NICOLAS RICHARD

La vérification formelle de systèmes probabilistes continus

Mémoire présenté
à la Faculté des études supérieures de l'Université Laval
pour l'obtention du grade de maître ès sciences

Département d'informatique et de génie logiciel
Faculté des sciences et de génie
UNIVERSITÉ LAVAL

Décembre 2003

Résumé

Dans ce mémoire, nous présentons l'évaluation de modèle, appelée en anglais « *model-checking* », pour les processus de Markov étiquetés. Ces derniers sont des systèmes de transitions probabilistes dont l'ensemble des états est non-dénombrable. Ils servent à modéliser des systèmes qui ont un comportement aléatoire ou à approximer des systèmes complexes. Les contributions que nous apportons à ce domaine sont la définition d'un langage formel pour décrire une famille de systèmes probabilistes continus, la formalisation d'algorithmes pour la vérification formelle ainsi qu'une implémentation en Java d'un outil pouvant vérifier si un système vérifie une formule d'une logique probabiliste. Nous présentons aussi dans le document une étude de cas ainsi que des résultats obtenus avec l'implémentation.

Remerciements

Je tiens à remercier fortement ma directrice de maîtrise, Mme Josée Desharnais, qui m'a initié au *model-checking*, domaine très intéressant qui regroupe beaucoup de concepts et de théories avec lesquels j'ai toujours aimés travailler, en mathématiques, en informatique théorique et en génie logiciel. J'aurais été bien triste de faire ma maîtrise sur un autre domaine et avec quelqu'un d'autre. Sa patience (surtour à me lire et à me relire) et sa maîtrise du domaine lui vondront ma reconnaissance éternelle. Merci Josée.

Je tiens aussi à remercier un collègue, Simon Paquette, pour avoir lu et relu mon mémoire, y avoir trouver *quelques* fautes de français et m'avoir donner des idées pour éclaircir certaines explications. Merci.

Finalement, j'aimerais remercier ma mère, Lise Denis, pour m'avoir donner l'encouragement dont j'avais besoin pour tenter des études graduées. Merci.

À ma mère, qui m'a toujours encouragé et qui a toujours cru en moi.

« *Program testing can at best show the presence of errors but never their absence.* »
Edsger W. Dijkstra

Table des matières

Chapitre 1

Introduction

Les systèmes informatiques se retrouvent pratiquement partout de nos jours. Du réfrigérateur offrant la navigation sur Internet, aux systèmes de pilotage automatique des avions et des fusées, en passant par les systèmes de contrôle de feux de signalisation ainsi que les contrôleurs d'ascenseur optimisant le temps d'attente, les systèmes informatiques prennent de plus en plus de place. On les utilise davantage pour des tâches sensibles et critiques, comme dans les systèmes d'aide à la prise de décision pour les médecins ou bien le contrôleur de vol d'une fusée. Malheureusement, les médias sont remplis d'histoires traitant d'erreurs coûteuses de conception de logiciel[1]. Ces erreurs sont parfois causées par une mauvaise conception ou par une mauvaise réutilisation d'un module fonctionnant correctement. Ces erreurs augmentent considérablement le coût de maintien des logiciels et peuvent avoir des conséquences désastreuses pour les utilisateurs.

Pour détecter et corriger ces erreurs, le monde informatique s'est longtemps appuyé sur la mécanique des tests unitaires et des tests d'intégration. Mais, comme le disait le père de la programmation structurée, Dijsktra, « *les tests ne peuvent démontrer que la présence d'erreur, jamais leur absence* »[2]. Les tests ont en effet une portée limitée et arrivent souvent tard dans le procesus de développement de logiciel. De plus, les tests ne peuvent pas détecter des erreurs d'incohérence des objectifs que doit remplir un système. Lorsqu'une erreur est détectée par un test, on ne connaît pas toujours l'origine de l'erreur et ce qui serait à faire pour la corriger.

[1] Le site Internet http ://www.cs.tau.ac.il/ nachumd/verify/horror.html contient une liste de 106 *histoires d'horreur* en conception de logiciel, comme l'écrasement de la sonde *Mars Climate Orbiter* sur Mars et la destruction de la fusée *Ariane 5*.

[2] Traduction libre de la citation « *Program testing can at best show the presence of errors but never their absence* ».

Il existe des techniques plus robustes pour nous assurer qu'un programme ne comporte pas d'erreurs. Certaines techniques, que l'on appelle méthodes formelles, se basent sur des fondements mathématiques pour démontrer qu'un système informatique respecte un ensemble de comportements désirés. Par exemple, pour une distributrice à café, un comportement voulu pourrait être : « *il est impossible d'avoir du café sans avoir payé* ». Un des avantages des méthodes formelles est qu'il n'y aura pas d'erreur, pour *toutes* les exécutions possibles d'un programme, d'après sa spécification. Certaines techniques, comme l'évaluatoin de modèle, permettent même de documenter les violations de la spécification en fournissant automatiquement une trace d'exécution menant à une erreur. Il existe plusieurs types de méthodes formelles. Entre autres, il y a la certification de code ([15],[20]) qui garantit qu'un programme respecte un ensemble de critères de sécurité et qui garantit la confidentialité de l'information traitée. Il y a aussi les preuves formelles basées sur la logique de Hoare [13] qui garantissent la bonne exécution d'un programme et le respect de sa spécification.

Dans le présent document, nous nous intéressons à une méthode formelle, appelée l'évaluation de modèle[3], qui est une approche automatique visant à déterminer si toutes les exécutions possibles d'un modèle respectent un comportement donné. Le modèle en question est habituellement une représentation d'un programme sous la forme d'un automate à transitions, décrivant son comportement d'après les entrées et le traitement qu'il effectue. La figure 1.1 présente un exemple d'automate modélisant le comportement d'une machine distributrice. Cette machine peut servir du café ou du thé. L'utilisateur doit insérer de l'argent et ensuite appuyer sur un des deux boutons étiquetés café et thé. Un automate est habituellement représenté par un ensemble d'états, les cercles, avec des transitions les reliant. Dans l'exemple, chacun des états possède un nom, permettant de les différencier. Sur les transitions, on retrouve aussi une étiquette, que l'on appelle action. Ces actions représentent un choix fait par l'environnement du système, ce qui amène la machine à effectuer une transition, d'un état vers un autre. L'état initial dans lequel la machine démarre est l'état `prêt`. Pour initier une transition, l'utilisateur doit insérer de l'argent. Ce faisant, la machine à café fait une transition vers l'état `choix`. L'utilisateur a alors trois choix : il peut demander du café avec le bouton café, il peut demander du thé, avec thé et il peut demander d'être remboursé, en appuyant sur remboursement. Après avoir servi une boisson, la distributrice revient à l'état initial, prête à servir un autre client.

Le but de la vérification formelle est de vérifier qu'un comportement est valide pour toutes les exécutions possibles. Par exemple, vérifions le comportement « *il est impossible d'avoir du café sans avoir mis de l'argent immédiatement avant* » sur la distributrice. Il faut alors vérifier tous les chemins possibles menant à `café_servi`.

[3] Dans la littérature, on retrouve habituellement l'expression anglaise « *model-checking* ».

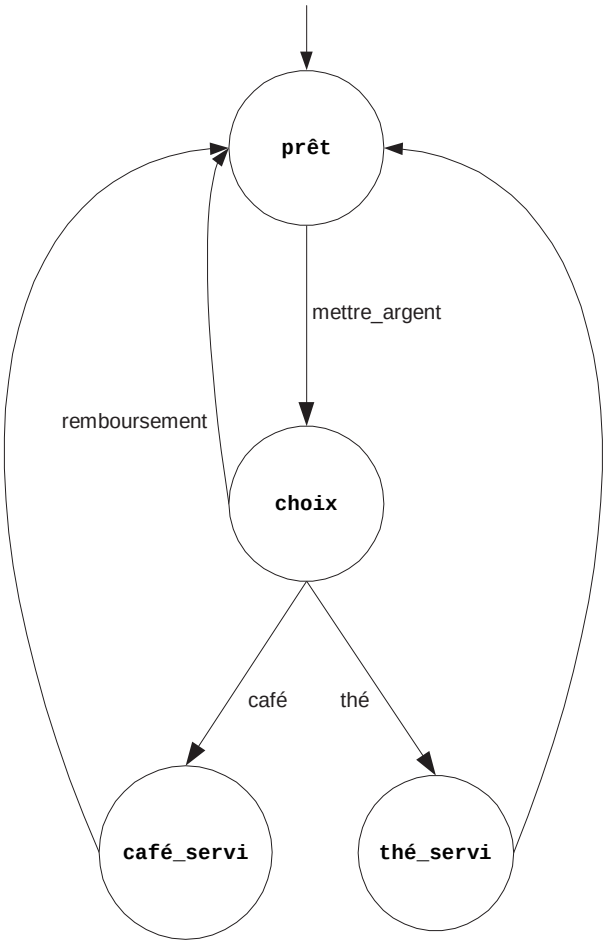

FIG. 1.1 – Automate représentant le comportement d'une distributrice automatique.

Comme la distributrice peut servir un nombre infini de clients, d'après l'automate de la figure 1.1, il peut paraître impossible de vérifier toutes les exécutions. Une vérification est tout de même réalisable, puisque les séquences possibles d'actions se répètent et l'on peut réduire les possibilités à quelques cas, commençant tous à l'état `prêt`. Étant donné que la propriété à étudier spécifie ce qui doit arriver avant d'être à l'état `café_servi`, nous devons étudier les chemins possibles pour y accéder. En fait, il n'y a qu'une seule possibilité, puisque cet état ne possède qu'une seule transition entrante. L'état précédent est donc `choix` pour toutes les exécutions possibles. De même, l'état précédent de ce dernier est toujours `prêt`. Or, pour initier la transition entre `prêt` et `choix`, l'utilisateur doit obligatoirement mettre de l'argent dans la machine. On est assuré alors que la machine n'offrira pas de choix à un utilisateur n'ayant pas payé. La propriété est donc vérifiée, puisque toutes les exécutions reviennent à l'état initial `prêt`.

Maintenant, introduisons une petite variante à l'exemple précédent. La figure 1.2

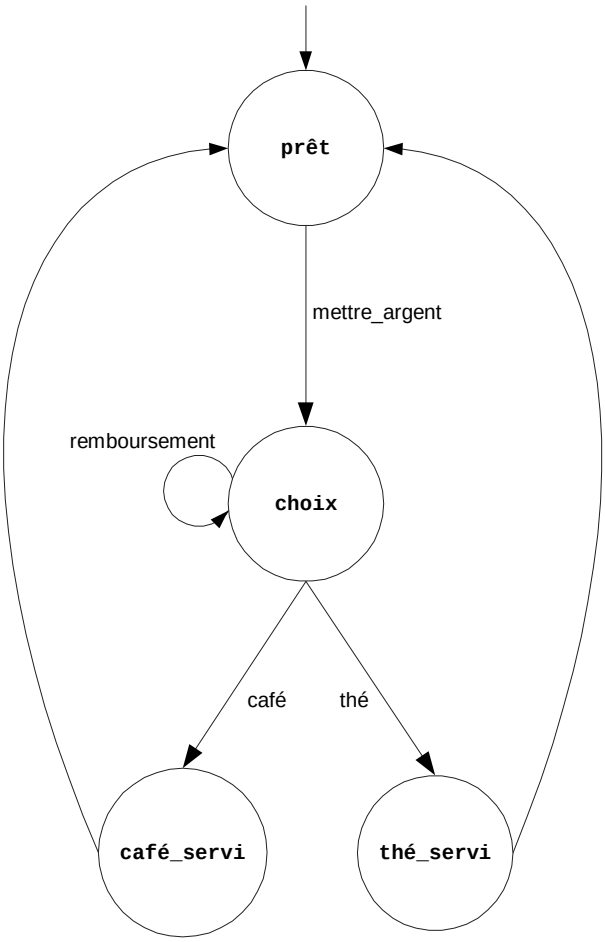

FIG. 1.2 – L'automate modifié de la distributrice.

présente un autre automate pour une distributrice automatique. Supposons que, suite à une erreur de conception, la distributrice revienne à l'état `choix` après avoir remboursé le client (par exemple si la machine laissait les boutons café et thé activés). On voit alors que pour la séquence d'actions [`mettre_argent`, remboursement, café] est possible. Si on suppose que l'action remboursement redonne au client l'argent qu'il avait mis au départ, la séquence permet d'avoir un café sans l'avoir payé. La vérification formelle permet de détecter des cas semblables pour conclure que cet automate ne respecte pas la propriété désirée. Cet exemple est très simple, mais il n'est pas rare que les systèmes contiennent des milliers d'états et qu'une vérification soit difficile à effectuer sans outil informatique.

En général, le comportement que l'on veut imposer au modèle est décrit avec une formule de logique, pour éviter toute ambiguïté d'interprétation et faciliter le traitement. Les techniques de vérification formelle s'intéressent chacune à un type de modèle précis et à une logique particulière. Elles utilisent chacune un algorithme basé sur une preuve

mathématique rigoureuse pour déterminer si le modèle respecte la formule.

Dans notre cas, le modèle est probabiliste, ce qui signifie que nous ne connaissons pas a priori dans quel état sera précisément le modèle après avoir effectué une transition. Le modèle pondère les différentes possibilités d'états successeurs par des probabilités. Par exemple, on pourrait avoir une machine avec un bouton tel que si on appuie, la machine effectue un traitement avec une probabilité de 99.9%. Dans le cas complémentaire, la machine ne répond pas. Il faut alors appuyer à nouveau. L'intérêt d'ajouter des probabilités dans un modèle vient de deux motivations différentes. La première est que le système représenté par le modèle peut avoir un comportement aléatoire. Par exemple, on peut penser à un algorithme qui choisit au hasard un individu parmi plusieurs candidats. On pourrait alors vérifier si l'algorithme est équitable pour chacun des candidats. L'équipe du vérificateur probabiliste PRISM[16] a plusieurs études de cas sur des algorithmes probabilistes[4]. La deuxième motivation est qu'il peut être plus facile et plus juste d'abstraire le comportement d'un système par des probabilités, bien que le système n'ait pas de comportement aléatoire en soi. On peut vouloir abstraire le comportement si trop de paramètres doivent être pris en compte pour établir quel est l'état successeur, après une transition. La machine dont nous avons parlé précédemment en est un exemple : nous ne connaissons pas exactement tous les facteurs pour lesquels la machine ne répondrait pas, mais nous savons que cette possibilité arrive dans 0.1% des cas. Un autre exemple serait le système de pilotatge automatique d'un avion. Il y a trop de paramètres physiques à mesurer pour déterminer précisément la trajectoire d'un avion. Par contre, on peut estimer le déplacement de l'avion en ne tenant compte que des principaux facteurs comme l'altitude, la vitesse et l'orientation du vent. On pourrait alors vérifier si l'avion demeurera en vol avec une probabilité supérieure à 99.99999%. Le chapitre suivant s'applique à décrire en détails le type de modèle avec lequel nous allons travailler.

Le but du présent travail est de développer un vérificateur formel pour une famille de processus de Markov étiquetés [6, 7, 8]. Pour atteindre ce but, nous présentons dans le chapitre suivant le modèle et la logique. Au chapitre 3, nous présentons deux langages formels que nous avons définis pour décrire les processus de Markov étiquettés et un exemple complexe qui nous permettra d'illustrer la puissance des langages. Ce chapitre présente aussi deux algorithmes pour effectuer la vérification formelle. Finalement, au chapitre 4, nous présentons l'implémentation en Java que nous avons faite, basée sur l'un des algorithmes, ainsi que certains résultats obtenus. Les contributions majeures de ce travail sont le langage de description pour les processus de Markov, la formalisation des algorithmes de vérification ainsi que l'implémentation. Sauf indications contraires, les définitions, lemmes et exemples que l'on retrouve dans ce document sont originaux.

[4]Pour plus de détails sur les études de cas de PRISM, voir http ://www.cs.bham.ac.uk/ dxp/prism/

Chapitre 2

Systèmes de transitions probabilistes continus

Dans ce chapitre, nous allons présenter d'une manière formelle les systèmes de transitions probabilistes continus. Nous énoncerons quelques définitions et présenterons de courts exemples.

Il y a plusieurs manières d'analyser et de représenter le comportement d'un système. Pour diverses raisons, certains optent pour l'analyse du traitement interne, mais nous nous intéressons plutôt aux interactions qu'un système peut avoir avec son environnement, c'est-à-dire les systèmes qui l'entourent et les utilisateurs. Dans une telle perspective, un système est dit *réactif*, puisqu'il réagit à son environnement, par un comportement directement déterminé par les stimuli auxquels il est exposé. L'environnement est toujours l'élément déclencheur d'une transition d'un état du système vers un autre. De notre point de vue, un système est étudié d'après son comportement extérieur, que l'on peut percevoir. Nous ne nous intéressons pas au traitement interne qu'il effectue, mais plutôt aux relations entre les entrées et les sorties. Ce choix nous permet d'analyser la composition de plusieurs systèmes, chacun étant un système en soi et faisant partie de l'environnement pour les autres. Pour illustrer ce qu'est un système réactif, prenons, par exemple, une machine distributrice : ce qu'elle servira dépendra du bouton appuyé par le client et l'action de le servir sera faite en réponse à ce choix. Le client représente ici l'environnement du système, qui lui dicte les actions à prendre et le système ne fait que répondre en prenant la transition associée à l'action demandée, c'est-à-dire de servir le bien demandé.

De plus, nos systèmes sont probabilistes puisqu'il existe plusieurs réponses à une action demandée et nous associons à chaque réponse une probabilité. La présence de pro-

babilités associées aux transitions permet de quantifier le *non-déterminisme*. D'autres types de modèle pour les systèmes réactifs permettent la présence de non-déterminisme, c'est-à-dire la présence, pour un état, de plusieurs transitions sortantes possédant la même action, vers des états différents. La signification d'une telle situation est que toutes ces transitions peuvent être prises par le système, mais que lors de l'exécution, une seule est choisie. A priori, rien ne permet de prédire ce choix, d'où l'expression *non-déterminisme*. Par exemple, le non-déterminisme réflète bien la situation où plusieurs processus se partagent un même processeur : pour analyser l'ordre d'exécution des processus, on doit considérer tous les ordonnancements possibles, si le choix d'un processus est purement aléatoire ou si on ne veut pas présumer de la politique d'ordonnancement.

Dans notre cas, si un état possède plusieurs transitions sortantes ayant la même action, on pondère par des probabilités le choix de la transition qui sera prise. Dans un sens, notre modèle est déterministe, puisque la distribution des probabilités est entièrement déterminée par l'état actuel du système et une action donnée. C'est le modèle introduit par Larsen et Skou pour modéliser les systèmes probabilistes ayant un espace d'états fini. Il est aussi appelé « système probabiliste réactif » [21], « fully probabilistic » [1] ou plus récemment « chaîne de Markov à temps discret »[16]. D'autres modèles probabilistes admettent plusieurs distributions différentes des probabilités, pour un même état de départ et une même action. On doit alors étudier tous les choix de distributions possibles. Ces modèles sont appelés « génératifs » [21],« probabilistes concurrents » [1] et plus récemment, « processus de décision Markov » [16]. Ces modèles ont tous un ensemble d'états fini, alors que nous nous intéressons aux systèmes probabilistes continus, ceux-ci ayant été introduits dans [2]

Pour illuster un peu plus nos systèmes probabilistes, reprenons l'exemple de la machine distributrice, mais avec une petite modification : la machine possède deux boutons, le premier étiqueté « Eau » et le second « Boissons ». Supposons que lorsqu'un client appuie sur le premier, il reçoit un verre d'eau avec 100% de chance et lorsqu'il appuie sur le second, il a 25% de chance de recevoir une boisson gazeuse, 25% de chance de recevoir un thé et aussi 25% de chance de recevoir un café. Le dernier 25% représente la possibilité que la machine ne réponde pas à l'action demandée. Ce dernier cas pourrait être dû à un blocage interne ou bien à un court-circuit. Le client devra appuyer à nouveau sur le bouton s'il désire une boisson. Cette machine distributrice peut réagir de plusieurs manières différentes à l'action Boissons, sans connaître a priori laquelle sera choisie. Mais, nous connaissons la probabilité de chacune des réponses que le système peut faire.

Nos systèmes probabilistes peuvent posséder un espace d'états non-dénombrable. Dans l'exemple précédent, la machine ne possède qu'un seul état, à savoir `Prêt-à-`

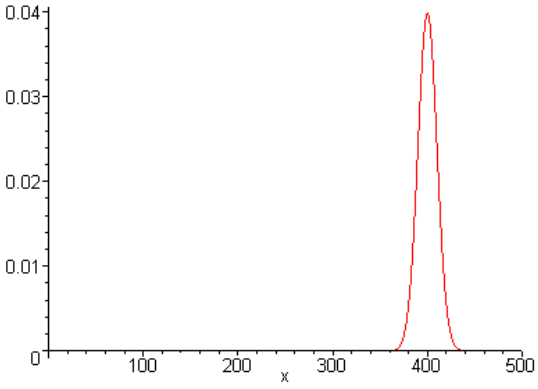

FIG. 2.1 – La fonction de densité normale avec une moyenne de 400 et une variance de 10.

`servir`. Dans d'autres exemples, comme la distributrice de café de la figure 1.1, l'ensemble des états peut être fini. Pour illustrer un système probabiliste possédant un ensemble d'états non-dénombrable, revenons avec notre exemple de la distributrice et supposons qu'il n'y ait qu'un seul bouton étiqueté Café. De plus, supposons que l'état, un nombre réel, représente la quantité de café versé lorsque le client appuie sur ce bouton. Fixons l'ensemble des états possibles à $[0, 500]$, signifiant que la distributrice peut verser entre 0 et 500 millilitres de café. Au départ, la distributrice est à l'état 0. Ici, toutes les valeurs entre 0 et 500 sont possibles et pas seulement un certain nombre fini (Ex : les nombres entiers entre 0 et 500). De plus, supposons que la distribution probabiliste des valeurs suit une loi normale, de moyenne 400 et de variance 10. La figure 2.1 nous rappelle que la fonction de densité d'une loi normale est symétrique autour de sa moyenne et que les valeurs ayant le plus de poids sont celles les plus proches de la moyenne.

Ainsi, il n'y a qu'une seule action possible et lorsque l'environnement la déclenche, la distributrice prend une transition vers un nouvel état, qui représente la quantité de café versé. Puisque l'ensemble des états est non-dénombrable, la probabilité d'avoir une quantité de café précise est 0. Par contre, on peut calculer la probabilité d'avoir une quantité appartenant à un ensemble particulier. Par exemple, la probabilité que la distributrice serve entre 400ml et 500ml de café est d'environ 1/2. De même, la probabilité d'avoir entre 0ml et 350ml est pratiquement nulle. Pour complexifier l'exemple, on pourrait avoir deux boutons, l'un pour verser une petite quantité et l'autre pour en verser une plus grande. La distribution des états pourrait être encore selon une loi normale, mais la moyenne et la variance pourraient être propres à chaque bouton. Le premier pourrait avoir une moyenne de 250ml, alors que l'autre pourrait avoir une moyenne de 400ml.

Dans ce qui suit, nous présentons formellement le modèle avec lequel nous travaillons, les processus de Markov étiquetés.

2.1 Quelques définitions et quelques exemples

Les systèmes probabilistes continus sont des systèmes réactifs, puisqu'ils réagissent à leur environnement, sont probabilistes dans le choix de leur nouvel état et leur ensemble d'états est non-dénombrable. Nous allons introduire dans ce qui suit des définitions formelles afin de mieux développer notre théorie. Les quatre premières définitions sont classiques dans le domaine des systèmes de transitions probabilitstes continus[2].

Définition 2.1.1 *Soit X un ensemble. Une* ***σ-algèbre*** *$\Sigma \in 2^X$ sur X est telle que*

1. $X \in \Sigma$
2. $\forall (A_i)_{i\in\mathbb{N}} \in \Sigma$, $\bigcup_{i\in\mathbb{N}} A_i \in \Sigma$
3. $\forall A \in \Sigma$, $\overline{A} := X - A \in \Sigma$.

On appelle alors (X, Σ) un *espace mesurable*. Dans le cadre de ce travail, nous nous restreignons à l'espace mesurable des nombres réels avec la σ-algèbre des intervalles. La théorie des systèmes de transitions probabilistes continus a tout de même été développée dans un cadre plus général, pour un espace mesurable quelconque. Les trois définitions suivantes se situe dans ce cadre général.

Définition 2.1.2 *Une* ***mesure de probabilité partielle*** *sur un espace mesurable (X, Σ) est une application $\mu : \Sigma \to [0, 1]$ telle que*

1. $\mu(\emptyset) = 0$,
2. $\mu(\cup A_i) = \cup\mu(A_i)$, *où $\cup A_i$ est une union dénombrable d'ensembles disjoints et telle que*
3. $\mu(X) \in [0, 1]$.

Dans la définition suivante, la notation $\mu(x, \cdot)$ représente la fonction μ auquel on fixe le premier paramètre, mais où le second demeure libre.

Définition 2.1.3 *Une* ***fonction de transition probabiliste partielle*** *sur un espace mesurable (X, Σ) est une fonction $\mu : X \times \Sigma \to [0, 1]$ telle que pour chaque $x \in X$, la*

fonction $\mu(x, \cdot)$ est une mesure de probabilité partielle et pour chaque $E \in \Sigma$, la fonction $\mu(\cdot, E)$ est mesurable, c'est-à-dire $\mu^{-1}(A, E) \in \Sigma$, pour $A \in \Sigma$.

Dans le cadre de cette recherche, nous nous restreignons aux systèmes dont l'ensemble des états est un sous-ensemble des nombres réels. Dans le présent document, l'espace mesurable sera donc toujours les réels et nous fixons Σ comme étant la σ-algèbre engendrée par les intervalles réels. Pour ce qui est des fonctions de probabilité, $\mu(x, E)$ représente la probabilité que le système prenne une transition à partir de l'état x vers un état dans E. Cette probabilité est *conditionnelle* puisqu'elle donne la probabilité d'aller vers un état de E *étant donné* que le système est présentement à l'état x. Par contre, comme on le remarquera dans la prochaine définition, la probabilité ne dépend pas de l'historique du système, à savoir quels sont les états qui ont été visités avant x et dans quel ordre. Cette condition est connue sous le nom de *Condition de Markov*.

Les fonctions de transition sont dites *partielles* parce que leur image n'est pas $\{0, 1\}$ mais plutôt $[0, 1]$, ce qui permet des situations où $0 < \mu(x, E) < 1$. Le fait que cette probabilité soit strictement supérieure à zéro signifie que la transition est possible, mais le fait qu'elle soit strictement inférieure à 1 signifie que le système pourrait ne pas répondre à l'action demandée. Pour illustrer cette situation, pensons à un système de transition probabiliste comme à une boîte noire sur laquelle il y a plusieurs boutons, un pour chaque action. Cette idée de la boîte noire a été introduite par Milner [18] pour illustrer l'analyse du calcul CSS. Chaque bouton peut être actif, signifiant que l'environnement peut choisir d'appuyer dessus ou bien il peut être désactivé, si l'action associée est impossible pour cet état. Alors, à chaque fois que l'environnement déclenche une action, c'est comme s'il appuyait sur un bouton actif et le système peut répondre en faisant une transition vers un autre état avec une configuration potentiellement différente de boutons activés et désactivés. Il se peut aussi que le système ne réponde pas à l'action demandée, ce qui signifie qu'il ne fait rien et reste au même état. Il est à noter que cette situation est différente de celle où le système fait un traitement en réponse à l'action et revient au même état, puisqu'ici, le système a répondu au stimulus de son environnement. Notons toutefois que l'explication ci-haut est une interprétation possible et que le modèle avec probabilités totales est aussi étudié. Néanmoins, c'est un débat mineur qui n'influence les résultats d'aucune façon.

Nous allons maintenant énoncer la définition d'un processus de Markov étiqueté, qui est une forme de système probabiliste continu. Pour plus de détails, le lecteur est invité à consulter [2].

Définition 2.1.4 *Un* ***processus de Markov étiqueté*** *ou* ***LMP*** [1] *avec un ensemble d'étiquettes* Act *est un quadruplet* $(S, s_0, \Sigma, \{\mu_{\mathsf{a}} \mid \mathsf{a} \in \mathsf{Act}\})$*, où* S *est l'ensemble des états,* s_0 *est l'état initial,* Σ *est une* σ*-algèbre borélienne sur* S*, et*

$$\forall \mathsf{a} \in \mathsf{Act}, \mu_{\mathsf{a}} : S \times \Sigma \longrightarrow [0, 1]$$

est une fonction de transition probabiliste partielle.

Comme nous l'avons mentionné plus haut, les LMP ne conservent pas la liste ni l'ordre des états visités pour définir la distribution des probabilités pour les états. La probabilité dépend seulement de l'état courant, de l'action et de l'ensemble des états d'arrivée.

Comme nous nous restreignons aux systèmes dont les états sont des nombres réels, nous allons omettre Σ de la syntaxe, dans le reste du document. Nous fixons Σ comme étant la σ-algèbre engendrée par les intervalles réels. De plus, nous allons permettre des étiquettes dans les états d'un système. Les étiquettes, prises dans un ensemble $\mathtt{Atom}$, servent à identifier certains groupes d'états, en les qualifiant d'une valeur, *vrai*, pour ces états. Les étiquettes servent habituellement à donner une valeur sémantique à certains états du système. Donc, nous ajoutons dans la définition l'ensemble des étiquettes ainsi qu'une fonction, $label : \mathtt{Atom} \to \Sigma$, qui retourne l'ensemble des états qui satisfont une étiquette donnée. Nous allons donner une nouvelle définition des LMP pour refléter ces modifications de la définition traditionnelle.

Définition 2.1.5 *Un* **LMP*** *est un tuple de la forme*

$$(S, i, \mathsf{Act}, \mathtt{Atom}, label, \{\mu_{\mathsf{a}} \mid \mathsf{a} \in \mathsf{Act}\})$$

où $S \in \Sigma$ *est l'ensemble des états,* $i \in S$ *est l'état initial,* Act *est l'ensemble des actions,* $\mathtt{Atom}$ *est l'ensemble dénombrable des étiquettes sur les états,* $label : \mathtt{Atom} \to \Sigma$ *est une fonction mesurable retournant l'ensemble des états qui satisfont une étiquette donnée et*

$$\forall \mathsf{a} \in \mathsf{Act}, \mu_{\mathsf{a}} : S \times \Sigma \longrightarrow [0, 1]$$

est une fonction de transition probabiliste partielle.

Dans le présent document, nous utiliserons l'expression *systèmes probabilistes continus* pour désigner l'ensemble des LMP*, puisque ces systèmes ont une fonction de

[1] Dans la littérature, on retrouve le sigle LMP, qui provient de l'expression anglaise « labelled Markov processes ».

transition *probabiliste* et parce que l'état dans lequel se trouve un de ces systèmes au cours de son exécution évolue à la manière d'une variable aléatoire *continue*.

Nous allons présenter ici quelques exemples pour illustrer la définition précédente. Nous dénotons par $P_{\mathsf{a}}(x, E)$ la probabilité d'aller de l'état x vers un état appartenant à l'ensemble E, en faisant l'action a. Le premier exemple reprend celui sur la machine distributrice vu plus haut, avec quelques précisions de plus.

Exemple 2.1.6 *La machine distributrice peut servir deux quantités de café, en millilitres. Nous la présentons sous forme d'un LMP* :*

$$\mathcal{S} = ([0, 750], 0, \{\mathsf{P}, \mathsf{G}\}, \{\texttt{deborde}\}, \{\texttt{deborde} \mapsto [500, 750]\}, \{\mu_{\mathsf{P}}, \mu_{\mathsf{G}}\})$$

L'ensemble des états de la machine distributrice est $[0, 750]$ *pour représenter la quantité de café qui peut être servie. La machine possède deux boutons,* P *et* G*, pour demander un petit verre et un grand verre de café respectivement. L'étiquette* `deborde` *est là pour identifier les états où nous jugeons qu'il pourrait y avoir un débordement du verre. Les fonctions de transitions probabilistes sont données comme suit :*

$$\begin{aligned} \mu_{\mathsf{P}}(x, [a, b]) &= \Phi\left(\frac{b - 250}{10}\right) - \Phi\left(\frac{a - 250}{10}\right) \\ \mu_{\mathsf{G}}(x, [a, b]) &= \Phi\left(\frac{b - 400}{50}\right) - \Phi\left(\frac{a - 400}{50}\right) \end{aligned}$$

où $x \in [0, 750]$ *et* $[a, b] \subseteq [0, 750]$*. La fonction* Φ *est la fonction de répartition d'une variable normale standard de moyenne 0 et de variance 1. L'annexe A sur les probabilités nous rappelle que la fonction de répartition* $F(z)$ *d'une variable* Z *suivant une loi normale de moyenne* μ *et de variance* σ^2 *est :*

$$F(z) = \Phi\left(\frac{z - \mu}{\sigma}\right)$$

où $\Phi(z)$ *est la fonction de répartition de la loi normale standard. On remarque que la fonction de transition probabiliste pour l'action* P *suit une loi normale de moyenne 250 et d'écart-type 10. Celle pour* G *suit une loi normale de moyenne 400 et d'écart-type 50. La figure 2.2 nous montre les deux fonctions de densité associées aux actions. On remarque que celle pour* P *est plus concentrée autour de sa moyenne, alors que celle pour* G *est plus évasée, ce qui implique que la quantité de café versée par cette action peut varier plus que celle pour* P*.*

À partir de ce système, on peut calculer la probabilité d'avoir un verre qui déborde,

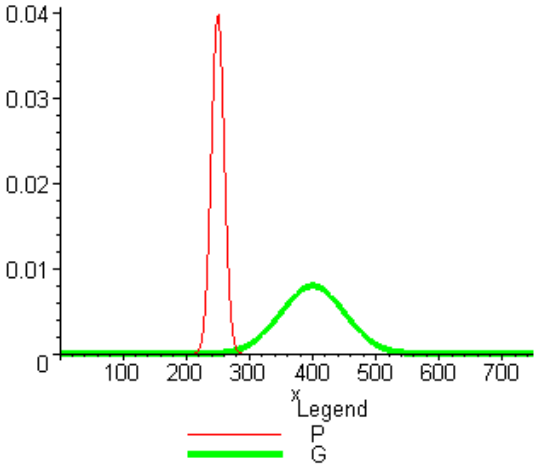

FIG. 2.2 – Les fonctions de densité associées à P et à G

pour chacune des actions.

$$\begin{aligned} P_{\mathsf{P}}(\texttt{deborde}) &= \mu_{\mathsf{P}}(x, [500, 750]) \\ &= \Phi\left(\frac{750-250}{10}\right) - \Phi\left(\frac{500-250}{10}\right) \\ &\approx 0 \\ P_{\mathsf{G}}(\texttt{deborde}) &= \mu_{G}(x, [500, 750]) \\ &= \Phi\left(\frac{750-400}{50}\right) - \Phi\left(\frac{500-400}{50}\right) \\ &\approx 0.0227 \end{aligned}$$

Donc, si un client demande un grand verre de café, il y a plus de 2% de chance que le verre déborde.

Supposons qu'un client puisse être insatisfait si la quantité versée par la distributrice est inférieure à un certain seuil. Fixons ce seuil à 200 *pour l'action* P *et à* 350 *pour l'action* G*. Calulons la probabilité qu'il soit insatisfait, pour chacune des actions.*

Pour $x \in [0, 750]$,

$$\begin{aligned} P_{\mathsf{P}}(x, [0, 200]) &= \mu_{\mathsf{P}}(x, [0, 200]) \\ &= \Phi\left(\frac{200 - 250}{10}\right) - \Phi\left(\frac{-250}{10}\right) \\ &\approx 0 \\ P_{\mathsf{G}}(x, [0, 350)) &= \mu_G(x, [0, 350]) \\ &= \Phi\left(\frac{350 - 400}{50}\right) - \Phi\left(\frac{-400}{50}\right) \\ &\approx 0.1587\ . \end{aligned}$$

Ainsi, pour l'action P*, il est presque impossible que le client soit insatisfait, alors que pour* G*, il y a une probabilité d'environ 15% qu'il le soit.*

L'exemple précédent est simpliste, pour deux raisons. Premièrement, les deux actions sont toujours possibles et ce, pour tous les états du système, puisque les probabilités $P_{\mathsf{P}}(x, [0, 750])$ et $P_{\mathsf{G}}(x, [0, 750])$ pour $x \in [0, 750]$ sont supérieures à 0. Si ce n'était pas le cas, c'est qu'une action serait en fait *désactivée* pour un certain état x, puisque la probabilité de faire une transition vers n'importe quel état du système serait 0, à partir de x. Deuxièmement, le système est trivial parce que $P_{\mathsf{P}}(x, [0, 750])$ et $P_{\mathsf{G}}(x, [0, 750])$ sont égales à 1, ce qui implique que le système répondra toujours à l'action demandée, comme nous l'avons expliqué plus haut. Donc, si nous faisons abstraction de la quantité de café versée, l'exemple se simplifie en un système à un seul état et à deux actions possibles, toujours activées. En effet, en faisant abstraction de la quantité versée, la machine distributrice évolue à la manière d'un automate à un seul état qui possède deux boucles, l'une étiquettée P et l'autre G.

FIG. 2.3 – Un automate à un état et deux transitions.

Nous allons présenter ici un exemple moins simpliste de système probabiliste tiré de [6]. Plus académique que réel, cet exemple nous servira à illustrer le formalisme que nous voulons introduire plus bas. Pour la description des transitions, nous utilisons la notation $[0, 1)$ pour désigner l'ensemble $\{x \in \mathbb{R} \mid 0 \leq x \text{ et } x < 1\}$.

Exemple 2.1.7 *Notre exemple est un système dont l'ensemble des états* S *est* $[0, 3] \cup \{4, 5\}$*, où l'état initial est* 1 *et l'ensemble des actions possibles est* $\{\mathsf{a}, \mathsf{b}\}$*. Voici les transitions :*

- *Si* $x \in [0,1]$, $P_{\mathsf{a}}(x,[0,z)) = \frac{x+z}{4}$, *où* $0 \leq z \leq 1$.
  $P_{\mathsf{a}}(x,\{1\}) = \frac{1-x}{4}$,
  $P_{\mathsf{a}}(x,(1,1+z]) = \frac{z}{4}$,
  $P_{\mathsf{a}}(x,(2,2+z]) = \frac{xz}{4}$,
- *Si* $x \in (1,2]$, $P_{\mathsf{a}}(x,\{4\}) = 1$.
- *Si* $x \in (2,3]$, $P_{\mathsf{b}}(x,\{5\}) = 1$.

Dans la définition du système, l'on remarque qu'il manque des probabilités : par exemple, il n'y a pas de probabilité pour définir les transitions entre les états dans $[0,1]$ *et l'état* $\{4\}$*. Dans de tels cas, nous supposons qu'ils n'y pas de transition possible et qu'ainsi, les probabilités sont nulles.*

La figure 2.4 illustre le système d'une manière informelle, où les états sont représentés par des ensembles, tout comme dans les probabilités ci-haut, et où les transitions sont représentées par l'action et la probabilité.

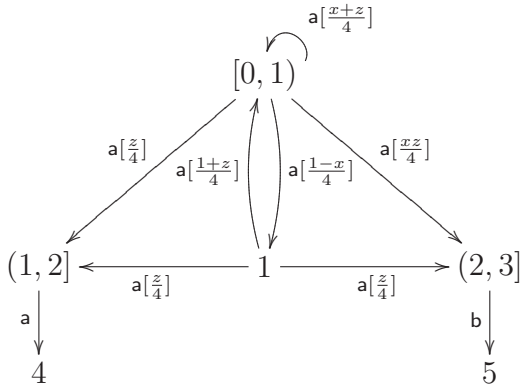

FIG. 2.4 – Un système continu

Plus loin, lorsque nous parlerons du calcul d'approximation et la satifaction de formule de logique, nous devrons être en mesure de calculer diverses probabilités de transitions à travers l'ensemble des états. Pour cela, nous devrons combiner des probabilités afin de calculer $P_{\mathsf{a}}(x,E)$, $\forall\, x \in S$ *et* $\forall\, E \subseteq S$.

Calculons la probabilité $P_{\mathsf{a}}(x,S)$ *pour* $x \in [0,1]$*, qui représente la probabilité d'aller d'un état dans* $[0,1]$ *vers un autre dans tout* S*. Pour y arriver, nous pouvons additionner les probabilité* $P_{\mathsf{a}}(x,[0,3])$, $P_{\mathsf{a}}(x,\{4\})$ *et* $P_{\mathsf{a}}(x,\{5\})$*, puisque les ensembles d'arrivée sont disjoints.*

Par conséquent, pour $x \in [0,1]$, nous obtenons :

$$\begin{aligned} P_{\mathsf{a}}(x,S) &= P_{\mathsf{a}}(x,[0,3]) + P_{\mathsf{a}}(x,\{4\}) + P_{\mathsf{a}}(x,\{5\}) \\ &= P_{\mathsf{a}}(x,[0,1)) + P_{\mathsf{a}}(x,\{1\}) + P_{\mathsf{a}}(x,(1,2]) + P_{\mathsf{a}}(x,(2,3]) + 0 \\ &= \frac{x+1}{4} + \frac{1-x}{4} + \frac{1}{4} + \frac{x}{4} \\ &= \frac{3+x}{4} \end{aligned}$$

Cette probabilité sert à déterminer si l'action a *est possible, ce qui est le cas pour* $x \in [0,1]$*, puisqu'alors,* $P_{\mathsf{a}}(x,S) > 0$*. Pour l'état initial du système, 1, cette probabilité vaut 1, ce qui signifie que nous sommes certain que le système fera une transition si nous demandons l'aciton* a*. À partir de la formule ci-haut, nous pouvons calculer la probabilité de rester à l'état* $x \in [0,1]$*, avec l'action* a*. En effet, cela revient à calculer le complément du résultat précédent, ce qui vaut* $\frac{1-x}{4}$*.*

De même, nous pouvons calculer les probabilités suivantes :

$$\begin{aligned} P_{\mathsf{a}}(x,\{0\}) &= \frac{x}{4} \\ P_{\mathsf{a}}(x,(2,3]) &= \frac{x}{4} \\ P_{\mathsf{a}}(x,(0,1)) &= \frac{x+1}{4} - \frac{x}{4} = \frac{1}{4} \\ P_{\mathsf{a}}(x,[0,2]) &= \frac{x+1}{4} + \frac{1-x}{4} + \frac{1}{4} = \frac{3}{4} \\ P_{\mathsf{a}}(x,(0,2]) &= \frac{3}{4} - \frac{x}{4} = \frac{3-x}{4} \end{aligned}$$

Le première probabilité se calcule en prenant $z = 0$ *dans la première branche de la définition du système. La seconde se calcule de la même manière en prenant* $z = 1$ *dans la quatrième branche. La suivante est la différence entre* $P_{\mathsf{a}}(x,[0,1))$ *et* $P_{\mathsf{a}}(x,\{0\})$*. La quatrième se calcule en additionnant les trois probabilités* $P_{\mathsf{a}}(x,[0,1))$*,* $P_{\mathsf{a}}(x,\{1\})$ *et* $P_{\mathsf{a}}(x,(1,2])$*. La dernière probabilité est la soustraction de la quatrième par la première.*

Comme nous désirons automatiser le processus de vérification formelle, nous avons besoin d'un langage formel pour décrire les comportements qu'un système doit ou ne doit pas avoir. Ce langage de description doit être sans ambiguïté et permettre facilement de vérifier si un système respecte le comportement. La section suivante vise à formuler ce langage.

2.2 Logique

Dans la section qui précède, nous avons défini et présenté le modèle qui nous servira à la vérification formelle. Nous allons présenter ici la logique qui nous servira à spécifier des propriétés que nous voudrons vérifier à l'aide du vérificateur formel. Avec cette logique, il est possible de décrire des comportements que l'on veut imposer à un système. Nous allons définir tout d'abord la syntaxe et la sémantique, pour ensuite présenter quelques exemples de formules de logique.

La logique est dérivée de celle introduite par Hennessys et Milner [12] et adaptée par Larsen et Skou [17] pour la vérification de la bisimulation de processus stochastiques. Nous l'avons aussi adaptée en y enlevant l'opérateur Δ_{a} et en y ajoutant les propositions atomiques sur les états en tant que formules.

Voici la syntaxe de la logique que nous acceptons

$$\phi := \mathsf{T} \mid \mathtt{p} \mid \phi \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \langle \mathsf{a} \rangle_q \phi$$

où $\mathtt{p} \in \mathtt{Atom}$ est une proposition atomique sur des états, $\mathsf{a} \in \mathsf{Act}$ est une action et q est un rationnel dans $[0, 1)$.

La sémantique de cette logique est la suivante, pour un LMP* $\mathcal{S} = (S, i, \mathsf{Act}, \mathtt{Atom}, \mathit{label}, \rightarrow)$. Soit $s \in S$ un état, $s \models \phi$ signifie que s satisfait la formule ϕ. Par abus de notation, nous écrirons $E \models \phi$ pour signifier que $\forall s \in E$, $s \models \phi$. La définition de la relation $\models$ se fait par induction. Tout d'abord, tous les états de S satisfont la formule T [2]. $s \models \mathtt{p}$ si et seulement si $s \in \mathit{label}(\mathtt{p})$. $s \models \neg\phi$ si et seulement si s ne satisfait pas la formule ϕ, i.e. $s \not\models \phi$. $s \models \phi_1 \wedge \phi_2$ si et seulement si $s \models \phi_1$ et $s \models \phi_2$. Nous notons $s \models \langle \mathsf{a} \rangle_q \phi$ si et seulement si il existe un ensemble $E \subseteq S$ tel que $\forall s' \in E$, $s' \models \phi$ et tel que $P_{\mathsf{a}}(s, E) > q$. Autrement dit, il est possible de faire une a-transition, avec une probabilité strictement supérieure à q, vers un état qui satisfait ϕ. Le système $\mathcal{S}$ satisfait une formule, noté $\mathcal{S} \models \phi$, si son état initial satisfait la formule. D'une manière générale, nous écrivons $[\![\phi]\!]$ pour dénoter l'ensemble des états qui satisfont ϕ, i.e. l'ensemble $\{s \in S \mid s \models \phi\}$.

Il est facile de voir que nous pouvons résumer la sémantique de la logique de la manière suivante.

[2] de « true », vrai en anglais.

Lemme 2.2.1 *Pour* $\mathcal{S} = (S, i, \mathsf{Act}, \mathtt{Atom}, label, \rightarrow)$, $s \in S$ *et* $\mathtt{p} \in \mathtt{Atom}$,

$$\begin{array}{rcl} \mathcal{S} \models \phi & \iff & i \models \phi \\ s \models \phi & \iff & s \in [\![\phi]\!] \\ [\![\mathsf{T}]\!] & = & S \\ [\![\mathtt{p}]\!] & = & label(\mathtt{p}) \\ [\![\neg\phi]\!] & = & S \setminus [\![\phi]\!] \\ [\![\phi_1 \wedge \phi_2]\!] & = & [\![\phi_1]\!] \cap [\![\phi_2]\!] \\ [\![\phi_1 \vee \phi_2]\!] & = & [\![\phi_1]\!] \cup [\![\phi_2]\!] \\ [\![\langle \mathsf{a} \rangle_q \phi]\!] & = & \{s \in S \mid P_{\mathsf{a}}(s, [\![\phi]\!]) > q\} \ . \end{array}$$

Pour illustrer la sémantique de la logique, nous présentons ici quelques exemples de formules, leur signification et nous déterminons si elles sont satisfaites par un modèle donné.

Exemple 2.2.2 *Comme modèle, prenons le système* $\mathcal{S} = (S, i, \mathsf{Act}, \mathtt{Atom}, label, \rightarrow) = ([0,3] \cup \{4,5\}, 1, \{\mathsf{a}, \mathsf{b}\}, \{\mathtt{safe}, \mathtt{exit}\}, \{\mathtt{safe} \mapsto [1.5, 2], \mathtt{exit} \mapsto \{4,5\}\}, \rightarrow)$ *donné à l'exemple 2.1.7, dans lequel nous avons ajouté les propositions atomiques* `safe` *et* `exit` *sur des états.*

Considérons tout d'abord la formule $\langle \mathsf{a} \rangle_0 \mathsf{T}$. *On l'interprète en « l'action* a *est possible » ou « activée », puisque la probabilité de faire* a *doit être strictement supérieure au seuil de 0. Un état* x *satisfait cette formule si et seulement si* $P_{\mathsf{a}}(x, S) > 0$. *Dans l'exemple 2.1.7, nous avons montré que* $P_{\mathsf{a}}(x, S) = \frac{3+x}{4}$ *pour* $x \in [0,1]$. *Donc, l'état initial 1 de* $\mathcal{S}$ *satisfait cette formule. D'ailleurs, l'état 1 satisfait toutes les formules de la forme* $\langle \mathsf{a} \rangle_{1-\epsilon} \mathsf{T}$, $\forall \epsilon \in (0,1]$ *puisque* $[\![\langle \mathsf{a} \rangle_{1-\epsilon} \mathsf{T}]\!] = \{s \in S | P_{\mathsf{a}}(s, S) > 1 - \epsilon\}$ *et* $P_{\mathsf{a}}(1, S) = 1 > 1 - \epsilon$.

Une autre formule est $\langle \mathsf{a} \rangle_0 \langle \mathsf{a} \rangle_0 \mathtt{exit}$ *qui représente le fait que le système puisse faire l'action* a *deux fois de suite et qu'il soit ensuite dans un état qui satisfait l'étiquette* `exit`. *Pour vérifier si le système de l'exemple satisfait cette formule, calculons* $[\![\langle \mathsf{a} \rangle_0 \mathtt{exit}]\!]$. *Pour cela, nous devons calculer :*

$$P_{\mathsf{a}}(x, \mathtt{exit}) = P_{\mathsf{a}}(x, \{4,5\}) = \begin{cases} 1 & si\ x \in (1,2] \\ 0 & si\ x \in [0,1] \cup (2,3] \cup \{4,5\} \end{cases}$$

Donc, $[\![\langle \mathsf{a} \rangle_0 \mathtt{exit}]\!] = (1,2]$. *La probabilité d'aller vers* $(1,2]$ *avec l'action* a *est*

$$P_{\mathsf{a}}(x, (1,2]) = \begin{cases} \frac{1}{4} & \text{si } x \in [0,1] \\ 0 & \text{si } x \in (1,3] \cup \{4,5\} \end{cases}$$

d'après l'équation

$$P_{\mathtt{a}}(x, (1, 1+z]) = \frac{z}{4},$$

en prenant $z = 1$.

Alors, seuls les états dans $[0, 1]$ *satisfont* $\langle \mathtt{a} \rangle_0 \langle \mathtt{a} \rangle_0 \mathtt{exit}$. *Comme l'état initial fait partie de cet ensemble, le système satisfait la formule.*

Par contre, le système ne satisfait pas la formule $\langle \mathtt{a} \rangle_{1/4} \langle \mathtt{a} \rangle_0 \mathtt{exit}$, *parce que la probabilité d'aller de l'état initial vers* $(1, 2]$ *avec l'action* a *est précisément* $1/4$.

Notre modèle, les LMP*, permet de représenter des systèmes informatiques au comportement aléatoire ou bien des systèmes pour lesquels il y a trop de paramètres à considérer pour déterminer leur comportement. Notre point de vue étant l'analyse des interactions avec l'environnement, ce modèle permet d'étudier la composition de plusieurs systèmes d'après les relations entrées-sorties. Pour être en mesure de vérifier automatiquement qu'un système respecte certaines propriétés, il est nécessaire de définir un langage formel dont l'interprétation est directe, sans ambiguïté. L'utilisation de la logique que nous avons introduite permet d'atteindre cet objectif. Par contre, la manière de définir les fonctions de probabilités des LMP* n'est pas unique. Nous devrons choisir un formalisme, afin de fixer la manière avec laquelle les systèmes seront fournis à l'outil informatique que nous voulons implémenter. Le prochain chapitre s'intéresse à cette question.

Chapitre 3

Un vérificateur formel pour les LMP*

Dans le chapitre précédent, nous avons défini et présenté les systèmes probabilistes continus et nous avons introduit la logique. Le but du présent chapitre est de définir les bases pour construire un vérificateur formel pour les LMP*. Dans un premier temps, nous présentons deux langages formels pour décrire les fonctions de transitions probabilistes d'un système. Par la suite, nous présentons un algorithme de vérification formelle.

3.1 Un langage de description pour les transitions

Les deux langages introduits dans cette section, développés au cours de nos recherches, servent à décrire et à représenter les systèmes probabilistes continus. Le premier langage est basé sur les fonctions de densité pour décrire les probabilités. Le second utilise les fonctions de répartition. Les deux langages ont comme point central de la syntaxe la définition des transitions du système, qui sont en fait des blocs denses de notations saisissant l'aspect dynamique et probabiliste du système. Pour chacune des transitions, on regroupe dans une même notation l'ensemble des états de départ, l'ensemble des états d'arrivée, l'action que doit effectuer le système et la fonction qui définit la probabilité d'un sous-ensemble d'états d'arrivée. C'est en fait une manière particulière de décrire l'ensemble $\{\mu_{\mathsf{a}} \mid \mathsf{a} \in \mathsf{Act}\}$ des fonctions de probabilité pour un LMP*, tel que spécifié à la définition 2.1.5. À la fin de la section, nous ferons une comparaison des deux langages afin d'en choisir un pour l'implantation du vérificateur

formel.

3.1.1 Langage basé sur les fonctions de densité

Syntaxe

Comme nous l'avons vu dans les exemples de la section 2.1, il y a plusieurs manières différentes de décrire les fonctions de transitions pour les LMP*s. Nous définisson ici un langage formel pour fixer la notation, spécifiant ainsi comment les systèmes probabilistes seront fournis au vérificateur formel. Nous verrons aussi que le langage détermine grandement la manière de calculer les probabilités sur le système. Comme nous l'avons vu à la section 2.2 sur la logique, le calcul des probabilités occupe une place importante pour déterminer si une formule du type $\langle \mathsf{a} \rangle_q \phi$ est satisfaite.

Tout d'abord, nous énonçons le formalisme, ensuite nous donnerons des exemples et nous montrerons, au lemme 3.1.5, qu'un système décrit avec ce formalisme est effectivement un LMP*.

Définition 3.1.1 *Soit $\mathcal{S} = (S, i, \mathsf{Act}, \mathtt{Atom}, label, \rightarrow)$, un système de transition probabiliste, où $S \in \Sigma$ est l'ensemble des états, $i \in S$ est l'état initial, Act est l'ensemble des actions, $\mathtt{Atom}$ est l'ensemble dénombrable des étiquettes sur les états et $label : \mathtt{Atom} \rightarrow \Sigma$ est la fonction retournant l'ensemble des états qui satisfont une étiquette. On définit la fonction de transition $\rightarrow$ de la manière suivante :*

$$\rightarrow = \sum_{\substack{X \in \boldsymbol{X} \\ Y \in \boldsymbol{Y} \\ \mathsf{a} \in \mathsf{Act}}} f_{X \xrightarrow{\mathsf{a}} Y}(x, y)$$

Où :

- *$\boldsymbol{X}$ et $\boldsymbol{Y}$ sont des partitions de S contenant seulement des intervalles et des ensembles finis, et*
- *$f_{X \xrightarrow{\mathsf{a}} Y} : S \times S \rightarrow [0, 1]$ est une fonction mesurable sur sa première variable et intégrable sur sa deuxième. De plus, elle doit satisfaire cette contrainte : $\forall x \in S$ et $\forall \mathsf{a} \in \mathsf{Act}$*

$$0 \leq \sum_{Y \in \overline{\boldsymbol{Y}^{\boldsymbol{d}}}} \int_S f_{X \xrightarrow{\mathsf{a}} Y}(x, y)\, \mathrm{d}y + \sum_{Y \in \boldsymbol{Y}^{\boldsymbol{d}}} \sum_{y \in Y} f_{X \xrightarrow{\mathsf{a}} Y}(x, y) \ \leq 1 \tag{3.1}$$

où X est l'unique ensemble de $\boldsymbol{X}$ contenant x, $\boldsymbol{Y}^{\boldsymbol{d}}$ est l'ensemble des éléments dénombrables de $\boldsymbol{Y}$ et $\overline{\boldsymbol{Y}^{\boldsymbol{d}}} = \boldsymbol{Y} \setminus \boldsymbol{Y}^{\boldsymbol{d}}$.

Les ensembles dans $\mathbf{X} \subseteq \Sigma$ sont interprétés comme des ensembles d'états de départ pour les transitions, alors que les ensembles dans $\mathbf{Y}$ sont des ensembles d'états d'arrivée. Si $(x, y) \notin X \times \overset{\circ}{Y}$ [1], $f_{X \xrightarrow{\mathsf{a}} Y}(x, y) = 0$. Si $Y \in \overline{\mathbf{Y^d}}$, $f_{X \xrightarrow{\mathsf{a}} Y}(x, y)$ est une fonction de densité qui donne le poids de la transition $x \xrightarrow{\mathsf{a}} y$, pour $x \in X$ et $y \in Y$. Si $Y \in \mathbf{Y^d}$, $f_{X \xrightarrow{\mathsf{a}} Y}(x, y)$ représente la probabilité d'aller vers l'état $y \in Y$ à partir de l'état $x \in X$. La contrainte (3.1) nous assure que lorsqu'on additionne toutes ces fonctions, nous avons une valeur entre 0 et 1.

Remarque 3.1.2 *Notons que nous imposons que les ensembles Y soient disjoints parce qu'il y a ambiguïté lorsque deux fonctions définissent les probabilités d'aller d'un même état de départ vers un ensemble d'états d'arrivée. Par exemple, supposons qu'il y a deux fonctions de densité, $f_{X \xrightarrow{\mathsf{a}} Y}(x, y)$ et $f_{X \xrightarrow{\mathsf{a}} Y'}(x, y)$ où $Y \cap Y' \neq \emptyset$, qui définissent toutes deux le poids des transitions entre X et $Y \cap Y'$. Cette situation est ambiguë, puisqu'il y a plusieurs interprétations possibles : on pourrait prendre le minimum en chacun des points des deux fonctions sur $Y \cap Y'$ ou bien on pourrait prendre la somme des deux fonctions. Il est clair que ces deux visions amèneraient des probabilités très différentes. Alors, nous préférons éliminer toute ambiguïté en imposant que les ensembles soient disjoints.*

Montrons ici un exemple d'une transition représentée dans le formalisme décrit plus haut.

Exemple 3.1.3 *Supposons que les ensembles de départ et d'arrivée sont respectivement $[1, 2]$ et $(4, 6]$ et que l'action est a. La fonction de densité pourrait être :*

$$f_{[1,2] \xrightarrow{\mathsf{a}} (4,6]}(x, y) = \left[(x - 1) \left(\frac{y - 4}{2} \right) \right]_{[1,2] \xrightarrow{\mathsf{a}} (4,6]}.$$

Pour ce qui est de la notation, nous laissons en indice les ensembles de départ et d'arrivée ainsi que l'action pour mettre en évidence les valeur de (x, y) où la fonction n'est pas identiquement nulle. La fonction de probabilité ici dépend de l'état de départ x ainsi que de l'état d'arrivée y. Pour ce qui est de la fonction de probabilité comme telle, le lecteur averti aura reconnu une loi uniforme pour ce qui est des états d'arrivée, pondérée par une fonction croissante en x. Le lecteur trouvera à l'annexe A un court rappel sur les probabilités. Le sens d'une telle fonction pourrait être que tous les états d'arrivée sont équiprobables, mais la transition a plus de chance d'être effectuée si l'état de départ x est proche de 2. Rappelons brièvement que la fonction de probabilité vaut zéro si $x \notin [1, 2]$ ou si $y \notin [4, 6]$.

[1] La fermeture d'un ensemble Y, notée $\overset{\circ}{Y}$, est l'intersection de tous les ensembles fermés contenant Y. Dans notre cas, la fermeture d'un intervalle réel dont les bornes sont a et b est toujours $[a, b]$.

Sémantique

À partir de cette notation, nous pouvons calculer la probabilité $P_{\mathsf{a}}(x, E)$, qui représente la probabilité qu'on puisse aller d'un état $x \in S$ vers un état de l'ensemble $E \in \Sigma$, en posant l'action a. Cette probabilité nous permettra de dégager le comportement d'un LMP* décrit dans le langage défini précédemment. Nous verrons plus loin que le calcul des probabilités joue un rôle important pour déterminer si une formule de logique est satisfaite par un système probabiliste continu.

D'après la définition 3.1.1, cette probabilité se calcule de la manière suivante, pour $x \in S$ et $E \in \Sigma$:

$$P_{\mathsf{a}}(x, E) = \sum_{\substack{X \in \mathbf{X} \\ Y \in \mathbf{Y^d}}} \int_E f_{X \xrightarrow{a} Y}(x, y)\, \mathrm{d}y + \sum_{\substack{X \in \mathbf{X} \\ Y \in \mathbf{Y^d}}} \sum_{y \in E} f_{X \xrightarrow{a} Y}(x, y)\ . \tag{3.2}$$

Pour calculer la probabilité $P_{\mathsf{a}}(x, E)$, on doit s'y prendre différemment pour les ensembles dénombrables et non-dénombrables. En effet, on sépare la manière de calculer les probabilités puisque les ensembles d'arrivée sont tous disjoints et que :

$$P\{x \in A \cup B\} = P\{x \in A\} + P\{x \in B\} - P\{x \in A \cap B\}\ .$$

Donc, on additionne les intégrales sur E pour toutes les fonctions dont l'ensemble d'arrivée est non-dénombrable. En effet, d'après les notions de base en probabilités énoncées à l'annexe A, la probabilité qu'une variable aléatoire continue Z de densité $g(z)$ prenne une valeur dans l'ensemble $E \in \Sigma$ se calcule par

$$\int_E g(z)\mathrm{d}z\ .$$

Pour une variable aléatoire discrète Z dont la loi de probabilité est $h(z)$, la probabilité $P\{Z \in E\}$ se calcule par la sommation

$$\sum_{z \in E} h(z)\ .$$

Pour les fonctions dont l'ensemble d'arrivée est dénombrable, on fait la somme pour $y \in E$, car $f_{X \xrightarrow{\mathsf{a}} Y}(x, y)$ donne directement la probabilité d'aller vers l'état y, étant donné que l'état de départ est x et que l'action est a.

La formule (3.2) se simplifie beaucoup, puisque'elle contient un très grand nombre de termes nuls. De même, la sommation sur $y \in E$ est peut être infinie, mais elle se ramène à une somme finie. Nous allons démontrer un petit résultat qui facilitera le calcul de la probabilité précédente, en utilisant le fait que les éléments de $\mathbf{X}$ sont disjoints.

Lemme 3.1.4 *Pour $x \in S$ et $E \subseteq S$,*

$$P_{\mathsf{a}}(x, E) = \sum_{Y \in \overline{\boldsymbol{Y}^{\boldsymbol{d}}}} \int_{Y \cap E} f_{X \xrightarrow{\mathsf{a}} Y}(x, y)\, \mathrm{d}y + \sum_{Y \in \boldsymbol{Y}^{\boldsymbol{d}}} \sum_{y \in E \cap Y} f_{X \xrightarrow{\mathsf{a}} Y}(x, y) \tag{3.3}$$

où X est l'unique élément de $\boldsymbol{X}$ qui contient x.

Démonstration : Tout d'abord, notons que si $x \notin \cup \mathbf{X}$, c'est qu'il est impossible de faire une transition à partir de x et donc que la probabilité est 0. Dans cette situation, $f_{X \xrightarrow{\mathsf{a}} Y}(x, y) = 0$, $\forall X \in \mathbf{X}$ et $\forall Y \in \mathbf{Y}$ par définition et le lemme est démontré. Par contre, si $x \in \mathbf{X}$, c'est qu'il existe un et un seul $X \in \mathbf{X}$ tel que $x \in X$, puisque les ensembles sont disjoints. De son côté, y prend des valeurs dans E, d'après l'intégrale et la sommation de la formule (3.2). Aussi, l'on remarque que $f_{X' \xrightarrow{\mathsf{a}} Y}(x, y) = 0$, pour $X' \neq X$ ou pour Y tel que $Y \cap E = \emptyset$. Ainsi, d'après la formule (3.2), nous pouvons conclure que

$$\begin{aligned}
P_{\mathsf{a}}(x, E) &= \sum_{\substack{X \in \mathbf{X} \\ Y \in \overline{\mathbf{Y}^{\mathbf{d}}}}} \int_E f_{X \xrightarrow{\mathsf{a}} Y}(x, y)\, \mathrm{d}y + \sum_{\substack{X \in \mathbf{X} \\ Y \in \mathbf{Y}^{\mathbf{d}}}} \sum_{y \in E} f_{X \xrightarrow{\mathsf{a}} Y}(x, y) \\
&= \sum_{Y \in \overline{\mathbf{Y}^{\mathbf{d}}}} \int_E f_{X \xrightarrow{\mathsf{a}} Y}(x, y)\, \mathrm{d}y + \sum_{Y \in \mathbf{Y}^{\mathbf{d}}} \sum_{y \in E} f_{X \xrightarrow{\mathsf{a}} Y}(x, y) \\
&= \sum_{\substack{Y \in \overline{\mathbf{Y}^{\mathbf{d}}} \\ Y \cap E \neq \emptyset}} \int_E f_{X \xrightarrow{\mathsf{a}} Y}(x, y)\, \mathrm{d}y + \sum_{Y \in \mathbf{Y}^{\mathbf{d}}} \sum_{y \in E \cap Y} f_{X \xrightarrow{\mathsf{a}} Y}(x, y)
\end{aligned}$$

■

À la lumière de la formule (3.3), nous voyons que la contrainte (3.1) dans la définition du langage impose en fait que la probabilité $P_{\mathsf{a}}(x, S)$ soit comprise entre 0 et 1, pour chacun des états $x \in S$. Une probabilité plus petite que 1 représente une situation où il est possible de rester à un état même si l'on pose une action. En fait, cela correspond au cas où le système ne répond pas à l'action demandée, mais pas au cas où le système revient à l'état x après avoir *fait* l'action a. De même, si un état $x \in S$ est tel que $P_{\mathsf{a}}(x, S) = 0$, c'est qu'il n'est pas possible de faire l'action a à partir de x.

Dans l'implémentation du vérificateur formel, l'utilisateur n'a pas à vérifier cette contrainte a priori, lorsqu'il fournit un système de transition probabiliste. Le programme est en mesure de la vérifier pour avertir l'utilisateur que les fonctions de transition donnent une probabilité plus grande que 1 ou plus petite que 0, pour certains états du système.

Afin de lier le langage défini plus haut aux LMP*, nous allons démontrer que la formule (3.3) respecte la condition sur μ_{a} donnée à la définition 2.1.5 .

Lemme 3.1.5 *Soit $\mathcal{S} = (S, i, \mathsf{Act}, \mathtt{Atom}, label, \rightarrow)$, tel que donné à la définition 3.1.1. Alors,*

1. *Si on fixe $x \in S$, $P_{\mathsf{a}}(x, \cdot)$ est une mesure de probabilité partielle, $\forall \mathsf{a} \in \mathsf{Act}$;*
2. *Si $E \in \Sigma$, $P_{\mathsf{a}}(\cdot, E)$ est une fonction mesurable, $\forall \mathsf{a} \in \mathsf{Act}$;*
3. *$\mathcal{S}$ est un LMP*.*

Démonstration Soit $\mathcal{S} = (S, i, \mathsf{Act}, \mathtt{Atom}, label, \rightarrow)$, un système de transitions probabiliste, donné d'après la définition 3.1.1. Nous démontrerons le lemme en trois parties. Pour la première, fixons $x \in S$ et montrons que $P_{\mathsf{a}}(x, \cdot)$ est une mesure,$\forall \mathsf{a} \in \mathsf{Act}$, en montrant que $P_{\mathsf{a}}(x, \emptyset) = 0$ et que $P_{\mathsf{a}}(x, \bigcup_{i\in I} E_i) = \sum_{i\in I} P_{\mathsf{a}}(x, E_i)$, où $\bigcup_{i\in I} E_i$ est une union dénombrable d'ensembles disjoints.

D'après la formule du lemme 3.1.4,

$$P_{\mathsf{a}}(x, \emptyset) = \sum_{Y \in \overline{\mathbf{Y}^{\mathbf{d}}}} \int_{\emptyset} f_{X \xrightarrow{\mathsf{a}} Y}(x, y)\, \mathrm{d}y + \sum_{Y \in \mathbf{Y}^{\mathbf{d}}} \sum_{y \in \emptyset} f_{X \xrightarrow{\mathsf{a}} Y}(x, y) \ .$$

Comme $f_{X \xrightarrow{\mathsf{a}} Y}(x, y)$ est une fonction mesurable en x et intégrable en y, $P_{\mathsf{a}}(x, \emptyset) = 0$.

Soit $\bigcup_{i\in I} E_i$, une union dénombrable d'ensembles disjoints, où $E_i \in \Sigma$. Nous allons démontrer que $P_{\mathsf{a}}(x, \bigcup_{i\in I} E_i) = \sum_{i\in I} P_{\mathsf{a}}(x, E_i)$. Tout d'abord, nous remarquons que

$$Y \cap (\bigcup_{i\in I} E_i) = \bigcup_{i\in I} (Y \cap E_i)$$

pour $Y \in \mathbf{Y}$. De plus, l'intégrale d'une fonction mesurable sur une union dénombrable d'ensembles disjoints est égale à la somme des intégrales sur les ensembles disjoints. Par conséquent,

$$\begin{aligned} P_{\mathsf{a}}(x, \bigcup_{i\in I} E_i) &= \sum_{Y \in \overline{\mathbf{Y}^{\mathbf{d}}}} \sum_{i\in I} \int_{Y \cap E_i} f_{X \xrightarrow{\mathsf{a}} Y}(x, y)\, \mathrm{d}y + \sum_{Y \in \mathbf{Y}^{\mathbf{d}}} \sum_{i\in I} \sum_{y \in Y \cap E_i} f_{X \xrightarrow{\mathsf{a}} Y}(x, y) \\ &= \sum_{i\in I} \left[\sum_{Y \in \overline{\mathbf{Y}^{\mathbf{d}}}} \int_{Y \cap E_i} f_{X \xrightarrow{\mathsf{a}} Y}(x, y)\, \mathrm{d}y + \sum_{Y \in \mathbf{Y}^{\mathbf{d}}} \sum_{y \in Y \cap E_i} f_{X \xrightarrow{\mathsf{a}} Y}(x, y) \right] \\ &= \sum_{i\in I} P_{\mathsf{a}}(x, E_i) \ . \end{aligned}$$

Pour conclure que $P_{\mathsf{a}}(x, \cdot)$ est une mesure de probabilité partielle, il ne reste qu'à montrer que $0 \leq P_{\mathsf{a}}(x, \cdot) < 1$. Or, par la contrainte 3.1 de la définiton 3.1.1, cette condition est vérifiée.

Pour la deuxième partie de la démonstration, montrons que $P_{\mathsf{a}}(\cdot, E)$ est une fonction mesurable, pour $\mathsf{a} \in \mathsf{Act}$ et $E \in \Sigma$. Si nous fixons $\mathsf{a} \in \mathsf{Act}$ et $E \in \Sigma$, nous remarquons que $P_{\mathsf{a}}(x, E)$ se simplifie, puisque nous pouvons calculer les intégrales ainsi que les sommations sur $y \in E \cap Y$. Le résultat de cette opération est une sommation d'expressions en x, lesquelles proviennent de $f_{X \xrightarrow{\mathsf{a}} Y}(x, y)$. Comme la somme de fonctions mesurables est une fonction mesurable et que $f_{X \xrightarrow{\mathsf{a}} Y}(x, y)$ est une fonction mesurable en ses deux coordonnées, alors $P_{\mathsf{a}}(x, E)$ est mesurable.

Les deux premières parties de cette démonstration impliquent que $P_{\mathsf{a}}(x, E)$ est une fonction de transitions probabiliste partielle, tel que spécifé à la définition 2.1.3. $\mathcal{S}$ est donc un LMP*, ce qui démontre la troisième partie du lemme.

■

Ainsi, un système décrit avec le langage défini précédemment est un LMP*.

Quelques exemples

Maintenant, nous allons revenir sur l'exemple 2.1.7 et réexprimer ce système dans le langage introduit plus haut.

Exemple 3.1.6 *Rappelons que $S = [0, 3] \cup \{4, 5\}$, l'état initial est 1 et les actions possibles sont $\mathsf{Act} = \{\mathsf{a}, \mathsf{b}\}$.*

Les ensembles de départ et d'arrivée X et Y sont, d'après l'exemple 2.1.7,

$$\begin{aligned} \boldsymbol{X} &= \{\, [0,1],\ (1,2],\ (2,3] \,\} \\ \boldsymbol{Y^d} &= \{\, \{0\},\ \{1\},\ \{4\},\ \{5\} \,\} \\ \overline{\boldsymbol{Y^d}} &= \{\, (0,1) \cup (1,2],\ (2,3] \,\}\ . \end{aligned}$$

Les fonctions de transitions sont :

$$\begin{aligned}
\to &= \sum_{\substack{X \in \boldsymbol{X} \\ Y \in \boldsymbol{Y} \\ \mathsf{a} \in \mathsf{Act}}} f_{X \xrightarrow{\mathsf{a}} Y}(x,y) \\
&= f_{[0,1] \xrightarrow{\mathsf{a}} \{0\}}(x,y) + f_{[0,1] \xrightarrow{\mathsf{a}} (0,1)\cup(1,2]}(x,y) + f_{[0,1] \xrightarrow{\mathsf{a}} \{1\}}(x,y) + f_{[0,1] \xrightarrow{\mathsf{a}} (2,3]}(x,y) \\
&\quad + f_{(1,2] \xrightarrow{\mathsf{a}} \{4\}}(x,y) + f_{(2,3] \xrightarrow{\mathsf{b}} \{5\}}(x,y) \\
&= \left[\frac{x}{4}\right]_{[0,1] \xrightarrow{\mathsf{a}} \{0\}} + \left[\frac{1}{4}\right]_{[0,1] \xrightarrow{\mathsf{a}} (0,1)\cup(1,2]} + \left[\frac{1-x}{4}\right]_{[0,1] \xrightarrow{\mathsf{a}} \{1\}} + \left[\frac{x}{4}\right]_{[0,1] \xrightarrow{\mathsf{a}} (2,3]} \\
&\quad + \left[1\right]_{(1,2] \xrightarrow{\mathsf{a}} \{4\}} + \left[1\right]_{(2,3] \xrightarrow{\mathsf{b}} \{5\}} .
\end{aligned}$$

Les ensembles dans $\boldsymbol{X}$ se trouvent en prenant les trois intervalles de départ, tel qu'énoncé dans l'exemple. Les ensembles dans $\overline{\boldsymbol{Y^d}}$ sont pris d'après les ensembles d'arrivée, en s'assurant que les points qui possèdent une probabilité > 0 soient dans un ensemble d'arrivée dénombrable. Par exemple, nous avons calculé que $P_{\mathsf{a}}(x, \{0\})$ vaut $\frac{x}{4}$. Alors, il faut placer l'état 0 dans un singleton de $\boldsymbol{Y^d}$.

Voyons comment l'on détermine les fonctions $f_{X \xrightarrow{\mathsf{a}} Y}(x,y)$ pour $Y \in \boldsymbol{Y^d}$ et $a \in A$. Comme les ensembles d'arrivée sont dénombrables, on interprète ces fonctions comme des probabilités. $f_{[0,1] \xrightarrow{\mathsf{a}} \{0\}}(x,y)$ est égal à $P_{\mathsf{a}}(x, \{0\}) = \frac{x}{4}$. Pour $f_{[0,1] \xrightarrow{\mathsf{a}} \{1\}}(x,y)$, la probabilité $P_{\mathsf{a}}(x, \{1\})$ pour $x \in [0,1]$ est donnée dans l'énoncé de l'exemple 2.1.7. De même, $f_{(1,2] \xrightarrow{\mathsf{a}} \{4\}}(x,y)$ et $f_{(2,3] \xrightarrow{\mathsf{b}} \{5\}}(x,y)$ valent uniformément 1.

Maintenant, voyons comment déterminer les fonctions $f_{X \xrightarrow{\mathsf{a}} Y}(x,y)$ pour $Y \in \overline{\boldsymbol{Y^d}}$ et $\mathsf{a} \in \mathsf{Act}$. Nous avons les probabilités sous forme de fonctions dépendant de l'état de départ et de l'ensemble d'arrivée. Alors, il faut calculer les dérivées des fonctions de probabilité pour retrouver les fonctions de densité.

Pour $f_{[0,1] \xrightarrow{\mathsf{a}} (0,1)}(x,y)$, nous avons besoin de la fonction de densité dont la répartition est $\frac{y}{4}$. En prenant la dérivée par rapport à y, nous retrouvons :

$$f_{[0,1] \xrightarrow{\mathsf{a}} (0,1)}(x,y) = \frac{\mathrm{d}}{\mathrm{d}y} \frac{y}{4} = \frac{1}{4} .$$

Ainsi, nous pouvons retrouver la probabilité $P_{\mathsf{a}}(x, (0,z))$ pour $x \in [0,1]$ et $z \in [0,1]$ en

calculant l'intégrale de $f_{[0,1]\xrightarrow{\mathsf{a}}(0,1)}(x,y)$ *pour* y *entre* 0 *et* z*, à l'aide de la formule (3.3) :*

$$\begin{aligned} P_{\mathsf{a}}(x,(0,z)) &= \sum_{\substack{Y\in\overline{\boldsymbol{Y}^{\boldsymbol{d}}}\\ Y\cap(0,z)\neq\emptyset}} \int_0^z f_{X\xrightarrow{\mathsf{a}}Y}(x,y)\,\mathrm{d}y + \sum_{Y\in\boldsymbol{Y}^{\boldsymbol{d}}}\sum_{y\in Y\cap(0,z)} f_{X\xrightarrow{\mathsf{a}}Y}(x,y) \\ &= \int_0^z f_{[0,1]\xrightarrow{\mathsf{a}}(0,1)}(x,y)\,\mathrm{d}y + 0 \\ &= \int_0^z \frac{1}{4}\,\mathrm{d}y = \frac{z}{4}\ . \end{aligned}$$

$f_{[0,1]\xrightarrow{\mathsf{a}}(1,2]}(x,y)$ *se calcule de la même manière que* $f_{[0,1]\xrightarrow{\mathsf{a}}(0,1)}(x,y)$*. Par contre, il faut remplacer* $z\in[0,1]$ *par* $y\in[1,2]$*. On obtient*

$$f_{[0,1]\xrightarrow{\mathsf{a}}(1,2]}(x,y) = \frac{\mathrm{d}}{\mathrm{d}y}\,\frac{y-1}{4} = \frac{1}{4}\ .$$

Comme $f_{[0,1]\xrightarrow{\mathsf{a}}(0,1)}(x,y)$ *et* $f_{[0,1]\xrightarrow{\mathsf{a}}(1,2]}(x,y)$ *ont le même ensemble de départ, la même action et la même fonction de densité, nous les regroupons en une fonction :*

$$f_{[0,1]\xrightarrow{\mathsf{a}}(0,1)\cup(1,2]}(x,y) = \frac{1}{4}\ .$$

$f_{[0,1]\xrightarrow{\mathsf{a}}(2,3]}(x,y)$ *se calcule aussi de la même façon, à la différence que la constante multiplicative dépend de* x*. Pour adapter la présentation de l'exemple 2.1.7 au formalisme de notre langage, réécrivons la probabilité en* $y\in[2,3]$*, plutôt qu'en* $z\in[0,1]$*. Cela donne* $\frac{x(y-2)}{4}$ *Alors,*

$$f_{[0,1]\xrightarrow{\mathsf{a}}(2,3]}(x,y) = \frac{\mathrm{d}}{\mathrm{d}y}\frac{x(y-2)}{4} = \frac{x}{4}\ .$$

Maintenant, nous allons présenter un système plus compliqué. Entre autres, ce système possède des fonctions de densité normale et exponentielle.

Exemple 3.1.7 *L'ensemble des états du système est* $S=\{0\}\cup[1,3]\cup[10,15]$*. L'état initial du système est* 0*. Les ensembles de départ et d'arrivée sont les suivants :*

$$\begin{aligned} \boldsymbol{X} &= \{\{0\},[1,1.5]\} \\ \boldsymbol{Y} &= \overline{\boldsymbol{Y}^{\boldsymbol{d}}} = \{[1,2],(2,3],[10,15]\}\ . \end{aligned}$$

Les fonctions de transitions sont

$$\begin{aligned}
\to &= \sum_{\substack{X\in \boldsymbol{X}\\ Y\in \boldsymbol{Y}\\ \mathsf{a}\in\mathsf{Act}}} f_{X\xrightarrow{\mathsf{a}}Y}(x,y)\\
&= \left[\frac{2-y}{8}\right]_{\{0\}\xrightarrow{\mathsf{a}}[1,2]} + \left[\frac{y-2}{8}\right]_{\{0\}\xrightarrow{\mathsf{a}}(2,3]}\\
&\quad + \left[\frac{1}{4}\left(\frac{1}{\sqrt{\pi}}e^{-(y-12.5)^2}\right)\right]_{\{0\}\xrightarrow{\mathsf{a}}[10,15]} + \left[\frac{x}{8}\left(\frac{4}{5}\,e^{-4/5\,y}\right)\right]_{[1,1.5]\xrightarrow{\mathsf{a}}[1,2]}\\
&\quad + \left[\frac{xy}{16}\right]_{[1,1.5]\xrightarrow{\mathsf{a}}(2,3]} + \left[\frac{3x}{8}\left(\frac{1}{14}\,e^{-y/14}\right)\right]_{[1,1.5]\xrightarrow{\mathsf{a}}[10,15]}.
\end{aligned}$$

L'on remarque que la fonction $f_{\{0\}\xrightarrow{\mathsf{a}}[10,15]}$ *est une fonction de densité normale multipliée par* $1/4$*, dont la moyenne est* $\mu = 12.5$ *et la variance est* $\sigma^2 = 0.5$*. Aussi,* $f_{[1,1.5]\xrightarrow{\mathsf{a}}[1,2]}$ *est une fonction de densité exponentielle multipliée par* $x/8$*, de paramètre* $\lambda = 5/4$*.*

Afin de nous assurer que ce système est bien formé, nous allons vérifier la contrainte (3.1) de la définition 3.1.1. La vérification peut se faire en prenant les fonctions par ensembles de départ. Commençons avec $X = \{0\}$*.*

$$\begin{aligned}
P_{\mathsf{a}}(0,S) &= \sum_{Y\in\overline{\boldsymbol{Y}^{\boldsymbol{d}}}}\int_S f_{\{0\}\xrightarrow{a}Y}(x,y)\,\mathrm{d}y + \sum_{Y\in\boldsymbol{Y}^{\boldsymbol{d}}}\sum_{y\in Y} f_{\{0\}\xrightarrow{a}Y}(x,y)\\
&= \int_1^2 \frac{2-y}{8}\mathrm{d}y + \int_2^3 \frac{y-2}{8}\mathrm{d}y + \int_{10}^{15}\frac{1}{4}\left(\frac{1}{\sqrt{\pi}}e^{-(y-12.5)^2}\right)\mathrm{d}y\\
&\approx 0.37
\end{aligned}$$

Comme la probabilité $P_{\mathsf{a}}(0,S)$ *est comprise entre 0 et 1, la contrainte est vérifiée pour cet ensemble.*

Fixons x *dans* $[1,1.5]$ *et calculons* $P_{\mathsf{a}}(x,S)$ *:*

$$\begin{aligned}
P_{\mathsf{a}}(x,S) &= \int_1^2 \frac{x}{8}\left(\frac{4}{5}\,e^{-4/5\,y}\right)\mathrm{d}y + \int_2^3 \frac{xy}{16}\mathrm{d}y + \int_{10}^{15}\frac{3x}{8}\left(\frac{1}{14}\,e^{-y/14}\right)\mathrm{d}y\\
&= -\frac{x}{8}(e^{-8/5}-e^{-4/5}) + \frac{5x}{32} - \frac{3x}{8}(e^{-15/14}-e^{-10/14})\,.
\end{aligned}$$

Cette probabilité est linéaire en x *et est comprise dans l'intervalle* $[0.24, 0.37]$ *pour* $x\in[1,1.5]$*. Ainsi, la contrainte est aussi vérifiée pour cet ensemble.*

Pour montrer le comportement de la formule (3.3) lorsque E *chevauche plusieurs* Y*, nous allons calculer la probabilité* $P_{\mathsf{a}}(x,[1.5,2.5])$*, pour* $x\in\cup\boldsymbol{Y}$*. Commençons par*

$x \in X = \{0\}$

$$\begin{aligned} P_{\mathsf{a}}(0, [1.5, 2.5]) &= \sum_{\substack{Y \in \overline{\boldsymbol{Y}^{\boldsymbol{d}}} \\ Y \cap [1.5,2.5] \neq \emptyset}} \int_{Y \cap [1.5,2.5]} f_{\{0\} \xrightarrow{\mathsf{a}} Y}(x, y)\, \mathrm{d}y \\ &\quad + \sum_{Y \in \boldsymbol{Y}^{\boldsymbol{d}}} \sum_{y \in [1.5,2.5] \cap Y} f_{\{0\} \xrightarrow{\mathsf{a}} Y}(x, y) \\ &= \int_{1.5}^{2} f_{\{0\} \xrightarrow{\mathsf{a}} [1,2]}(x, y)\, \mathrm{d}y + \int_{2}^{2.5} f_{\{0\} \xrightarrow{\mathsf{a}} (2,3]}(x, y)\, \mathrm{d}y + 0 \\ &= \int_{1.5}^{2} \frac{2 - y}{8}\, \mathrm{d}y + \int_{2}^{2.5} \frac{y - 2}{8}\, \mathrm{d}y \\ &= \frac{1}{32}\,. \end{aligned}$$

Calculons la même probabilité pour $x \in X = [1, 1.5]$.

$$\begin{aligned} P_{\mathsf{a}}(x, [1.5, 2.5]) &= \int_{1.5}^{2} f_{[1,1.5] \xrightarrow{\mathsf{a}} [1,2]}(x, y)\, \mathrm{d}y + \int_{2}^{2.5} f_{[1,1.5] \xrightarrow{\mathsf{a}} (2,3]}(x, y)\, \mathrm{d}y + 0 \\ &= \int_{1.5}^{2} \frac{x}{10} e^{-4y/5}\, \mathrm{d}y + \int_{2}^{2.5} \frac{xy}{16}\, \mathrm{d}y \\ &= \frac{x}{128} \left(-16\, e^{-8/5} + 16\, e^{-6/5} + 9\right) \\ &\approx 0.083\, x \end{aligned}$$

3.1.2 Langage basé sur les fonctions de répartition

Le langage présenté dans la sous-section précédente utilise des fonctions de densité pour définir les probabilités des transitions. Il y a une autre manière qui consiste à utiliser des fonctions de répartition. L'impact d'un tel changement se retrouve principalement dans le calcul des probabilités : on remplace chaque intégrale d'une fonction de densité par la différence de deux évaluations d'une fonction de répartition. L'avantage est intéressant puisque qu'une intégrale est généralement plus longue à calculer, numériquement ou symboliquement, que deux évaluations d'une fonction. La sous-section suivante présente une comparaison des deux langages, afin de choisir lequel servira pour l'implémentation du vérificateur formel.

La définition 3.1.1 revisitée

Dans la définition 3.1.1, les fonctions sur des ensembles non-dénombrables sont des fonctions de densité. Dans cette section, nous tenterons d'illustrer des systèmes en

utilisant plutôt des fonctions de répartition. Pour faire le lien avec les fonctions de densité, rappelons que la fonction de répartition F d'une variable aléatoire Y de densité f est définie par

$$F(z) = P\{Y \leq z\} = \int_{-\infty}^{z} f(y)\mathrm{d}y \ .$$

Pour plus détails, le lecteur est invité à consulter l'annexe A.

Pour adapter la définition aux fonctions de répartition, nous allons simplement noter par $F_{X \xrightarrow{\mathsf{a}} Y}(x, y)$ les fonctions de transition. Si $Y \in \mathbf{Y}$, la signification de $F_{X \xrightarrow{\mathsf{a}} Y}(x, y)$ est la même que $f_{X \xrightarrow{\mathsf{a}} Y}(x, y)$ dans la définition originale. Par contre, pour $Y \in \overline{\mathbf{Y}^{\mathbf{d}}}$, $F_{X \xrightarrow{\mathsf{a}} Y}(x, y)$ représente la probabilité qu'à partir de l'état x, en faisant l'action a, le système aille vers un état plus petit que y. Autrement dit,

$$F_{X \xrightarrow{\mathsf{a}} Y}(x, y) = \begin{cases} P_{\mathsf{a}}(x, (-\infty, y)) & \text{si } (x, y) \in X \times \overset{\circ}{Y} \\ 0 & \text{sinon} \end{cases}$$

où $\overset{\circ}{Y}$ dénote la fermeture de Y.

Alors, pour calculer $P_{\mathsf{a}}(x, E)$, où $E = \dot{\cup}_{i \in I} E_i$ est une union d'intervalles disjoints, le calcul devient

$$\begin{aligned} P_{\mathsf{a}}(x, E) \ = \ & \sum_{i \in I} \left[\sum_{\substack{Y \in \overline{\mathbf{Y}^{\mathbf{d}}} \\ Y \cap E_i \neq \emptyset}} \left[F_{X \xrightarrow{\mathsf{a}} Y}(x, \sup(Y \cap E_i)) - F_{X \xrightarrow{\mathsf{a}} Y}(x, \inf(Y \cap E_i)) \right] \right] \\ & + \sum_{Y \in \mathbf{Y}^{\mathbf{d}}} \sum_{y \in E \cap Y} F_{X \xrightarrow{\mathsf{a}} Y}(x, y) \ . \end{aligned} \tag{3.4}$$

Pour appliquer la formule précédente telle quelle, les $Y \in \overline{\mathbf{Y}^{\mathbf{d}}}$ doivent être des intervalles plutôt que des unions d'intervalles. De là vient la restriction sur les Y dans la définition 3.1.1.

Avec les fonctions de répartition, la contrainte (3.1) devient

$\forall x \in S$ et $\forall \mathsf{a} \in \mathsf{Act}$

$$\begin{aligned} 0 \ \leq \ & \sum_{i \in I} \sum_{\substack{Y \in \overline{\mathbf{Y}^{\mathbf{d}}} \\ Y \cap S_i \neq \emptyset}} \left[F_{X \xrightarrow{\mathsf{a}} Y}(x, \sup(Y \cap S_i)) - F_{X \xrightarrow{\mathsf{a}} Y}(x, \inf(Y \cap S_i)) \right] \\ & + \sum_{Y \in \mathbf{Y}^{\mathbf{d}}} \sum_{y \in S \cap Y} F_{X \xrightarrow{\mathsf{a}} Y}(x, y) \leq 1 \end{aligned} \tag{3.5}$$

qui est simplement

$\forall x \in S$ et $\forall \mathsf{a} \in \mathsf{Act}$,

$$0 \leq P_{\mathsf{a}}(x, S) \leq 1 \ .$$

Il est possible d'obtenir un résultat similaire au lemme 3.1.5, pour conclure qu'un système décrit avec des fonctions de répartition est aussi un LMP*.

Lemme 3.1.8 *Soit* $\mathcal{S} = (S, i, \mathsf{Act}, \mathtt{Atom}, label, \rightarrow)$*, un système probabiliste continu, décrit avec des fonctions de répartition. Alors,* $\mathcal{S}$ *est un LMP*.*

La démonstration de ce résultat est similaire à celle du lemme 3.1.5.

Quelques exemples revisités

Dans les prochains exemples, nous allons transformer les fonctions de densité en fonctions de répartition. Nous allons montrer ainsi l'approche à suivre pour effectuer cette traduction. Le lecteur trouvera à la section A.2 quelques lois continues, avec leur fonction de répartition. Le premier exemple présente les principales étapes pour transformer un système décrit avec des fonctions de densité en un système exprimé à l'aide de fonctions de répartition.

Exemple 3.1.9 *Nous reprenons le système de l'exemple 3.1.6. L'ensemble des états est* $S = [0,3] \cup \{4,5\}$*. Le système est*

$$
\begin{aligned}
\mathcal{S} &= \sum_{\substack{X \in \boldsymbol{X} \\ Y \in \boldsymbol{Y} \\ \mathsf{a} \in \mathsf{Act}}} F_{X \xrightarrow{\mathsf{a}} Y}(x,y) \\
&= F_{[0,1] \xrightarrow{\mathsf{a}} \{0\}}(x,y) + F_{[0,1] \xrightarrow{\mathsf{a}} (0,1)}(x,y) + F_{[0,1] \xrightarrow{\mathsf{a}} (1,2]}(x,y) \\
&\quad + F_{[0,1] \xrightarrow{\mathsf{a}} \{1\}}(x,y) + F_{[0,1] \xrightarrow{\mathsf{a}} (2,3]}(x,y) \\
&\quad + F_{(1,2] \xrightarrow{\mathsf{a}} \{4\}}(x,y) + F_{(2,3] \xrightarrow{\mathsf{b}} \{5\}}(x,y) \\
&= \left[\frac{x}{4}\right]_{[0,1] \xrightarrow{\mathsf{a}} \{0\}} + \left[\frac{y}{4}\right]_{[0,1] \xrightarrow{\mathsf{a}} (0,1)} + \left[\frac{y}{4}\right]_{[0,1] \xrightarrow{\mathsf{a}} (1,2]} + \left[\frac{1-x}{4}\right]_{[0,1] \xrightarrow{\mathsf{a}} \{1\}} \\
&\quad + \left[\frac{x(y-2)}{4}\right]_{[0,1] \xrightarrow{\mathsf{a}} (2,3]} + \left[1\right]_{(1,2] \xrightarrow{\mathsf{a}} \{4\}} + \left[1\right]_{(2,3] \xrightarrow{\mathsf{b}} \{5\}}.
\end{aligned}
$$

Voyons comment nous avons obtenu ce système. Tout d'abord, notons que nous avons scindé en deux l'ensemble d'états d'arrivée $(0,1) \cup (1,2]$*, pour respecter le fait que les ensembles d'arrivée doivent être des intervalles. Aussi, notons que les fonctions* $F_{X \xrightarrow{\mathsf{a}} Y}(x,y)$ *pour* $Y \in \boldsymbol{Y^d}$ *sont les mêmes que dans l'exemple 3.1.6. Dans le cas où*

$Y \in \overline{\boldsymbol{Y^d}}$, *nous devons tranformer les fonctions de densité en fonctions de répartition, en calculant l'intégrale* $\int_{-\infty}^{y} f_{X \xrightarrow{\mathsf{a}} Y}(x,z)\mathrm{d}z$.

$F_{[0,1]\xrightarrow{\mathsf{a}}(0,1)}(x,y)$ *se calcule avec l'intégrale*

$$\begin{aligned} F_{[0,1]\xrightarrow{\mathsf{a}}(0,1)}(x,y) &= \int_{-\infty}^{y} f_{[0,1]\xrightarrow{\mathsf{a}}(0,1)}(x,z)\,\mathrm{d}z \\ &= \int_{0}^{y} \frac{1}{4}\,\mathrm{d}z \\ &= \frac{y}{4} \end{aligned}$$

pour $y \in (0,1)$. *Le résultat est le même pour* $F_{[0,1]\xrightarrow{\mathsf{a}}(1,2]}(x,y)$.

On calcule $F_{[0,1]\xrightarrow{\mathsf{a}}(2,3]}(x,y)$ *comme ceci :*

$$\begin{aligned} F_{[0,1]\xrightarrow{\mathsf{a}}(2,3]}(x,y) &= \int_{-\infty}^{y} f_{[0,1]\xrightarrow{\mathsf{a}}(2,3]}(x,z)\,\mathrm{d}z \\ &= \int_{2}^{y} \frac{x}{4}\,\mathrm{d}z \\ &= \frac{x(y-2)}{4} \end{aligned}$$

pour $y \in (2,3]$.

À partir de ce système, nous allons montrer comment utiliser la formule (3.4), en calculant plusieurs probabilités. Dans ce qui suit, nous allons utiliser la notation $F_{X\xrightarrow{\mathsf{a}}Y}(x,y)\Big|_a^b = F_{X\xrightarrow{\mathsf{a}}Y}(x,b) - F_{X\xrightarrow{\mathsf{a}}Y}(x,a)$ servant pour l'intégration définie en mathématiques. Il est à noter que nous l'utilisons toujours sur la seconde variable des fonction $F_{X\xrightarrow{\mathsf{a}}Y}(x,y)$

Exemple 3.1.10 *Calculons la probabilité* $P_{\mathsf{a}}(x,[0,3])$, *pour* $x \in [0,1]$. *Par la formu-*

le (3.4), cette probabilité est

$$\begin{aligned}
P_{\mathsf{a}}(x,[0,3]) &= F_{[0,1]\xrightarrow{\mathsf{a}}(0,1)}(x,y)\Big|_0^1 \\
&\quad + F_{[0,1]\xrightarrow{\mathsf{a}}(1,2]}(x,y)\Big|_1^2 \\
&\quad + F_{[0,1]\xrightarrow{\mathsf{a}}(2,3]}(x,y)\Big|_2^3 \\
&\quad + F_{[0,1]\xrightarrow{\mathsf{a}}\{0\}}(x,0) + F_{[0,1]\xrightarrow{\mathsf{a}}\{1\}}(x,1) \\
&= \Big(F_{[0,1]\xrightarrow{\mathsf{a}}(0,1)}(x,1) - F_{[0,1]\xrightarrow{\mathsf{a}}(0,1)}(x,0)\Big) \\
&\quad + \Big(F_{[0,1]\xrightarrow{\mathsf{a}}(1,2]}(x,2) - F_{[0,1]\xrightarrow{\mathsf{a}}(1,2]}(x,1)\Big) \\
&\quad + \Big(F_{[0,1]\xrightarrow{\mathsf{a}}(2,3]}(x,3) - F_{[0,1]\xrightarrow{\mathsf{a}}(2,3]}(x,2)\Big) \\
&\quad + F_{[0,1]\xrightarrow{\mathsf{a}}\{0\}}(x,0) + F_{[0,1]\xrightarrow{\mathsf{a}}\{1\}}(x,1) \\
&= \frac{1}{2} + \frac{x}{4} + \frac{x}{4} + \frac{1-x}{4} \\
&= \frac{3+x}{4}\ .
\end{aligned}$$

Nous retrouvons le même résultat aux exemples 2.1.7 et 3.1.6.

À partir de la probabilité précédente, nous allons calculer $P_{\mathsf{a}}(x,(0,2])$*, pour* $x \in [0,1]$*, comme étant*

$$\begin{aligned}
P_{\mathsf{a}}(x,(0,2]) &= P_{\mathsf{a}}(x,[0,3]) - P_{\mathsf{a}}(x,\{0\}) - P_{\mathsf{a}}(x,(2,3]) \\
&= \frac{3+x}{4} - \frac{x}{4} - \frac{x}{4} \\
&= \frac{3-x}{4}
\end{aligned}$$

qui est exactement ce que nous avons calculé dans l'exemple original.

Reprenons l'exemple 3.1.7. L'intérêt d'un tel exemple réside dans la transformation de ses fonctions de densité en des fonctions de répartition. Comme nous l'avons déjà vu, la fonction de répartition d'une variable normale est plutôt particulière.

Exemple 3.1.11 *L'ensemble des états est* $S = \{0\} \cup [1,3] \cup [10,15]$. *Le système est :*

$$\begin{aligned}
\to &= \sum_{\substack{X \in \boldsymbol{X} \\ Y \in \boldsymbol{Y} \\ \mathsf{a} \in \mathsf{Act}}} F_{X \xrightarrow{\mathsf{a}} Y}(x,y) \\
&= \left[\frac{-y^2+2y-3}{16}\right]_{\{0\} \xrightarrow{\mathsf{a}} [1,2]} + \left[\frac{y^2-4y+4}{16}\right]_{\{0\} \xrightarrow{\mathsf{a}} (2,3]} \\
&\quad + \left[\frac{1}{4}\Phi\left(2y-25\right)\right]_{\{0\} \xrightarrow{\mathsf{a}} [10,15]} + \left[\frac{-x}{8}e^{-4/5\,y}\right]_{[1,1.5] \xrightarrow{\mathsf{a}} [1,2]} \\
&\quad + \left[\frac{xy^2}{32}\right]_{[1,1.5] \xrightarrow{\mathsf{a}} (2,3]} + \left[\frac{-3x}{8}\,e^{-y/14}\right]_{[1,1.5] \xrightarrow{\mathsf{a}} [10,15]}
\end{aligned}$$

où $\Phi(z)$ *est la fonction de répartition de variable normale standard. Pour plus de détails, voir la section A.2.*

Voyons comment les fonctions de répartition ont été obtenues à partir des fonctions de densité, i.e. celles pour lesquelles $Y \in \overline{\boldsymbol{Y}^{\boldsymbol{d}}}$.

$F_{\{0\} \xrightarrow{\mathsf{a}} [1,2]}(x,y)$ *se calcule d'après l'intégrale*

$$\begin{aligned}
F_{\{0\} \xrightarrow{\mathsf{a}} [1,2]}(x,y) &= \int_{-\infty}^{y} f_{\{0\} \xrightarrow{\mathsf{a}} [1,2]}(x,z)\,\mathrm{d}z \\
&= \int_{1}^{y} \frac{2-z}{8}\,\mathrm{d}z \\
&= \frac{-y^2+4y-3}{16}
\end{aligned}$$

où $z \in [1,2]$.

$F_{\{0\} \xrightarrow{\mathsf{a}} (2,3]}(x,y)$ *est*

$$\begin{aligned}
F_{\{0\} \xrightarrow{\mathsf{a}} (2,3]}(x,y) &= \int_{-\infty}^{y} f_{\{0\} \xrightarrow{\mathsf{a}} (2,3]}(x,z)\,\mathrm{d}z \\
&+ \int_{2}^{y} \frac{z-2}{8}\,\mathrm{d}z \\
&= \frac{y^2-4y+4}{16}
\end{aligned}$$

où $y \in (2,3]$.

Pour calculer $F_{\{0\} \xrightarrow{\mathsf{a}} [10,15]}(x,y)$, *l'on se rappelle que la fonction de densité* $f_{\{0\} \xrightarrow{\mathsf{a}} [10,15]}(x,y)$ *est celle d'une variable aléatoire normale de moyenne* $\mu = 12.5$ *et de variance* $\sigma^2 = 0.5$, *multipliée par 1/4. Les variables normale ont des propriétés bien*

intéressantes. En particulier, la fonction de répartition d'une variable Y de densité normale $N(\mu, \sigma^2)$ peut être évalué à l'aide de celle d'une vairable normale standard $N(0,1)$. En fait, on ne peut que calculer approximativement une telle fonction par des méthodes numériques, puisque la fonction de densité de la loi normale n'est pas intégrable. Il existe plusieurs tables qui donnent toutes la fonction de répartition $\Phi(x)$, à des degrés de précision différents.

Donc,

$$F_{\{0\} \xrightarrow{a} [10,15]}(x,y) \;=\; \frac{1}{4}\Phi\left(\frac{y-12.5}{\sqrt{0.5}}\right) = \frac{1}{4}\Phi\left(\sqrt{2}(y-12.5)\right) \;.$$

Il est à noter que le $\frac{1}{4}$ se factorise à l'avant parce que cette expression ne dépend pas de y.

Pour $F_{[1,1.5] \xrightarrow{a} [1,2]}(x,y)$, il faut calculer l'intégrale suivante :

$$\begin{aligned} F_{[1,1.5] \xrightarrow{a} [1,2]}(x,y) &= \int_{-\infty}^{y} f_{[1,1.5] \xrightarrow{a} [1,2]}(x,z)\,\mathrm{d}z \\ &= \int_{1}^{y} \frac{x}{8}\left(\frac{4}{5}\,e^{-4/5\,z}\right)\,\mathrm{d}z \\ &= \frac{-x}{8}e^{-4/5\,y} \end{aligned}$$

pour $y \in [1,2]$.

Pour la fonction$F_{[1,1.5] \xrightarrow{a} (2,3]}(x,y)$, nous avons :

$$\begin{aligned} F_{[1,1.5] \xrightarrow{a} (2,3]}(x,y) &= \int_{-\infty}^{y} \frac{xz}{16}\,\mathrm{d}z \\ &= \frac{xy^2}{32} \end{aligned}$$

pour $y \in (2,3]$.

Finalement, pour $F_{[1,1.5] \xrightarrow{a} [10,15]}(x,y)$, nous avons :

$$\begin{aligned} F_{[1,1.5] \xrightarrow{a} [10,15]}(x,y) &= \int_{-\infty}^{y} \frac{3x}{8}\left(\frac{1}{14}\,e^{-z/14}\right)\,\mathrm{d}z \\ &= \frac{-3x}{8}\,e^{-y/14} \end{aligned}$$

pour $y \in [10,15]$.

3.1.3 Comparaison des langages

Le langage tel que nous l'avons défini au départ utilise des fonctions de densité pour modéliser le comportement aléatoire d'un système probabiliste. Dans la sous-section précédente, nous avons introduit une variante au langage, remplaçant les fonctions de densité par des fonctions de répartition. Nous allons comparer les deux options, afin de fixer un choix en vue de l'implémentation du vérificateur formel. Comme les deux langages sont les mêmes mis à part les fonctions de transitions, nous allons surtout comparer les fonctions de densité et les fonctions de répartition.

D'après l'annexe A, pour une variable aléatoire continue X, nous savons qu'il existe un lien direct entre sa fonction de densité $f(x)$ et sa fonction de répartition $F(x)$, à savoir que

$$\frac{\mathrm{d}}{\mathrm{d}x}F(x) = f(x)$$

Donc, à la base, l'expressivité des deux langages est la même, puisqu'il est possible de dériver les fonctions de répartition pour obtenir celles de densité et d'intégrer les fonctions de densité pour obtenir celles de répartition. Mais, il n'est pas toujours possible d'intégrer une fonction de densité et l'exemple classique de ceci est la loi normale (voir l'annexe A.2). En effet, l'intégrale

$$\int_E e^{t^2}\mathrm{d}t$$

ne peut pas être calculée symboliquement. En pratique, la fonction de répartition d'une variable normale est calculée *numériquement*, par des méthodes numériques ou bien en prenant des valeurs d'après des tables. Évidemment, ces manières de calculer la fonction de répartition ne sont pas aussi précises que si l'intégrale pouvait se calculer symboliquement.

Alors, on serait tenté de penser que les fonctions de densité sont plus expressives et plus précises, mais ce n'est pas le cas, puisqu'on devra recourrir à des méthodes numériques pour calculer certaines probabilités. Cela se produit précisément si la fonction de densité n'est pas intégrable, dû à la manière de calculer les probabilités. La manière de déterminer la valeur de $P\{x \in [a,b]\}$ est différente pour les deux formalismes puisque, pour une fonction de densité $f(x)$, $P\{x \in [a,b]\} = \int_a^b f(x)\,\mathrm{d}x$ alors que, pour une fonction de répartition $F(x)$, $P\{x \in [a,b]\} = F(b) - F(a)$. Dans le premier cas, il faut intégrer $f(x)$ pour ensuite l'évaluer en a et b alors que dans le second, il est suffisant d'évaluer la fonction $F(x)$ en a et b. On devra donc utiliser des méthodes numériques si l'on doit calculer des probabilités à partir d'une fonction de densité pour laquelle il n'existe pas d'équivalent symbolique à $\int_a^b f(x)\,\mathrm{d}x$,

En somme, si nous résumons les deux situations plus haut, s'il est impossible de calculer l'intégrale $\int g(x)\mathrm{d}x$, pour transformer une fonction de densité $g(x)$ en une fonction de répartition, il sera tout aussi impossible de calculer symboliquement la valeur de $P\{x \in [a,b]\}$. Donc, dans les deux cas, il faudra recourrir à des méthodes numériques, ce qui amènera des erreurs de précision. En somme, une variable aléatoire qui ne se décrit sous forme d'une fonction simple que par sa fonction de densité est en soi un cas pathologique, pour lequel on ne peut pas calculer symboliquement ni la fonction de répartition, ni les probabilités du type $P\{x \in [a,b]\}$.

Par contre, nous pensons que les fonctions de répartition ont quelques avantages sur les fonctions de densité. Nous ne nous servont des fonctions de transitions que dans le calcul de la probabilité d'effectuer une transition (La section 2.2 met bien en évidence le rôle des probabilités dans la vérification de la satisfiabilité d'une formule de logique). Comme nous l'avons fait ressortir plus haut, la formule pour calculer $P_{\mathsf{a}}(x,E)$ est plus simple dans le cas des fonctions de répartition, puisqu'on évite de calculer des intégrales à chaque fois. En effet, en passant par les fonctions de répartitions, on substitue le calcul de plusieurs intégrales au *calcul*[2] d'une seule lors de la modélisation. L'intégrale n'est calculée qu'une seule fois, en pré-traitement lors de la modélisation. Cette approche est plus simple et beaucoup plus rapide, si nous voulons l'implémenter sur un ordinateur.

De plus, il y a un autre avantage à utiliser les fonctions de répartition. Comme nous l'avons mentionné plus haut, il y a des cas où certaines fonctions de répartitions doivent être calculées par des tables. Alors, s'il y a erreur de précision inhérente à cette approche, aussi bien la diminuer le plus possible. Cela peut être fait en calculant numériquement l'intégrale $\int_{-\infty}^{x} f(t)\,\mathrm{d}t$, avec la précision voulue, puisqu'on peut régler la précision de la méthode numérique utilisée au niveau désiré. Il serait très couteux en temps de calcul de vouloir atteindre une même précision en utilisant les fonctions de densité, puisqu'alors, chaque calcul numérique de l'intégrale serait plus long. En utilisant les fonctions de répartition, on *factorise* les calculs de précision, parce que la table d'une fonction de répartion peut être aussi précise que l'on veut et, une fois calculée, une table peut reservir pour des utilisations où la précision désirée est la même ou moindre.

Alors, nous fixons notre choix sur les fonctions de répartition, dans le but de développer un vérificateur automatique.

[2]Pour la plupart des grandes lois connues, la fonction de répartition est aussi connue, évitant ainsi de calculer l'intégrale.

3.2 Un exemple tempéré

Nous allons présenter un exemple plus complexe qui se veut l'exemple central du présent travail, dans le but de démontrer l'expressivité du langage pour les LMP*. Dans ce qui suit, nous allons surtout nous attarder à interpréter un système donné sous forme de LMP*, afin de comprendre son comportement, d'après sa définition et ses fonctions de transition données sous forme de fonctions de répartition. Plus loin, ce même exemple nous servira de modèle pour interpréter et vérifier la satisfiabilité de quelques formules de logique ainsi que pour tester l'implémentation de notre vérificateur formel.

3.2.1 Le système

La situation de base pour notre exemple est un système de contrôle de la température. Les états représentent la température en Celsius d'une pièce fermée, qu'un utilisateur peut demander de chauffer ou de climatiser. À intervalles de temps fixes, disons 30 minutes, l'utilisateur peut demander que la température soit augmentée ou bien diminuée, en appuyant sur l'un des deux boutons étitquetés « a » et « d ». Chacun des appareils, le chauffage et la climatisation, peut tomber en panne, empêchant du coup l'augmentation ou la diminution de la température, respectivement. La probabilité que l'un de ces appareils tombe en panne est définie par une fonction linéraire dépendant de la température dans la pièce au moment où l'utilisateur fait sa demande. S'il appuie sur l'un des boutons, la température de la pièce après trente minutes suit une loi normale, dont la moyenne dépend de la température courante. Voici une modélisation de ce système de contrôle de la température[3] :

$$\mathcal{S} = (S, i, \mathsf{Act}, \mathtt{Atom}, \mathit{label}, \rightarrow)$$

où

- $S = [0, 40] \cup [100, 140] \cup [200, 240] \cup [300, 340]$
- $i = 15$
- $\mathsf{Act} = \{\mathsf{a}, \mathsf{d}\}$
- $\mathtt{Atom} = \{\mathtt{confort}, \mathtt{chauf\_ok}, \mathtt{clim\_ok}\}$
- $\mathit{label} = \{\mathtt{confort} \mapsto [18, 23] \cup [118, 123] \cup [218, 223] \cup [318, 323],$
  $\mathtt{chauf\_ok} \mapsto [0, 40] \cup [200, 240],\ \mathtt{clim\_ok} \mapsto [0, 40] \cup [100, 140]\}$
- $\mathbf{X} = \{[0, 20), [20, 30), [30, 40], [120, 140], [200, 230], [300, 340]\}$
- $\mathbf{Y} = \overline{\mathbf{Y}^{\mathbf{d}}} = \{[0, 40], [100, 140], [200, 240], [300, 340]\}$

[3] Le code source de cet exemple, dans le formalisme de la section 4.3.3, se trouve à la l'annexe C.

– La fonction de transition probabiliste est donnée comme suit :

$$
\begin{aligned}
\rightarrow \;=\; & \left[(x/300+0.8)\,\Phi\left(\frac{y-(x+x/10+2)}{1/2}\right)\right]_{[0,20)\cup[20,30)\xrightarrow{\mathtt{a}}[0,40]} + \\
& \left[(-x/300+0.15)\,\Phi\left(\frac{y-(x+102)}{1/2}\right)\right]_{[0,20)\cup[20,30)\xrightarrow{\mathtt{a}}[100,140]} + \\
& \left[(-x/300+0.15)\,\Phi\left(\frac{y-(x+102)}{1/2}\right)\right]_{[0,20)\cup[20,30)\xrightarrow{\mathtt{a}}[100,140]} + \\
& \left[(-3x/400+1)\,\Phi\left(\frac{y-(x+x/5-8)}{1/2}\right)\right]_{[20,30)\cup[30,40]\xrightarrow{\mathtt{d}}[0,40]} + \\
& \left[(3x/400-0.05)\,\Phi\left(\frac{y-(x+198)}{1/2}\right)\right]_{[20,30)\cup[30,40]\xrightarrow{\mathtt{d}}[200,240]} + \\
& \left[(-3x/400+1.75)\,\Phi\left(\frac{y-(1.2x-28)}{1/2}\right)\right]_{[120,140]\xrightarrow{\mathtt{d}}[100,140]} + \\
& \left[(-3x/400+1.75)\,\Phi\left(\frac{y-(1.2x-28)}{1/2}\right)\right]_{[120,140]\xrightarrow{\mathtt{d}}[100,140]} + \\
& \left[(3x/400-0.8)\,\Phi\left(\frac{y-(x+198)}{1/2}\right)\right]_{[120,140]\xrightarrow{\mathtt{d}}[300,340]} + \\
& \left[(x/300+2/15)\,\Phi\left(\frac{y-(x+x/10-18)}{1/2}\right)\right]_{[200,230]\xrightarrow{\mathtt{a}}[200,240]} + \\
& \left[(-x/300+49/60)\,\Phi\left(\frac{y-(x+102)}{1/2}\right)\right]_{[200,230]\xrightarrow{\mathtt{a}}[300,340]}
\end{aligned}
$$

Voici comment interpréter cette modélisation. La température dans la pièce peut varier entre 0 et 40 degrés Celsius. Pour refléter le fait que l'appareil de chauffage et/ou celui de climatisation peuvent être en panne, nous prenons quatre segments sur la droite réelle comme espace d'états : sur le premier segment, $[0, 40]$, les deux appareils fonctionnent, sur le second, $[100, 140]$, l'appareil de chauffage est en panne, sur le troisième, $[200, 240]$, le climatiseur est en panne et sur le dernier, $[300, 340]$, les deux sont en panne. On retrouve la température de la pièce à partir d'un état en prenant la valeur de l'état modulo 100. L'étiquette `chauf_ok` est là pour désigner les états où l'appareil de chauffage peut fonctionner et `clim_ok`, pour désigner ceux où le climatiseur peut fonctionner. Pour les besoins de l'exemble, nous estimons que que la zone de température confortable est entre 18℃et 23℃. Comme il y a plusieurs segments, la fonction *label* associe à l'étiquette `confort` l'ensemble $[18, 23] \cup [118, 123] \cup [218, 223] \cup [318, 323]$ de tous les états représentant une température dite confortable.

Les actions possibles sont `a` pour augmenter la température et `d` pour la diminuer. Si l'utilisateur appuie sur l'un des boutons, il est possible que le système réponde à la demande, mais il est aussi possible que l'appareil solicité soit endommagé par l'action,

auquel cas il tombe en panne. Si cela se produit, le système fera une transition vers un état dans un segment où cette appareil est considéré comme étant brisé.

Pour illustrer la modélisation, prenons la fonction probabiliste $F_{[0,20)\cup[20,30)\xrightarrow{\mathsf{a}}[0,40]}(x,y)$. Elle lie les états de départ dans $[0,30]$ aux états d'arrivée dans $[0,40]$ par l'action a. Ces intervalles appartiennent au premier segment décrit plus haut et, par conséquent, les deux appareils fonctionnent et fonctionneront *après* la transition. L'action a est possible seulement pour les états dans $[0,30]$ pour représenter le fait qu'on bloque la possibilité de chauffer si la température est supérieure à 30℃. La fonction de probabilité contient deux parties majeures : la première est une fonction linéraire en x et la seconde est une fonction de répartition suivant une loi normale, qui dépend de x et de y. On voit que la seconde partie est une loi normale de moyenne $x + x/10 + 2$ et d'écart-type $1/2$. Cela signifie qu'en moyenne, la température augmentera de $x/10 + 2$, puisque la température courante est x. Si la température au moment de prendre la transition est 0℃, alors elle augmentera en moyenne de 2 et si elle est 30℃, alors elle augmentera en moyenne de 5. Ce choix sert à représenter le fait qu'il est plus difficile et plus long de chauffer une pièce lorsque sa température est basse. Après cette transition, si l'utilisateur veut encore augmenter la température, il n'aura qu'à appuyer à nouveau sur a. Il est important de noter qu'au moment de prendre la transition, la valeur x est connue et c'est y qui est la variable aléatoire, ce qui nous permet de considérer x dans le calcul de la moyenne. La première partie de la fonction quant à elle sert à modéliser le fait que l'appareil de chauffage puisse tomber en panne. Ici, la fonction est linéaire, signifiant que, si la température courante est de 0℃, il y a $0/300 + 0.8 = 80\%$ de chance que l'appareil demeure intact et puisse fournir l'augmentation de température demandée. Si la température courante est 30℃, cette probabilité augmente à 90%, pour refléter le fait qu'une température plus basse demande plus d'effort et que l'appareil a plus de chance de se briser. Donc cette fonction linéaire pondère la probabilité de demeurer dans l'intervalle $[0,40]$. En effet, pour $x \in [0,30]$, $0.8 \leq P_{\mathsf{a}}(x,[0,40]) \leq 0.9$.

La figure 3.1 montre une représentation graphique de la fonction de densité associée à la fonction de répartition ci-haut. Le lecteur remarquera que le graphe de la fonction est une *vague* ou une *onde* formée des courbes normales. En effet, pour un x fixé, la fonction se comporte comme une loi normale, dont la moyenne augmente lorsque x augmente.

L'autre transition possible pour l'action a est que le chauffage tombe en panne, amenant le système dans le deuxième segment, à savoir $[100,140]$. Si on regarde la fonction $F_{[0,20)\cup[20,30)\xrightarrow{\mathsf{a}}[100,140]}(x,y)$, on voit que la fonction linéaire, qui pondère la loi normale, vaut 0.15 à 0 et 0.05 à 30. De son côté, la loi normale de cette fonction est de moyenne $x + 102$ pour représenter le fait qu'en moyenne, la température va augmenter

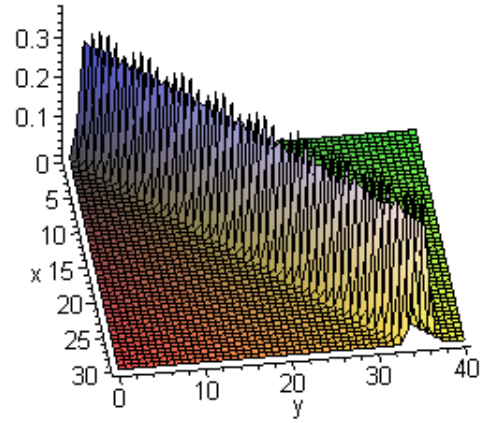

FIG. 3.1 – L'illustration de la fonction de densité associée à $F_{[0,20)\cup[20,30)\xrightarrow{a}[0,40]}(x,y)$

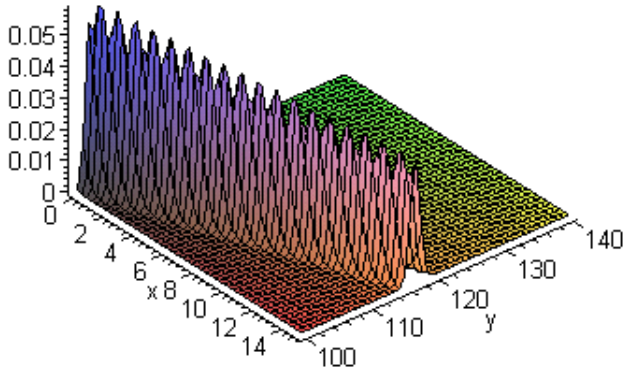

FIG. 3.2 – L'illustration de la fonction de densité associée à $F_{[0,20)\cup[20,30)\xrightarrow{a}[100,140]}(x,y)$

de 2 avant que le chauffage ne cesse de fonctionner, pour amener ensuite le système dans le bon segment, d'où le +102. La figure 3.2 illustre le graphe de la fonction. On peut remarquer que l'amplitude de l'*onde* normale diminue lorsque x augmente, ce qui s'explique par le fait que l'expression $(-x/300 + 0.15)$ dans la fonction passe de 0.15 à 0.05.

3.2.2 Calcul de quelques probabilités

Pour dégager le comportement du système, nous allons présenter quelques calculs de probabilités. Par exemple, nous déterminerons si $0 \leq P_{\mathsf{a}}(x, S) \leq 1$ pour $x \in [0, 30]$. Nous calculerons aussi $P_{\mathsf{a}}(15, \texttt{confort})$ pour déterminer s'il est possible que la température soit dans la zone de confort après avoir demandé de chauffer, à partir d'une température de 15℃.

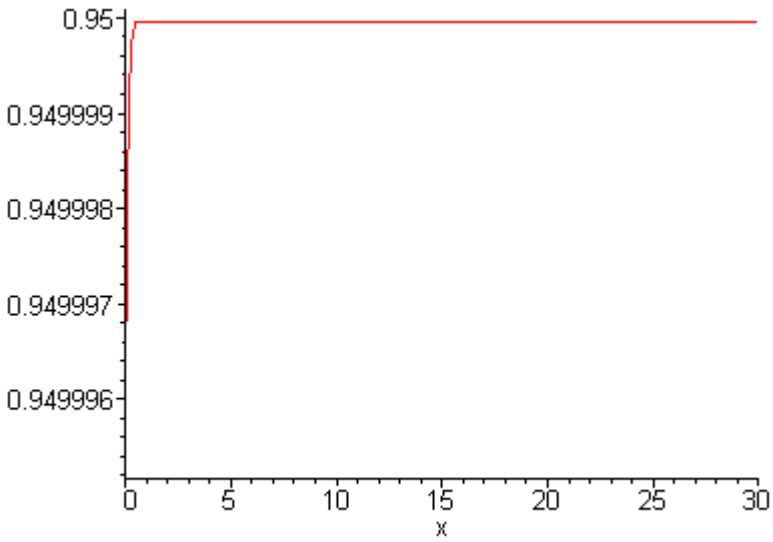

FIG. 3.3 – Le graphe de la fonction $P_{\mathsf{a}}(x, S)$ pour $x \in [0, 30)$.

Calculons la probabilité $P_{\mathsf{a}}(x, S)$. Cette valeur nous permettra de déduire si l'action a est *toujours* possible et s'il est possible que le système ne réponde pas à cette action. Pour $x \in [0, 30)$,

$$\begin{aligned} P_{\mathsf{a}}(x,S) &= F_{[0,20)\cup[20,30)\xrightarrow{\mathsf{a}}[0,40]}(x,y)\Big|_0^{40} + F_{[0,20)\cup[20,30)\xrightarrow{\mathsf{a}}[100,140]}(x,y)\Big|_{100}^{140} \\ &= (x/300+0.8)\left[\Phi\left(\frac{38-x-x/10}{1/2}\right) - \Phi\left(\frac{-x-x/10-2}{1/2}\right)\right] \\ &\quad +(-x/300+0.15)\left[\Phi\left(\frac{38-x}{1/2}\right) - \Phi\left(\frac{-2-x}{1/2}\right)\right] . \end{aligned}$$

Il est difficile d'obtenir une simplification de l'expression précédente pour faciliter la compréhension. La figure 3.3 présente le graphe de la fonction $P_{\mathsf{a}}(x, S)$, pour $x \in [0, 30)$. Comme le suggère cette figure, la valeur de la fonction est inclus dans $[0.949, 0.95]$. Par contre, ce que la figure ne révèle pas, c'est que la fonciton ne vaut pas 0.95 pour aucun x. Pour que ce soit le cas, il faudrait que

$$\frac{38-x-x/10}{1/2} = -\frac{-x-x/10-2}{1/2}$$

et que

$$\frac{38-x}{1/2} = -\frac{-2-x}{1/2}$$

ce qui n'est pas le cas. À l'aide du logiciel *Maple*, on remarque qu'en $x = 0$, $P_{\mathsf{a}}(x, S) \approx 0.9499952493$. Lorsque x augmente, la fonction est plutôt asymptotique à 0.95. La valeur de $P_{\mathsf{a}}(x, S)$ sur $x \in [200, 230]$ est semblable à celle sur $[0, 30)$. On peut donc résumer comme ceci :

$$P_{\mathsf{a}}(x,S) \in \begin{cases} [0.949, 0.95) & \text{si } x \in [0,30) \cup [200,230] \\ \{0\} & \text{sinon} \end{cases} . \tag{3.6}$$

Par conséquent, on note que l'action a n'est pas toujours possible, puisqu'il existe des états où la probabilité de faire a est 0. On note aussi qu'il est possible que le système ne réponde pas à l'action a même si elle est possible, puisque $0 < P_{\mathsf{a}}(x, S) < 1$, $\forall x \in [0, 30) \cup [200, 230]$.

Le lecteur arrivera à des résultats semblables pour l'action d.

Le climatiseur se comporte de la même manière que l'appareil de chauffage, à quelques paramètres près. Nous laissons le soin au lecteur de vérifier les chances de panne étant donnée la température courante pour le climatiseur ainsi que son comportement en moyenne.

Pour illustrer le calcul des probabilités et le comportement du système, nous allons calculer la probabilité qu'à partir de l'état initial et en demandant au système de chauffer la pièce, la température résultante soit dans la zone de confort, c'est-à-dire entre 18℃ et 23℃.

$$\begin{aligned} P_{\mathsf{a}}(15, \texttt{confort}) &= P_{\mathsf{a}}(15, [18, 23] \cup [118, 123] \cup [218, 223] \cup [318, 323]) \\ &= F_{[0,20)\cup[20,30)\xrightarrow{\mathsf{a}}[0,40]}(15, y)\Big|_{18}^{23} + F_{[0,20)\cup[20,30)\xrightarrow{\mathsf{a}}[100,140]}(15, y)\Big|_{118}^{123} \\ &= 0.85\ \Phi\left(\frac{y - 18.5}{0.5}\right)\Big|_{18}^{23} + 0.1\ \Phi\left(\frac{y - 117}{0.5}\right)\Big|_{118}^{123} \\ &\approx 0.7174 \end{aligned}$$

3.2.3 Vérification de quelques formules

Pour illustrer davantage la sémantique de la logique, nous allons vérifier si quelques formules sont satisfaites par le modèle du système de contrôle de température. Entre autres, nous étudierons les formules $\langle \mathsf{a} \rangle_{0.95-\epsilon} \mathsf{T}$ pour un certain ϵ, $\langle \mathsf{d} \rangle_{0.78} \texttt{clim\_ok}$ et $[\![\langle \mathsf{a} \rangle_0 \langle \mathsf{a} \rangle_0 \langle \mathsf{a} \rangle_0 \mathsf{T}]\!]$.

À l'équation (3.6), nous avons énoncé que

$$P_{\mathsf{a}}(x, S) \in \begin{cases} [0.949, 0.95) & \text{si } x \in [0, 30) \cup [200, 230] \\ \{0\} & \text{sinon} \end{cases}.$$

Donc, $\mathcal{S} \models \langle \mathsf{a} \rangle_{0.95-\epsilon} \mathsf{T}$, $\forall \epsilon \in (0, 0.95]$, car l'état initial 15 satisfait cette formule. Mais, il demeure un 5% de chance que le système n'effectue pas la transition demandée.

Comme l'état initial du système est 15, le système satisfait la formule $\neg\texttt{confort}$, puisque $15 \notin [\![\texttt{confort}]\!]$. Donc, si l'utilisateur désire que la température courante

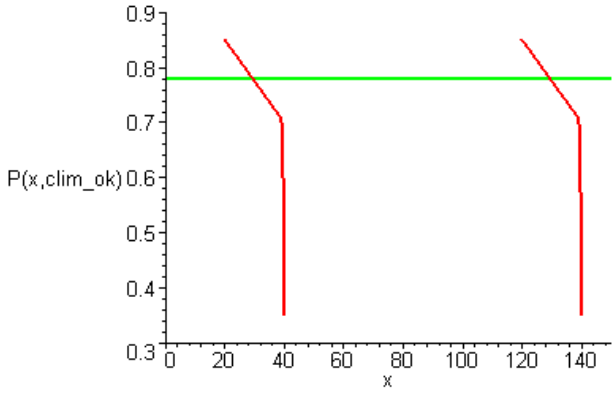

FIG. 3.4 – La fonction $P_{\mathsf{d}}(x, \texttt{clim\_ok})$

soit dans la zone de confort, il doit appuyer sur le bouton étiqueté `a` afin que la température soit augmentée. D'ailleurs, il ne servirait à rien qu'il appuie sur `d`, puisque l'action associée n'est pas activée. Le lecteur peut s'en convaincre en vérifiant que $P_{\mathsf{d}}(15, S) = 0$. Or, même si l'utilisateur demande que la pièce soit chauffée, la probabilité que la température résultante soit dans la zone de confort est d'environ 71.74%, ce qui laisse 28.26% de chance que la température soit plus basse, que le système se brise ou qu'il ne réponde pas à l'action demandée. En somme, le système respecte la formule $\langle \mathsf{a} \rangle_{0.7174-\epsilon} \texttt{confort}$ pour $\epsilon \in (0, 0.7174]$.

Pour évaluer la satisfiabilité d'une formule, nous pouvons déterminer numériquement quels états la satisfont. Par exemple, prenons la formule $\langle \mathsf{d} \rangle_{0.78} \texttt{clim\_ok}$. Tout d'abord, nous devons calculer la probabilité $P_{\mathsf{d}}(x, \texttt{clim\_ok})$ pour ensuite déduire pour quels états cette fonction est supérieure à 0.78. Posons $x \in [20, 40]$, alors

$$\begin{aligned} P_{\mathsf{d}}(x, \texttt{clim\_ok}) &= P_{\mathsf{d}}(x, [0, 40] \cup [100, 140]) \\ &= F_{[20,30)\cup[30,40] \xrightarrow{\mathsf{d}} [0,40]}(x, y)\Big|_0^{40} \\ &= \left(\frac{-3x}{400} + 1\right)\left[\Phi\left(96 - \frac{12x}{5}\right) - \Phi\left(8 - \frac{12x}{5}\right)\right] \end{aligned}$$

Pour $x \in [120, 140]$,

$$\begin{aligned} P_{\mathsf{d}}(x, \texttt{clim\_ok}) &= P_{\mathsf{d}}(x, [0, 40] \cup [100, 140]) \\ &= F_{[120,140] \xrightarrow{\mathsf{d}} [120,140]}(x, y)\Big|_{100}^{140} \\ &= \left(\frac{-3x}{400} + 1.75\right)\left[\Phi\left(336 - 2.4x\right) - \Phi\left(256 - 2.4x\right)\right] \end{aligned}$$

Les états qui satisfont $\langle \mathsf{d} \rangle_{0.78} \texttt{clim\_ok}$ sont ceux où $P_{\mathsf{d}}(x, \texttt{clim\_ok}) > 0.78$. Il est

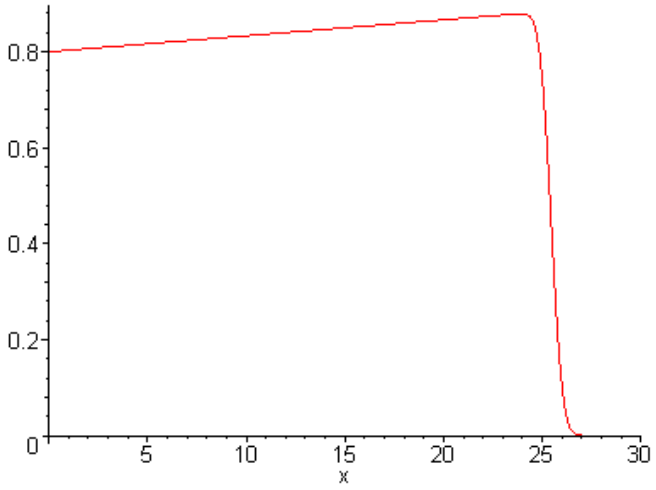

FIG. 3.5 – La fonction $P_{\mathsf{a}}(x, [0,30) \cup [200,230])$ sur $[0,30)$.

possible de déterminer numériquement quels sont les points d'intersection de la fonction $P_{\mathsf{d}}(x, \texttt{clim\_ok})$ avec la droite $f(x) = 0.78$. La figure 3.4 illustre le graphe de la fonction $P_{\mathsf{d}}(x, \texttt{clim\_ok})$, définie pour $x \in [20,40] \cup [120,140]$, ainsi que le la fonction constante $f(x) = 0.78$. Nous détaillerons à la section 4.3.2 quelles techniques peuvent être utilisées pour calcluer les points d'intersection. Ici, l'application d'une méthode numérique donne que pour $x \in \{29.33, 129.33\}$, $P_{\mathsf{d}}(x, \texttt{clim\_ok}) = 0.78$. Ensuite, par une étude de signe de la fonction $g(x) = P_{\mathsf{d}}(x, \texttt{clim\_ok}) - 0.78$, on déduit que $[\![\langle \mathsf{d} \rangle_{0.78} \texttt{clim\_ok}]\!] = [20, 29.33) \cup [120, 129.33)$.

Calculons $[\![\langle \mathsf{a} \rangle_0 \langle \mathsf{a} \rangle_0 \langle \mathsf{a} \rangle_0 \mathsf{T}]\!]$. Comme nous avons calculé précédemment $P_{\mathsf{a}}(x, S)$, nous savons que $[\![\langle \mathsf{a} \rangle_0 \mathsf{T}]\!] = [0,30) \cup [200,230]$. Alors, calculons $P_{\mathsf{a}}(x, [0,30) \cup [200,230])$ pour déterminer $[\![\langle \mathsf{a} \rangle_0 \langle \mathsf{a} \rangle_0 \mathsf{T}]\!]$. D'après la formule sur le calcul de probabilités,

$$\begin{aligned} P_{\mathsf{a}}(x, [0,30) \cup [200,230]) &= F_{[20,30)\cup[30,40] \xrightarrow{\mathsf{a}} [0,40]}(x,y)\Big|_0^{30} \\ &\quad + F_{[200,300) \xrightarrow{\mathsf{a}} [200,240]}(x,y)\Big|_{200}^{230} \\ &= \left(\frac{x}{300} + 0.8\right)\left[\Phi\left(56 - 2.2x\right) - \Phi\left(-4 - 2.2x\right)\right] \\ &\quad + \left(\frac{x}{300} + 2/15\right)\left[\Phi\left(496 - 2.2x\right) - \Phi\left(436 - 2.2x\right)\right] \ . \end{aligned}$$

La figure 3.5 montre le graphe de cette fonction, sur $[0,30)$. La fonction semble valoir 0 pour $x \in (27,30)$, mais ce n'est pas le cas, dû au comportement asymptotique de la fonction de répartition d'une variable normale. En effet, bien que $\lim_{x\to\infty} \Phi(x) = 1$, la fonction $\Phi(x)$, ne vaut jamais 1. On remarque le même phénomène pour $x \in [200,230]$. Donc, les états pour lesquels $P_{\mathsf{a}}(x, [0,30) \cup [200,230]) > 0$ sont $[0,30) \cup [200,230]$. On remarque que ce sont les mêmes états pour $[\![\langle \mathsf{a} \rangle_0 \mathsf{T}]\!]$. Finallement, nous pouvons conclure que $[\![\langle \mathsf{a} \rangle_0 \langle \mathsf{a} \rangle_0 \langle \mathsf{a} \rangle_0 \mathsf{T}]\!] = [0,30) \cup [200,230]$, car nous serons amené à calculer exactement

la même fonction de probabilités, sur le même ensemble d'états $[0, 30) \cup [200, 230]$.

Nous avons illustré dans cette section le comportement du système, ses possibilités de bris, ses possibilités de non-réponse à l'action demandée, etc. Cet exemple nous servira plus loin de plateforme pour les tests de performance, ainsi que pour illustrer la satisfiabilité de certaines formules de logiques.

L'exemple étudié montre aussi l'importance du calcul des probabilités, lorsqu'on doit déterminer si un système satisfait une formule de la forme $\langle \mathsf{a} \rangle_q \phi$. En fait, la démarche pour calculer $[\![\langle \mathsf{a} \rangle_q \phi]\!]$ est la suivante. Premièrement, il faut calculer $[\![\phi]\!]$. Si ϕ est T ou une proposition atomique, le calcul est simple. Si ϕ est plutôt composé d'un opérateur de la logique, il faudra s'en remettre aux égalités du lemme 2.2.1. Deuxièmement, on doit calculer $P_{\mathsf{a}}(x, [\![\phi]\!])$, qui peut former une fonction à branche. Troisièmement, on doit calculer les zéros de la fonction $g(x) = P_{\mathsf{a}}(x, [\![\phi]\!]) - q$. Finalement, on doit faire une étude de signe autour des zéros de $g(x)$ pour déterminer pour quels x, $g(x) > 0$. Ce type de procédure devra se retrouver dans l'implantation du vérificateur formel afin que le programme puisse calculer les formules de la forme $\langle \mathsf{a} \rangle_q \phi$. Donc, en plus des opérations au niveau de la logique, l'implantation devra être en mesure de calculer la fonction de probabilité $P_{\mathsf{a}}(x, E)$ pour $E \subseteq S$, de faire une étude de signe ainsi que de trouver les zéros d'une fonction réelle.

3.3 Vérification formelle pour les LMP*

La vérification formelle de modèle est une technique automatique qui, pour un modèle d'un système et une propriété énoncée sous la forme d'une formule de logique, vérifie systématiquement si la propriété est respectée dans tous les états du sytème. Nous présentons deux approches pour effectuer une vérification formelle d'un LMP*, l'une basée sur une exploration de l'espace d'état, l'autre basée sur le calcul d'un système probabiliste fini *équivalent*.

3.3.1 Vérification directe

La vérification directe d'une formule ϕ sur un modèle $\mathcal{S}$ consiste en une exploration judicieuse de l'espace d'état basée sur la sémantique des opérateurs logiques formant ϕ. En effet, il ne sert à rien de parcourir tous les états d'un modèle pour retenir ceux qui vérifient une formule. Par exemple, supposons qu'un système $\mathcal{S}$ possède deux étiquettes

sur ses états, $\mathtt{p}$ et $\mathtt{q}$ et que l'on veut vérifier une formule comme $\mathtt{p}$, $\neg\mathtt{q}$ et $\mathtt{p} \vee \mathtt{q}$. Alors, on peut parcourir tous les états pour vérifier leurs étiquettes et ainsi déduire s'ils satisfont la formule. Cette méthode est très coûteuse en temps, surtout si l'ensemble des états est infini, comme pour les LMP*. Par contre, il est plus judicieux de calculer $[\![\mathtt{p}]\!]$ et $[\![\mathtt{q}]\!]$, donnés par *label*, et déduire ensuite quels états satisfont la formule donnée. Alors, les calculs se résument à deux appels à la fonction *label* et quelques opérations sur les ensembles (union, intersection et différence). En somme, cette approche se veut une implémentation directe du calcul de $[\![\phi]\!]$, tel que donné au lemme 2.2.1. Dans ce qui suit, nous allons présenter les algorithmes nécessaires à la vérification directe.

L'algorithme de vérification directe prend en entrées un LMP*, $\mathcal{S}$, et une formule, ϕ, et renvoie l'ensemble des états de $\mathcal{S}$ qui vérifient la formule ϕ. Il procède de manière récursive pour déterminer l'ensemble, à la manière de l'algorithme de model-checking de CTL. Voici en détails l'algorithme $Sat(\mathcal{S}, \phi)$.

Algorithme 3.3.1 Sat(LMP* $\mathcal{S}$, Formule ϕ)
Dépendant de ϕ, Choisir :
- Cas T
  - Retourner S
- Cas $\mathtt{p}$
  - Retourner $label(\mathtt{p})$
- Cas $\neg\psi$
  - Retourner $S \setminus Sat(\mathcal{S}, \psi)$
- Cas $\psi_1 \wedge \psi_2$
  - Retourner $Sat(\mathcal{S}, \psi_1) \cap Sat(\mathcal{S}, \psi_2)$
- Cas $\psi_1 \vee \psi_2$
  - Retourner $Sat(\mathcal{S}, \psi_1) \cup Sat(\mathcal{S}, \psi_2)$
- Cas $\langle \mathsf{a} \rangle_q \psi$
  - $E := Sat(\mathcal{S}, \psi)$
  - $Prob(x) := CalculerProbabilite(\mathcal{S}, E, \mathsf{a})$
  - Retourner $Separation(Prob(x), q)$

Fin Choisir
Fin Algorithme
□

On voit clairement que l'algorithme est une implantation directe de ce qui est décrit au lemme 2.2.1. Les calculs à l'intérieur de l'algorithme utilisent les opérations de bases de l'arithmétique des ensembles (union, intersection et différence). Les premiers cas sont directement construits à partir de la sémantique. Le dernier cas, celui où $\phi = \langle \mathsf{a} \rangle_q \psi$, est plus compliqué puisqu'il faut calculer l'ensemble $\{x \in S | P_{\mathsf{a}}(x, [\![\psi]\!]) > q\}$. Pour ce

faire, nous utilisons deux sous-procédures. La première, $CalculerProbabilite(\mathcal{S}, E)$, retourne une fonction de S dans $[0, 1]$ qui donne la probabilité $P_{\mathsf{a}}(x, E)$ pour tout $x \in S$. Habituellement, cette fonction est une fonction à branche puisque les fonctions de probabilités sont généralement différentes pour chaque $X \in \mathbf{X}$, donnant potentiellement une branche pour chaque X. Cette sous-procédure, implémente la formule (3.3) si les fonctions de probabilité sont des fonctions de densité ou bien la formule (3.4) si elles sont sous formes de fonctions de répartition. La dernière approche est détaillée à l'algorithme 3.3.2, plus bas. Pour calculer cette fonction, la sous-procédure suppose que x est dans chacun des ensembles de départ de $\mathbf{X}$. Par conséquent, ceci nous donne des fonctions partiellement évaluées puisque le deuxième argument y a été intégré ou bien évalué, laissant le premier argument toujours non-évalué. Nous avons vu ceci dans les exemples 3.1.6, 3.1.7, 3.1.10. La deuxième sous-procédure prend une fonction et une valeur et retourne l'ensemble des états pour lesquels la fonction est strictement supérieure à cette valeur. Étant donné que la fonction ici est la fonction de probabilité $P_{\mathsf{a}}(x, E)$, la sous-procédure retourne l'ensemble des états tel que la probabilité d'aller vers $[\![\psi]\!]$ avec l'action a est supérieur à q.

En somme, l'essence de l'approche utilisée dans l'algorithme pour déterminer $[\![\langle \mathsf{a} \rangle_q \phi]\!]$ se retrouve dans la manière de calculer cette valeur sur papier. On voit clairement un lien entre l'approche et ce qui est fait dans l'exemple de la section 3.2.1 et les exemples 2.2.2 et 3.2.3, à savoir qu'il faut calculer d'abord $[\![\phi]\!]$, déterminer ensuite $P_{\mathsf{a}}(x, [\![\phi]\!])$, pour conserver finalement les états x tels que $P_{\mathsf{a}}(x, [\![\phi]\!]) > q$.

Nous allons présenter en détails ici les deux sous-procédures mentionnées plus haut. En premier, nous présentons l'algorithme $CalculerProbabilite(\mathcal{S}, E)$, qui calcule la fonction qui associe à x la probabilité d'aller vers un état de $E \subseteq S$, sachant que les fonctions de probabilité des transitions sont données sous forme de fonctions de répartition. Dans l'algorithme,les fonctions $Borne\_Sup(Y)$ et $Borne\_Inf(Y)$ renvoient la borne supérieure et la borne inférieure de l'intervalle Y, respectivement. Ces fonctions sont bien définies pour $Y \in \overline{\mathbf{Y}^{\mathbf{d}}}$ puisque ces ensembles sont des intervalles et non pas des unions d'intervalles.

Nous avons remarqué plus haut que la fonction $P_{\mathsf{a}}(x, E)$ pouvait former une fonction à branche. Pour l'instant, nous ne spécifions pas la manière dont un tel type de fonction est stocké en mémoire, mais nous supposerons seulement deux choses :

1. Une fonction à branche est un ensemble de paires intervalle-fonction, l'intervalle définissant le domaine d'une branche, et la fonction, la valeur sur celui-ci. Les intervalles doivent être disjoints entre-eux.
2. Pour évaluer une fonction à branche en x, on recherche si un intervalle contient x parmi les paires intervalle-fonction. On évalue alors la fonction associée en x. S'il

n'existe pas de tel intervalle, on suppose alors que la fonction à branche vaut 0 en x.

Algorithme 3.3.2 *CalculerProbabilite*(LMP* $\mathcal{S}$, Ensemble E, Action a)
Initialiser la fonction $Prob(x)$ à 0, $\forall x \in S$.
Pour chaque X dans $\mathbf{X}$
  Initialiser $f(x) := 0$
  Pour chaque intervalle E_i de E
    Pour chaque $Y \in \overline{\mathbf{Y^d}}$ tel que $F_{X \xrightarrow{\mathsf{a}} Y}(x, y)$ existe
      $Y' := Y \cap E_i$
      Si $Y' \neq \emptyset$
        $f(x) := f(x)$
          $+ \left[F_{X \xrightarrow{\mathsf{a}} Y}(x, Borne\_Sup(Y')) - F_{X \xrightarrow{\mathsf{a}} Y}(x, Borne\_Inf(Y'))\right]$
        Si $Y' = E_i$
          Sortir du Pour chaque Y
        Fin Si
      Fin Si
    Fin Pour
  Fin Pour
  Pour chaque $Y \in \mathbf{Y^d}$
    $Y' := E \cap Y$
    Pour chaque point $y \in Y'$
      $f(x) := f(x) + F_{X \xrightarrow{\mathsf{a}} Y}(x, y)$
    Fin Pour
  Fin Pour
  Ajouter la branche $f(x)$ à $Prob(x)$ avec le domaine X
Fin Pour
Fin Algorithme
□

L'algorithme implémente la formule (3.4),

$$P_{\mathsf{a}}(x, E) = \sum_{i \in I} \left[\sum_{\substack{Y \in \overline{\mathbf{Y^d}} \\ Y \cap E_i \neq \emptyset}} \left[F_{X \xrightarrow{\mathsf{a}} Y}(x, \sup(Y \cap E_i)) - F_{X \xrightarrow{\mathsf{a}} Y}(x, \inf(Y \cap E_i))\right] \right] + \sum_{Y \in \mathbf{Y^d}} \sum_{y \in E \cap Y} F_{X \xrightarrow{\mathsf{a}} Y}(x, y) \ ,$$

en supposant que les fonctions de probabilité des transitions sont données sous forme de fonctions de répartition. Pour déterminer la fonction de probabilité, il est nécessaire

et suffisant d'itérer sur chaque $X \in \mathbf{X}$: nécessaire, car ils peuvent chacun définir une branche de la fonction $P_{\mathsf{a}}(x, E)$; suffisant, car les états qui ne sont pas dans $\cup\mathbf{X}$ ne sont pas des états d'où on peut effectuer une transition et il est inutile de les considérer dans le cacul de la probabilité.[4] En effet, nous supposons que si un état $x \in S$ n'appartient pas au domaine d'une des branches de $P_{\mathsf{a}}(x, E)$, c'est que la fonction vaut 0 en ce point. Une fois X fixé par la boucle extérieure, l'algorithme calcule la fonction telle que donnée par la formule.

La seconde sous-procédure est $Separation(f(x), q)$, qui détermine pour quelles valeurs en x, la fonction $f(x)$ est strictement supérieure à q, c'est-à-dire l'ensemble $\{x \in S | f(x) > q\}$.

Algorithme 3.3.3 *Separation*(Fonction $f(x)$, Réel q)
1 Initialiser l'ensemble E à vide
2 Pour chaque branche de la fonction $f(x)$
3 $X :=$ domaine de la branche courante
4 $g(x) :=$ la branche de $f(x)$ dont le domaine est X
5 Si $g(x)$ est constant sur X et $g(x) > q$ pour un $x \in X$
6 Ajouter X à E
7 Sinon
8 Liste $[x_i]_{i \in I} := TrouverZeros(g(x) - q, X)$
9 $n := |I|$
10 Si $n = 0$
11 {*Aucun point d'intersection*}
12 Si $g(Borne\_Inf(X)) > q$
13 Ajouter X à E
14 Fin Si
15 Sinon
16 {*Début du traitement des points d'intersection*}
17 Si $g(Borne\_Inf(X)) > q$
18 {*Il y a une partie de la fonction au dessus de* $h(x) = q$}
19 Si $Borne\_Inf(X) \in X$
20 Ajouter $[Borne\_Inf(X), x_1)$ à E
21 Sinon
22 Ajouter $(Borne\_Inf(X), x_1)$ à E
23 Fin Si
24 Fin Si
25 Pour i de 2 à n

[4] Une amélioration faite à l'implantation est d'itérer seulement sur les X pour lesquels une a-transition existe. La même chose a été faite pour la boucle sur les Y, puisque X et a sont fixés.

```
26                  Si g(\frac{x_{i-1}+x_i}{2}) > q
27                      Ajouter (x_{i-1}, x_i) à E
28                  Fin Si
29              Fin Pour
30              Si g(Borne_Sup(X)) > q
31                  Si sup(X) est incluse
32                      Ajouter (x_n, Borne_Sup(X)] à E
33                  Sinon
34                      Ajouter (x_n, Borne_Sup(X)) à E
35                  Fin Si
36              Fin Si
37          Fin Si
38      Fin Si
39 Fin Pour
40 Retourner E
41 Fin Algorithme
□
```

Pour trouver toutes les valeurs de x pour lesquelles $f(x) > q$, la sous-procédure doit déterminer d'abord les points d'intersection entre chacune des branches de $f(x)$ et la droite $h(x) = q$. Pour cela, l'algorithme itère sur chacune des branches (ligne 2). Si la branche courante, appelons-la $g(x)$, est une fonction constante, c'est-à-dire de la forme $g(x) = k$ pour $k \in \mathbb{R}$, alors il pourrait y avoir une infinité de points d'intersection. À la ligne 5, on vérifie si $g(x)$ est une droite strictement supérieure à q et si c'est le cas, tout le domaine de la branche est ajouté à E, l'ensemble des états qui sera retourné.

Si $g(x)$ n'est pas constant, on doit d'abord déterminer quels sont les zéros de la fonction $g(x) - q$. Pour cela, on fait appel à $TrouverZeros(f(x), E)$, qui retourne les zéros de la fonction $f(x)$ appartenant à la fermeture de l'intervalle E. À la section 4.3.2, nous donnerons plus de détails sur l'implémentation de cette sous-procédure, qui utilise une méthode numérique. Une fois l'appel fait, on doit déduire pour quels x, $g(x) > q$. Pour y arriver, on utilise le fait que les points d'intersections sont stockés en ordre, dans la liste $[x_i]_{i \in I}$ et nous supposons qu'il n'en existe pas d'autres. Alors, l'algorithme traite les points à la suite, en commençant par étudier la valeur de la fonction $g(x)$ entre la borne inférieure de X et le premier point d'intersection. À la ligne 17, si $g(Borne\_Inf(X)) > q$, alors tout $g(x)$ sur $(Borne\_Inf(X), x_1)$ est strictement supérieur, puisque nous supposons que la fonction est continue et qu'il n'existe pas d'autres points d'intersection que ceux dans la liste $[x_i]_{i \in I}$. Ensuite, on vérifie si $g(x) > q$ pour un x choisi entre deux points d'intersection successifs, x_{i-1} et x_i. Si c'est le cas, on conserve l'intervalle (x_{i-1}, x_i). À la fin, on effectue un traitement pour le dernier point et la borne supérieure

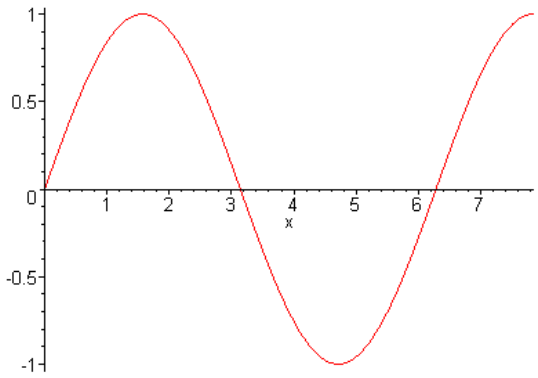

FIG. 3.6 – $\sin(x)$ sur $(0, \frac{5\pi}{2})$.

de X. Il est à noter que le cas où $f(x) = q$ pour l'une des bornes de X ne pose pas de problème, puisque'alors, les conditions des *Si* aux lignes 17 et 30 seraient fausses. Ainsi, la ou les bornes seraient traitées en tant que points d'intersection, dans la boucle *Pour* de la ligne 25.

Nous allons présenter une trace de l'application de cet algorithme sur une fonction, puisqu'il est plus compliqué que les autres et puisque c'est la première fois que l'on rencontre ce genre de traitement dans le présent document. La trace illustre bien l'analyse des bornes de l'intervalle lorsqu'une d'entre elles est un zéro de la fonction. L'exemple montre aussi l'importance de considérer la fermeture de l'intervalle lors de la recherche des zéros.

Exemple 3.3.4 *Nous désirons déterminer les zéros de la fonction à branche suivante*

$$f(x) = \begin{cases} \sin(x) & si\ x \in (0, \frac{5\pi}{2}) \\ 0 & sinon \end{cases}$$

La figure 3.6 montre le graphe de $sin(x)$ *sur* $(0, \frac{5\pi}{2})$. *Nous allons faire la trace de l'appel* $Separation(f(x), 0)$.

Tout d'abord, notons que $f(x)$ *n'a qu'une seule branche non-nulle, à savoir la paire* $(0, \frac{5\pi}{2})$ — $\sin(x)$. *Donc, la boucle de la ligne 2 de l'algorithme 3.3.3 ne se fera qu'une seule fois. Alors,* $X = (0, \frac{5\pi}{2})$ *et* $g(x) = \sin(x)$. *Comme la fonction n'est pas constante, il y aura un appel à la sous-procédure* $TrouverZero$, *qui retournera la liste* $[0, \pi, 2\pi]$. *La valeur* 0 *se retrouve dans la liste puisque* $TrouverZero$ *retourne tous les zéros sur la fermeture de* $(0, \frac{5\pi}{2})$. *À la ligne 17, la condition du* Si *est fausse, puisque* $g(x) = 0$. *Alors, l'algorithme poursuit en effectuant la boucle sur les éléments de la liste, aux lignes 25 à 29. Pour* 0 *et* π, *la valeur de* $g(\pi/2) = 1$ *est supérieure à* 0, *alors l'ensemble* E *devient* $(0, \pi)$. *Pour* π *et* 2π, $g(3\pi/2) = -1 \not> 0$ *et* E *n'est pas modifié. Après la*

boucle, à la ligne 30, la condition du Si *est vraie, puisque* $g(5\pi/2) = 1$. *Comme la borne supérieure n'est pas incluse, l'ensemble* E *devient* $(0, \pi) \cup (2\pi, 5\pi/2)$. *Finalement, l'algorithme retourne l'ensemble* E.

Dans l'exemple, si la procédure $TrouverZero$ n'avait pas considéré la fermeture de l'intervalle $(0, \frac{5\pi}{2})$, alors elle aurait retourné la liste $[\pi, 2\pi]$. Par conséquent, *Separation* n'aurait pas analysé les points entre 0 et π et aurait retourné l'ensemble $(2\pi, 5\pi/2)$.

Une implémentation de tous ces algorithmes devrait être en mesure de déterminer si une formule de logique est satisfaite par un état, par un système ainsi qu'être en mesure de calculer quels états la satisfont. L'approche de la vérification directe a été implémentée par l'auteur. Le lecteur trouvera au chapitre 4 des indications concernant cette implémentation.

3.3.2 Vérification par le calcul d'approximations

Dans [8], les auteurs ont développé une méthode pour construire un système probabiliste fini à partir d'un LMP*. Ce système fini, que nous appelons l'approximation, est construit de telle sorte qu'il est possible d'inférer si une formule est satisfaite par le LMP*, à partir du fait qu'elle est satisfaite par une approximation bien choisie. L'intérêt d'une telle approche est que le calcul de $[\![\phi]\!]$ se fait sur un système fini et peut être plus simple que de calculer $[\![\phi]\!]$ sur le système infini. Il existe d'ailleurs quelques outils [16][3] pour faire la vérification formelle de systèmes probabilistes finis, nous évitant d'implémenter un algorithme de vérification. Un autre avantage de cette approche est qu'il est possible de caractériser l'ensemble des formules pour lesquelles on peut déduire la satisfiabilité à partir d'une approximation, permettant ainsi de la réutiliser.

L'algorithme qui suit a été extrait en bonne partie d'une définition dans [8] et raffinée dans [4]. Il calcule un système fini, une approximation, pour un LMP* $\mathcal{S}$, un entier n et un rationnel $\epsilon > 0$. Les auteurs de l'approche ont démontré qu'un LMP* simule toutes ses approximations. Donc, si une formule est satisfaite par une approximation, alors le système infini la satisfait aussi.

Les paramètres de l'algorithmes, n et ϵ, servent à déterminer la précision de l'approximation. On choisit n comme étant la profondeur maximale d'un ensemble de formules que l'on désire vérifier. La profondeur d'une formule, ou « depth », est définie de

manière récursive, comme ceci :

$$\begin{aligned}
\text{depth}(\mathsf{T}) &= 0 \\
\text{depth}(\mathtt{p}) &= 0 \\
\text{depth}(\neg\phi) &= 1 + \text{depth}(\phi) \\
\text{depth}(\langle \mathsf{a} \rangle_q \phi) &= 1 + \text{depth}(\phi) \\
\text{depth}(\phi \wedge \psi) &= 1 + \max(\text{depth}(\phi), \text{depth}(\psi)) \\
\text{depth}(\phi \vee \psi) &= 1 + \max(\text{depth}(\phi), \text{depth}(\psi))
\end{aligned}$$

où $\mathtt{p}$ est une proposition atomique sur les états. La paramètre $\epsilon = 1/\epsilon'$ est choisi tel que ϵ' est le plus petit commun multiple de tous les dénominateurs des probabilités apparaissant dans les formules du type $\langle \mathsf{a} \rangle_q \phi$.

L'algorithme procède comme suit. Il construit un système probabiliste fini $\mathcal{S}(n, \epsilon)$ à partir d'un LMP* $\mathcal{S} = (S, i, \mathsf{Act}, \mathtt{Atom}, label, \rightarrow)$. Chaque état de ce dernier est une paire (A, i) où $A \in \Sigma$ et i est un entier entre 0 et n. Les états sont séparés en $n + 1$ *niveaux*, identifiés par la deuxième partie de la paire. Chaque niveau est une partition de S. Le niveau 0 n'a qu'un seul état, l'état inital $(S, 0)$, et le niveau suivant est défini d'après les probabilités d'aller vers S. En fait, l'algorithme calcule $P_{\mathsf{a}}(x, S)$, $\forall x \in S$, et divise ensuite cette fonction en couche d'épaisseur ϵ. Tous les x qui appartiennent à une même couche formeront un état du niveau suivant, de la forme $(E, 1)$. Une fois toutes les couches séparées, on donne des transitions entres les états du niveau 1 et celui du niveau 0 en prenant le inf de la fonciton $P_{\mathsf{a}}(x, S)$ sur E. Une fois la couche 1 terminée, on continue avec la prochaine, en séparant en couche toutes les fonctions du type $P_{\mathsf{a}}(x, E)$ où $(E, 1) \in \mathcal{S}(n, \epsilon)$.

Algorithme 3.3.5 *Approximation**(LMP*$\mathcal{S}$, Naturel n, Rationnel ϵ)
Initialiser l'approximation vide $\mathcal{S}(n, \epsilon)$
Ajouter l'état $(S, 0)$ à $\mathcal{S}(n, \epsilon)$
Pour l de 1 à n
  m := Nombre d'état à la couche $l - 1$
  Copier tous les ensembles d'états de la couche $l - 1$ vers la couche l
  Pour chaque C tel que $(C, l - 1)$ existe
    Pour chaque action $\mathsf{a} \in \mathsf{Act}$
      Calculer $f(x) := P_{\mathsf{a}}(x, C)$
      $Couches := DecoupageEnCouche(f(x), \epsilon/m)$
      Pour chaque ensemble E qui forme une couche de $P_{\mathsf{a}}(x, U)$
        Pour chaque ensemble Y d'état du niveau l
          Si $Y \cap E \neq \emptyset$ et $Y \cap E \neq Y$
            Enlever l'état (Y, l)

Ajouter l'état $(Y \cap E, l)$
Ajouter l'état $(Y \backslash E, l)$ [5]
Fin Si
Fin Pour Y
Fin Pour E
Fin Pour a
Fin Pour C
{Traitement des transitions}
Pour chaque M tel que (M, l) existe
Pour chaque N tel que (N, l) existe
Pour chaque action $\mathsf{a} \in A$
Calculer $inf := \inf_{x \in M} P_{\mathsf{a}}(x, N)$
Si $inf \neq 0$, ajouter la transition $(M, l) \xrightarrow{\mathsf{a}[inf]} (N, l)$
Fin Pour a
Fin Pour N
Fin Pour M
Pour chaque M tel que (M, l) existe
Pour chaque N tel que $(N, l-1)$ existe
Pour chaque action $\mathsf{a} \in A$
Calculer $inf := \inf_{x \in M} P_{\mathsf{a}}(x, N)$
$k :=$ indice de l'intervalle contenant la borne inf. de N dans l
$somme$:=0
Tant que k est inférieur au nombre d'ensemble dans la couche l
et que $k^{\text{ième}}$ ensemble de l s'instersecte avec N
$E :=$ le $k^{\text{ième}}$ ensemble de l
somme := somme + valeur de la transition de E à N par a
k := k+1
Fin tant que
Si $inf - somme \neq 0$, ajouter la transition $(M, l) \xrightarrow{\mathsf{a}[inf - somme]} (N, l-1)$
Fin Pour a
Fin Pour N
Fin Pour M
Fin Pour l
Retourner l'approximation $\mathcal{S}(n, \epsilon)$
Fin Algorithme
□

Pour séparer les fonctions du type $P_{\mathsf{a}}(x, E)$ en couches d'épaisseur ϵ, nous avons

[5] Si $Y \backslash E = E_1 \dot{\cup} E_2$, alors on ajoute deux états (E_1, l) et (E_2, l)

développé l'algorithme suivant. Il est en fait une généralisation de l'algorithme 3.3.3 *Separation*.

Algorithme 3.3.6 *DecoupageEnCouche*(Fonction $f(x)$, Réel ϵ)
Initialiser E, un ensemble d'intervalles
Pour chaque branche de la fonction $f(x)$
$X :=$ domaine de la branche courante
$g(x) :=$ la branche de $f(x)$ dont le domaine est X
Liste $[x_i]$ qui conserve les points en ordre croissant
Pour k de ϵ à 1, pas ϵ
$[x_i] := [x_i] \cup TrouverZeros(f(x) - k, X)$ Fin Pour k
S'il n'y a aucun point d'intersection
Ajouter $[Borne\_Inf(X), Borne\_Sup(X)]$ à E
Sinon
{*Début du traitement des points d'intersection*}
Si $x_1 \neq Borne\_Inf(X)$
Si $f(Borne\_Inf(X)) < f(x_1)$
Ajouter $[Borne\_Inf(X), x_1]$ à E
Sinon
Ajouter $[Borne\_Inf(X), x_1)$ et $[x_1, x_1]$ à E
Fin Si
Sinon
Ajouter $[x_1, x_1]$ à E
Fin Si
Pour tous les autres points d'intersection $x_i, i > 1$
et en dernier pour $Borne\_Sup(X)$
Si $f(x_{i-1}) < f(x_i)$
Ajouter $(x_{i-1}, x_i]$ à E
Sinon
Si $f(x_{i-1}) = f(x_i)$
Si $f(\frac{x_{i-1}+x_i}{2}) > f(x_i)$
Ajouter (x_{i-1}, x_i) et $[x_i, x_i]$ à E
Sinon Ralonger l'intervalle finissant par $x_{i-1}]$ jusqu'à $x_i]$
Fin Si
Sinon
$\{f(\frac{x_{i-1}+x_i}{2}) > f(x_i)\}$
Ralonger l'intervalle finissant par $x_{i-1}]$ jusqu'à $x_i)$
Ajouter $[x_i, x_i]$ à E
Fin Si
Fin Si

Fin Pour
Fin Si
Fin Pour
Retourner E
Fin Algorithme
□

Comme le le but du présent travail n'est pas de développer l'approche par approximation, le lecteur est invité à consulter [8, 6] pour plus de détails et des exemples.

3.3.3 Comparaison des approches

À prime abord, pour vérifier un ensemble de formules $\mathcal{L}$, on pourrait penser que la vérification directe prendrait plus de temps que le calcul d'une approximation satisfaisante suivi de la vérification sur cette approximation finie. Il faut noter qu'une approximation $\mathcal{S}(n, 1/q)$ possède $\sum_{i=0}^{n} q^i$ états en pire cas. Alors, pour n et q arbitrairement grands, même si le nombre de formules à vérifier est petit, le calcul de l'approximation sera long. De plus, la procédure *TrouverZero*, qui influence grandement le temps de calcul (voir la section 4.5), n'est appelée qu'une seule fois pour chaque opérateur du type $\langle \mathsf{a} \rangle_q$ lors de la vérification directe. Dans le cas du calcul d'une approximation $\mathcal{S}(n, 1/q)$, ce nombre d'appels correspond à $q \sum_{i=0}^{n} q^i$. En somme, la vérification par le calcul d'une approximation peut être plus rapide, mais pour un ensemble de formules de cardinalité très élevée.

De plus, la vérification directe d'un ensemble de formules pourrait être améliorée en exploitant la présence de sous-formules identiques. Pour illustrer ce propos, pensons à l'exemple dégénéré où l'ensemble des formules à vérifier serait

$$\mathcal{L} = \{\phi,\ \phi \vee \psi,\ \phi \wedge \langle \mathsf{a} \rangle_q \phi,\ \langle \mathsf{a} \rangle_q \langle \mathsf{a} \rangle_q \langle \mathsf{a} \rangle_q \langle \mathsf{a} \rangle_q \psi\}\ .$$

Nous aurions alors avantage à calculer $[\![\phi]\!]$ et $[\![\psi]\!]$ une seule fois et à les garder en mémoire, pour déduire ensuite la satisfiabilité des formules de $\mathcal{L}$. Une telle approche demande un traitement supplémentaire, mais pourrait être implémentée facilement.

La section 4.5 présente une analyse des résultats de l'implémentation de l'approche directe.

Chapitre 4

Implémentation

Dans cette section, nous allons présenter l'implémentation de la vérification directe des systèmes probabilistes continus. Rappelons que le but d'une telle vérification est de déterminer si un système décrit dans le langage de la section 3.1 satisfait une formule de la logique de la section 2.2. Nous allons procéder comme suit : tout d'abord, nous commençons par décrire l'architechture générale du programme en présentant les différents paquetages et leur utilité. Ensuite, nous allons montrer quelques exemples qui ont été résolus avec notre programme, accompagnés de statistiques pour en évaluer la performance. Finalement, nous allons discuter des limites et travaux futurs qui seraient réalisables.

4.1 Présentation générale du programme

Un vérificateur formel est un programme complexe pouvant rassembler plusieurs fonctionnalités. Dans cette section, nous allons présenter celles que nous avons choisis, comme objectifs pour arriver à bâtir la première version de l'implémentation.

Voici les fonctionnalités que nous avons retenus, pour l'implémentation :
- La possibilité de charger un LMP* $\mathcal{S}$ à partir d'un fichier, dans un format préétabli, utilisant le langage de description des transitions de la section 3.1 ;
- La saisie à l'écran d'une formule de logique ϕ ;
- La validation du système par rapport à la contrainte $0 \leq P_{\mathsf{a}}(x, S) < 1$, $\forall x \in S$ et $\forall \mathsf{a} \in \mathsf{Act}$;
- Le calcul de $[\![\phi]\!]$, l'ensemble des états qui satisfont la formule ;

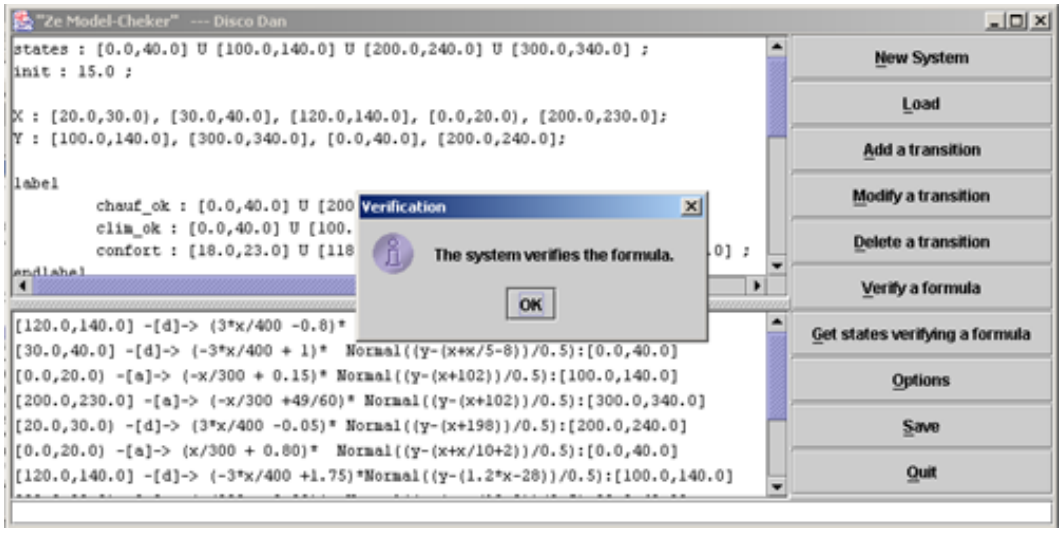

FIG. 4.1 – Le vérificateur formel déclare que le système vérifie la formule.

– La vérification de $\mathcal{S} \models \phi$.

La façon de procéder pour vérifier que $\mathcal{S} \models \phi$ est d'abord de décrire le LMP* dans un fichier, suivant le format donné plus loin à la section 4.3.3, ensuite de saisir la formule à l'écran et de cliquer sur un bouton pour déclencher la vérification. Une fois les calculs terminés, le progamme répond par oui ou par non, tout dépendant si le système vérifie la formule ou non. La figure 4.1 donne un apperçu de l'interface du programme et du message qu'il donne lorsque le système chargé vérifie la formule donnée par l'utilisateur.

Avec les fonctionnalités énumérées plus haut et les algorithmes nécessaires à la vérification formelle, nous avons choisi de développer le programme suivant le modèle présenté à la figure 4.2. En somme, l'utilisateur doit utiliser l'interface graphique pour entrer son système et sa formule. Dans le premier cas, ce sera par le chargement d'un fichier, dans le second, ce sera par la saisie dans une boîte de discussion à l'écran. Alors, dans les deux cas, l'information sous forme de chaînes de caractères devra être analysée pour les traduire en une formule de logique et un LMP*. S'il n'y a aucune erreur de syntaxe, les analyseurs vont créer deux entités, afin de conserver l'information. Ces entités seront alors données au moteur de calcul afin qu'il détermine si le système satisfait la formule. Comme nous l'avons vu à la section 3.3.1 sur la vérification directe, il y a 3 sous-procédures importantes, dont deux sont complètement extérieures au domaine, à savoir $TrouverZero$ et $Separation$. Nous avons choisi de les représenter à l'extérieur du moteur de calcul dans le modèle d'analyse pour conserver l'idée qu'au niveau de l'imlémentation, ces deux sous-procédures pourraient être facilement remplacées plus tard par du code extétieure. Finalement, une fois la vérification terminée, le moteur de calculs envoie un message à l'interface graphique déclarant si le système probabiliste vérifie la formule.

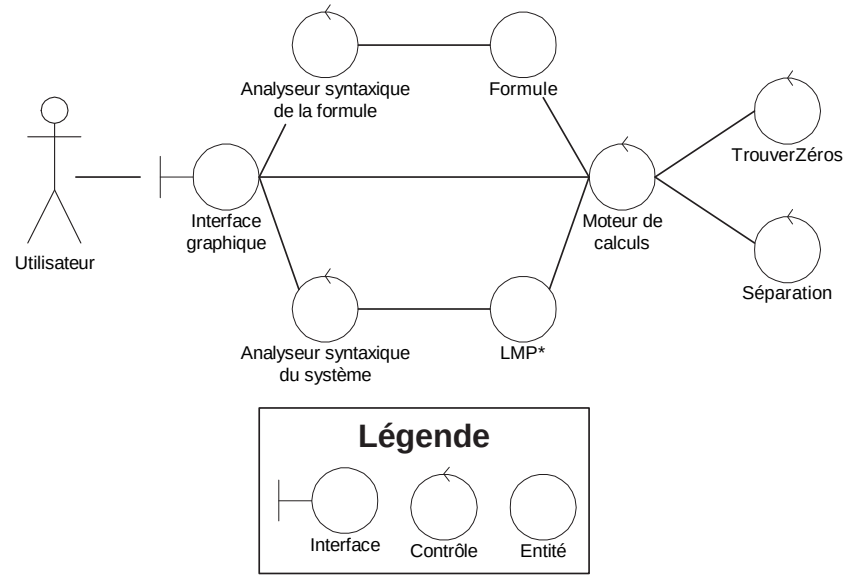

FIG. 4.2 – Le modèle d'analyse selon le formalisme de Jacobson.

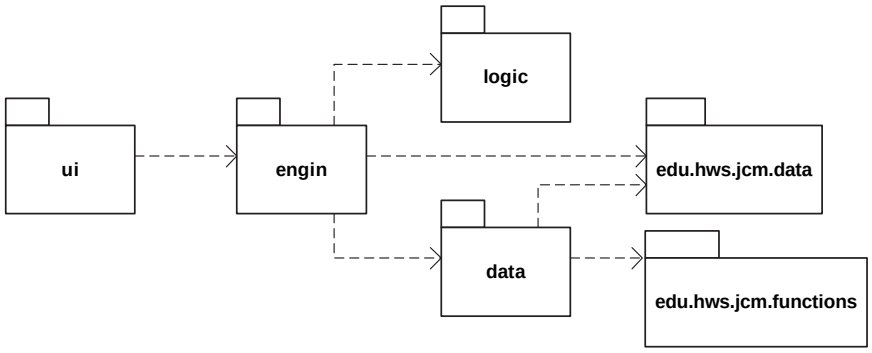

FIG. 4.3 – L'arborescence et les dépendances des paquetages

4.2 Architecture du model-checker

Nous donnons ici une vue d'ensemble de l'architecture et du découpage en paquetages du programme. Les paquetages et les classes qu'ils contiennent seront décrits plus en détails dans les sections suivantes.

Comme le montre la figure 4.3, le model-checker est découpé en six paquetage : deux récupérés du projet JCM[9] initié par M. David Eck[1] et quatre entièrement développés par l'auteur.

Les paquetages `edu.hws.jcm.data` et `edu.hws.jcm.functions` servent de base

[1]Pour plus d'information, visiter http ://math.hws.edu/javamath/

pour représenter et évaluer tout ce qui est constante, variable, expression mathématique, fonction, etc. Nous nous en servons surtout pour représenter et évaluer les fonctions de répartition. Le paquetage `data` contient toutes les classes nécessaires à représentation des données, comme `Interval` pour représenter un intervalle réel, `Transition`, `ProbSystem` pour représenter un système probabiliste et quelques autres classes utilitaires. Le paquetage `logic` sert à représenter et à manipuler des formules de logique, d'après la syntaxe donnée à la section 2.2. Entre autres, ce paquetage contient un analyseur syntaxique qui permet de créer une formule à partir d'une chaîne de caractère, comme par exemple « <a>[0] (safe AND (NOT <b>[0.5] T)) », qui représente la formule $\langle \mathsf{a} \rangle_0 \left(\mathtt{safe} \wedge \neg \langle \mathsf{b} \rangle_{1/2} \mathsf{T} \right)$. Le paquetage `engin` qui ne contient que la classe `Engin` est le paquetage principal, celui où se font tous les calculs sur les systèmes probabilistes en vue d'implémenter l'algorithme de vérification. Finalement, le paquetage `ui` contient les classes s'occupant de l'interface graphique pour l'utilisateur.

Dans ce qui suit, nous écrirons le nom des classes et des fonctions avec la police de caractère `typewriter`, afin que le lecteur saisisse bien que nous faisons référence à l'implémentation et non au concept mathématique ou informatique qui lui est généralement associé, comme par exemple la classe `Interval`. Parfois, lorsque la distinction ne sera pas nécessaire, nous écrirons en roman le nom de certains concepts pour y faire référence, tant au niveau mathématique qu'au niveau de l'implémentation en Java.

4.2.1 JCM

Le projet JCM [9] (*Java Components for Mathematics*) a pour but de développer un ensemble de composants logiciels mathématiques écrits dans le langage de programmation Java. Regroupées en quatre paquetages, les classes qui s'y trouvent forment une collection de composants servant à manipuler, évaluer et représenter graphiquement des expressions mathématiques. Pour nos besoins, nous nous servons surtout des classes `Constant`, `Variable`, `ExpressionProgram`, `SimpleFunction` et `TableFunction`. Les deux premières servent respectivement à représenter des constantes réelles(exemple « 3 ») et des variables (exemple « x ») prenant des valeurs réelles pour former des expressions comme $x + 3$. La troisième est en fait une implémentation d'un automate à pile pour évaluer des expressions mathématiques. Via la classe `edu.hws.jcm.data.Parser` qui sert d'analyseur syntaxique, on peut créer une instance de la classe `ExpressionProgram` à partir d'une chaîne de caractère. Il suffit d'enregistrer auprès du `Parser` utilisé les variables dont on se sert pour avoir la possibilité de traduire des chaînes de caractères en `ExpressionProgram` pouvant être évaluées par la suite. Par exemple, pour les fonctions probabilites des transitions, on enregistre les variables

x et y auprès du `Parser` pour permettre à l'usager de définir des fonctions comme $e^{(y-x-5)^2}$. Il est à noter que cet analyseur syntaxique possède déjà quelques constantes mathématique, pensons à e et π, et certaines fonctions, par exemple sin, cos, tan pour faciliter la saisie des expressions. Aussi, il est possible d'enregistrer des fonctions mathématique supplémentaires. Nous nous en sommes servi pour ajouter à l'analyseur la fonction `Normal` qui prend un paramètre et dont la valeur retournée correspond à celle donnée par la table normale standard (voir 4.2.2 plus bas). Pour implémenter cette fonction, nous avons utilisé la classe `TableFunction` qui définit une fonction par un ensemble de coordonnées dans le plan cartésien. Avec une instance de la classe `ExpressionProgram`, on peut construire une `SimpleFunction`, qui est une fonction à une variable, en donnant la `Variable` qui servira à évaluer la fonction. Le site Internet du projet[9] contient la documentation nécessaire pour utiliser leur code.

Nous avons choisi de récupérer ces classes du projet JCM parce qu'ils implémentent plusieurs fonctionnalités recherchées. Nous avons identifié au départ que les expressions mathématiques des fonctions probabilistes comporteraient des variables et des fonctions utilitaires (log, sin, exp, etc.) et que nous aurions besoin d'en ajouter des nouvelles. Entre autres, nous avons besoin de représenter la fonction de répartition d'une variable normale standard, définie d'après un ensemble de coordonnées dans le plan. Tous ces éléments se retrouvent dans JCM, dont la manière de calculer les expressions repose sur un automate à pile.

Il y a tout de même quelques limites à ce paquetage par rapport aux opérations recherchées. Par exemple, il est impossible d'évaluer partiellement une fonction à deux paramètres, comme il est requis pour calculer les probabilité, d'après la formule (3.4). Nous avons été en mesure de contourner ce problème en traduisant la fonction dans une chaîne de caractères et en remplaçant toutes les occurrences de la variable y par la valeur désirée. Aussi, la fonction d'évaluation de la classe `TableFunction`, qui sert à représenter la table normale standard, n'est pas très efficace. En effet, son temps d'exécution est de l'ordre de $O(n)$ où n est le nombre de points dans la table. Pourtant, il aurait été facile de programmer une telle fonction dont le temps aurait été dans l'ordre de $O(\log_2(n))$, en effectuant une recherche dichotomique sur les valeurs triées.

4.2.2 Le paquetage `data`

Le paquetage `data` sert surtout à représenter les principaux concepts pour stocker en mémoire un système probabiliste continu. Pratiquement aucun traitement n'est fait à ce niveau de l'application. La figure B.1 en annexe présente le diagramme UML des classes de ce paquetage.

Les classes `Interval` et `IntervalList`

Les classes `Interval` et `IntervalList` servent à représenter l'ensemble des états d'un système continu. Une instance de `Interval` est précisement un intervalle réel ayant des bornes qui peuvent être incluses ou non. On peut représenter aussi un singleton, par exemple $\{3\}$ par un `Interval` dont les deux bornes sont égales et incluses, ici $[3, 3]$. Un constructeur prenant une `String` permet aussi de construire un intervalle directement à partir d'une chaîne de caractère, pour faciliter le travail durant le traitement de ce qui est saisi à l'écran par l'utilisateur. La chaîne en question peut être soit un singleton ou un intervalle, tel qu'écrit plus haut.

La classe `IntervalList` sert à représenter une union d'intervalles disjoints, qui sont conservés en mémoire de façon ordonnée, dans une liste. Cette union d'intervalles sert d'ensemble de valeurs réelles afin d'implémenter les opérateurs d'union d'ensembles, d'intersection, de soustraction et de complément tel que le demande l'algorithme 3.3.1. Toutes ces opérations sont implémentées de manière à respecter les conventions mathématiques qui les concernent. Par exemple, le résultat de l'union d'un intervalle vide, $(0, 0)$, avec un intervalle quelconque donne ce dernier. La soustraction d'un intervalle par un autre donne toujours un ensemble inclus ou égal au premier. Pour faciliter l'utilisation de la classe `IntervalList`, nous avons implémenté un constructeur à partir d'une `String`. Pour l'utiliser correctement, il suffit de lui fournir une chaîne de caractères contenant des `Interval`s séparés par la lettre `U` en majuscule, pour union. Par exemple, la chaîne « `[0,5] U {8} U (-10,0)` » est valide pour ce constructeur. Il est à noter qu'il n'est pas nécessaire que les `Interval`s soient ordonnés. Étant donné que les intervalles sont conservés en ordre dans la liste, nous avons pu implémenter les opérations d'union, d'intersection et de soustraction de deux `IntervalList` en temps linéraire par rapport au nombre d'intervalles. De plus, la recherche pour déterminer si une valeur précise est dans une liste se fait en temps logarithmique, suivant l'algorithme de recherche dichotomique.

La Classe `Transition`

La classe `Transition` sert à conserver une transition d'un système probabiliste. On peut la voir comme un contenant, à la manière de la notation $F_{X \xrightarrow{\mathtt{a}} Y}(x, y)$, reliant la clé $(X, \mathtt{a}, Y)$ à l'expression définissant la fonction. Une `Transition` est construite à partir d'un `Interval` d'états de départ, d'un `Interval` d'états d'arrivé et d'une action, auxquels on associe une expression mathématique définissant la fonction de répartition des états, à la manière de la section 3.1.2. Il n'y a aucune précondition sur les différents

objets qui constituent une `Transition`. Les seules méthodes que cette classe contient sont des méthodes du type (*getXXXX*) pour récupérer les objets qu'elle regroupe. En somme, cette classe sert surtout à lier un intervalle de départ, un intervalle d'arrivée et une action à une fonction de répartition.

La classe `StandardNormalTable`

Cette classe sert à représenter et évaluer la fonction de répartition d'une variable normale standard. Elle hérite de la classe `TableFunction` de JCM afin faciliter l'évaluation d'une fonction définie par un ensemble de points du plan. La classe mère implémente une fonction d'interpolation pour l'évaluation, qui peut faire plusieurs types d'interpolation (directe, linéaire et cubique). Pour calculer une valeur à partir de la table normale, on utilise habituellement l'interpolation linéaire.

Afin d'éviter de programmer dans le code l'ensemble des valeurs de la table normale standard, ces dernières sont lues dans un fichier. La permière ligne du fichier contient le nombre de points et sur chacune des lignes suivantes, il y a la valeur en x, un espace et la valeur de $\Phi(x)$. Par défaut, le nom du fichier est « `StandardNormalTable.txt` ». Ainsi, un utilisateur du programme peut fournir sa table normale avec la précision qu'il désire.

On peut enregistrer une instance de cette classe auprès d'un `Parser` de JCM afin que la fonction $Normal(x)$ soit automatiquement disponible. Ainsi, le `Parser` sera en mesure de créer une expression à partir de la chaîne « `Normal((y-2)/0.1)` »

La classe `ProbSystem`

`ProbSystem` est la classe centrale du paquetage `data`. Elle sert à stocker en mémoire un LMP*. Pour y arriver, elle se sert de plusieurs table de dispersion du paquetage `java.util`, afin d'accélérer la recherche lors du traitement. Tout d'abord, il y a un `HashSet` pour les actions, un `TreeSet` pour les propositions atomiques et une `HashMap` pour conserver les associations entre une proposition atomique et l'ensemble des états où elle est vraie. Cette dernière sert en fait à stocker la fonction *label* d'un LMP*. Il y a aussi trois `HashMap` pour accélérer le calcul des probabilités. La première contient tous les X pour lesquels il est possible de faire une action donnée. La seconde contient tous les Y vers où il est possible d'aller partant d'un X donné et avec une action donnée. La troisième sert à récupérer la `Transition` qui est définie pour un X, une action et

un Y donnés. L'utilisation de table de dispersion accélère le traitement, puisque dans le calcul des probabilités, il ne sert à rien de considérer les X pour lesquels une action donnée est impossible, de même pour les Y.

Comme il n'y pas de traitement à ce niveau de l'application, la plupart des méthodes de `ProbSystem` servent à ajouter de l'information dans ses tables de dispersion ou bien à les interroger. Pour ajouter simplement de l'information, on peut utiliser la méthode

```
public void addTransition(Transition t) throws ProbSystemException
```

qui s'occupe de vérifier si la `Transition` peut être ajoutée. C'est possible si elle respecte les critères suivants :

- l'ensemble de départ doit être inclus dans l'ensemble des états du système ;
- l'ensemble de départ de la transition est soit un X du système ou bien n'intersecte pas les X qui sont déjà stockés ;
- De même pour l'ensemble d'arrivée de la transition ;
- il ne doit pas déjà y avoir une transition pour le même ensemble de départ, la même action et le même ensemble d'arrivée.

Si la nouvelle transition repecte ces critères, la méthode s'occupe de placer l'information dans les différentes tables de dispersion. Sinon, elle lance une `ProbSystemException`. Il est à noter qu'une instance de `ProbSystem` ne peut pas vérifier par elle-même le respect de la contrainte $0 \leq P_{\mathsf{a}}(x, S) < 1$, puisqu'il faut un moteur de calcul pour déterminer la fonction de probabilité et pour vérifier si l'inéquation est respectée.

La classe possède aussi une méthode, `toString()`, qui permet de décrire le LMP* stocké dans une chaîne de caractère. Le format de sortie dans la chaîne respecte ce qui est spécifié à la section 4.3.3, pour faciliter l'enregistrement dans un fichier.

4.2.3 Le paquetage `logic`

Le paquetage `logic` vise à implémenter la syntaxe de la logique de la section 2.2. Comme une formule de logique peut comporter de 0 à deux sous-formules, nous avons défini les opérateur d'après une même classe dont ils héritent tous certaines propriétés. Cette super-classe, `Formula`, est abstraite et définit ce qu'ont en commun tous les opérateurs ainsi que des méthodes virtuelles devant être spécifiées dans les sous-classes. Elle possède un tableau de sous-formules, une méthode `toString` générique et une

méthode `depth` qui revoie la profondeur, telle que définie par

$$\begin{aligned}
\text{depth}(\mathsf{T}) &= 0 \\
\text{depth}(\mathtt{p}) &= 0 \\
\text{depth}(\neg\phi) &= 1 + \text{depth}(\phi) \\
\text{depth}(\langle \mathtt{a} \rangle_q \phi) &= 1 + \text{depth}(\phi) \\
\text{depth}(\phi \wedge \psi) &= 1 + \max(\text{depth}(\phi), \text{depth}(\psi)) \\
\text{depth}(\phi \vee \psi) &= 1 + \max(\text{depth}(\phi), \text{depth}(\psi)) \ .
\end{aligned}$$

L'implémentation de cette méthode ne tient compte que du tableau de sous-formules. S'il est vide, la méthode renvoie 0, sinon elle retourne 1 plus le maximum de la profondeur des sous-formules.

Pour représenter les opérateurs de la logiques, il y a les classes `True` pour T, `AtomicProposition` pour les propositions atomiques, `LogicalNot` pour $\neg$, `LogicalAnd` pour $\wedge$, `LogicalOr` pour $\vee$ et `ProbabilisticNext` pour $\langle \mathtt{a} \rangle_q$. Toutes ces classes héritent de `Formula` et redéfinissent ses méthodes abstraites. Comme les formules sont données dans des chaînes de caractères, chacune des classes sauf `AtomicProposition` et `ProbabilisticNext` possède une constante `TYPE` qui contient la chaîne associée à chaque opérateur. La chaîne pour l'opérateur T est « `T` », pour $\neg$, « `NOT` », pour $\wedge$, « `AND` » et pour $\vee$, « `OR` ».

Pour compléter le paquetage, la classe `LogicParser` sert d'analyseur syntaxique pour les formules. On doit lui fournir au préalable l'ensemble des propositions atomiques, dans des `String` pour qu'elle puisse les reconnaître lors de la lecture. La classe possède une méthode

```
public Formula parse(String s) throws LogicParserException
```

pour effectuer l'analyse syntaxique. Dans une chaîne de caractère, une proposition atomique doit être un mot débutant par une lettre minuscule et l'opérateur $\langle \mathtt{a} \rangle_q$ doit être donné selon le format « `<a>[q]` ».

4.2.4 Le paquetage `engin`

Le paquetage `engin` contient la classe `Engin`, qui est le moteur de calculs de l'application. Tous les algorithmes de la section 3.3.1 sur la vérification directe sont implémentés par des méthodes de la classe `Engin`. Cette dernière s'occupe aussi de gérer l'analyse syntaxique des chaînes de caractères obtenues de l'interface graphique. Elle possède une instance de l'analyseur syntaxique du paquetage `edu.hws.jcm.data` afin de traiter

les chaînes contenant les fonctions de probabilités. Elle possède aussi une instance de l'analyseur du paquetage `logic`. L'utilisateur n'a donc qu'à s'adresser à une instance de `Engin` pour générer les objets associés aux expressions qu'il saisit à l'écran ou bien inscrite dans un fichier. La classe possède d'ailleurs une méthode permettant de charger un `ProbSystem` à partir d'un fichier respectant le formalisme donné à la section 4.3.3. Comme la méthode `toString()` de `ProbSystem` permet d'écrire un LMP* dans une `String` suivant le même formalisme, il est possible d'enregistrer et de lire facilement un système à partir d'un fichier.

La classe `Engin` implémente directement l'algorithmes 3.3.1 qui calcule $[\![\phi]\!]$ par la méthode

```
public IntervalList getStatesVerifing(Formula f) .
```

Cette dernière fait à la méthode

```
public HashMap computeProbabilities(IntervalList list, String act)
```

qui implémente l'algorithme 3.3.2 pour calculer la fonction de probabilité d'aller vers les états dans `list` avec l'action `action`. La `HashMap` retournée sert à stocker la fonction à branche définissant la probabilité : les clés sont des `Interval` pour lesquels il est possible de récupérer une `Expression` définissant la fonction de probabilité sur cet intervalle. La méthode `getStatesVerifing` fait aussi appel à

```
public IntervalList separation(HashMap fct,Variable var,double q)
```

qui implémente l'algorithme 3.3.3. Cette méthode détermine pour quelles valeurs de `var` la fonction à branche `fct` est strictement supérieure à `q`.

Pour implémenter l'algorithme 3.3.3, nous avons besoin d'une méthode qui trouve les zéros d'une fonction. La section 4.3.2 explique comment a été implémentée la fonction

```
ArrayList findZeroes(Expression exp, Variable var,
                     Interval inter, double threshold)
```

qui recherche les points d'intersection de l'expression `exp` avec la fonction constante `threshold`. L'expression est évaluée à l'aide de la variable `var` sur la fermeture de l'intervalle `inter`. La méthode retourne, dans une `ArrayList` tous les points d'intersection qu'elle a trouvés.

L'utilisateur n'a pas à vérifier lui-même si un `ProbSystem` respecte la contrainte $0 \leq P_{\mathsf{a}}(x, S) < 1$. En effet, la méthode

```
public boolean verifySystemConstraints() throws Exception
```

renvoie `true` si le système stocké dans l'instance courante de `Engin` vérifie la contrainte. Sinon, la méthode lance une `Exception` qui contient alors un message décrivant pour quelle action et pour quels x l'équation $0 \leq P_{\mathsf{a}}(x, S) < 1$ n'est pas vérifiée.

4.2.5 Tests unitaires

Ceux qui ont/auront la chance de consulter le code source verront que chacun des paquetages développés contient une classe Test, excepetés ceux de JCM. Celle-ci contient un ensemble de tests unitaires sur les méthodes des classes du paquetage en question et contient une méthode `main(String arg[])` pour la démarrer. Si un test unitaire échoue, c'est qu'une assertion a été violée. Un message décrivant l'assertion apparaîtra alors dans la console.

Aussi, la plupart des classes possèdent une méthode

```
public boolean verifierInvariants()
```

qui permet de vérifier si les invariants de la classe sont respectés. Les invariants sont des prédicats sur des attributs qui doivent être vrais tout au long de la vie des objets d'une classe. Par exemple, pour la classe `Interval`, un des invariants est que la borne supérieure doit toujours être plus grande ou égale à la borne inférieure. À aucun moment durant la vie d'un `Interval`, il ne sera toléré que ce prédicat soit faux. L'approche par invariants permet de transposer des contraintes au niveau logique à des prédicats au niveau de l'implémentation. La manière d'utiliser la méthode `verifierInvariants()` est de l'appeler à la fin des méthodes qui modifient l'état de l'objet courant, de la manière suivante :

```
assert verifierInvariants();
```

Si les assertions sont activées à l'exécution de la machine Java, toute méthode violant un invariant créera une AssertionException avec un texte descriptif dans la console. Si les assertions sont désactivées, aucun appel ne sera fait à la méthode, qui retournerait simplement `true` si jamais elle étati appelée directement. Nous avons utilisé les invariants pour faciliter le développement de l'application, puisqu'elles permettent de trouver plus facilement la source d'une erreur.

4.3 Choix faits pour l'implémentation

Pour implémenter les algorithmes nécessaires à la vérification directe, nous avons dû faire des choix, compte tenu du langage de programmation choisi, de la bibliothèque de classes de JCM et de certains problèmes inhérents à la programmation avec les nombres à virgule flotante de la norme IEEE 754. Dans les pages qui suivent, nous reviendrons sur le choix des fonctions de répartition et nous présenterons l'implémentation d'un algorithme calculant les zéros d'une fonction ainsi que le format de fichier que l'on doit respecter pour donner un LMP* au vérificateur formel.

4.3.1 Fonctions de densité ou fonctions de répartition ?

Le premier choix que nous avons eu à faire a été de fixer le langage de description des fonctions de transitions. Aux sections 3.1.1 et 3.1.2, nous avons décrit deux langages, pratiquement semblables, sauf en ce qui concerne la manière de définir la distribution des probabilités de la variable y : le premier langage utilise les fonctions de densité, alors que le second utilise les fonctions de répartition. À la section 3.1.3, nous avons discuté des différences entre ces langages, remarquant que les fonctions de répartition ont quelques avantages sur les fonctions de densité, de notre point de vue. Entre autres, l'utilisation des fonctions de répartition évite le calcul de plusieurs intégrales pour déterminer $P_{\mathsf{a}}(x, E)$. Aussi, pour certain cas pathologiques, l'utilisation des fonctions de répartition permet de calculer la valeur d'une fonction d'après une table, qui peut être aussi précise que l'on veut. Cet table doit être bâtie en pré-traitement, mais elle peut être réutilisée pour toutes les vérifications subséquentes, en autant qu'elle possède la précision désirée.

Donc, nous avons fixé que les transitions des systèmes probabilistes devrait être représentées avec le langage utilisant les fonctions de répartition, de la section 3.1.2.

4.3.2 La fonction `findZeroes`

La fonction `findZeroes` sert à trouver les points d'intersection d'une fonction à une variable sur la fermeture avec une fonction constante, sur d'un intervalle donné. Sa signature est

```
ArrayList findZeroes(Expression exp, Variable var,
                     Interval inter, double threshold)
```

La fonction prend en paramètre une expression mathématique représentant la fonction, sa variable, l'intervalle sur lequel elle doit trouver les points d'intersection et la valeur de la fonction constante. Elle retourne dans une *ArrayList*, qui est en fait un vecteur, toutes les valeurs pour lesquelles l'expression moins le seuil vaut zéro.

L'algorithme

Il existe plusieurs algorithmes pour implémenter la recherche numérique des zéros d'une fonction. La plupart sont des méthodes itératives et nécessite une valeur en x de la fonction $F(x)$ proche d'un zéro pour débuter la recherche. Pour y arriver, la fonction `findZeroes` commence par découper l'intervalle `inter` en plusieurs segments et évalue

la fonction à chaque borne des segments. Si le signe de la fonction change entre les bornes d'un segment, c'est que $f(x)$ possède un zéro sur ce segment. Alors, la recherche proprement dite d'un zéro peut débuter, sur ce segment.

La méthode numérique la plus connue pour trouver un zéro d'une fonction est celle de Newton. Elle calcule par approximation successive la valeur d'un zéro, se basant sur la formule

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

où x_0 est une valeur de départ, dans notre cas l'une des bornes du segment où le changement de signe a été détecté. La méthode se termine lorsque la différence entre x_{n+1} et x_n n'est plus significative, d'après un ϵ donné, ou bien lorsque n dépasse une valeur pré-déterminée.

La convergence de cette méthode est au moins quadratique. Tel que défini dans [10], on dit qu'une méthode des points fixes *converge à l'ordre p* si :

$$|e_{n+1}| \approx C|e_n|^p$$

où C est une constante et $e_n = x_n - r$ est la différence entre la $n^{\text{ième}}$ approximation et la racine r. Ce type d'analyse d'une méthode numérique nous indique sa *vitesse* de convergence : plus l'exposant p est élevé, plus la méthode converge rapidement vers la valeur de la racine.

Donc, la méthode de Newton est rapide, mais elle demande de calculer la dérivée de la fonction $f(x)$. L'implémentation d'une machine de dérivation symbolique est très complexe et nous préférons l'éviter. Nous serions alors obligés de nous rabattre sur la dérivation numérique. Il existe des méthodes numériques pour calculer la dérivée, mais ceci amène une erreur supplémentaire à la méthode de Newton et augmente le temps de calcul de façon considérable.

Alors, pour éviter cette surcharge, on utilise plutôt la méthode de la sécante, qui approxime directement la dérivée par l'expression

$$f'(x_n) \approx \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}} \ .$$

En fait, on remplace la dérivée par la pente de la sécante passant par les points $(x_{n-1}, f(x_{n-1}))$ et $(x_n, f(x_n))$. L'expression pour x_{n+1} devient

$$x_{n+1} = x_n - \frac{f(x_n)(x_n - x_{n-1})}{f(x_n) - f(x_{n-1})} \ .$$

Une différence notable par rapport à la méthode de Newton est que la méthode de la sécante est une méthode à *deux pas*, c'est-à-dire nécessitant deux valeurs initiales. Dans notre cas, cela ne pose pas de problème puisqu'on peut prendre les valeurs des deux bornes du segment où le changement de signe a été détecté.

L'algorithme associée à la méthode de la sécante est le suivant, adapté de [10]. Il prend en entrées la fonction $f(x)$, la précision $\epsilon > 0$, le nombre d'itérations maximal N et deux valeurs de départ, x_0 et x_1.

Algorithme 4.3.1
MethodeSecante(Fonction $f(x)$, Réel ϵ, Naturel N, Réel x_0, Réel x_1)
$n := 2$
Tant que $(n-1) \leq N$ faire
  $x_{n+1} := x_n - [f(x_n)(x_n - x_{n-1})] \,/\, [f(x_n) - f(x_{n-1})]$
  Si $x_{n+1} = 0$ et $|x_n| < \epsilon$
    {*Zéro trouvé*}
    Retourner x_{n+1}
  Sinon
    Si $|\frac{x_{n+1}-x_n}{x_{n+1}}| < \epsilon$
      {*Zéro trouvé*}
      Retourner x_{n+1}
  Fin Si
Fin Tant que
{*Zéro non-trouvé. Retourner une valeur hors borne* [2]}
Retourner **NUL**
□

Pour l'implémentation de cet algorithme, la classe `Engin` possède trois atributs : `precision`, `nbIteration` et `segmentation`. Le premier représente le ϵ de l'algorithme, c'est-à-dire la précision des zéros. Le second représente le N, le nombre maximal d'itération et le dernier contient le nombre de segments qu'il faut utiliser pour découper le domaine de la fonction. Rappelons que la fonction est évaluée à chaque borne des segments, pour détecter les changements de signe.

Le taux de convergence de la méthode de la sécante est $p = \frac{1+\sqrt{5}}{2} \approx 1.618$. Comme ce taux est entre 1 et 2, cette méthode est plus lente que celle de Newton, mais est tout de même mieux qu'une méthode dont la convergence est linéraire.

[2] L'implémentation dictera comment ce cas sera géré. En java, il est possible de retourner la constante `double` NaN, qui est une valeur hors borne. On pourrait aussi gérer ce cas avec une `Exception`.

Nous avons implémenté cet algorithme en Java et nous l'avons testé. La prochaine section détaille quelques problèmes que nous avons eus avec la programmation numérique en nombres à virgule flotante.

La précision de calcul et les nombres à virgule flotante

En implémentant l'algorithme 4.3.1, nous avons redécouvert certains problèmes liés à la programmation en nombre à virgule flotante. Ces problèmes sont bien connus [11] et sont surtout dû à la troncature des nombres. Les deux causes principales de troncatures sont les suivantes.

Premièrement, pour mettre une infinité de nombres réels dans un nombre fini de bits, il faut nécessairement une représentation approximative. En fait, certains nombres réels peuvent ne pas avoir d'équivalent dans la représentation choisie par le programmeur. En Java, la norme IEEE 754[14] émise par l'organisme *Institute of Electrical and Electronics Engineers* (*IEEE*) détermine la manière dont les nombres réels sont stockés en mémoire, avec un nombre fixe de bits. Avec cette norme, le nombre 0.1 ne peut être représenté exactement et ce, parce que ce nombre a un développement périodique en nombre binaire, tout comme 1/3 n'a pas d'équivalent en nombres décimaux.

Deuxièmement, il y a le problème que, étant donné un nombre de bits fixé, la plupart des opérations arithmétiques peuvent produire des nombres qui ne peuvent pas être représentés précisément en utilisant ce nombre de bit. L'addition, la soustraction et la multiplication peuvent causer un débordement, alors que la division peut demander un nombre supérieur de bit pour la précision du résultat.

Pour illustrer le problème de troncature, nous allons montrer le résultat de l'exécution d'un programme simple, en Java. Voici le code :

```
double n=0.1;
for(int i =0; i<1000;i++){
   System.out.println("n : " + n);
   n+=0.1;
}
```

Une partie du résultat de l'exécution du code précédent se trouve à la figure 4.4. On voit que la troisième ligne, qui est supposée contenir 0.3, affiche plutôt un nombre à 17 décimales, suggérant qu'il y a eu un décalage dans les calculs quelque part.

Afin d'éviter le problème de troncature, il possible de définir et d'utiliser une classe qui conserve les nombres réels avec toute la précision nécessaire, utilisant un nombre

```
n : 0.1
n : 0.2
n : 0.30000000000000004
n : 0.4
n : 0.5
n : 0.6
n : 0.7
n : 0.7999999999999999
n : 0.8999999999999999
n : 0.9999999999999999
```

FIG. 4.4 – $0.1 \neq 0.1$

de bit variable jusqu'à une certaine limite. Pour notre implémentation, nous avons choisi d'utiliser la classe `java.util.BigDecimal`, qui stocke les nombres réels dans un tableau de `int`, où chacune des cases contient un chiffre de la représentation décimale. Comme ce tableau n'a pas de taille prédéfinie, il peut être arbitrairement grand. Les deux problèmes majeurs avec l'utilisation d'une telle classe sont l'augmentation du temps de calcul et l'augmentation de la mémoire nécessaire. Nous avons donc décider d'implémenter la fonction `findZeroes` de deux manières : avec le type primitif `double` basé sur la norme IEEE 754 et avec la classe `BigDecimal`. Ainsi, l'utilisateur peut choisir s'il préfère minimiser le temps de calcul ou bien maximiser la précision. La section 4.5 présente une analyse des résultats de l'outil. Entre autres, on y compare la vitesse d'exécution des deux versions de la fonction `findZeroes`.

4.3.3 Format du fichier d'entrée et de sortie

Voici le format de fichier en entrée et en sortie, pour un système probabiliste continu. Les expressions entre accolades représentent toute expression valide du type. Les espaces, tabulations et retours de chariot ne sont pas nécessaires, mais ils sont là pour rendre la lecture plus facile. Pour insérer des commentaires dans le fichier, on ajoute `//` au début d'une ligne et alors celle-ci est ignorée.

```
// Pour des commentaires, au début d'une ligne
states : {IntervalList};
init : {double};

X : {Interval}, {Interval}, ... ;
Y : {Interval}, {Interval}, ... ;
```

```
label
   {String} : {IntervalList};
   {String} : {IntervalList};
   ...
endlabel

transition
   {Interval} -[{String}]-> {Expression} : {Interval};
   {Interval} -[{String}]-> {Expression} : {Interval};
   ...
endtransition
```

Comme nous l'avons mentionné à la section 4.3.1, les expressions doivent définir des fonctions de répartition. Aussi, il est possible d'utiliser des fonctions de base (sin, cos, tan, log, exp , Normal (table normale standard, voir 4.2.2), etc.).

Si nous reprenons l'exemple 3.1.9, nous aurons le fichier suivant :

```
// Un système probabiliste
states : [0,3] U {4} U {5} ;
init : 1;

X: [0,1],(1,2],(2,3];
Y: {0},(0,1),{1},(1,2],(2,3],{4},{5};

label
   safe : [1,3];
   exit : {4} U {5};
endlabel

transition
   [0,1] -[a]-> x          :{0};
   [0,1] -[a]-> y/4        :(0,1);
   [0,1] -[a]-> y/4        :(1,2];
   [0,1] -[a]-> (1-x)/4    :{1};
   [0,1] -[a]-> x(y-2)/4   :(2,3];
   (1,2] -[a]-> 1          :{4};
   (2,3] -[b]-> 1          :{5};
endtransition
```

Ce format permet de définir complètement un LMP*. Un autre exemple de LMP* décrit selon ce format se trouve à l'annexe C.

4.4 Analyse de l'erreur

Il y a quelques sources d'erreur numérique dans notre implémentation. Dans ce qui suit, nous tenterons des les identifier et de les quantifier.

Il y a trois fonctionnalités du vérificateur qui peuvent donner des réponses légèrement imprécises : la vérification de la contrainte, le calcul de $[\![\phi]\!]$ et la vérification de $\mathcal{S} \models \phi$. D'après la manière dont ils sont implémentés, les erreurs proviennent surtout de la procédure qui calcule $P_{\mathsf{a}}(x, E)$ et de celle qui recherche les zéros d'une fonction. Il est à noter que si ϕ ne comporte pas de sous-formule du type $\langle \mathsf{a} \rangle_q \psi$, le résultat de $[\![\phi]\!]$ et de $\mathcal{S} \models \phi$ est précis, puisqu'aucun calcul arithmétique n'est nécessaire.

Pour la validation de la contrainte du système et pour la vérification directe d'une formule comportant au moins une sous-formule du type $\langle \mathsf{a} \rangle_q \phi$, des erreurs de calcul peuvent provenir de plusieurs sources. L'évaluation d'une expression mathématique, comme une fonction de transition probabiliste, est aussi précise que l'arithmétique sur le type primitif `double` basée sur la norme IEEE 754. Le lecteur trouvera dans [11] des informations complémentaires sur le calcul de la précision des principaux opérateurs arithmétiques de la norme IEEE 754. En effet, les fonctions probabilistes sont conservées et évaluées par la classe `ExpressionProgram` de JCM et celle-ci se base sur les `double` pour effectuer l'évaluation. Les erreurs de calcul peuvent provenir aussi de l'utilisation de la table normale, dans la définition des fonctions de transition. L'évaluation de `Normal(z)` se fait en recherchant la valeur parmi celles stockées. Si elle y est, l'imprécision ne provient que de la table et du type `double`. Si elle n'y est pas, il y a interpolation linéaire. L'erreur peut d'interpolation être estimée à $h^2/8$ [10], où h est la distance entre les valeurs en x de la table. Pour la table fournie avec le programme, $h = 0.01$, ce qui nous permet d'estimer l'erreur d'interpolation à 0.0000125.

À la section 4.3.2 nous avons présenté la fonction `findZeroes`, qui utilise le partionnement en segment et la méthode de la sécante pour trouver les zéros d'une fonction sur un domaine donné. Lorsqu'elle trouve un zéro, nous savons que la valeur est précise à la valeur de `precision` près[10], un des paramètres de la classe `Engin`. Par contre, il est possible que la fonction ne découvre pas un zéro, pour deux raisons. La première est que le nombre d'itérations fait par la méthode de la sécante n'est pas suffisant. Dans ce cas, la valeur approximée x_n est retournée, valeur qui est sûrement imprécise. Pour contour-

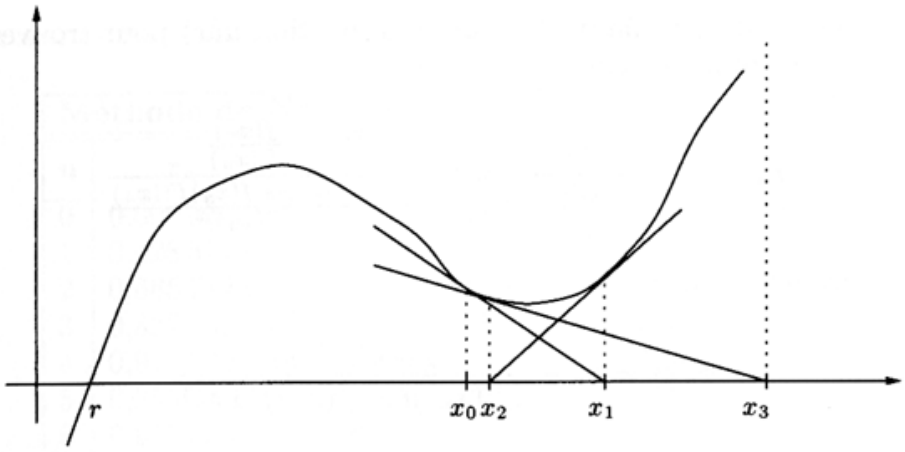

FIG. 4.5 – Un cas pathologique de la méthode de la sécante.

ner ce problème, il est conseiller d'augmenter la valeur du paramètre `nbIteration` de la classe `Engin`. La deuxième raison est qu'aucun changement de signe a été détecté sur le segment où est situé le zéro. Par exemple, pensons au cas où la racine double de la fonction $f(x) = x^2$ est strictement contenue dans un segment. La valeur aux deux bornes du segment serait alors positives et la fonction poursuiverait sa recherche avec le prochain segment. Même en lançant la méthode de la sécante sur l'intervalle [-1,1], la procédure ne convergerait pas parce qu'il y aurait une division par zéro. En fait, il existe des cas pathologique de la méthode de la sécante. La figure 4.5, tirée de [10] en présente un exemple.

Pour illustrer la précision des calculs, nous proposons de présenter quelques exemples de résultats obtenus avec l'implémentation, afin de documenter son comportement.

Nous n'avons pas remarqué d'erreur de la part de la fonction `computeProbabilities`, qui détermine l'expression de la fonction $P_{\mathsf{a}}(x, E)$, à partir des tests que nous avons effectués. Par exemple, le programme est en mesure de calculer précisément l'expression $P_{\mathsf{a}}(x, E)$ pour $E = S$, $E = \{0\}$, $E = (2, 3]$, $E = (0, 1)$, $E = [0, 2)$ et $E = (0, 2]$ tel que nous l'avons montré à l'exemple 2.1.7. Il n'y a que dans l'évaluation en `double` de l'expression que nous avons détecté un manque de précision.

Avec ce même exemple, exprimé avec des fonctions de répartition (voir l'exemple 3.1.9), on remarque quelques erreurs de précision. Par exemple, nous avons déjà montré que

$$P_{\mathsf{a}}(x, S) = \frac{3 + x}{4}$$

pour $x \in [0, 1]$. Si on demande au programme de calculer les états qui satisfont la

formule $\langle \mathsf{a} \rangle_{0.85} \mathsf{T}$ en utilisant les `double`, il retourne la réponse

$$(0.3999999999999998, 1.0] \ .$$

Si on utilise les `BigDecimal`, la réponse retournée est plutôt $(0.4, 1.0]$, ce qui exact d'après l'équation ci-haut. Donc, on remarque qu'il y a un manque de précision avec les `double`.

Nous avons aussi testé directement la fonction `findZeroes`, avec quelques expressions, comme x^2 et $\sin(x)$. Nous avons remarqué que la version de `findZeroes` utilisant des `double` peut ne pas trouver tous les zéros. Par exemple, cette version ne trouve pas la racine $x = 0$ de la fonction $f(x) = x^2$ et ne trouve pas tous les zéros de $\sin(x)$ sur $[0, 3\pi]$. La version de `findZeroes` utilisant des `BigDecimal` semble plus fiable, puisqu'elle est en mesure de déterminer toutes les racines des deux fonctions mentionnées précédemment.

4.5 Analyse des résultats et retour sur l'étude de cas

Dans cette section, nous allons présenter quelques tests que nous avons effectués sur notre implémentation de la vérification directe. Nous présenterons le contexte dans lequel nous avons fait ces tests pour ensuite décrire et analyser les résultats que nous avons obtenus.

Les quatre tests que nous présentons ici ont été effectués sur deux exemples. Le premier est l'exemple 3.1.9 qui nous a servi principalement pour illustrer notre langage et le second est celui de la section 3.2.1 sur le système de contrôle de la température. Sur chaque exemple, nous avons mesuré le temps de vérification de la contrainte (3.5) qui vérifie que $0 \leq P_{\mathsf{a}}(x, S) < 1$. Nous avons aussi mesuré le temps et la précision du calcul de $[\![\phi]\!]$ pour une formule ϕ, sur chacun des exemples. Sur le premier exemple, nous avons testé la formule $\phi_1 = \langle \mathsf{a} \rangle_{0.8} \mathsf{T}$, pour laquelle $[\![\phi_1]\!] = (0.2, 2]$. Sur le deuxième, nous avons testé la formule

$$\phi_2 = \langle \mathsf{a} \rangle_0 \langle \mathsf{a} \rangle_0 \langle \mathsf{a} \rangle_0 \langle \mathsf{a} \rangle_0 \langle \mathsf{a} \rangle_0 \langle \mathsf{a} \rangle_0 \langle \mathsf{a} \rangle_0 \langle \mathsf{a} \rangle_0 \langle \mathsf{a} \rangle_0 \langle \mathsf{a} \rangle_0 \mathsf{T}$$

pour laquelle $[\![\phi_2]\!] = [0, 30] \cup [200, 230]$. D'une manière générale, les temps de calcul ne représentent pas pour nous une mesure absolue de la performance de notre programme. Nous nous en servons plutôt comme un outil pour évaluer l'impact de la variation de certains paramètres de la classe `Engin`.

```
Engin engin  = new Engin();

// ... Ajustement des paramètres de l'engin ici
// ... Chargement du ProbSystem ici

long debut = 0;
long fin = 0;
long duree = 0;
boolean rep;


debut = System.currentTimeMillis(); //--- Heure au début du test

rep = engin.verifySystemConstraints();

fin = System.currentTimeMillis();   //--- Heure à la fin du test
duree = (fin - debut);

System.out.println(duree+"\t"+rep);
//...
```

FIG. 4.6 – Exemple du code servant à tester le programme.

Pour chaque test, nous avons effectué vingt exécutions complètes d'un petit programme. Ce programme a été redémarré à chaque fois, pour être certain que l'utilisation de la mémoire et le temps de calcul n'étaient pas optimisés par des exécution successives du même code. Nous avons effectué ces tests sur un ordinateur muni d'un processeur Intel Pentium 4 avec 1 gigaoctet de mémoire RDRAM, sur Windows 2000 SP4. Le programme de tests que nous avons écrit n'utilise pas l'interface graphique du vérificateur formel. Nous avons préféré utiliser directement la classe `Engin`, pour mesurer le plus précisément possible l'exécution, sans tenir compte du traitement fait pour gérer l'affichage graphique. La figure 4.6 montre le patron que nous avons utilisé pour faire nos tests. Ce programme ne fait qu'instancier un objet de la classe `Engin`, charger en mémoire un ProbSystem et exécuter une vérification. L'heure, en milisecondes, est prise avant et après l'appel d'une méthode de `Engin`. Les quatre tests ci-dessous présentent une moyenne des temps d'exécution ainsi que les réponses obtenues, en utilisant des `double` et en utilisant des `BigDecimal`. Pour chacune des vérifications, nous avons fait varier la précision et le nombre de segmentation pour la recherche des zéros des fonctions de probabilités. Le nombre de segments a été fixé à 1000 pour étudier l'impact de la variation de la précision alors que la précision a été fixée à 10^{-8} pour étudier la variation de la segmentation.

Lors de chacun des tests de la vérication de la contrainte, la réponse retournée par le programme a été `true`, pour les deux exemples. La figure 4.7 présente les temps de calcul obtenus sur l'exemple 3.1.9, et la figure 4.8, ceux obtenus sur l'exemple de

Temps (ms) de vérification de la Contrainte		
Précision	avec des double	avec des BigDecimal
1,00E-04	47,00	196,48
1,00E-08	47,00	195,71
1,00E-16	47,00	225,38
1,00E-25	47,00	231,43
Segmentation		
100	31,29	74,43
1000	55,86	200,19
10000	217,19	814,00
100000	2184,43	6717,38

FIG. 4.7 – Résultats des tests de la vérification de la contrainte sur l'exemple 3.1.9.

Temps (ms) de vérification de la Contrainte		
Précision	avec des double	avec des BigDecimal
1,00E-04	276,67	677,05
1,00E-08	279,00	674,86
1,00E-16	279,67	775,33
1,00E-25	282,05	793,10
Segmentation		
100	103,38	220,33
1000	279,00	680,62
10000	1962,00	5085,00
100000	19441,76	48642,38

FIG. 4.8 – Résultats des tests de la vérification de la contrainte sur l'exemple de la section 3.2.1.

la section 3.2.1. La figure 4.9 présente les résultats obtenus pour le calcul de $[\![\phi_1]\!]$ sur l'exemple 3.1.9 et la figure 4.10 ceux pour $[\![\phi_2]\!] \cap [0, 40]$. Nous rappelons que les temps sont la moyenne des temps de calcul sur 20 exécutions.

Les résultats semblent indiquer que la variation de la précision influence peu le temps de calcul et la réponse, qu'on utilise des `double` ou des `BigDecimal`, que ce soit pour la vérification de la contrainte ou pour le calcul des états qui satisfont une formule donnée. En effet, pour chacun des quatre tests, on remarque que le temps de calcul sont sensiblement les mêmes, pour des précisions de 10^{-4}, 10^{-8}, 10^{-16} et 10^{-25}. Pour les deux derniers tests, on remarque que les réponses retournées par le programme ne sont pas modifiées par une variation de la précision. Donc, pour nos tests, la précision n'influence pas vraiment le temps de calcul et ni les réponses. Nous ne nous attendions pas à cela. Le phénomène s'explique probablement par le fait que la méthode de la sécante, qui est

Calcul des états qui satisfont <a>[0.8] T				
	avec des double		*avec des BigDecimal*	
Précision	Temps (ms)	Réponse	Temps (ms)	Réponse
1,00E-04	34,24	(0.20000000000000015,2.0]	149,62	(0.2,2.0]
1,00E-08	33,48	(0.20000000000000015,2.0]	148,14	(0.2,2.0]
1,00E-16	32,76	(0.20000000000000015,2.0]	171,86	(0.2,2.0]
1,00E-25	32,71	(0.20000000000000015,2.0]	177,29	(0.2,2.0]
Segmentation				
100	24,48	(0.20000000000000004,2.0]	51,29	(0.2,2.0]
1000	35,62	(0.20000000000000015,2.0]	147,29	(0.2,2.0]
10000	111,67	(0.20000000000000007,2.0]	489,00	(0.2,2.0]
100000	1298,48	(0.20000000000000018,2.0]	3517,10	(0.2,2.0]

FIG. 4.9 – Résultats de l'évaluation de $[\![\phi_1]\!]$ sur l'exemple 3.1.9.

Calcul des états qui satisfont <a>[0] (10 fois) T				
	avec des double		*avec des BigDecimal*	
Précision	Temps (ms)	Réponse restreinte à [0,40]	Temps (ms)	Réponse
1,00E-04	396,48	[0.0,19.67999999999967)	1045,19	[0.0,19.68)
1,00E-08	399,62	[0.0,19.67999999999967)	1103,33	[0.0,19.68)
1,00E-16	395,19	[0.0,19.67999999999967)	1372,71	[0.0,19.68)
1,00E-25	398,19	[0.0,19.67999999999967)	1384,62	[0.0,19.68)
Segmentation				
100	124,29	[0.0,19.99999999999996)	289,38	[0.0,20.0)
1000	400,33	[0.0,19.67999999999967)	1104,76	[0.0,19.68)
10000	3013,48	[0.0,19.637999999999995)	9279,14	[0.0,19.638)
100000	30028,20	[0.0,19.63419999998092)	90119,33	[0.0,19.6342)

FIG. 4.10 – Résultats de l'évaluation de $[\![\phi_2]\!]$ sur $[0, 40]$ pour l'exemple de la section 3.2.1.

le seul bout de code influencé par une variation de la précision, converge rapidement vers les zéros. La condition d'arrêt est vraie lorsque l'erreur relative de x_{n+1} et x_n est inférieure à la précision.

La variation du nombre de segments, quant à elle, fait varier le temps de calcul de façon significative. Dans la recherche des zéros, le nombre de vérification de changement de signe est directement proportionnel au nombre de segments, d'où la conjecture que le temps de calcul est au moins d'ordre linéaire par rapport au nombre de segments. Pour le premier système, on remarque à la figure 4.9 que la variation du nombre de segments influence un peu les résultats, dans le cas des `double`. Si on arrondit les réponses à la précision près (10^{-8}), on s'apperçoit par contre que ce sont les même valeurs et que les différences ne sont pas significatives. Les résultats de la figure 4.10 révèlent que

le nombre de segments peut influencer la précision des résultats : plus le nombre de segments est élevé, plus les zéros sont précis, que ce soit en utilisant des `double` ou des `BigDecimal`.

On s'aperçoit par contre que les résultats obtenus sont très différents de la réponse correcte, $[0, 30]$, sur $[0, 40]$. Cela est dû à l'utilisation de la table normale, approximée par un ensemble de points, et à la fonction de probabilité calculée lors de l'évaluation de $[\![\phi_2]\!]$. À l'exemle 3.5 de la page 46, nous avons traité du calcul de $[\![\langle \mathsf{a} \rangle_0 \langle \mathsf{a} \rangle_0 \langle \mathsf{a} \rangle_0 \mathsf{T}]\!]$ et du fait que cet ensemble est égal à $[\![\langle \mathsf{a} \rangle_0 \langle \mathsf{a} \rangle_0 \mathsf{T}]\!]$ et à $[\![\langle \mathsf{a} \rangle_0 \mathsf{T}]\!]$ pour conclure que $[\![\langle \mathsf{a} \rangle_0 \langle \mathsf{a} \rangle_0 \langle \mathsf{a} \rangle_0 \mathsf{T}]\!] = [0, 30) \cup [200, 230]$. Nous pouvons étendre ce résultat à $[\![\phi_2]\!]$. La différence entre le résultat théorique et ceux pratiques donnés à la figure 4.10 est que notre implémentation ne peut pas tenir compte du caractère asymptotique de $\Phi(x)$. Comme la table normale est approximée par un nombre fini de points, la fonction $\Phi(x)$ implémentée vaut 1 pour des x supérieurs à une certaine borne. Nous nous retrouvons alors avec $P_{\mathsf{a}}(x_0, [0, 30) \cup [200, 230]) = 0$ pour un certain x_0 entre 25 et 30. Comme nous imbriquons dix opérateurs logiques du type $\langle \mathsf{a} \rangle_0$, cette erreur se propage de plus en plus, pour finalement donner les valeurs de la figure 4.10.

Pour minimiser les erreurs de précision notées dans la présente section et dans la précédente, voici ce que nous conseillons. Pour maximiser la recherche des racines d'une fonction, il est préférable d'utiliser la classe `BigDecimal`. Des exemples ont démontré que certaines racines n'étaient pas trouvées par la fonction `findZeroes` utilisant des `double`. La nombre de segments semble augmenter la précision des réponses retournées par le programme, d'après les résultats obtenus. De plus, comme le temps de calcul est d'ordre linéaire par rapport au nombre de segments, il est préférable d'en prendre un nombre élevé, pour la recherche des zéros. Bien que les fonctions de répartition facilitent le calcul des probabilités, il demeure que l'utilisation d'une table pour évaluer la fonction $\Phi(x)$ implique une imprécision des résultats. Nous avons déjà mentionné que pour ce type de variable aléatoire, même si nous avions utilisé des fonctions de densité, nous aurions eu à intégrer numériquement, amenant aussi une imprécision dans les résultats.

Chapitre 5

Conclusion

Les processus de Markov étiquetés servent à modéliser des systèmes qui ont un comportement aléatoire ou à approximer des systèmes complexes. Ils permettent l'analyse de la composition de plusieurs systèmes, puisqu'on étudie pour chacun les interactions avec leur environnement. À l'aide d'un modèle probabiliste, il est possible de vérifier que le comportement d'un système respecte un certain nombre de propriétés. Cette approche, que l'on appelle vérification formelle, permet d'établir, de façon automatique, l'équation

$$\mathcal{S} \models \phi$$

afin de nous assurer que le modèle $\mathcal{S}$ vérifie le comportement décrit par la formule ϕ.

Dans le second chapitre, nous avons présenté la famille de modèles que nous étudions, les LMP*, qui regroupe des systèmes probabilistes dont l'ensemble des états est non-dénombrable. Ils permettent, entre autres, de représenter des systèmes au comportement aléatoire, comme un protocole qui choisit au hasard un individu parmi des candidats. Ils permettent aussi de mieux modéliser des systèmes dont le taux de bris est connu, comme un tampon mémoire qui perdrait en moyenne un bit sur un million. Pour les LMP*, la manière traditionnelle d'énumérer les transitions une à une, comme pour les automates finis, n'est pas suffisante pour les définir dans le cas où l'ensemble des états est non-dénombrable. À travers quelques exemples, nous avons vu qu'il existe plusieurs manières de décrire les fonctions de probabilités définissant les transitions. La logique que nous avons présentée permet de décrire des propriétés que l'on veut imposer à un système. L'utilisation de cette logique est nécessaire, puisqu'ainsi, l'interprétation est unique et qu'il est aussi possible de détecter une incohérence dans la spécification d'un logiciel. En effet, une spécification peut comporter deux comportements contradictoires, ce qui se traduira par deux formules de logique, l'une étant équivalente sémantiquement à la négation de l'autre. En étudiant l'ensemble des formules spécifiant un système, avec

un démonstrateur automatique, il est possible de détecter de telles erreurs, ce qui permet de les corriger au début du processus de développement de logiciel.

Dans le troisième chapitre, nous avons défini deux langages formels pour spécifier les fonctions de transitions probabilistes, l'un utilisant des fonctions de densité, l'autre, des fonctions de répartition. Comme ces deux langages sont équivalents du point de vue de l'expressivité et que le calcul de probabilités est plus simple en utilisant le second, nous avons choisi les fonctions de répartition pour développer les algorithmes en vue d'implémenter la vérification formelle. Nous pensons que les langages que nous avons introduits permettront la représentation de plusieurs systèmes que nous retrouvons dans les systèmes informatisés. À travers l'exemple du contrôle de la température, nous avons illustré la puissance du langage ainsi que les principales opérations à effectuer dans le but d'implémenter la vérification formelle. Nous avons présenté deux approches pour relever ce défi. La vérification directe est une recherche à travers l'ensemble des états de ceux qui satisfont une formule donnée. Cette recherche est directement basée sur la sémantique de la logique et est dirigée par les opérateurs constituant la formule, l'évaluation ne se faisant pas sur tous les états. La vérification par le calcul d'une approximation permet de réduire un système dont l'ensemble des états est non-dénombrable en un système fini, pour ensuite effectuer la vérification sur ce dernier. Cette approche paraît prometteuse, bien que nous pensons qu'elle soit profitable seulement dans les cas où un très grand ensemble de formules est à vérifier.

Nous avons présenté au quatrième chapitre une implémentation en Java de la vérification directe pour les LMP*. L'implémentation fournit toutes les fonctionnalités que nous avions choisies au départ, comme le chargement à partir d'un fichier, la validation de la contrainte et le calcul des états qui satisfont une formule donnée. Nous avons présenté des résultats que nous avons obtenus. Dans certains cas, les réponses sont exactes, pour d'autres, l'erreur de calcul n'est pas négligeable. Comme la plupart des calculs sont effectués en utilisant le type primitif `double`, il y a beaucoup de troncatures effectuées, autant à la lecture des fonctions de probabilités qu'au moment de les évaluer. Nous avons aussi montré que l'utilisation d'une table normale évaluée avec des `double` amène beaucoup d'imprécision dans les calculs.

Les prochains efforts devront être faits pour bâtir des études de cas complexes, pour vérifier l'expressivité du langage et pour tester la puissance de l'outil développé. Une autre approche pour la vérification pourrait être étudiée pour tenter d'introduire des techniques symboliques comme les MTBDD [5]. De plus, comme nous l'avons mentionné à la section 3.3.3 la vérification directe d'un ensemble de formules pourrait être améliorée en exploitant la présence de sous-formules identiques. Nous pourrions ainsi diminuer le temps de calculs. Parce que nous travaillons avec des fonctions réelles

définies sur un domaine non-dénombrable, notre modèle est aussi très sensible aux erreurs de l'arithmétique des nombres en virgule flotante. De ce côté, nous préconisons de réimplémenter en utilisant la classe `BigDecimal` tout le moteur de calcul ainsi que les classes servants à représenter les fonctions réelles. Nous devrons aussi étudier en pratique l'impact de l'utilisation des fonctions de densité pour les fonctions probabilistes et améliorer l'implémentation de la recherche des zéros d'une fonction.

Bibliographie

[1] C. Baier. On algorithmic verification methods for probabilistic systems. Univ. Mannheim, 1998. Habilitation thesis.

[2] R. Blute, J. Desharnais, A. Edalat et P. Panangaden. Bisimulation for labelled Markov processes. Dans *Proceedings of the Twelfth IEEE Symposium On Logic In Computer Science, Warsaw, Poland*, 1997.

[3] S. V. A. Campos, E. M. Clarke, W. R. Marrero et M. Minea. Verus : A tool for quantitative analysis of finite-state real-time systems. Dans *Workshop on Languages, Compilers, & Tools for Real-Time Systems*, pages 70–78, 1995.

[4] V. Danos et J. Desharnais. Labeled Markov processes : stronger and faster approximations. Sera publié dans *Proceedings of LICS03*, 2003.

[5] L. de Alfaro, M. Kwiatkowska, G. Norman, D. Parker et R. Segala. Symbolic model checking of concurrent probabilistic processes using MTBDDs and the Kronecker representation. Dans S. Graf et M. Schwartzbach, editors, *Proc. 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'00)*, volume 1785 of *LNCS*, pages 395–410. Springer, 2000.

[6] J. Desharnais. *Labelled Markov Processes.* Thèse de doctorat, McGill University, Novembre 1999.

[7] J. Desharnais, A. Edalat et P. Panangaden. Bisimulation for labelled Markov processes. *Information and Computation*, 179(2) :163–193, Dec 2002.

[8] J. Desharnais, V. Gupta, R. Jagadeesan et P. Panangaden. Approximating labeled Markov processes. *Information and Computation*, 184(1) :160–200, juillet 2003.

[9] D. Eck, K. Mitchell et J. Ryan. JCM – Java Components for Mathematics. Pour plus de détails, voir http ://math.hws.edu/javamath/.

[10] A. Fortin. *Analyse numérique pour ingénieurs.* Presses internationales Polytechniques, deuxième édition, 2001.

[11] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1) :5–48, 1991.

[12] M. Hennessy et R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the Association for Computing Machinery*, 32(1) :137–161, 1985.

[13] C. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10) :567–580, 1969.

[14] IEEE 1985. ANSI/IEEE std 754-1985, standard for binary floating-point arithmetic. Dans *ACM SIGPLAN*, volume 22, pages 9–25, 1987.

[15] D. Kozen. Language-based security. Dans *Mathematical Foundations of Computer Science*, pages 284–298, 1999.

[16] M. Z. Kwiatkowska, G. Norman et D. Parker. PRISM : Probabilistic symbolic model checker. Dans *Computer Performance Evaluation / TOOLS*, pages 200–204, 2002.

[17] K. G. Larsen et A. Skou. Bisimulation through probablistic testing. *Information and Computation*, 94 :1–28, 1991.

[18] R. Milner. *A Calculus for Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[19] S. M. Ross. *Initiation aux probabilités*. Presses polytechniques et universitaires romandes, quatrième édition, 1996.

[20] F. B. Schneider, G. Morrisett et R. Harper. A language-based approach to security. *Lecture Notes in Computer Science*, Vol. 2000 :86–101, 2001.

[21] R. van Glabbeek, S. Smolka, B. Steffen et C. Tofts. Reactive generative and stratified models for probabilistic processes. Dans *Proceedings of the 5th Annual IEEE Symposium On Logic In Computer Science*, 1990.

Annexe A

Rappels sur les probabilités

A.1 Variables aléatoires continues

Voici un bref rappel sur les probabilités, à propos des variables aléatoires continues. Pour plus de détails, voir [19].

Définition A.1.1 *X est une **variable aléatoire continue** s'il existe une fonction f non-négative définie pour tout $x \in \mathbb{R}$ et vérifiant que $\forall E \subseteq \mathbb{R}$:*

$$P\{X \in E\} = \int_E f(x)\mathrm{d}x$$

*f est la **fonction de densité** associée à la variable X. Donc, la probabilité que X prenne une valeur dans l'ensemble E se calcule en prenant l'intégrale sur E. Pour que le tout donne une probabilité, f doit respecter la contrainte suivante :*

$$P\{X \in (-\infty, \infty)\} = \int_{-\infty}^{\infty} f(x)\mathrm{d}x = 1$$

De cette définition, on déduit que :

1. $P\{a \leq X \leq b\} = \int_a^b f(x)\mathrm{d}x$
2. $P\{X = a\} = \int_a^a f(x)\mathrm{d}x = 0$
3. $P\{a \leq X \leq b\} = P\{a < X < b\}$

Définition A.1.2 *La **fonction de répartition** F d'une variable aléatoire continue*

X, dont la fonction de densité est f est définie par

$$F(a) = P\{X \leq a\} = \int_{-\infty}^{a} f(x)\mathrm{d}x$$

ou, autrement,

$$\frac{\mathrm{d}}{\mathrm{d}a}F(a) = f(a)$$

Alors,

1. $P\{a \leq X \leq b\} = F(b) - F(a)$
2. $P\{X = a\} = F(a) - F(a) = 0$
3. $\lim_{x\to-\infty} F(x) = 0$
4. $\lim_{x\to+\infty} F(x) = 1$

Définition A.1.3 *L'**espérance mathématique** d'une variable aléatoire continue X, dont la fonction de densité est f, est définie par*

$$\mathrm{E}[X] = \int_{-\infty}^{\infty} xf(x)\mathrm{d}x$$

Définition A.1.4 *La **variance** d'une variable aléatoire continue X est définie par*

$$\mathrm{Var}(X) = \mathrm{E}[X^2] - \mathrm{E}[X]^2$$

Définition A.1.5 *L'**écart-type** d'une variable aléatoire X, qui se note σ, est définie par*

$$\sigma = \sqrt{\mathrm{Var}(X)}$$

A.2 Quelques lois continues

Loi uniforme (α, β)

Une variable aléatoire X est dite **uniformément** distribuée sur l'intervalle (α, β) si sa fonction de densité est

$$f(x) = \begin{cases} \frac{1}{\beta-\alpha} & \text{si } \alpha < x < \beta \\ 0 & \text{sinon} \end{cases}$$

De cette fonction, on peut déduire que

1. $\mathrm{E}[X] = (\beta + \alpha)/2$
2. $\mathrm{Var}(X) = (\beta - \alpha)^2/12$
3. $F(x) = \begin{cases} 0 & \text{si } x \leq a \\ \frac{x-\alpha}{\beta-\alpha} & \text{si } \alpha < x \leq \beta \\ 1 & \text{si } x \geq b \end{cases}$

Loi normale (μ, σ^2)

Une vairable aléatoire X est dite **normale** avec les paramètres μ et σ^2 si la densité de X est donnée par :

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{\frac{-(x-\mu)^2}{2\sigma^2}} \quad \forall x \in \mathbb{R}$$

La fonction de répartition d'une variable Normale(0,1) est notée $\Phi(x)$ et est définie par :

$$\Phi(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{\frac{-y^2}{2}} \mathrm{d}y$$

De la définition de la densité et de Φ, on déduit que :

1. $\mathrm{E}[X] = \mu$
2. $\mathrm{Var}[X] = \sigma^2$
3. Si X est une variable suivant une loi Normale(μ, σ^2), alors $Y = \alpha X + \beta$ suit une loi Normale$(\alpha\mu + \beta, \alpha^2\sigma^2)$.
4. La fonction de répartition d'une variable X Normale(μ, σ^2) est :

$$F(x) = \Phi\left(\frac{x-\mu}{\sigma}\right)$$

5. Si X est une variable Normal(μ, σ^2), alors :

$$P(a \leq X \leq b) = \Phi\left(\frac{b-\mu}{\sigma}\right) - \Phi\left(\frac{a-\mu}{\sigma}\right)$$

Loi exponentielle $(\lambda > 0)$

La fonction de densité d'une variable aléatoire suivant une loi **exponentielle** de paramètre $\lambda > 0$ est :

$$f(x) = \begin{cases} \lambda e^{-\lambda x} & \text{si } x \geq 0 \\ 0 & \text{sinon} \end{cases}$$

De cette densité, on déduit que :

1. $F(x) = 1 - e^{-\lambda x}$, pour $x \geq 0$.
2. $\mathrm{E}[X] = 1/\lambda$
3. $\mathrm{Var}[X] = 1/\lambda^2$
4. Les variables exponentielles ont la propriété d'être sans mémoire, c'est-à-dire :

$$P\{X > s + t \mid X > t\} = P\{X > s\} \text{ pour tous } s, t \geq 0 \ .$$

Cette propriété est appelée *propriété de Markov* et elle s'applique aux LMP*.

Annexe B

Diagrammes UML des classes

Dans les pages qui suivent, nous présentons les diagrammes UML des classes des paquetages. La figure B.1 présente les principales classes du paquetage `data`. La figure B.2 présente la classe centrale du programme, à savoir la classe `engin.Engin`. Finalement, la figure B.3 présente presque tout le paquetage `logic`. Les classes qui ne sont pas dans ces figures ont été mises de côté pour diverses raisons : certaines sont plutôt accessoires, d'autre sont des *Exceptions* et nous avons aussi omis les classes de tests (voir 4.2.5).

FIG. B.1 – Les classes du paquetage `data`

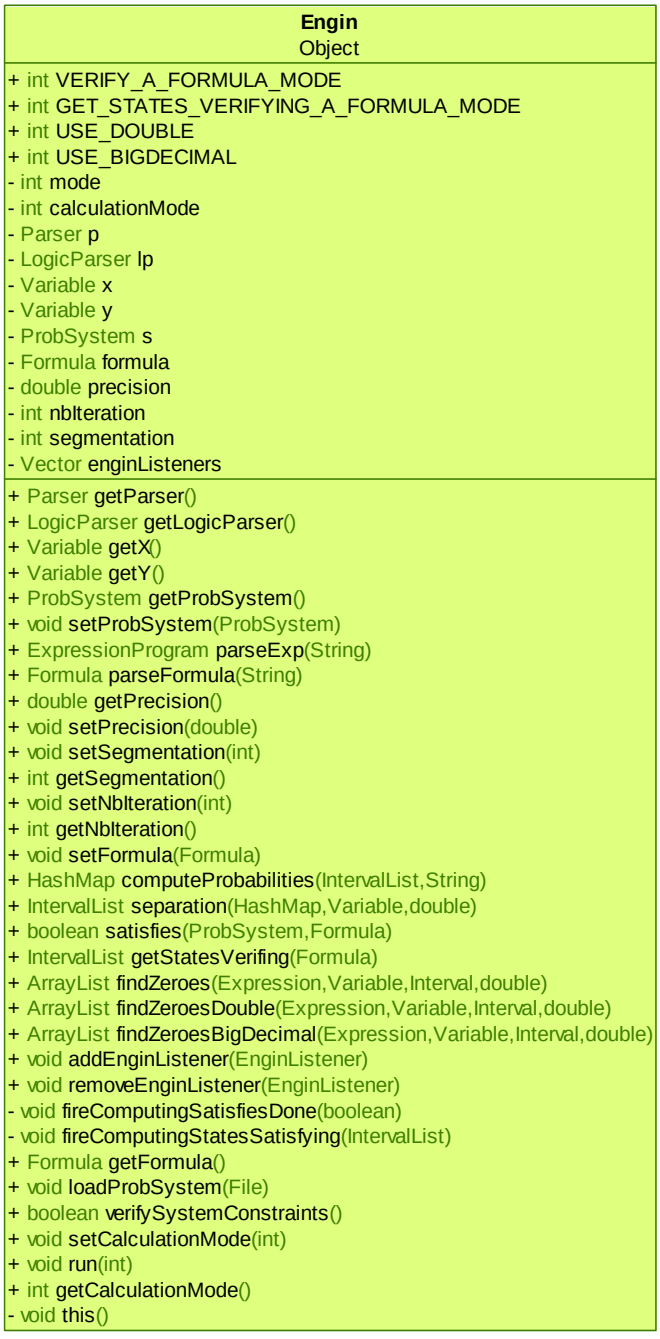

FIG. B.2 – La classe `Engin`

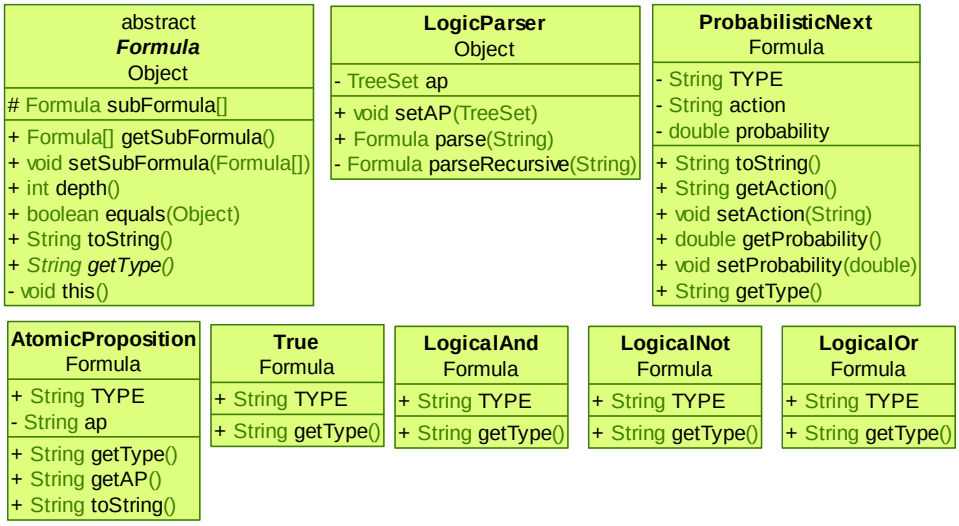

FIG. B.3 – Le paquetage `logic`

Annexe C

Le code source du système de contrôle de la température

Voici le code source de l'exemple 3.2.1, à la page 39. Ici, `Normal`$(z) = \Phi(z)$, de la table normale standard.

```
states : [0,40] U [100,140] U [200,240] U [300,340];

init :  15;

X : [0,20) , [20,30) ,[30,40] , [120,140] ,[200,230] , [300,340];
Y : [0,40] , [100,140] , [200,240] , [300,340];

label
   confort  : [18,23] U [118,123] U [218,223] U [318,323] ;
   chauf_ok : [0,40] U [200,240];
   clim_ok  : [0,40] U [100,140];
endlabel

transition
 [0,20)   -[a]->(x/300 + 0.80)*  Normal((y-(x+x/10+2))/0.5) :[0,40];
 [20,30)  -[a]->(x/300 + 0.80)*  Normal((y-(x+x/10+2))/0.5) :[0,40];
 [0,20)   -[a]->(-x/300 + 0.15)* Normal((y-(x+102))/0.5)    :[100,140];
 [20,30)  -[a]->(-x/300 + 0.15)* Normal((y-(x+102))/0.5)    :[100,140];
 [20,30)  -[d]->(-3*x/400 + 1)*  Normal((y-(x+x/5-8))/0.5)  :[0,40];
```

```
 [30,40]  -[d]->(-3*x/400 + 1)*  Normal((y-(x+x/5-8))/0.5)  :[0,40];
 [20,30)  -[d]->(3*x/400 -0.05)* Normal((y-(x+198))/0.5)    :[200,240];
 [30,40]  -[d]->(3*x/400 -0.05)* Normal((y-(x+198))/0.5)    :[200,240];
 [120,140]-[d]->(-3*x/400 +1.75)*Normal((y-(1.2*x-28))/0.5) :[100,140];
 [120,140]-[d]->(3*x/400 -0.8)*  Normal((y-(x+198))/0.5)    :[300,340];
 [200,230]-[a]->(x/300 + 2/15)*  Normal((y-(x+x/10-18))/0.5):[200,240];
 [200,230]-[a]->(-x/300 +49/60)* Normal((y-(x+102))/0.5)    :[300,340];
endtransition
```