 4.3.8 et 4.4.1, nous avons que :*

1. $MTBDD(S) \equiv_{10^{-3}} MTBDD_\epsilon(S)$.
2. $S \sim_{3 \cdot 10^{-3}} S'$.

Nous savons depuis le chapitre 3 que la formule Φ *n'est pas satisfaite au départ par le système de transition* S *mais qu'elle l'est dans le système de transition* S' *qui est* $3 \cdot 10^{-3}$*-bisimilaire à* S *d'où la réponse du model-checker qui dit que* Φ *est satisfaite par* S *alors que ce n'est pas le cas.*

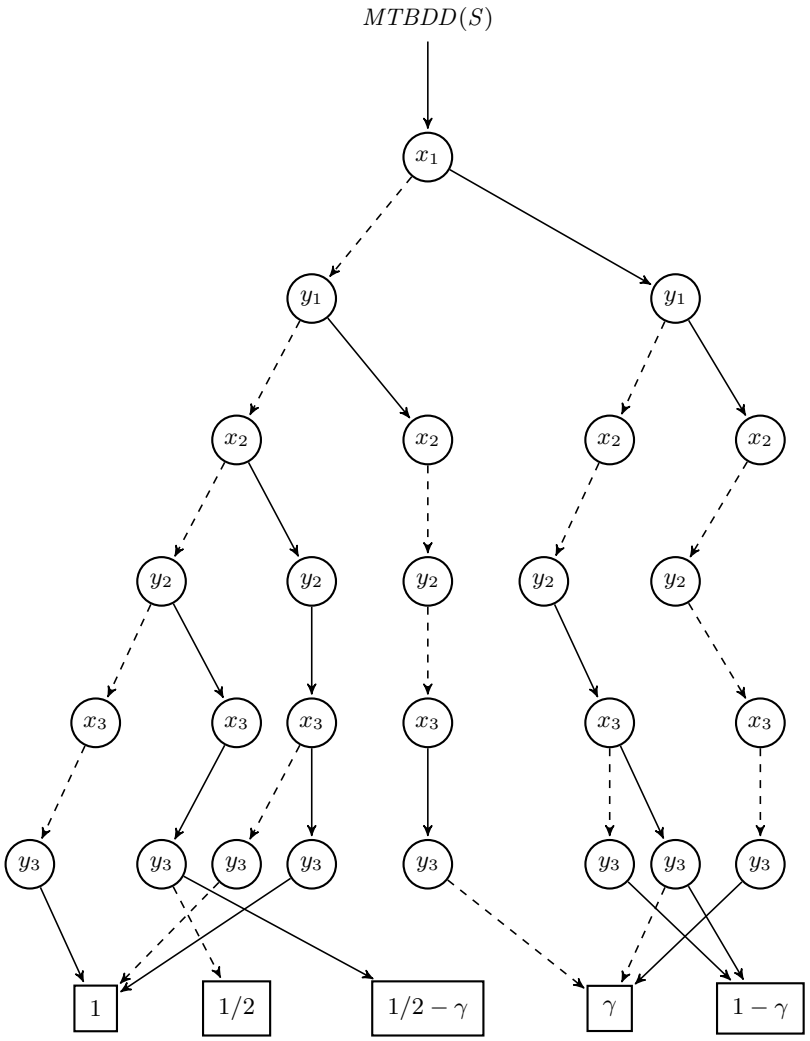


FIGURE 4.8: $MTBDD(S)$ correspondant au système de transition de la figure 4.7.

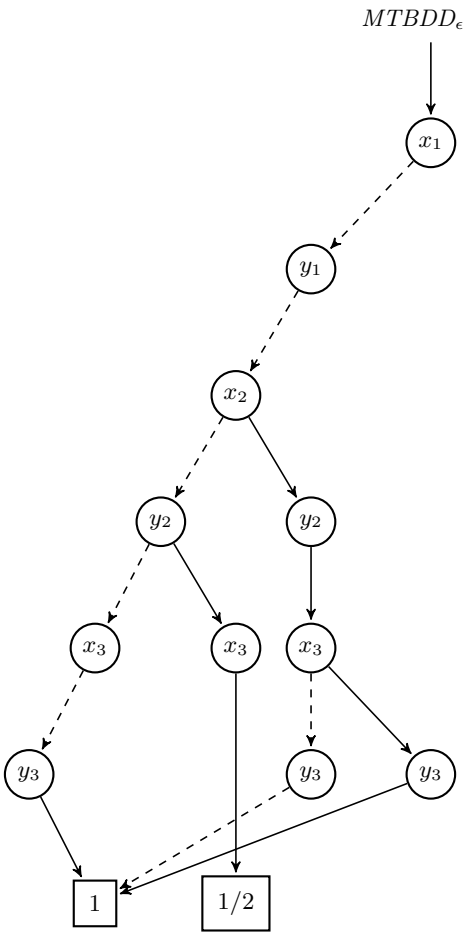


FIGURE 4.9: $MTBDD_\epsilon(S)$ correspondant au système de transition de la figure 4.7.

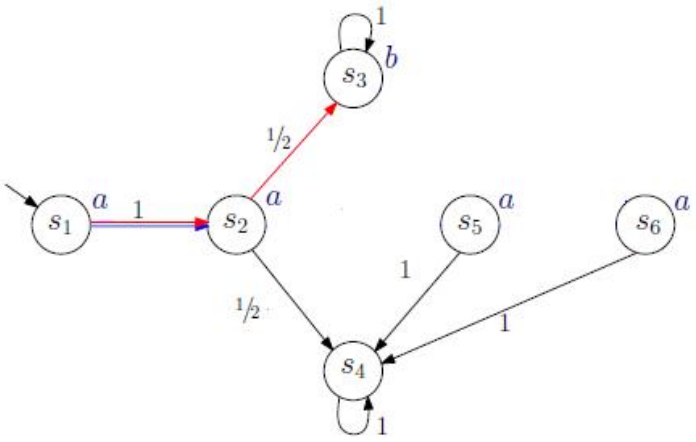


FIGURE 4.10: Système de transition S' correspondant au MTBDD de la figure 4.9.

Dans ce chapitre, nous avons montré qu'en ayant recours à la technique du model-checking symbolique pour vérifier un système probabiliste S, PRISM fait que le système encodé correspond en réalité à un système "*approximatif*" S' du modèle à l'étude. Nous avons utilisé la notion d'équivalence approchée pour montrer que les deux systèmes sont tels que $S \sim_{\delta\epsilon} S'$ où δ représente le nombre maximal de transitions sortant vers un état s du système à l'étude, et ϵ, la précision utilisée par PRISM pour générer les MTBDD. Il est assez aisé de remarquer que $\delta < |S|$. Dans le chapitre qui suit, nous analyserons l'impact des erreurs d'arrondi (arithmétique flottante et recours au model-checking symbolique) sur le résultat retourné par PRISM.

Chapitre 5

Impact sur le résultat et améliorations éventuelles

Au chapitre 3, nous avons montré à quels moments du processus de model-checking, les erreurs apparaissent. Dans ce chapitre, nous analysons l'impact des erreurs sur le résultat retourné par PRISM. Rappelons que le model-checking probabiliste consiste à vérifier si un modèle probabiliste M (ou un état s de M) vérifie une propriété donnée. Le modèle M est représenté sous forme de chaîne de Markov (DTMC, MDP, CTMC) tandis que la propriété est énoncée à l'aide de la logique PCTL pour le cas des DTMC qui nous intéressent. L'algorithme de model-checking décrit à la section 2.3 comporte deux phases selon qu'il s'agit de vérifier une formule énoncée avec l'opérateur probabiliste ou pas :

1. *Analyse qualitative* : vérifier une propriété énoncée sous forme de formule non probabiliste,

2. *Analyse quantitative ou numérique* : vérifier une propriété énoncée à l'aide de l'opérateur probabiliste.

Les erreurs répertoriées apparaissent lors de l'analyse numérique en vérifiant des propriétés quantitatives c'est à dire celles que l'on énonce avec l'opérateur probabiliste P. L'analyse qualitative ne génère pas d'erreurs. Plus concrètement, les erreurs apparaissent quand il s'agit de vérifier une propriété de la forme $P_{\sim p}(\varphi)$. Nous avons déjà montré que vérifier une propriété de cette forme revient à résoudre un système d'équations linéaires de type $Ax = b$. Comme le système à l'étude est composé d'un ensemble S d'états, la matrice A est de taille $|S| \times |S|$. Elle découle de la matrice P des transitions de probabilité du modèle M, matrice à laquelle on applique certaines manipulations que nous décrivons dans la suite de ce chapitre.

En résolvant le système d'équations linéaires $Ax = b$, les erreurs apparaissent :

1. en utilisant la représentation des probabilités à l'aide de nombres à virgule flottante,

2. en ayant recours à des méthodes itératives (au lieu de méthodes directes) pour résoudre le système d'équations linéaires,
3. en représentant la matrice A sous forme de MTBDD.

L'ensemble de ces erreurs constituent l'*erreur numérique*. Nous pouvons dégager les *erreurs d'arrondi* (arithmétique à point flottant, usage des MTBDDs) des *erreurs de troncature* (méthodes itératives) qui proviennent du fait qu'on a remplacé une méthode de calcul direct par des opérations mettant en jeu un nombre fini d'étapes.

Dans ce chapitre, nous voulons déterminer l'impact des erreurs d'arrondi sur le résultat retourné par PRISM. La première section analyse l'impact des erreurs d'arrondi sur le résultat du système d'équations linéaires $Ax = b$. La section suivante parle des méthodes éventuelles pour améliorer la précision et/ou la fiabilité des réponses fournies. Suite à ce travail, il pourrait être intéressant d'analyser l'effet combiné des erreurs d'arrondi et de troncature sur le résultat retourné par le model-checker.

5.1 Impact des erreurs d'arrondi sur le résultat

Nous débutons cette section en énonçant les différents notions mathématiques qui vont nous permettre d'analyser l'effet des erreurs d'arrondi sur le résultat retourné par l'outil PRISM.

5.1.1 Définitions et notations

Nous considérons seulement les matrices réelles et nous désignons par $\mathbb{R}^{m\times n}$, l'ensemble des matrices réelles $m \times n$. Les entrées d'une matrice A sont désignées par a_{ij} ou A_{ij} et $|A|$ est la matrice avec les entrées $|a_{ij}|$. L'inégalité $A \geq B$ pour les matrices signifie que $a_{ij} \geq b_{ij}$ pour tous i,j, et l'inégalité $v \geq w$ pour les vecteurs signifie que $v_i \geq w_i$ pour toutes les entrées des vecteurs. De la même façon, l'inégalité $v \geq 0$ pour le vecteur v signifie que $v_i \geq 0$ pour toutes ses entrées. Nous désignons par I_s, la matrice identité de taille $s \times s$ et par 0_s la matrice nulle de taille $s \times s$. Nous écrivons simplement par I et 0 quand leurs tailles sont évidentes à partir du contexte. Les définitions de cette section sont basées sur celles de Fortin [17].

Définition 5.1.1. *Une* norme vectorielle *est une application de $\mathbb{R}^n$ dans $\mathbb{R}$ qui associe à un vecteur $\overrightarrow{x}$ un scalaire noté $||\overrightarrow{x}||$ et qui vérifie les propriétés suivantes :*

1. *La norme d'un vecteur est toujours strictement positive, sauf si le vecteur a toutes ses composantes nulles :*
$$||\overrightarrow{x}|| > 0, \text{ sauf si } \overrightarrow{x} = 0$$
2. *Si α est un scalaire, alors :*
$$||\overrightarrow{\alpha x}|| = |\alpha|||\overrightarrow{x}||$$

3. *L'*inégalité triangulaire *est toujours vérifiée entre deux vecteurs* $\overrightarrow{x}$ *et* $\overrightarrow{y}$ *quelconques :*

$$||\overrightarrow{x} + \overrightarrow{y}|| \leq ||\overrightarrow{x}|| + ||\overrightarrow{y}||$$

Toute application qui vérifie ces trois propriétés est une norme vectorielle. La plus connue est la *norme euclidienne.*

Définition 5.1.2. *La norme euclidienne d'un vecteur* $\overrightarrow{x}$ *est notée* $||\overrightarrow{x}||_2$ *est définie par :*

$$||\overrightarrow{x}||_2 = \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2}$$

En plus de la norme euclidienne, nous définissons les normes l_1 et l_∞ qui vérifient aussi les trois propriétés nécessaires.

Définition 5.1.3. *La norme* l_1 *est définie par* $||\overrightarrow{x}||_1 = \sum_{i=1}^{n} |x_i|$ *tandis que la norme* l_∞ *est définie par* $\max_{1 \leq i \leq n} |x_i|$.

Exemple 5.1.4. *Pour le vecteur* $\overrightarrow{x} = \begin{pmatrix} 1 \\ -3 \\ 8 \end{pmatrix}$, *on a :*

$$\begin{array}{rcccc} ||\overrightarrow{x}||_1 & = & 1+3+8 & = & 12 \\ ||\overrightarrow{x}||_2 & = & \sqrt{1+9+64} & = & \sqrt{74} \\ ||\overrightarrow{x}||_\infty & = & \max(1,3,8) & = & 8 \end{array}$$

Puisque nous nous intéressons aux systèmes d'équations linéaires, nous introduisons la notion de norme relative aux matrices.

Définition 5.1.5. *Une* norme matricielle *est une application qui associe à une matrice* A *un scalaire noté* $||A||$ *vérifiant les quatre propriétés suivantes :*

1. *La norme d'une matrice est toujours strictement positive, sauf si la matrice a toutes ses composantes nulles :*

$$||A|| > 0, \text{ sauf si } A = 0$$

2. *Si* α *est un scalaire, alors :*

$$||\alpha A|| = |\alpha| ||A||$$

3. *L'*inégalité triangulaire *est toujours vérifiée entre deux matrices* A *et* B *quelconques :*

$$||A + B|| \leq ||A|| + ||B||$$

4. Une quatrième propriété est nécessaire pour les matrices :

$$||AB|| \leq ||A||\ ||B||$$

Comme pour les vecteurs, toute application qui vérifie ces quatre propriétés est une norme matricielle.

Définition 5.1.6. *Parmi les normes matricielles existantes, nous considérons les normes suivantes :*

$$\begin{aligned} ||A||_1 &= \max_{1\leq j\leq n} \sum_{i=1}^{n} |a_{ij}| \\ ||A||_2 &= \sqrt{\mu_{\max}} \\ ||A||_\infty &= \max_{1\leq i\leq n} \sum_{j=1}^{n} |a_{ij}| \\ |||A||_f &= \sqrt{\sum_{i=1}^{n}\sum_{j=1}^{n} |a_{ij}^2|} \end{aligned}$$

où $\mu_{\max}$ désigne le rayon spectral, c'est à dire la plus grande valeur propre de la matrice A.

Remarque 5.1.7. *La norme $||A||_1$ consiste à sommer en valeur absolue chacune des colonnes de A et à choisir la plus grande somme. La norme $||A||_\infty$ fait un travail similaire sur les lignes. La norme $||A||_2$ qui correspond au rayon spectral de A porte le nom de norme spectrale ou norme euclidienne car il s'agit d' une extension pour les matrices de la norme euclidienne pour les vecteurs. La norme $||A||_f$ ou norme de Frobenius correspond à la racine carrée de la somme des carrés de toutes les entrées de A.*

Exemple 5.1.8. *[17] Soit la matrice :*

$$A = \begin{pmatrix} 3 & -1 & -5 \\ -8 & 2 & 4 \\ 4 & 3 & -1 \end{pmatrix}$$

Pour les normes $||\cdot||_1$; $||\cdot||_\infty$ et $||\cdot||_f$, nous avons que :

$$\begin{aligned} ||A||_1 &= \max(15, 6, 10) &&= 15 \\ ||A||_\infty &= \max(9, 14, 8) &&= 14 \\ ||A||_f &= \sqrt{1+4+25+9+1+25+1+81} &&= \sqrt{147} \end{aligned}$$

Après avoir défini les notions de normes vectorielles et matricielles, nous pouvons définir la notion de conditionnement d'une matrice.

Définition 5.1.9. *Le conditionnement d'une matrice $\kappa(A)$ est défini par :*

$$\kappa(A) = ||A||\ ||A^{-1}||$$

Il s'agit simplement du produit de la norme de A et de la norme de son inverse.

Le conditionnement d'une matrice est toujours supérieur ou égal à 1 et il dépend de la norme matricielle utilisée. On utilise le plus souvent la norme $||A||_\infty$. Le conditionnement d'un système d'équation linéaire $Ax = b$ correspond au conditionnement $\kappa(A)$ de la matrice A. Plus le conditionnement est proche de 1, moins le système est sensible aux perturbations des données, donc plus stable. Par contre, plus il est grand, plus la solution du système linéaire est sensible aux perturbations des données. Pour les systèmes d'équations linéaires $Ax = b$ qui sont générés en solutionnant une formule PCTL, nous allons utiliser le conditionnement de la matrice A afin d'analyser l'effet des erreurs d'arrondi sur le résultat calculé.

Définition 5.1.10. *Une matrice $A = [a_{ij}] \in \mathbb{R}^{n\times n}$ est dite diagonale dominante si $|a_{ii}| \geq \sum_{i\neq j}^{n} |a_{ij}|$ pour $i = 1, \cdots, n$.*

La matrice A est dite strictement diagonale dominante si l'inégalité précédente est une inégalité stricte.

Définition 5.1.11. *Une matrice $A \in \mathbb{R}^{n\times n}$ admet une factorisation LDU s'il existe une matrice triangulaire inférieure L, une matrice triangulaire supérieure U et une matrice diagonale D qui sont telles que $A = LDU$.*

Définition 5.1.12. *[32] Une matrice $A = [a_{ij}] \in \mathbb{R}^{n\times n}$ est une Z-matrice si $a_{ij} \leq 0$ pour tout $i \neq j$. La matrice A est une M-matrice si A est une Z-matrice et $A^{-1} \geq 0$.*

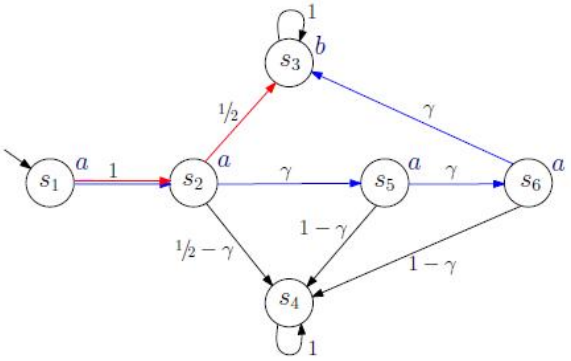
5.1.2 Analyse numérique

Nous partons de l'exemple qui suit pour expliquer l'effet des erreurs d'arrondi sur le résultat retourné par PRISM. Considérons le DTMC introduit dans les chapitres précédents, que nous reprenons à la figure 5.1 ; dans ce DTMC γ représente une constante de petite valeur inférieure à 1/2 et P, la matrice de transition du DTMC à l'étude. Nous voulons déterminer si un état du DTMC satisfait la formule PCTL :

$$\Phi = P_{\leq 1/2}(a\,\mathcal{U}\,b).$$

Un état s satisfait la formule Φ si la probabilité d'emprunter un chemin sur lequel a est toujours satisfait avant d'atteindre un état qui satisfait b est inférieure ou égale à 1/2, i.e., $Pr(s, a\,\mathcal{U}\,b) \leq 1/2$.

Remarque 5.1.13. *Nous avons choisi d'analyser, dans le cas du model-checking des DTMC, l'effet des erreurs d'arrondi sur les formules PCTL de la forme $P_{\sim p}(\Phi_1\,\mathcal{U}\,\Phi_2)$ où Φ_1 et Φ_2 sont des formules d'état sans l'opérateur probabiliste. En effet, nous voulons analyser l'effet de l'erreur numérique sans qu'aucune erreur de calcul ne soit introduite en cours de chemin.*



$$P = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1/2 & 1/2-\gamma & \gamma & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1-\gamma & 0 & \gamma \\ 0 & 0 & \gamma & 1-\gamma & 0 & 0 \end{pmatrix}$$

FIGURE 5.1: Un DTMC et sa matrice de transition.

Cette sous-catégorie de formule PCTL constitue la forme la plus générale des formules PCTL probabilistes et celle où les erreurs sont générées à la base sans effet de propagation sur le chemin. Ainsi par exemple nous analyserons la formule $P_{\leq 1/2}(a \mathcal{U} b)$ *mais pas la formule PCTL* $P_{\leq 1/2}((P_{\geq 3/4}(\mathcal{X} a)) \mathcal{U} b)$.

Soient M un DTMC, S l'ensemble de ses états et $P_{\sim p}(a \mathcal{U} b)$ une formule PCTL que nous voulons vérifier sur M. L'algorithme de model-checking décrit à la section 2.3 consiste à calculer pour chaque état s_i de S, la probabilité $Pr(s_i, a \mathcal{U} b)$ de satisfaire $a \mathcal{U} b$ et de déterminer par après si $Pr(s_i, a \mathcal{U} b) \sim p$ avec $\sim \in \{<, \leq, \geq, >\}$. Le calcul des probabilités $Pr(s_i, a \mathcal{U} b)$ est effectué en résolvant un système d'équations linéaires $Ax = b$ où chaque entrée $x(i)$ du vecteur solution correspond à la probabilité $Pr(s_i, a \mathcal{U} b)$ pour l'état s_i.

L'erreur d'arrondi fait qu'au lieu de résoudre $Ax = b$, nous résolvons dans les faits $(A+E)x' = b$ où la matrice E représente les perturbations dues aux erreurs d'arrondi. Nous voulons analyser l'écart entre la solution calculée x' et la solution exacte x. Pour cela, nous allons utiliser la notion de conditionnement d'une matrice, une métrique permettant de mesurer l'écart entre une solution calculée et la solution exacte.

Pour obtenir le système d'équations à résoudre, on considère une partition de l'ensemble S en trois ensembles disjoints S^0, S^1 et $S^?$. L'ensemble S^0 correspond aux états qui ne satisfont pas $a \mathcal{U} b$. L'ensemble S^1 correspond aux états dont la probabilité de satisfaire $a \mathcal{U} b$ est égale à 1. Pour une formule $P_{\sim p}(a \mathcal{U} b)$, l'ensemble S^1 correspond aux états qui satisfont d'emblée b. Les ensembles S^0 et S^1 sont calculés par deux algorithmes PROB0 et PROB1 présentés par Baier et al. [6] et Parker [37]. Les algorithmes PROB0 et PROB1 constituent la partie pré-calcul de l'algorithme de model-checking. Dans ces algorithmes, les ensembles S^0 et S^1 sont déterminés aisément par un parcours en profondeur sans qu'il y ait des erreurs générées. L'ensemble $S^?$ est déterminé comme suit, $S^? = S \backslash (S^0 \cup S^1)$.

Le système d'équations linéaires $Ax = b$ est alors construit comme suit :

1. de la matrice P de transition de M, on dérive une matrice P' qui est telle que :

$$P'(s,s') \quad = \quad \begin{cases} P(s,s') & si\ s \in S^? \\ 0 & autrement. \end{cases}$$

La matrice P' est obtenue en remplaçant par 0 les entrées des lignes correspondant aux états appartenant aux ensembles S^0 et S^1. Il est important de rappeler que pour les DTMC, la somme des entrées de P sur une ligne est égale à 1. Pour le DTMC de l'exemple de la figure 5.1 et la formule $P_{\sim p}(a\,\mathcal{U}\,b)$, les ensembles S^0, S^1 et S^1 correspondent à $S^0 = \{s_4\}$, $S^1 = \{s_3\}$ et $S^? = \{s_1, s_2, s_5, s_6\}$. On a ainsi que les matrices P et P' correspondent à :

$$P = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1/2 & 1/2-\gamma & \gamma & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1-\gamma & 0 & \gamma \\ 0 & 0 & \gamma & 1-\gamma & 0 & 0 \end{pmatrix} \quad \text{et} \quad P' = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1/2 & 1/2-\gamma & \gamma & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1-\gamma & 0 & \gamma \\ 0 & 0 & \gamma & 1-\gamma & 0 & 0 \end{pmatrix}$$

2. la matrice A du système $Ax = b$ à résoudre est donnée par $A = I - P'$ où I est la matrice identité. Pour l'exemple considéré de la figure 5.1, la matrice A correspond à :

$$A = \begin{pmatrix} 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 1 & -1/2 & \gamma - 1/2 & -\gamma & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & \gamma - 1 & 1 & -\gamma \\ 0 & 0 & -\gamma & \gamma - 1 & 0 & 1 \end{pmatrix}$$

3. le vecteur b est donné par :

$$b(s) \quad = \quad \begin{cases} 1 & si\ s \in S^1 \\ 0 & autrement. \end{cases}$$

Le vecteur b est un vecteur-colonne sur les états avec $b(s) = 1$ si $s \in S^1$ et 0 autrement. Il nous renseigne si l'état correspondant appartient à l'ensemble S^1 ou pas. Dans l'exemple susmentionné, le vecteur b correspond à :

$$b = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

Tel que mentionné à la section 2.3, chaque entrée $x(i)$ du vecteur solution correspond à la probabilité $Pr(s_i, a \mathcal{U} b)$ pour l'état s_i. Il est important de mentionner que le système linéaire généré n'altère pas les probabilités des états dans S^0 et S^1 puisque pour ceux-ci, $x(i) = 0$ si $s_i \in S^0$ et $x(i) = 1$ si $s_i \in S^1$ car les entrées de la ligne i dans la matrice A sont par construction tous à 0 sauf a_{ii} qui vaut 1. Cela oblige que $x(s) = 0$ pour $s \in S^0$ et $x(s) = 1$ pour $s \in S^1$, puisque dans ces cas, $b(s)$ vaut respectivement 0 et 1.

Soient M est un DTMC, S l'ensemble de ses états et une formule PCTL $P_{\sim p}(a \mathcal{U} b)$. Au vu de la manière dont est construit le système d'équations linéaires $Ax = b$ correspondant, nous pouvons dire que la matrice A et le vecteur b sont tels que :

(i) $\forall i \neq j \mid -1 \leq a_{ij} \leq 0$
(ii) $\forall i = 1 \cdots n \mid b(i) \in \{0, 1\}$
(iii) $\forall i = 1 \cdots n \mid 0 < a_{ii} \leq 1$
(iv) $s_i \in S^? \Rightarrow \left(\sum_{j=1, j \neq i}^{n} a_{ij} = -a_{ii}\right) \wedge \left(b(i) = 0\right)$
(v) $s_i \in S^0 \Rightarrow (a_{ii} = 1) \wedge \left(\sum_{j=1, j \neq i}^{n} a_{ij} = 0\right) \wedge (b(i) = 0)$
(vi) $s_i \in S^1 \Rightarrow (a_{ii} = 1) \wedge \left(\sum_{j=1, j \neq i}^{n} a_{ij} = 0\right) \wedge (b(i) = 1)$
(vii) $\exists i \mid s_i \in S^1$

Les énoncés (i) et (ii) découlent directement des définitions de A et de b. Ils indiquent que les éléments non-diagonaux sont compris entre -1 et 0 inclusivement tandis que les entrées du vecteur b sont soit égales à 1, soit nulles (un état appartient à S^1 ou pas).

Concernant l'énoncé (iii), on note surtout que $a_{ii} > 0$. Cela vient du fait que pour avoir $a_{ii} = 0$, il faudrait que l'état s_i correspondant possède une transition boucle de valeur égale à 1, i.e., $P(s_i, s_i) = 1$ et que $s_i \in S^?$. Or un état pareil ne peut pas appartenir à $S^?$.

L'énoncé (iv) décrit les lignes de la matrice A et du vecteur b correspondant aux états dans l'ensemble $S^?$. Pour un état s_i dans cet ensemble, il est évident que $b(i) = 0$ puisque $s_i \notin S^1$. Pour les entrées a_{ij} de la matrice A, on a par construction que :

$$a_{ii} = 1 - P(s_i, s_i) \tag{5.1}$$

$$\sum_{j=1,j\neq i}^{n} a_{ij} = -\sum_{j=1,j\neq i}^{n} P(s_i, s_j) \tag{5.2}$$

De plus, pour un DTMC on a que la somme des entrées de la matrice de transition est égale à 1 :

$$P(s_i, s_i) + \sum_{j=1, j\neq i}^{n} P(s_i, s_j) = 1 \tag{5.3}$$

En remplaçant dans 5.2, $\sum_{j=1,j\neq i}^{n} P(s_i, s_j)$ par sa valeur de l'équation 5.3, l'équation 5.2 peut être réécrite comme suit :

$$\sum_{j=1, j\neq i}^{n} a_{ij} = -(1 - P(s_i, s_i)) = -a_{ii} \tag{5.4}$$

En particulier si l'état correspondant $s_i \in S^?$ ne comporte pas une transition boucle dans le système de transition, alors $a_{ii} = 1$ et $\sum_{j=1,j\neq i}^{n} a_{ij} = -1$.

Les énoncés (v) et (vi) décrivent les lignes de la matrice A correspondant aux états dans les ensembles S^0 et S^1 respectivement. Ils décrivent le fait que les entrées de la ligne i dans la matrice A sont par construction tous à 0 sauf a_{ii} qui vaut 1. L'énoncé (vii) traduit le fait que pour un cas non trivial, l'ensemble S^1 soit non vide, i.e., il existe au moins un état vérifiant b sinon tous les états ne pourraient vérifier $a \,\mathcal{U}\, b$.

Il apparaît clairement que la matrice A est une diagonale dominante mais pas strictement. Plus intéressant, la matrice A est une M-matrice comme le stipule le théorème 5.1.14 [2, 38].

Théorème 5.1.14. *Si A est une matrice diagonale dominante avec $a_{ii} > 0$ et $a_{ij} \leq 0$ pour tout $i \neq j$, alors A est une M-matrice.*

Pour le cas du model-checking, le théorème 5.1.14 nous permet de dire que :

Théorème 5.1.15. *Soient un DTMC M et une formule PCTL $P_{\sim p}(a \,\mathcal{U}\, b)$. La matrice A du système d'équations linéaires $Ax = b$ généré pour vérifier $P_{\sim p}(a \,\mathcal{U}\, b)$ sur M est une M-matrice.*

Puisque A est une M-matrice diagonale dominante, elle satisfait plusieurs propriétés : elle est non-singulière, son inverse A^{-1} existe et elle admet une décomposition LDU [38, 42]. En ayant une caractérisation de la matrice A, nous pouvons utiliser le conditionnement de A pour analyser la solution du système d'équations $Ax = b$. D'emblée, nous rappelons le théorème qui suit tiré de [17].

Théorème 5.1.16. *Soient $Ax = b$ un système d'équations linéaires et x', la solution calculée en utilisant l'arithmétique flottante, on a que :*

$$\frac{\|x - x'\|_\infty}{\|x'\|_\infty} \leq \epsilon_m \|A\|_\infty \|A^{-1}\|_\infty.$$

où ϵ_m représente la précision machine. Pour le cas du model-checking, nous avons aussi les deux théorèmes qui suivent :

Théorème 5.1.17. *Soient un DTMC M et une formule PCTL $P_{\sim p}(a \,\mathcal{U}\, b)$. La matrice A du système d'équations linéaires $Ax = b$ généré pour vérifier $P_{\sim p}(a \,\mathcal{U}\, b)$ sur M est telle que :*

$$\|A\|_\infty \leq 2.$$

Démonstration : Si on fait la somme des valeurs absolues des éléments sur chaque ligne de la matrice A, la valeur maximale que peut prendre une telle somme vient d'une ligne qui est telle que $a_{ii} = 1$ et $\sum_{j=1, j\neq i}^{n} a_{ij} = -1$. Comme on l'a déjà remarqué, une ligne pareille correspond à un état $s_i \in S^?$ sans une transition boucle. Pour une telle ligne, on a que $\sum_{j=1}^{n} |a_{i,j}| = 2$.

□

Théorème 5.1.18. *Soient un DTMC M, une formule PCTL $P_{\sim p}(a \,\mathcal{U}\, b)$, $Ax = b$ le système d'équations linéaires généré pour vérifier $P_{\sim p}(a \,\mathcal{U}\, b)$ sur M et x', la solution calculée en utilisant l'arithmétique flottante, on a que :*

$$\|x\|_\infty = \|x'\|_\infty = 1.$$

Démonstration : La démonstration est assez identique de celle du théorème 5.1.17. La ligne qui produit la valeur maximale de $x(i)$ correspond à un état qui remplit la condition de la propriété vii, état qui existe toujours selon la même propriété. Pour une telle ligne, la valeur de $x(i)$ ne varie pas au cours des itérations et elle n'est pas affectée par les erreurs d'arrondi d'où $x'(i) = 1$ aussi.

□

En utilisant les valeurs de $\|x'\|_\infty$ et $\|A\|_\infty$ tirées des théorèmes 5.1.17 et 5.1.18, le théorème 5.1.16 peut s'appliquer comme suit pour le cas du model-checking :

Théorème 5.1.19. *Soient un DTMC M, une formule PCTL $P_{\sim p}(a \,\mathcal{U}\, b)$, $Ax = b$ le système d'équations linéaires généré pour vérifier $P_{\sim p}(a \,\mathcal{U}\, b)$ sur M et x', la solution calculée en utilisant l'arithmétique flottante, on a que :*

$$\|x - x'\|_\infty \leq 2 \cdot \epsilon_m \|A^{-1}\|_\infty.$$

Le théorème 5.1.19 nous fournit une borne supérieure sur l'écart entre la probabilité recherchée et celle calculée par le model-checker. Nous insistons sur le fait que cette dernière n'est pas la probabilité retournée par le model-checker vu qu'elle dépend aussi de la méthode de résolution appliquée par le model-checker. Nous distinguons la solution recherchée x, celle calculée après l'introduction des erreurs d'arrondi x' et x_k qui est retournée par le model-checker après k itérations si la méthode de résolution est itérative. Par contre, si elle est directe, alors dans ce cas la solution retournée par le model-checker est x'. Cela nous permet dans certains cas, comme nous le démontrons dans la section suivante, de fournir un intervalle de confiance à une réponse fournie par le model-checker.

La borne du théorème 5.1.19 présente l'inconvénient qu'elle implique le calcul de l'inverse de A qui est plus coûteux ($\simeq \frac{4n^3}{3}$) que la résolution du système linéaire ($\simeq \frac{n^3}{3}$)[17]. Pour des systèmes de grande taille, cette borne est incalculable. Il aurait été intéressant d'avoir une borne supérieure de $\|A^{-1}\|_\infty$ en fonction des entrées ou de la taille de A pour une M-matrice diagonale dominante mais nous n'avons pas pu établir une telle borne. Certes, il y a la borne de [48] qui a été plusieurs fois raffinée [11, 34] mais elle s'applique aux matrices strictement diagonale dominantes. Il existe d'autres bornes mais elles s'appliquent pour certaines catégories de M-matrices [23, 27, 33].

Nous présentons dans la section suivante quelques techniques qui ont été déjà proposées pour améliorer, garantir ou certifier le résultat retourné par un model-checker.

5.2 Améliorations éventuelles

Plusieurs travaux [9, 12, 15, 49] se sont intéressés à la problématique de la fiabilité des résultats avec la technique du model-checking et différentes techniques [12, 49] ont été proposées pour faire face aux conséquences liées aux erreurs déjà mentionnées. Ces techniques ont pour objectif soit de corriger une source d'erreur précise, soit de prévenir l'utilisateur d'un possible résultat erroné ou de lui fournir une garantie d'exactitude. Une technique est considérée pertinente si elle peut être mise en pratique pour des grands systèmes, c'est-à-dire si elle ne coûte pas autant (en temps et en mémoire) que ce qu'il en coûte pour résoudre le problème à l'étude et si elle n'altère pas les solutions déjà appliquées. Par exemple, Wimmer et al. [49] ont proposé le recours à une méthode directe pour la résolution du système linéaire comme l'élimination gaussienne pour corriger les effets dues aux méthodes itératives alors qu'il est prouvé que ces méthodes directes ne sont pas adaptés pour de larges systèmes creux surtout quand on utilise le model-checking symbolique. En effet les méthodes directes détruisent la structure creuse des matrices et par voie de conséquence la représentation compacte ne peut plus être exploitée, la taille de la matrice ainsi que celle du MTBDD correspondant augmentent dès lors. Nous décrivons dans cette section, les techniques que nous avons estimées les plus intégrables dans

le model-checker PRISM sans nuire à son fonctionnement actuel. En effet, il est important que les améliorations suggérées n'entrainent pas une régression dans la liste des fonctionnalités déjà offertes par le model-checker.

5.2.1 Degré de confiance

Dans l'exemple donné à la section 3.1 que nous avons repris à la figure 5.1.2, si l'utilisateur savait que $s_1 \models P_{\leq 0.5}(a \mathcal{U} b)$ puisque $Pr(s_1, a \mathcal{U} b) = 0.5$, il est évident que la réponse devrait être considérée avec plus d'attention puisqu'une petite perturbation aurait pu changer la réponse retournée. Pour de larges systèmes (plus de 10^5 états), on ne peut pas fournir à l'utilisateur les probabilités de chaque état. A la place, on peut proposer une mesure qui nous indique le "*degré de confiance*" d'une formule donnée pour tout le système à l'étude. Cette mesure proposée par Wimmer et al. [49] et noté *db* (de l'anglais *degree of belief*) est donné par :

$$db_{P_{\sim p}(\psi)} = \min\{|Pr(s, \psi) - p| \big| s \in S\}$$

Pour chaque sous-formule $\phi = P_{\sim p}(\psi)$ de Φ, on calcule pour chaque état s la différence $|Pr(s, \psi) - p|$. La plus petite différence correspond au degré de confiance de Φ. Plus le degré de confiance est proche de zéro, plus l'utilisateur est informé d'un résultat pouvant comporter des erreurs comme pour l'exemple de la figure 5.1.2 où $db_{P_{\leq 0.5}(a \mathcal{U} b)} = 0$. Le degré de confiance d'une formule Φ renseigne l'utilisateur si une erreur peut être commise lors de la vérification d'une sous formule de Φ. La mesure du degré de confiance présente l'avantage d'être facile à intégrer dans les outils de model-checking déjà existant. Par contre, il faut mentionner qu'elle ne permet pas de remédier à une quelconque source d'erreurs.

5.2.2 Intervalle de confiance

Pour garantir le résultat retourné par un model-checker, Wimmer et al. [49] ont aussi proposé l'idée de calculer des intervalles sûrs qui contiennent la probabilité avec laquelle une formule de chemin est satisfaite dans un état, c'est à dire :

$$I_\psi(s) = [a, b] \text{ tel que } a \leq Pr(s, \psi) \leq b$$

Une formule $P_{\sim p}(\psi)$ est satisfaite à coup sûr dans un état s si $\forall x \in I_\psi(s) : x \sim p$ avec $\sim \in \{<, \leq, >, \geq\}$. A l'opposé, on est sûr que $P_{\sim p}(\psi)$ n'est pas satisfaite dans s si $P_{\not\sim p}(\psi)$. Dans les autres cas, on ne peut faire aucune affirmation quant à la validité de la formule.

Exemple 5.2.1. *Soit la formule PCTL* $\Phi = P_{\leq 0.5}(a \mathcal{U} b)$*, 3 états* s_1*,* s_2 *et* s_3 *tels que* $I_{(a \mathcal{U} b)}(s_1) = [0.3, 0.5]$*,* $I_{(a \mathcal{U} b)}(s_2) = [0.6, 0.8]$ *et* $I_{(a \mathcal{U} b)}(s_3) = [0.4, 0.7]$*. Dans ce cas, nous pouvons affirmer que* $s_1 \models \Phi$ *et* $s_2 \not\models \Phi$*. Pour le cas de* s_3*, nous ne pouvons faire aucune affirmation.*

Le principal défi reste le calcul de tels intervalles sûrs. Fecher et al. [15], Wimmer et al. [49] ont proposé une technique à cet effet mais de tels intervalles peuvent être assez grands de telle manière qu'ils risquent de ne pas être utiles à l'utilisateur.

En partant de cette idée et du résultat du théorème 5.1.19, nous pouvons aussi fournir une garantie de certitude à une formule PCTL de la forme $P_{\sim p}(a \mathcal{U} b)$. Soient $Ax = b$, le système linéaire correspondant et $\delta = 2\epsilon_m ||A^{-1}||_\infty$, on dira à coup sûr qu'un état $s \models P_{\sim p}(a \mathcal{U} b)$ si et seulement si :

- $s \models P_{\sim_{p-\delta}}(a \mathcal{U} b)$ si $\sim \in \{<, \leq\}$
- $s \models P_{\sim_{p+\delta}}(a \mathcal{U} b)$ si $\sim \in \{>, \geq\}$

En conclusion, nous avons analysé dans ce chapitre l'impact des erreurs d'arrondi sur le résultat retourné par PRISM. Pour la forme la plus générale d'une formule PCTL avec l'opérateur probabiliste, nous avons fourni une borne supérieure de l'erreur commise même si cette borne implique l'inverse d'une matrice. Nous avons aussi présenté certaines techniques d'améliorations du résultat retourné qui peuvent être facilement intégrées dans l'outil PRISM.

Conclusion

Les systèmes issus des technologies de l'information et de la communication font désormais partie intégrante de notre vie quotidienne. Internet n'est plus l'apanage de quelques spécialistes et des universitaires, il s'est répandu dans toutes les couches de la société. Des systèmes intégrés comme les cartes de crédit, les téléphones portables et les téléviseurs intelligents sont autant de preuves de cette présence dans notre réalité quotidienne.

Des techniques et des outils ont donc été conçus pour permettre aux concepteurs de faire la vérification de façon automatique. Dans ce mémoire, nous nous sommes intéressés à une technique de vérification formelle basée sur le modèle appelée *évaluation de modèle* (*model-checking* en anglais). L'évaluation de modèle probabiliste est une technique de vérification qui consiste à déterminer si un modèle probabiliste M vérifie une propriété donnée. Les modèles sont décrits par des systèmes de transitions appelés *modèles* tandis que la logique temporelle est utilisée comme langage de spécification des propriétés. L'évaluation de modèle exige aussi un algorithme d'évaluation de modèle qui se charge de déterminer si le système à l'étude vérifie ou non la propriété énoncée. Les algorithmes sont implémentés sous forme d'outil informatique pour l'évaluation de modèle appelés *vérificateurs* (*model-checkers* en anglais).

Il existe différentes façons de modéliser un système et différents types de modèles existent. Dans le cas du model-checking probabiliste, nous utilisons des modèles qui intègrent l'information sur la probabilité qu'une transition entre états se produise. Plusieurs outils ont été développés pour permettre de faire le model-checking probabiliste automatique. PRISM, qui est l'outil qui nous intéresse dans le cadre de ce travail, est un model-checker probabiliste qui supporte le model-checking pour différents types de modèles probabilistes comme les chaînes de Markov à temps discret ou DTMC (de l'anglais *Discrete-Time Markov Chain*), les processus de décision de Markov ou MDP (de l'anglais *Markov Decision Process*), les chaînes de Markov à temps continu ou CTMC (de l'anglais *Continuous-Time Markov Chains*), etc. En entrée, l'outil prend deux paramètres, une description du modèle dans le langage PRISM et une liste de spécifications du modèle. Il détermine ensuite quels états du modèle satisfont chaque spécification donnée. Dans le cadre de ce travail, nous avons choisi d'analyser le model-checking pour les DTMC. Nous avons souligné en particulier le fait que l'algorithme de model-checking

pour DTMC consiste essentiellement à la résolution d'un système d'équations linéaires $Ax = b$.

Dans ce mémoire, nous avons montré à quelles étapes du processus de model-checking les erreurs apparaissent. Nous avons répertorié dans la littérature quatre sources d'erreur à savoir : l'*arithmétique à virgule flottante* utilisée par PRISM pour représenter les probabilités, les *méthodes itératives* utilisées pour résoudre les systèmes d'équations linéaires $Ax = b$, la *librairie CUDD* utilisée pour le model-checking symbolique et la *méthode d'uniformisation* (ou méthode de Jensen) utilisée lors du model-checking pour CTMC. Comme nous avions choisi de décrire le model-checking pour les DTMC, nous avons documenté, avec exemples à l'appui, les trois premières sources d'erreurs répertoriées. Suite à ce travail, il pourrait être intéressant d'analyser la dernière source d'erreurs qui concerne le model-checking pour CTMC. L'analyse effectuée pour les DTMC pourrait s'étendre sur une partie ou la totalité des CTMC. De manière générale, nous avons regroupé ces sources d'erreurs en deux groupes à savoir les *erreurs d'arrondi* (arithmétique à point flottant, model-checking symbolique) des *erreurs de troncature* (méthode itérative, méthode d'uniformisation) qui proviennent du fait qu'on a remplacé une méthode de calcul direct par des opérations mettant en jeu un nombre fini d'étapes.

Nous nous sommes particulièrement intéressés aux erreurs qui apparaissent lors du model-checking symbolique. Nous avons montré qu'en ayant recours à la technique du model-checking symbolique pour vérifier un système probabiliste S, PRISM fait que le système encodé correspond en réalité à un système "*approximatif*" S' du modèle à l'étude. Comme la notion de bisimulation classique n'était pas adaptée pour comparer les deux systèmes, nous avons utilisé la notion d'équivalence approchée, une variante de la bisimulation classique, pour montrer que les deux systèmes sont tels que $S \sim_{\delta\epsilon} S'$ où δ représente le nombre maximal de transitions sortant vers un état s du système à l'étude (avec $\delta < |S|$), et ϵ, la précision utilisée par PRISM lors du model-checking symbolique.

Enfin, nous avons analysé l'impact des erreurs d'arrondi sur le résultat retourné par PRISM. Nous avons montré que pour le cas du model-checking pour DTMC, l'erreur d'arrondi fait qu'au lieu de résoudre le système d'équations linéaires $Ax = b$, nous résolvons dans les faits $(A + E)x' = b$ où la matrice E représente les perturbations dues aux erreurs d'arrondi. Nous avons analysé l'écart entre la solution calculée x' et la solution exacte x pour une formule PCTL de la forme $P_{\sim p}(a \mathcal{U} b)$. Nous avons montré que pour une formule de ce type, $\|x - x'\|_\infty \leq 2 \cdot \epsilon_m \|A^{-1}\|_\infty$ et partant, nous avons fourni une borne supérieure de l'erreur commise pour cette catégorie de formule suite aux erreurs d'arrondi. Nous avons aussi suggéré quelques techniques pour faire face aux conséquences liées aux erreurs déjà mentionnées. Ces techniques ont été proposées car nous considérons qu'elles peuvent être intégrées aux outils actuels sans pour autant altérer les solutions déjà appliquées.

Comme travaux futurs au présent travail, il serait intéressant d'analyser aussi l'impact des

erreurs de troncature sur le résultat retourné par PRISM. Nous avons déjà vu que les erreurs d'arrondi font que nous résolvons dans les faits $(A + E)x' = b$, les erreurs de troncature font que le résultat retourné x_k après k itérations est une approximation de la solution calculée x'. Une analyse de l'écart entre x_k et x' serait un grand pas en avant pour borner l'erreur sur le résultat retourné par le model-checker.

Nous estimons aussi que la borne fournie de l'écart entre x et x' pour une formule PCTL de la forme $P_{\sim p}(a \,\mathcal{U}\, b)$ peut être améliorée puisqu'elle nécessite le calcul de l'inverse d'une matrice. Au vu des expériences réalisées dans le cadre de ce travail et en tenant compte de la structure de la matrice A générée pour le model-checking d'une formule de ce type, nous croyons qu'une borne supérieure de $|A^{-1}\|_\infty$ en fonction des entrées de A, par exemple en fonction de la taille n de la matrice A, peut être obtenue.

Bibliographie

[1] Abarbanel-vinov, Y., N. Aizenbud-reshef, I. Beer, C. Eisner, D. Geist, T. Heyman, I. Reuveni, E. Rippel, I. Shitsevalov, Y. Wolfsthal et T. Yatzkar-haham, « On the effective deployment of functional formal verification », *In Proc. 7th International SPIN Workshop, LNCS 1885*, p. 72–83, Springer-Verlag, 2001.

[2] Alfa, A. S., J. Xue et Q. Ye, « Entrywise perturbation theory for diagonally dominant M-matrices with applications », *Numerische Mathematik*, 90(3), p. 401–414, 2002.

[3] Alur, R. et T. A. Henzinger, « Reactive modules », *Proceedings of the 11th Annual Symposium on Logic in Computer Science (LICS)*, p. 207–218, IEEE Computer Society, 1996.

[4] Baier, C., J.-P. Katoen et H. Hermanns, « Approximate symbolic model-checking of continuous-time Markov chains », *Proc. 10th International Conference on Concurrency Theory*, vol. 1664, p. 146–161, 1999.

[5] Baier, C., B. Haverkort, H. Hermanns et J.-P. Katoen, *Model checking continuous-time markov chains by transient analysis*, *Computer Aided Verification*, vol. 1855, Lecture Notes in Computer Science, p. 358–372, Springer Berlin Heidelberg, 2000.

[6] Baier, C. et J.-P. Katoen, *Principles of Model Checking*, MIT-Press, 2008.

[7] Baier, C. et M. Kwiatkowska, « Model checking for a probabilistic branching time logic with fairness », *Distributed Computing*, 11, p. 125–155, 1998.

[8] Baier, C., *On algorithmic verification methods for probabilistic systems*, Habilitation thesis, Fakultät für Mathematik & Informatik, Universität Mannheim, 1998.

[9] Bell, A. et B. R. Haverkort, *Untold horrors about steady-state probabilities : What reward-based measures won't tell about the equilibrium distribution*, *Formal Methods and Stochastic Models for Performance Evaluation*, vol. 4748, Lecture Notes in Computer Science, p. 2–17, Springer Berlin Heidelberg, 2007.

[10] Bryant, R. E., « Graph-based algorithms for boolean function manipulation », *IEEE Transactions on Computers*, C-35(8), p. 677–691, 1986.

[11] Cheng, G.-H. et T.-Z. Huang, « An upper bound for of strictly diagonally dominant M-matrices », *Linear Algebra and its Applications*, 426(2 - 3), p. 667–673, 2007.

[12] Daws, C., *Symbolic and parametric model-checking of discrete-time Markov chains*, *Theoretical Aspects of Computing - ICTAC 2004*, vol. 3407, Lecture Notes in Computer Science, p. 280–294, Springer Berlin Heidelberg, 2005.

[13] Desharnais, J., V. Gupta, R. Jagadeesan et P. Panangaden, « Metrics for labelled Markov processes », *Theoretical Computer Science*, 318(3), p. 323 – 354, 2004.

[14] Desharnais, J., F. Laviolette et M. Tracol, « Approximate analysis of probabilistic processes : Logic, simulation and games », *Quantitative Evaluation of Systems, International Conference on*, 0, p. 264–273, 2008.

[15] Fecher, H., M. Leucker et V. Wolf, *Don't know in probabilistic systems*, *Proceedings of the 13th International Conference on Model Checking Software*, vol. 3925, Lecture Notes in Computer Science, p. 71–88, Springer Berlin Heidelberg, 2006.

[16] Ferns, N., P. Panangaden et D. Precup, « Metrics for Markov decision processes with infinite state spaces. », *UAI*, p. 201–208, AUAI Press, 2005.

[17] Fortin, A., *Analyse numérique pour ingénieurs*, Presses internationales Polytechnique, Quatrième edition, 2011.

[18] Giacalone, A., C. chang Jou et S. A. Smolka, « Algebraic reasoning for probabilistic concurrent systems », *Proc. IFIP TC2 Working Conference on Programming Concepts and Methods*, p. 443–458, North-Holland, 1990.

[19] Hansson, H. et B. Jonsson, « A framework for reasoning about time and reliability », *Proceedings of 10th IEEE Real Time System Symposium*, p. 102–111, 1989.

[20] Holzmann, G. J., « The theory and practice of a formal method : Newcore », *Proc. IFIP World Computer Congress*, vol. I, p. 35–44, Hamburg, Germany, North-Holland Publ., Amsterdam, The Netherlands, 1994, (invited paper).

[21] Holzmann, G. J., « The Model Checker SPIN », *IEEE Transactions on Software Engineering*, 23(5), p. 279–295, 1997.

[22] Honda recalls vehicles on software issue, http://www.reuters.com/article/2011/08/05/us-honda-recall-idUSTRE77432120110805, Consulté le 2 mai 2014.

[23] Huang, T.-Z., « Estimation of and the smallest singular value », *Computers and Mathematics with Applications*, 55(6), p. 1075–1080, 2008.

[24] « IEEE Recommended Practice on Software Reliability », *IEEE STD 1633-2008*, p. 3–5, Juin 2008.

[25] IEEE standard 754, http://grouper.ieee.org/groups/754/, Consulté le 20 avril 2013.

[26] Katoen, J.-P., M. Khattri et I. S. Zapreev, « A Markov reward model checker », *2nd Int. Conf. on Quantitative Evaluation of Systems (QEST)*, p. 243–244, IEEE Computer Society, 2005.

[27] Kolotilina, L. Y., « Bounds for the infinity norm of the inverse for certain M- and H-matrices », *Linear Algebra and its Applications*, 430(2-3), p. 692–702, 2009.

[28] Kwiatkowska, M., G. Norman et R. Segala, « Automated verification of a randomized distributed consensus protocol using Cadence SMV and PRISM », *Proc. 13th International Conference on Computer Aided Verification (CAV01)*, vol. 2102, Lncs, p. 194–206, Springer, 2001.

[29] Kwiatkowska, M., G. Norman et D. Parker, *PRISM : Probabilistic Symbolic Model Checker*, *Computer Performance Evaluation : Modelling Techniques and Tools*, vol. 2324, Lecture Notes in Computer Science, p. 200–204, Springer Berlin Heidelberg, 2002.

[30] Kwiatkowska, M., G. Norman et J. Sproston, « Probabilistic model checking of deadline properties in the IEEE 1394 FireWire root contention protocol », *Formal Aspects of Computing*, 14(3), p. 295–318, 2003.

[31] Kwiatkowska, M., G. Norman et D. Parker, « Probabilistic symbolic model checking with PRISM : A hybrid approach », *International Journal on Software Tools for Technology Transfer (STTT)*, 6(2), p. 128–142, 2004.

[32] Li, W. et Y. Chen, « Some new two-sided bounds for determinants of diagonally dominant matrices », *Journal of Inequalities and Applications*, 2012(1), 2012.

[33] Li, W., « The infinity norm bound for the inverse of nonsingular diagonal dominant matrices », *Applied Mathematics Letters*, 21(3), p. 258–263, 2008.

[34] Moraca, N., « Bounds for norms of the matrix inverse and the smallest singular value », *Linear Algebra and its Applications*, 429(10), p. 2589–2601, 2008, Special Issue in honor of Richard S. Varga.

[35] Nafa, T., *La vérification formelle de systèmes réactifs probabilistes finis*, mémoire de maîtrise, Université Laval, Québec, Canada, 2005.

[36] Paquette, S., *Evaluation symbolique de systèmes probabilistes à espace d'états continu*, mémoire de maîtrise, Université Laval, Québec, Canada, 2005.

[37] Parker, D., *Implementation of symbolic model checking for probabilistic systems*, thèse de doctorat, University of Birmingham, 2002.

[38] Plemmons, R. J., « M-matrix characterizations. I-nonsingular M-matrices », *Linear Algebra and its Applications*, 18(2), p. 175–188, 1977.

[39] Pnueli, A., « The temporal logic of programs », *Proceedings 18th IEEE Symposium on Foundations of Computer Science*, p. 46–57, 1977.

[40] Richard, N., *La vérification formelle de systèmes probabilistes continus*, mémoire de maîtrise, Université Laval, Québec, Canada, 2003.

[41] Schlipf, T., T. Buechner, R. Fritz, M. Helms et J. Koehl, « Formal verification made easy. », *IBM Journal of Research and Development*, 41(4-5), p. 567–576, 1997.

[42] Shao, J.-L., T.-Z. Huang et G.-F. Zhang, « Linear system based approach for solving some related problems of M-matrices », *Linear Algebra and its Applications*, 432(1), p. 327–337, 2010.

[43] Somenzi, F., *CUDD : CU decision diagram package, release 2.4.0*, University of Colorado at Boulder, 2005.

[44] Stewart, W. J., *Introduction to the Numerical Solution of Markov Chains*, Princeton Univ. Press, 1994.

[45] Tani, S., K. Hamaguchi et S. Yajima, *The complexity of the optimal variable ordering problems of shared binary decision diagrams*, *Algorithms and Computation*, vol. 762, Lecture Notes in Computer Science, p. 389–398, Springer Berlin Heidelberg, 1993.

[46] Tracol, M., J. Desharnais et A. Zhioua, « Computing distances between probabilistic automata », *QAPL*, vol. 57, EPTCS, p. 148–162, 2011.

[47] Tracol, M., *Vérification approchée de systèmes probabilistes*, thèse de doctorat, Universite Paris-Sud, 2010.

[48] Varah, J. M., « A lower bound for the smallest singular value of a matrix », *Linear Algebra and its Applications*, 11(1), p. 3–5, 1975.

[49] Wimmer, R., A. Kortus, M. Herbstritt et B. Becker, « Probabilistic model checking and reliability of results. », *DDECS*, p. 207–212, IEEE Computer Society, 2008.