Extensions de l'algorithme d'atteignabilité arrière dans le cadre de la vérification de modèles modulo théories

Mattias Roux

université PARIS-SACLAY

UNIVERSITÉ PARIS SUD
Comprendre le monde, construire l'avenir

NNT : 2019SACLS582

Thèse de doctorat

Extensions de l'algorithme d'atteignabilité arrière dans le cadre de la vérification de modèles modulo théories

Thèse de doctorat de l'Université Paris-Saclay
préparée à l'Université Paris-Sud

École doctorale n°580 Sciences et technologies de l'information et de la communication (STIC)
Spécialité de doctorat : Informatique

Thèse présentée et soutenue à Orsay, le 19 décembre 2019, par

MATTIAS ROUX

Composition du Jury :

Dominique Quadri Professeure, LRI - Université Paris-Sud (UMR 8623)	Présidente
Charlotte Truchet Maîtresse de conférence, LS2N - Université de Nantes (UMR 6004)	Rapportrice
Pascal Poizat Professeur, LIP6 - Sorbonne Université (UMT 7606)	Rapporteur
Étienne André Professeur, LORIA - Université de Lorraine (UMR 7503)	Examinateur
Philippe Quéinnec Professeur, IRIT - ENSEEIHT (UMR 5505)	Examinateur
Sylvain Conchon Professeur, LRI - Université Paris-Sud (UMR 8623)	Directeur de thèse

À Ornella

Remerciements

À l'origine je voulais me satisfaire d'un :

«Celleux que je remercie le savent.

Les autres le savent aussi.»

Ces remerciements lapidaires avaient le mérite de me permettre de n'oublier personne et de contenter tout le monde. Mais les démonstrations de gratitude sont généralement la première et la dernière chose qu'on lit dans ce genre de document. Les raisons sont assez simples :

— Je vais lire les remerciements, de toutes façons je ne comprendrai rien au reste
— Suis-je dans les remerciements ? «Quoi ?! Il m'a oublié·e ?! Quel sal*ud [1] ! J'avais tellement fait pour lui !»
— Qui d'autre est dans les remerciements ? «Quoi ?! Elle/Lui alors que pas moi ?! Quel sal*ud [2] !»

Pour ces raisons j'ai donc décidé d'en écrire des vrais. Mais c'est quoi, au fait, des vrais remerciements ?

Par exemple, dois-je remercier monsieur F. qui m'a pourtant oublié dans ses remerciements ? Et que dois-je faire de monsieur C. dont je n'ai pas encore lu la prose de gratitude, vais-je le remercier sans savoir s'il le fait lui-même ? Première question, donc : remercie-t-on en espérant l'être en retour ou en signe de gratitude ? Je penche naturellement pour la seconde option ce qui m'amène à une nouvelle question. Si je suis reconnaissant, jusqu'à quand dois-je remercier ? Et remercier pour quoi ? Remercier d'être né ? Remercier pour les études qui m'ont conduit jusqu'ici ? Remercier le climat sociologique qui favorise des gens comme moi pour accéder à des études supérieures ? Remercier donc mon milieu socio-culturel et son aspect privilégié au sein de la société française et par extension occidentale ? Suis-je ici, en train d'écrire ces remerciements parce que je suis né, parce que j'ai été éduqué, parce que j'ai appris, parce que c'était une trajectoire naturelle ? Et comment regrouper les remerciements ? Quels taxons ? Un taxon famille ? Un taxon collègues ? Un taxon ami·e·s ? Ça fait beaucoup de taxons mais il n'y en a jamais trop.

En terme de naissance, je remercie donc mes parents – pas seulement pour m'avoir fait naître, naturellement – Donatella (et ça se prononce avec l'accent sur le *e* prononcé [ɛ] et on prononce bien les deux

1. On me dit très souvent que je suis grossier, j'ai donc décidé d'écrire dans un langage châtié.
2. Ce n'est pas facile

l, si on n'y arrive pas on dit «madame» ou «Dona», ce sera mieux que «Donatéla») et Jean-Philippe (bon, ça, ce n'est pas dur à prononcer mais pas sûr qu'il entende immédiatement, mieux vaut rajouter un «Jean» pour amorcer le processus d'écoute, «Jean-Jean-Philippe» donc). Maman parce que tu m'as toujours presque compris, que tu m'as fait lire les livres les plus marquants de ma jeunesse et parce que même si je n'ai aucun souvenir du fait que tu faisais la cuisine à part les courgettes du mercredi midi, tu as fait tout le reste. Papa parce que malgré ton sourire ou parfois grâce à ton sourire tu as su être la force tranquille qui s'énerve rarement mais justement, l'homme qui sort des grandes phrases de philosophe pour les oublier dans l'heure qui suit et qui, comme moi, retient tout ce qu'il lit (ce qui n'est pas toujours passionnant). Eux-mêmes ayant été engendrés par mes grands-mères que j'ai (eu) la joie de connaître, Mamie et Nonna. Mamie parce que tu as toujours été là entre les repas du jeudi midi, les injonctions à te donner un arrière-petit-enfant ou les anecdotes familiales à chaque repas de famille, merci. Nonna, l'oracle, morta prima di conoscere Ornella e Pokam Amos e di sapermi in dottorato, una voce particolare, un sorriso stupendo e un «vai» che rimanerà per sempre nella mia memoria. Papi et Nonno, notre rencontre aura été trop brève pour que vous ne soyez autre chose que mentionnés ici.

Qui dit naissance dit possibilité d'avoir des frères et sœurs et, dans ce domaine, j'ai été gâté avec non pas un, non pas deux, non pas trois, non pas quatre, non pas cinq, non pas six mais bien sept frères et sœurs. Tou·te·s arrivé·e·s après moi mais m'ayant accompagné tout au long de ma vie. Les remercier pour une chose en particulier serait forcément réducteur mais les remercier pour tout en général serait assez tristouni. Je préviens toute personne étrangère aux membres de la famille Roux-Pavolini, le reste de ce paragraphe vous semblera grossier, stupide, mal écrit, inintéressant et ça tombe bien, ça ne s'adresse pas à vous. C'est la première fois que je peux écrire des remerciements et peut-être la dernière, je ne vais pas m'embarrasser de bonnes manières. Merci donc à Fel, toi mon frère presque jumeau qui m'as accompagné dans mes pugilats (avec toi) comme dans nos discussions chaque soir avec Agrid, la pause pipi (pas du chien) et le film du soir généralement comme centre névralgique de notre discussion vespérale. Merci d'avoir réussi à subir et aimer un frère comme moi. Merci aussi à Ramone… Des parents ont osé appeler leur fille Ramone ?! Non. Merci donc à Amélie, plutôt. Toi ma sœur pas jumelle mais presque à qui j'ai appris à lire, la seule personne que je connaisse qui lave le savon plutôt que l'inverse même si avec ta nouvelle occupation il me semble que les choses sont redevenues plutôt normales. Nous n'aurons sûrement plus jamais à organiser les tours de lecture d'Harry Potter mais nous avons encore de nombreuses histoires à vivre ensemble. Merci à Anne, toi la dernière de mes «jumelles» avec qui je partage un plaisir de la relation tempétueuse à défaut des goûts musicaux. Toi la première à avoir été coulée dans le même moule que moi avant la fournée suivante. J'étais peut-être un grand-frère difficile mais tu étais une bonne servante pour mon trône. Merci à Baltha qui ne se mouille pas mais qui ne se laisse pas mouiller. Toi mon premier vrai «petit frère» devenu grand, et beau (ah, Balthazar) et, fort heureusement, le seul à avoir pris le sourire de c*****d de notre père. Merci à Didi. Tu amèneras toujours ton humour particulier et l'amour de ton famille pour nous faire rire, nous consoler, nous souder et, plus généralement, pour nous montrer qu'il y a toujours de l'amour dans tous les moments de la vie que ce soit lors de la réception d'un ballon de foot sur quelque partie anatomique non prévue pour, un 3/20 au dernier contrôle de maths ou un film avec Hugh Grant ou Julia Roberts. Merci à toi aussi, Yolaine, la troisième chiante parmi les chiants, digne sœur de ton frère aîné, que j'aurai fait souffrir mais qui, je l'espère, sais que je l'aime énormément. J'attends toujours notre collaboration sur un jeu. Tu m'as suffisamment accompagné lorsque je jouais pour avoir une idée de ce qu'est un jeu maintenant. Enfin, merci à toi Couïlle-Couïlle, petit dernier, arrivé tard mais finalement arrivé (toujours dit qu'on serait 8 à la fin) et qui as presque plus été mon fils que mon frère mais quand il s'est agi de te laisser geeker de 10h (après mon petit-déjeuner !) à 20h c'est vrai que j'étais plus ton frère.

Je suis né, très bien. J'ai été entouré de l'amour de mes proches, encore mieux. J'ai grandi et évolué à leur contact et ils m'ont éduqué. Alors parlons d'éducation. Dois-je remercier ma géniale prof de français de 3e, madame Melhaoui, ainsi que mon excellent prof de français de 1ère, monsieur Lecornu ? Ai-je eu besoin d'eux pour développer mon amour de la langue bien maniée, du cinéma et pour perfectionner mes moyens d'expression ? Je pense que oui. Je pense qu'il faut aussi que je remercie beaucoup de mes profs, secrétaires, administratifs etc d'université où j'ai passé les meilleures années d'étude de ma vie et de labo où j'ai passé ces quatre dernières années. Merci donc à Jean-Christophe Léger, Nicolas Bredeche, Christian Jacquemin (pas uniquement pour l'enseignement mais aussi pour ces conversations fortuites aux Ulis, ces encouragements et ces mails), Christine Paulin, Nicole Bidoit, Anastasia Bezerianos, Denis Cousineau, Mehdi Ammi, Nicolas Sabouret (aussi pour ces conversations interminables où que nous nous rencontrions, bus, CESFO, couloirs du labo), Laurent Simon, Philippe Dague, Fabienne Robinson, Lionel Lacassagne, Laurent Rosaz, Kim Nguyễn, Sylvain Conchon, Denise Macé, Francine Hordesseaux, Murielle Bénard, Régine Bricquet, Katia Evrat, Gladys Bakayoko, Laurent Darré, Myriam Joseph et Martine Croissant. Toutes ces personnes ont fait en sorte que j'apprenne le plus possible et le mieux possible mais je souhaite tout de même remercier en particulier Christine et Sylvain qui ont fait énormément pour moi.

En parlant de Sylvain, tu as bien droit à ton paragraphe personnel puisqu'il s'agit quand-même de remerciements dans un manuscrit de thèse. Sans toi, pas de thèse. C'est tout. Tu m'as pris en stage, tu m'as permis d'obtenir une bourse pour mon master, tu m'as poussé à écrire mes articles. Notre relation fut… complexe. Riche. Étonnante. Fructueuse. Orageuse. Il est très difficile de créer ex nihilo une relation de travail presque personnelle comme celle qui se crée lors d'une thèse. Il y aura forcément des mauvais moments mais aussi de très bons moments et pour tout ça, Sylvain, merci.

Merci aussi à Pascal Poizat et Charlotte Truchet. En moins de deux mois vous avez appris que vous alliez devoir rapporter une thèse et qu'il faudrait que ce soit fait au maximum un mois après réception du document et vous n'avez pas failli. Je ne vous remercierai sûrement pas assez pour ce que vous avez fait et pour les rapports très agréables que vous avez faits sur ce document. Et merci à Dominique Quadri, Étienne André et Philippe Quéinnec d'avoir accepté de faire partie de mon jury de soutenance de thèse.

On arrive à ce moment où on a peur d'avoir oublié des gens. J'aurais pu t'oublier, Frédo, par exemple, le premier à m'avoir fait découvrir un labo de recherche (pas si éloigné géographiquement du mien) et ma première lecture d'article scientifique (ton souvenir, pas le mien).

Serge parce que… parce que tu sais pourquoi. J'ai eu beau t'enseigner que l'endroit où on gare généralement les voitures s'appelle un parking, ça n'est rien par rapport à ce que tu m'as donné. Un de mes plus grands mercis (poméridien ou pas) t'est donc destiné. Tu en découperas un bout pour la Riflette, quand-même. Pas facile de me supporter surtout quand on est ado et que je suis un jeune adulte con.

N'oublie pas le reste de ta famille, Mattias ! Oui, Fabienne et Fabrice et vos invitations des mardis soirs avec petit rhum à la fin, Gladys, Maëva et Myriam, cousines marrantes, pas loin et pourtant on se voit si peu !, Christian, parrain nul (mais Noir Désir et Noël Rock) mais oncle pas mal, Zoc e Antonella, tutte queste vacanze d'estate a litigare, ridere, mangiare, Tommaso (così impari) e Marta che ci vedevano andare in acqua subito dopo aver mangiato, Zia Rossi e Zio Massimo con cui ho scoperto Murakami Haruki, voi che mi avete chiamato nei momenti difficili e che non sapete servire il cibo per 15 persone ma chi lo sa fare ?, Micol e Matilde, le due cugine cosi differenti ma cosi vicine. Bacioni per tutti questi anni di Calco, Brivio, Toscana, chiamate di compleanni («Ah ! Chi è ! I Cirilli/Pavolini/La Nonna ! Mattias,

è per te!») e tante altre cose. La mia vita italiana è per sempre legata a voi tutti/e. Merci à toi, Coline, qui me rends heureux en rendant mon frère heureux mais surtout qui as une science particulière des messages avec 49 émoticônes qui racontent une histoire. Tu es aussi la seule personne qui as dit qu'elle tricoterait quelque chose pour mon fils et qui l'as fait ! Merci Wumi. Tu rends aussi ma sœur sereine et heureuse. C'est un plaisir de tous les jours de vous voir ensemble (Par contre monte pas trop haut, hein. T'es là, tu redescends, tu dis quand-même digital). Merci aussi au rugby. Les saveurs savent. Merci à Tantine et Tonton pour les weekends normands, les gros repas, la belotte, James Bond et la meilleure place du canapé qui était systématiquement réservée. Merci aussi Djougne, ma sœur jumelle en courage. Nos discussions me manquent un peu et ont ponctué ma thèse de disputes complètement inutiles donc parfaitement appropriées. Je n'écouterai jamais Beyoncé pareil depuis nos merveilleurses vacances de Noël à Dortmund. Tzoni/a/e/u/o, tu es vénale et tétue, c'est pour ça que je t'aime aussi. Merci pour les leçons d'allemand et tes avis plutôt tranchés sur des films au demeurant très bons. Djokou, handicapée des ordinateurs mais avec tant d'amour à donner, merci. Ta Jo pour les leçons de bansoa. Me chiè ne sohneh nerah la'a. Mes coos, ma nouvelle famille, merci. Merci à William pour les beuveries nocturnes dans les bars de Yaoundé, Ton Thierry pour être le grand frère que tout le monde rêverait d'avoir et Ma pour la même raison. Merci Tata Rosa parce que tu m'as donné Orné et les réveils à 4h du matin pour nous demander de nous marier et maman Pauline qui partage avec moi le même amour des douches et de sa fille (et merci à Ama Megré qui m'a tem guiet chouo, je n'aimerais pas être maudit par ses filles pour l'avoir oubliée ici et c'est bien la seule personne qui comprend quand je parle bansoa). Merci à Audrey et Noé qui m'ont rappelé que parfois le plus important c'est aussi de se débrouiller pour pouvoir jouer toute la journée. Merci enfin à Kartchia, Pasteur, Yebhe et Thierno pour les rares mais agréables weekends à Lille.

S'il y eut une chose de difficile pour un enfant comme moi entouré par tant d'amour familial, se fut d'avoir des amis. Jusqu'à récemment je pouvais les compter sur les doigts d'un manchot. Puis comme avec tout ce que la fac m'a offert, j'ai découvert des gens qui m'acceptaient (généralement) comme je suis. La première ce fut Sophie même si nous avons un désaccord sur les termes exacts ayant introduits notre rencontre nous n'en avons pas sur l'amitié qui en a découlé. Merci d'être toujours la présence proche et amicale qui a accompagné mes années universitaires. Malgré ton piètre niveau en informatique je ne te changerais pour rien au monde. Peu de temps après et pour un semestre seulement j'ai partagé les bancs des amphis ainsi que les files d'attentes de LoL avec Timothée. Monsieur «Je vais l'écraser, j'ai le match-up ! Ah l'bâtard, il a trop de chance». Discuter avec toi c'est comme discuter avec moi. Peu importe si vous avez raison ou pas, l'important c'est qu'à la fin j'ai raison. Nos conversations restent, pour la plupart, inachevées. Appelons ça un compromis nécessaire à notre longue relation d'amitié. Et de toutes façons, comme disait le docteur C. dans son magnum opus «Vérification des résultats de l'inférence du compilateur OCaml» :

«Merci Mattias [...], je ne l'admettrai qu'ici, tu auras souvent eu raison» [3]

Avec Sophie forcément il y a eu Bruno qui, fort heureusement pour les surnoms que j'aime donner, a changé de coiffure (oui, bravo, bien vu, c'est lui monsieur F.). Pour le coup, bon niveau en informatique gâché par un choix de trajectoire discutable mais accepté. On t'aura fait souffrir à ne jamais vouloir sortir et à ne jamais donner de nouvelles mais bon, on aura eu des LAN marrantes. Puis arriva Mathilde. Allergique aux chats tout en les adorant, ce paradoxe te résume assez bien. Je t'aime même si tu n'aimes

3. Bien évidemment, sa maîtrise de l'euphémisme ne cesse de m'épater.

pas mon torse. Robin, je ne te remercie pas pour les années fac, on ne va pas faire semblant, mais vu que je t'ai rencontré en L1 et que je t'ai apprécié en thèse je te mets quand-même ici ! On partage la même fierté pour nos sœurs qui font du rugby, ça ne s'invente pas un tel atome crochu. Tu hésites beaucoup mais tu me marqueras toujours par ton humilité assez irritante et ta capacité à m'écouter parler des heures durant de DotA 2 ou de Binding of Isaac (3.000.000% ça se fête). L3, Thibaut. Ah, Thibaut. L'électron libre. Le roi de la gaffe. L'empereur du romantisme. Le Tachon de nos soirées. Tu m'auras fait rire mais je t'aurai aussi fait découvrir le Plein Sud (R.I.P.). Tu fais tes courses avec une poche et tu ne joues que Riven mais je t'aime quand-même. Alwine naturellement. Nos repas ensemble tous les midis pendant ma première année de thèse ont été des moments très agréables et non, miaou peut-être mais Rengar n'était vraiment pas fait pour toi. Merci pour ces repas, ces pauses clopes, ces «flemme... mais une flemme !». Un merci aussi à Damien malgré tout. Tu m'as relancé sur DotA 2 et nous avons eu de belles années de fac. Merci aussi à Marie pour ces discussions reposantes dans le bus ou les couloirs du labo. Merci enfin à Loïs pour les pauses du stage de M2 et toutes ces discussions enthousiasmantes. Merci à Yohan, toujours dispo pour venir boire une bière et pour les discussions de doctorants. Merci aussi à Jérôme qui met dans notre amitié ce qu'il ne met pas dans ses chronosphères. Merci à toi pour les messages, les longues discussions et ton écoute permanente. Grazie a voi, Gabriele e Enea, uno troppo gentile e l'altro troppo tossico per le partite, le discussioni e la possibilità di parlare italiano quotidianamente. Merci à Michael pour les nocturnes sur TI et les discussions poussées sur notre jeu préféré. Merci à vous tous mes amis. Les années nous éloignent un peu mais pas suffisamment pour que je vous oublie et pour que je ne prenne pas plaisir à vous revoir quand l'occasion se présente (pas trop souvent quand-même).

Mon grand ami, Albin. Tu m'as lâchement abandonné pour aller te réfugier à Paris mais tu as gardé ma chatte, tu nous as aidés pour détruire et reconstruire notre appart, à monter les meubles IKEA (mais honnêtement c'est toi qui devrais nous remercier pour ça), tu es toujours là quand je plonge sous les t4 [4] et je ressens forcément ton absence par ma mort prématurée, nous en avons passées des années, des soirées, des midis, des journées, des parties ensemble. Je n'oublierai jamais OG - EG avec toi mais de toutes façons j'oublie très peu de choses et je ne peux que te remercier.

Il y a enfin les deux autres larrons avec qui Albin et moi avons partagé nos années de doctorat. Deux couples d'amis qui se sont rencontrés pour former un quartette d'amis. Pierrick, Rémy, vous n'avez jamais compris mes mails mais je ne vous en veux pas, vous avez fini par me comprendre et me supporter. Ce ne fut pas facile mais le résultat en vaut le coup. Nous étions quatre glandeurs, nous sommes maintenant quatre glandocteurs et c'est beau. Je ne sais pas vraiment comment ça tient mais ça tient pour le moment. Merci pour ces 14845 + 16738 + 18584 + 26330 messages échangés pendant trois ans (sans compter quand les glandeurs font de l'OCaml, parlent de la vie sexuelle d'un des quatre ou de l'actualité du sud de Paris) et tout ce qui va avec entre non-invitations, bières prévues trois mois à l'avance et les blagues sur les mamans. Rémy, il faut vraiment que tu te poses les bonnes questions.

Un doctorat, contrairement à ce que les films aiment à présenter (forcément, quand un film comme **La chute du Faucon Noir** [5] se présente comme réaliste on ne va pas s'attendre à un effort poussé pour présenter le monde de la recherche), ce n'est pas que des personnes qui réfléchissent dans leur coin jusqu'à crier «Euréka» ou «Boum, je suis l'invincible», c'est aussi des repas, des pauses et d'autres moments de convivialité. Merci donc à l'équipe VALS et tout particulièrement à Jean-Christophe avec qui

4. J'en profite pour remercier aussi IceFrog et Valve pour m'avoir donné le jeu qui a occupé toutes mes années de thèse et de productivité, DotA 2. Jouez-y. Ça vaut vraiment le coup – on ne peut pas faire de pied-de-page d'un pied-de-page mais pourquoi «coup» et pas «coût» ?

5. Qui, objectivement parlant, ne peut être apprécié que par des personnes que je ne vous conseille pas de fréquenter.

je ne partage presque aucun goût, signe que lui comme moi sommes des personnes de qualité sachant faire fi de leurs différences, Kim qui reste très jeune (et très grossier !), Andrei, Хоть тебе и не нравится, что я матерюсь одной-единственной фразой (мог бы и научить меня чему-нибудь ещё, если не хотел, чтобы я все время повторял “сука блядь”), я очень ценю наши беседы о фильмах и обо всем на свете, Fatiha et sa voix qui s’entend où qu’on soit, Frédéric qui n’assume pas sa ressemblance fortuite avec Jean Dujardin, Safouan et ses discussions diverses et variées, Véro que j’apprécie autant qu’elle peut parfois m’énerver et qui est une des rares personnes qui sait quelle quantité de crème il faut mettre dans la carbonara en plus de savoir parler la langue pour la cuisiner, Delphine, partie (du labo) avant que j’ai eu le temps de lui dire au revoir, qui fut ma prof avant d’être ma collègue et que j’ai appris à connaître et à apprécier alors, Guillaume qui n’a pas d’âme (Abaddon, franchement), qui a découvert que les bons films avec Nick Nolte sont ceux avec Jeff Bridges et avec qui j’ai parlé de longues heures de DotA, de DotA et parfois de DotA, Sylvie pour les discussions dans le bus et les différentes aides apportées aux innocents thésards que nous sommes, Fred, le doyen au fauteuil réservé, l’humour du vendredi mais du lundi au vendredi, le seul homme capable de vous faire croire qu’une collègue est morte grâce à un titre de mail de départ à la retraite, mon confrère judoka, Sensei Voisin. Je remercie aussi les thésards du groupe qui ont bien malheureusement dû subir mes quolibets liés à leurs choix de jeu de prédilection ou simplement parce que ça m’amusait, Quentin, Diane, Raphaël, Mário (qui m’a bien aidé avec Why3 et patience), Georges. Enfin, je ne vous oublie pas, les deux derniers, Julien (j’aurais pu aussi te mettre dans les amis, soyons honnêtes, mais il faut faire des choix dans la vie et ça prenait tout son sens de te mettre ici !) et Alexandrina. Julien, tu as été pendant trois ans mon co-bureau, ce n’est pas rien. J’ai presque réussi à te faire jouer à un MOBA et tu sais maintenant quelle quantité de crème il faut mettre dans la carbonara. Je n’ai pas toujours été là mais bon, j’avais un fils qui arrivait et Alexandrina m’a remplacé. On aura quand-même passé trois bonnes années ensemble dans un bureau à la fois studieux, musical et amical et pour ça, merci. Alexandrina, tu viens d’arriver mais tu es la seule personne qui m’a demandé si elle serait dans les remerciements. ET PAS QU’UNE FOIS ! Tu étais une étudiante peu bavarde mais tu te révèles une co-bureau extrêmement prolifique en sons parlés. Tu as fini par me battre aux Villes USA, je sais donc que je te laisserai forte d’une nouvelle connaissance et tu m’as aidé pour la phrase pour Andrei (avec une erreur !). Спасибо.

Merci à celles et ceux qui sont sûr·e·s que je les ai oublié·e·s et que ce n’était pas voulu de ma part ou qui estiment que la partie les mentionnant est trop courte ou factuellement incorrecte.

Merci aussi à la carbonara qui est, de loin, mon plat préféré, et qui me donnera toujours envie de continuer, à Sergio Leone, Billy Wilder, Stanley Kubrick, The Beatles, Fabrizio de André, Mozart, Beethoven, Vivaldi et tous ces artistes qui ont aussi façonné ce que je suis aujourd’hui.

Merci aussi à Pokam Amos Djoukui Roux, premier de ses noms, deuxième de ses prénoms, Grokam, Pokamour, Pokamos, Pokamore, Amos, Gramos, Amorino, Petit Pô, Popino et Kakarot. Pas pour mes heures de sommeil, pas pour mes heures de jeu, pas pour l’avancée de ma rédaction ni même de ma thèse mais pour tout ce qui est à venir et pour avoir été le premier jour du reste de ma vie. Tu es mon accomplissement m’ayant demandé le plus de temps et d’investissement mais tu en vaux le coup (par contre apprends vite à lire histoire que ces remerciements ne servent pas à rien).

Me voilà arrivé à la fin de ces courts remerciements. Bien entendu, cette thèse lui est dédicacée mais Le Chacal Djoukui Ornella Vanessa Paradis Maternelle Fentchen Bibiche, Djofeuh, amour de ma vie à venir et de ces 6,5 dernières années, mère de mon fils mais surtout pour tout ce que tu es pour moi et que tous les poèmes, mails et autres écrits que nous nous sommes envoyés ne sauraient décrire pleinement, merci. Merci d'avoir été ma coloc, ma femme, mon amie, mon amante, ma source d'engueulades (qu'est-ce que j'aime quand on s'engueule, c'est devenu un art), de rires, de joies et de tant d'autres émotions. Je pourrais encore écrire des pages et des pages pour te remercier. Te remercier pour les sériethons, les films, les jeux, cette capacité admirable que tu as de consoler les gens, ta façon inorthodoxe de jouer, tes danses et tes chants de joie et je pourrais continuer encore et encore mais bon, merci pour tout.

Et enfin, merci à moi. Pour tout.

Table des matières

Table des figures

Liste des algorithmes

Chapitre 1

Introduction

On le dit, on le répète, on l'entend, on le voit, on le ressent, les systèmes informatiques sont de plus en plus complexes, de plus en plus présents et de moins en moins bugués... Malheureusement non. Complexes ? Oui. Omniprésents ? Bien évidemment ! Sans bugs ? Absolument pas !

Comme la plupart des choses que les humains créent, on réfléchit aux conséquences trop tard et on y met moins de moyens. Combien de millions d'euros (de dollars, de yen, de yuans...) pour des millions de lignes de code pour de nouveaux systèmes d'exploitation, pour faire décoller à peu près tout ce qui peut décoller, pour des voitures intelligentes, pour des IA qui jouent à DotA 2, au Go, aux échecs puis qui jouent avec notre argent ? Combien de personnes dans ces grandes entreprises destinées à collecter nos données, à les manipuler, à nous aider à communiquer, à nous déplacer, à travailler ? Et pendant que l'on crée du neuf combien de systèmes d'exploitations exploités jusqu'à la moelle par quantité de virus, combien de catastrophes industrielles, combien de données confidentielles ayant fuité ? Et alors nous, pauvres humains, que pouvons nous faire ? Tester, vérifier, certifier, assurer, standardiser. Faire tout cela avec le peu de moyens qui nous sont donnés mais avec nos mains et nos têtes pour compenser.

Il faut malheureusement se rendre à l'évidence, le monde informatique fuit vers l'avant. Tout est trop complexe, trop gros, vérifier que les outils font ce qu'ils doivent faire et ne font pas ce qu'ils ne doivent surtout pas faire prend du temps, de l'argent, des gens.

1993, Windows NT 3.1, entre quatre et cinq millions de lignes de code.

Moins de 10 ans après Windows XP est commercialisé. Ce système contiendra à sa sortie 45 millions de lignes de code.

Le premier noyau de Linux contenait, lui, une dizaine de milliers de ligne de code en 1991.

En 2015 la dernière version du noyau en contenait 20 millions.

On peut discuter la pertinence d'une telle mesure mais il est assez naturel d'imaginer qu'il est impossible, humainement parlant, de vérifier que ces systèmes n'ont pas de bugs.

Un document de 70 pages listant l'ensemble des bugs connus sur les processeurs de la gamme Xeon produits par Intel a été publié en 2015 [41]. La plupart de ces bugs ne peuvent pas être réparés et ne

peuvent qu'être contournés en invitant a posteriori les utilisateurs à faire ou pas certaines choses afin de ne pas provoquer ces bugs comme on a pu le voir avec les deux dernières grandes failles ayant été révélées sur les microprocesseurs modernes que sont Spectre et Meltdown [38].

Une grande quantité de bugs pourraient être découverts avant même l'implémentation des logiciels ou la fabrication des composants et c'est dans ce cadre là que sont nées plusieurs disciplines afin de vérifier que les nouveaux systèmes que l'on souhaite implémenter ou fabriquer se comportent comme on le souhaite. Un domaine en particulier attirera tout particulièrement notre attention dans ce document : la vérification de modèles (appelée aussi *model checking*) qui fait partie du champ plus large des méthodes formelles.

De façon succincte, les méthodes formelles sont l'ensemble des méthodes permettant de raisonner à l'aide de la logique sur l'ensemble des composants constituant le matériel informatique actuel (programmes, matériel informatique ...). Ces méthodes sont utilisées pour vérifier que ces composants sont valides par rapport à des spécifications données. On le voit aujourd'hui avec l'importance donnée par certaines grandes compagnies aux langages fortement typés (Facebook avec Reason [61] ou Mozilla avec Rust [62] par exemple), mais d'autres méthodes sont utilisées telles que l'interprétation abstraite, la vérification déductive et, dans le cas qui nous intéresse, donc, la vérification de modèles que je présente rapidement ici avant de la décrire plus précisément dans le Chapitre 3. Cette technique se base sur l'introduction de la logique temporelle par Pnueli [58] et Owicki et Lamport [54] afin de pouvoir exprimer des propriétés de programmes et notamment :

— la *sûreté* : rien de mauvais n'arrive
— la *vivacité* : quelque chose de voulu finira par arriver

En se basant sur cette nouvelle logique, Clarke et Emerson [29, 22, 23] ainsi que Queille [59] accompagné ensuite par Sifakis [60] fondent les bases de la vérification de modèle. Le terme de modèle dans ce domaine se réfère à deux sens différents. Le premier est lié au fait qu'on modélise le système qu'on cherche à vérifier (qu'il soit électronique ou informatique) plutôt que de travailler directement sur le système (ce qui convient parfaitement à la problématique à laquelle on cherche à répondre, i.e. comment vérifier un système avant de le fabriquer ?). Le deuxième est lié au fait qu'on vérifie que cette modélisation est bien un modèle (dans le sens qu'on lui donne dans la théorie des modèles) des propriétés qu'on souhaite vérifier. La vérification de modèles est donc une technique cherchant à savoir automatiquement si une modélisation vérifie des propriétés données.

À l'origine limitée à des systèmes finis, la vérification de modèles a été étendue aux systèmes paramétrés (que l'on décrira plus précisément dans le Chapitre 3) par Clarke et al. [16], German et Sistla [32] et Abdulla et al. [1, 3, 5]. Le cadre théorique des systèmes paramétrés pose de gros problèmes de terminaison et d'indécidabilité comme l'ont montré Apt et Kozen [8] puis, plus tard, Abdulla et al. [1].

Plusieurs techniques ont été mises en place pour vérifier cette classe de système notamment la vérification par expressions régulières (*regular model checking*) [14, 13, 11, 12, 53] consistant à représenter les états par des mots d'un langage, les ensembles d'états par des expressions régulières et les transitions par des automates finis permettant de passer d'une expression régulière à une autre. Les techniques qui en découlent et que je vais brièvement décrire maintenant, contrairement aux autres citées, ont pour grand intérêt de ne pas essayer de réduire le problème traité à un problème fini (soit en mettant en évidence une bisimulation entre le système paramétré et le système fini, soit en créant des abstractions

finies de ces systèmes qu'on prouve correcte, soit en raisonnant sur des compteurs, par exemple) mais en embrassant pleinement leur aspect paramétré au moyen de l'utilisation de quantificateurs afin de représenter les états de la modélisation du système (dorénavant appelée "*système*" pour ne pas surcharger la lecture) au moyen de formules logiques.

Issus du model checking régulier, on s'intéresse aux deux outils que sont PFS [57] et Undip [66]. Ces deux outils ont été créés par Abulla et al. [2], Undip étant une amélioration de PFS dans le sens où il permet de manipuler des domaines de valeurs potentiellement infinis contrairement à PFS qui se limite aux variables booléennes et à des domaines finis tels que les énumérations. La contribution majeure qu'ils ont proposée est la notion de système bien structuré et la possibilité, donc, de modéliser les systèmes et leurs états par des représentations finies. Forts de ces représentations caractérisant des ensembles d'états potentiellement infini, il devient alors possible de vérifier qu'ils contiennent déjà de nouveaux états découverts afin d'atteindre rapidement un point fixe. Cette technique a permis de s'affranchir des techniques énumératives et a fait entrer la vérification de modèles paramétrés dans le domaine du symbolique. La problématique majeure liée à leur approche est due aux limitations sur la topologie des systèmes étudiés qui imposent de réimplémenter intégralement leurs algorithmes de vérification en cas de changement de topologie.

Ces techniques ont néanmoins permis d'apprécier la puissance des méthodes symboliques et de commencer à dessiner les contours d'une classe de systèmes où le problème de l'atteignabilité serait décidable et terminerait (ce qui n'est absolument pas le cas dans un cadre général autorisant tout type de formule, de quantification etc [1]). Ghirlardi et Ranise ont alors proposé une technique basée sur des systèmes à tableaux et la puissance des solveurs SMT dans [33, 35] appelée MCMT (pour Model Checking Modulo Theories). Contrairement aux deux outils précédents, cette technique utilise des formules du premier ordre pour représenter non seulement les variables des états mais aussi pour décrire la topologie du système. Le problème d'atteignabilité avec ses calculs de pré-image et de point fixe se réduit ainsi à manipuler des formules de la logique du premier ordre et à se servir de la puissance des solveurs SMT modernes mais, même dans ces conditions, il faut imposer des conditions assez strictes pour assurer la décidabilité et la terminaison de ces systèmes comme les auteurs le montrent dans [34].

Dans le cadre mis en place dans MCMT les systèmes ne sont représentés que par des tableaux de taille infinie et par des transitions entre ces tableaux et leur vérification est assurée par une analyse par atteignabilité arrière (donc par calcul de pré-images depuis la négation de la propriété qu'on souhaite prouver) et par un moteur de point fixe. Cette technique revient donc à calculer l'ensemble clos des états mauvais du système en vérifiant qu'il n'intersecte pas les états initiaux de celui-ci. C'est dans ce cadre que Cubicle a été implémenté comme décrit dans la thèse d'Alain Mebsout [51]. Je décris dans le Chapitre 3 le fonctionnement de ce logiciel qui a été implémenté afin de vérifier automatiquement des propriétés de sûreté de systèmes paramétrés complexes (avec, donc, les problématiques de passage à l'échelle inhérentes aux méthodes automatiques). L'approche fondamentale de cette thèse a été de générer des invariants du système en expurgeant les formules obtenues par calcul de pré-image de leurs littéraux inutiles à la résolution. Afin de vérifier que ces nouvelles formules obtenues en supprimant des littéraux sont a priori pertinentes, elles sont comparées à une exécution en avant sur un domaine fini du système appelée *oracle* et ne sont pas prises en compte s'il existe un état de l'oracle leur correspondant.

S'il est bien une chose, néanmoins, que PFS, Undip et MCMT ont réglé en la contournant, c'est la problématique des propriétés de vivacité ou, tout du moins, de la quantification universelle dans les propriétés à prouver. J'aborderai cette problématique plus précisément dans le Chapitre 6 mais le

problème découle du fait que ce type de formule fait sortir la vérification du cadre assurant la terminaison et la décidabilité. La solution communément trouvée par les auteurs de ces logiciels a donc été d'abstraire les systèmes afin de faire disparaître les quantificateurs universels [4, 6, 7]. La version actuelle de Cubicle elle-même se contente de les traiter comme s'ils n'existaient presque pas [51]. L'inconvénient de ces méthodes est que bien que si un système est prouvé correct avec leurs transformations alors il l'est sans, un système peut être prouvé incorrect à cause de ces transformations alors qu'il est en réalité tout à fait sûr.

Il est vain et orgueilleux de penser régler le problème de la vérification au moyen d'une seule méthode et, personnellement, je tends à penser que tant qu'on produira toujours plus et en changeant en permanence de façon de faire (que ce soit en changeant de paradigmes de programmation, de langages ou autre) sans laisser le temps aux scientifiques, théoricien·ne·s et autres personnes travaillant à élaborer des outils de vérification, à réfléchir aux théories de la conception de programme et à faire en sorte, globalement, qu'on sache pourquoi tout cela fonctionne ou pas, sans leur laisser le temps, donc, de trouver des solutions à ces problèmes aussi vite qu'ils sont créés, l'informatique tombera malade alors qu'elle donne l'impression de s'étendre à de plus en plus d'objets de notre vie quotidienne. N'étant ni devin ni n'ayant envie de l'être, je n'ai tenté donc que d'ajouter ma petite pierre au domaine de la vérification de programmes. Car il en est d'une thèse comme d'une vie, face à l'immensité du monde on se sent petit mais chaque pierre compte.

La problématique principale à laquelle j'ai tenté de répondre dans cette thèse a donc été la suivante :

Comment faire en sorte que Cubicle soit plus performant et puisse être utilisé pour vérifier des problématiques différentes ?

1.1 Contributions

Mes contributions sont les suivantes :

- Au moment de commencer cette thèse, Cubicle avait permis de prouver automatiquement pour la première fois l'architecture complexe multiprocesseur FLASH [26] au moyen de techniques novatrices d'inférence d'invariants, notamment. J'ai donc fait en sorte d'améliorer l'algorithme d'atteignabilité en en implémentant une nouvelle version présentée dans le Chapitre 4 ainsi qu'en améliorant l'inférence d'invariants au moyen de techniques issues du domaine de l'intelligence artificielle dans le Chapitre 5. Cette contribution se décompose donc en deux parties,
  - L'implémentation dans Cubicle d'un nouvel algorithme d'atteignabilité appelé FAR inspiré des travaux faits sur IC3 par Bradley ayant conduit à une publication [24],
  - L'implémentation de techniques nouvelles d'atteignabilité avant sur des systèmes finis afin d'obtenir un oracle plus performant et moins gourmand.
- La deuxième partie de cette thèse s'est concentrée sur la possibilité de prouver automatiquement d'autres types de propriétés étant donné l'intérêt que Cubicle provoquait dans les communautés de l'algorithmique distribuée (en France par exemple avec le projet ANR Pardi [55]). Les difficultés liées à ces propriétés sont notamment dues au caractère indécidable de leur vérification automatique dans le cadre paramétré. Je me suis donc inspiré de techniques venues du domaine

de la vérification déductive mais aussi de la vérification de modèles dans le cadre de formules universellement quantifiées pour prouver de façon automatique le splitter, protocole qui jusque là n'avait été prouvé que manuellement. C'est, à ma connaissance, la première fois qu'un système paramétré contenant des quantificateurs universels est prouvé non pas en contournant au moyen d'abstractions ou de manipulations syntaxiques ces quantificateurs mais en les incluant pleinement dans le travail de vérification. Ce travail est présenté dans le Chapitre 6 et a été présenté également dans une publication [27],

— La dernière contribution est une traduction automatique de protocoles de Cubicle vers Why3 afin de soulager une grande partie de la recherche d'invariants tout en laissant à tout utilisateur·rice la possibilité d'en ajouter manuellement. Cette traduction n'est pas triviale car nécessite de mettre en place des techniques afin de rendre séquentiels des systèmes naturellement concurrents et non déterministes.

Chapitre 2

État de l'art

2.1 IC3/PDR et autres algorithmes d'atteignabilité

Les algorithmes de vérification peuvent généralement être séparés en deux catégories, les algorithmes énumératifs et les algorithmes dirigés par les propriétés. Les algorithmes énumératifs, qui ne nous intéresseront plus pour le reste de ce document, calculent l'ensemble des états atteignables par un programme afin d'en vérifier les propriétés voulues. Ces méthodes sont malheureusement rapidement limitées par la cardinalité de l'ensemble de ces états et ne rentrent de toutes façons pas du tout dans le cadre de mes travaux. Les autres méthodes, celles dirigées par les propriétés, se basent donc sur celles-ci pour créer des abstractions du système, les raffiner, les étudier etc.

L'idée principale derrière la vérification de modèle dirigée par une propriété P est de vérifier que P reste vraie au cours de l'exécution du système auquel elle est associée. Dans le cadre spécifique de Cubicle, un système est représenté par un ensemble T de transitions, comme on le verra dans le chapitre suivant, et on cherche à savoir si $P \wedge T \Rightarrow P'$ [1]. Plusieurs méthodes ont été mises en place pour vérifier s'il existe un cube [2] c inclus dans P tel que $c \wedge T \not\Rightarrow P'$ et à raffiner P en ajoutant la négation de c. Ces algorithmes cherchent donc à trouver ces cubes afin de renforcer la propriété P jusqu'à trouver une propriété représentant un invariant inductif du système ou jusqu'à ce qu'on découvre que le système étudié n'est pas correct vis à vis de la propriété recherchée.

Au cours des années plusieurs méthodes ont tenté de résoudre ce genre de problèmes. On pense notamment à l'analyse d'atteignabilité arrière (partir de $\neg P$ et calculer l'ensemble des états l'atteignant jusqu'à atteindre un point fixe – cf. Chapitre 3), méthode ayant l'inconvénient de ne pas s'intéresser aux états atteignables depuis l'état initial et de faire une analyse non guidée mais dont les avantages liés aux requêtes SMT assez simples, aux possibilités d'approximations et au fait de se concentrer sur la propriété à prouver ont permis l'émergence d'outils très performants tels que NuSMV, Blast, TLC et Cubicle. On peut mentionner les techniques de vérification de modèle bornées (BMC pour *Bounded Model Checking*) qui ont l'inconvénient majeur de demander une grande puissance aux solveurs du fait de requêtes extrêmement lourdes et d'entrer bien trop souvent dans des fragments indécidables ce qui

1. Le $'$ permet d'indiquer qu'on parle de P dans le système après exécution de T
2. Toute formule de la forme $x_1 \wedge \neg x_2 \wedge x_3 \wedge \ldots$ avec x_i des littéraux

a été résolu en partie par la k-induction consistant à vérifier non pas que $P \wedge T \implies P'$ mais que $P^0 \wedge T \wedge P^1 \wedge T \wedge ... \wedge T \Rightarrow P^k$ donc vérifier que la propriété P reste vraie si elle était vraie dans les k exécutions précédentes.

On ne compte plus les améliorations pour ces algorithmes et cela ne nous intéresse pas beaucoup ici car c'est dans ce contexte que l'algorithme IC3 (pour *Incremental Construction of Inductive Clauses for Indubitable Correctness*) a été présenté pour la première fois par Bradley [15]. Cet algorithme a tenté d'utiliser la puissance des approches mentionnées et de les combiner en une seule. On construit donc des formules dont la première représente l'état initial et les suivantes représentent une sur-approximation des états atteignables du système exécution après exécution (l'analyse est donc restreinte par le fait de représenter des états atteignables). Chacune de ces formules doit néanmoins impliquer la propriété P et la recherche s'arrête s'il en existe une qui ne satisfait pas cette condition (l'analyse est ainsi guidée par la propriété à prouver). IC3 mélange donc l'approche incrémentale (prouver successivement un ensemble de petites formules qu'on ajoute une à une) de l'analyse d'atteignabilité arrière et l'approche monolithique (on crée une formule de plus en plus grosse qu'on essaye de prouver d'un seul coup) de la k-induction.

Cet algorithme a rapidement intéressé la communauté du fait de son mélange savant de techniques afin d'en accentuer les qualités tout en en effaçant les défauts. Il a donc été très rapidement suivi par de nombreux travaux de recherche que ce soit chez McMillan qui a amélioré son algorithme d'abstractions paresseuses [48, 49, 50] ou chez Cimatti et al. [17, 18, 19, 20, 21] ou encore chez les développeurs de Z3 [40] [3]. Cet algorithme ne se contente pas uniquement de renforcer l'ensemble des invariants permettant de prouver la propriété (donc de chercher un modèle) mais dirige précisément la recherche de contre-exemples (donc cherche à résoudre les conflits).

Lors de l'implémentation d'un algorithme similaire dans Cubicle, il a donc été tout à fait naturel de chercher à savoir ce qu'il en était autre part dans le monde. Cubicle a une caractéristique assez unique dans le domaine des vérificateurs de modèle, c'est un outil qui vérifie des systèmes non déterministes paramétrés donc infinis. La plupart des outils cités sont utilisés dans des contextes purement booléens ou sur des systèmes finis. On peut donc trouver des outils comme TLC [46] (dont on peut trouver une riche bibliographie sur le site de l'auteur) ou μZ [40], l'outil de Microsoft basé sur Z3. Bien que cet outil se targue de gérer les systèmes infinis, il s'avère que la documentation se contente malheureusement de ne traiter que des problèmes non paramétrés et dont l'infinité provient des domaines infinis des variables et que mes diverses expériences pour traduire des fichiers de Cubicle se sont avérées infructueuses.

Il convient de mentionner PFS, Undip et MCMT dans les vérificateurs de modèles paramétrés. La version de Cubicle sur laquelle je me suis basé était déjà plus compétitive et plus expressive ce qui que ces outils ce qui, dans le cas de cette thèse, rendait la comparaison peu pertinente étant donné que le but était d'étendre l'expressivité de Cubicle et de le rendre plus rapide.

3. Qui en profitent au passage pour renommer cette classe d'algorithme PDR pour *Property Directed Reachability*.

2.2 Algorithmes distribués et preuve déductive de programmes concurrents

Dans le dernier chapitre de cette thèse j'aborde la problématique de la vérification de modèle pour des algorithmes distribués. Cette nouvelle problématique touche à deux domaines distincts que j'ai tenté de rapprocher, la vérification automatique d'algorithmes distribués et la preuve déductive dans le cadre des programmes concurrents. Pourquoi ne pas s'être intéressé uniquement à la vérification automatique d'algorithmes distribués ? Car comme le prouvent les différents outils s'y étant attaqués, que ce soit ByMC dans [36, 43] justifiant leur décision de ne pas automatiser la recherche d'invariants par le fait que la paramétricité des systèmes rend la vérification de modèle indécidable (ce que l'on verra en effet dans le Chapitre 6) ou DVF [65] qui se rapproche plus d'un vérificateur de preuve que d'un vérificateur automatique de modèle et demande donc une certaine expertise humaine pour fonctionner, la vérification automatique de cette classe de problème est impossible. Étant donné, donc, que la preuve automatique s'avère impossible, je me suis tourné vers la preuve déductive de programme. Le problème immédiat lié à cette approche est que la plupart des outils (tous ceux à ma connaissance, du moins), ne gèrent pas nativement la concurrence. Il est possible de transformer des programmes concurrents en gardant la même sémantique comme ce qui est fait par Blanchard et al. [10] où les programmes sont transformés en programmes séquentiels dont on garde toutes les traces possibles d'exécution. L'idée que je développe dans le Chapitre 6 s'inspire de cette technique mais on remarquera que la bibliographie traitant de ce sujet dans la littérature actuelle est assez réduite notamment par volonté d'améliorer la preuve de programmes séquentiels.

2.3 Vérification de modèle et intelligence artificielle

Les techniques d'intelligence artificielle devenant de plus en plus performantes (on pense par exemple à AlphaGo [63] suivi de AlphaGo Zero [64] il est tout à fait naturel que la communauté de la vérification de modèle tente de s'en emparer comme on le voit par exemple dans le domaine de la vérification de modèle probabiliste [44] et notamment Goldman et al. [37] qui tentent d'utiliser la méthode de Monte-Carlo pour résoudre des problèmes de vérification probabiliste. On trouve néanmoins pour le moment surtout des travaux orientés vers l'adaptation des techniques de la vérification de modèle pour vérifier des IA [42] plutôt que l'utilisation des techniques de l'IA pour améliorer la vérification de modèle. Dans cette catégorie, la plupart des techniques récemment développées font appel à l'apprentissage automatique (*machine learning*) afin de découvrir des invariants [30] mettant en place une forme d'oracle en boîte noire qui apprend au fur et à mesure au moyen d'exemples, de contre-exemples et d'implications. Cet algorithme, ICE, a donné naissance aux algorithmes de type MLIS (pour *Machine Learning-based Invariant Synthesis*) qui ont été décrits et étudiés par Vizel et al. [67] et Garg et al. [31]. Ces types d'algorithmes offrent un nouveau paradigme où les invariants ne sont pas sûrs et peuvent conduire à des traces fallacieuses mais, grâce à un mécanisme de *teacher* et de *learner*, l'algorithme de vérification finit par faire émerger des invariants de plus en plus robustes permettant de vérifier les systèmes qui lui sont donnés. Vizel et al. [67] notent une forte similarité entre cette approche et les approches faites dans la vérification de modèle basées sur les problèmes SAT tout en remarquant des différences fondamentales

entre les deux[4]. Du fait de la relative jeunesse de ces travaux il s'avère que la plupart sont à l'état de prototypes ou d'idée mais qu'il semble évident qu'un nouveau domaine est en train de se créer grâce à la puissance combinée des solveurs SMT modernes et des techniques d'apprentissage.

4. On remarquerea notamment la similarité profonde qu'il y a entre BRAB, décrit au Chapitre 5, et ICE, bien que ni [67] ni [30] ne le citent. Cela est notamment dû au fait que beaucoup de techniques actuelles sont inspirées de techniques d'intelligence artificielle sans pour autant le mentionner explicitement.

Chapitre 3

Cubicle : présentation et état actuel

3.1 Qu'est-ce que Cubicle ?

Ce document se base sur le vérificateur de modèles pour systèmes paramétrés nommé Cubicle. Ce chapitre servira de familiarisation à tout·e lecteur·rice avec cet outil, les problèmes qu'il permet de résoudre ainsi qu'une certaine formalisation.

3.1.1 Un vérificateurs de modèles pour systèmes paramétrés

Voilà par quoi tout commence et ce qu'il convient donc d'expliquer dès à présent en décomposant cette description en ses unités élémentaires.

3.1.1.a La vérification de modèles

Tout système admet des propriétés qu'on peut énoncer lors de leur conception, de leur fonctionnement ou encore de leur panne. Lorsque je mets de l'argent dans un distributeur de boisson me permettant de produire une thèse j'espère que ce distributeur me la fournira [1]. Ceci est une propriété et même, plus précisément, une propriété de *vivacité*. Ce type de propriété indique que mon programme finira bien par faire ce qu'il est censé faire au bout d'un moment. Ces propriétés de vivacité incluent aussi bien «si je tourne la clé dans la serrure, la porte s'ouvrira» que «si je vote comme 50% des gens, notre candidat·e sera être élu·e» [2].

En revanche, lorsque je demande à ce distributeur de me fournir ma boisson, si celle-ci est froide alors que je l'avais demandée chaude, sucrée alors que je la voulais sans sucre ou tout autre «bug» du système, je suis en droit de m'offusquer au sujet de cette machine qui ne fait pas ce qu'elle est censée faire. Ce type de propriété est appelée *sûreté*. Elle indique des résultats que notre programme n'est jamais censé

1. La boisson, pas la thèse.
2. On remarque que cette propriété, en fonction des systèmes électoraux, n'est pas assurée.

atteindre. On y trouve des propriétés telles que «ma porte ne doit jamais pouvoir se fermer toute seule» ou «aucun·e président·e ne doit pouvoir prendre les pleins pouvoirs du pays» [3].

Un *vérificateur de modèles* (ou *model checker*) est un programme qui vérifie que des systèmes assurent bien ces deux types de propriétés. C'est donc un programme qui prend un système et les propriétés que l'on souhaite sur ce système et qui vérifie que le système les satisfait. Contrairement au test, le vérificateur de modèles n'opère pas directement sur la machine mais sur une abstraction de celle-ci. Dans notre exemple, plutôt que de fournir la machine à café je fournis une modélisation de celle-ci comme on peut le voir Figure 3.1.

À Retenir

La *vérification de modèles* (ou *model checking*) est un domaine dans lequel on fournit le modèle abstrait d'un système dont on vérifie qu'il satisfait certaines propriétés.

3.1.1.b Les systèmes paramétrés

Avant de comprendre ce que l'on entend par «paramétré» intéressons-nous à l'infini. Qu'est-ce qui distingue un système fini d'un système infini ? Reprenons notre distributeur [4]. Qu'est-ce qu'une modélisation simple de ce système ? On sait qu'il attend une commande et qu'à la réception de celle-ci, lorsque de l'argent a été fourni, il l'honore et se remet en attente. Ce fonctionnement est schématisé à la Figure 3.1.

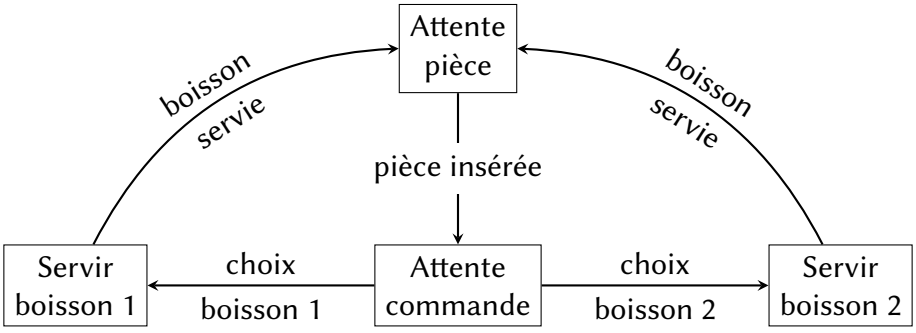


FIGURE 3.1 – Modélisation simple du distributeur

Les états possibles de ce distributeur sont, comme on peut le voir, au nombre de quatre :

— en attente d'une pièce
— en attente de commande
— servant la première boisson
— servant la seconde boisson

Ce système est donc un système fini car le nombre d'états possibles différents qu'il peut atteindre est fini (bien que son exécution soit infinie, on reviendra forcément à un état déjà vu précédemment donc le nombre d'états à considérer est fini).

3. Et, comme avec tous les systèmes politiques, cette propriété n'est malheureusement pas non plus toujours assurée.
4. Qui nous sert définitivement beaucoup.

En revanche, ce distributeur pourrait proposer un nombre infini de boissons comme on peut le voir Figure 3.2 ou garder en mémoire la quantité de pièces reçues comme représenté Figure 3.3.

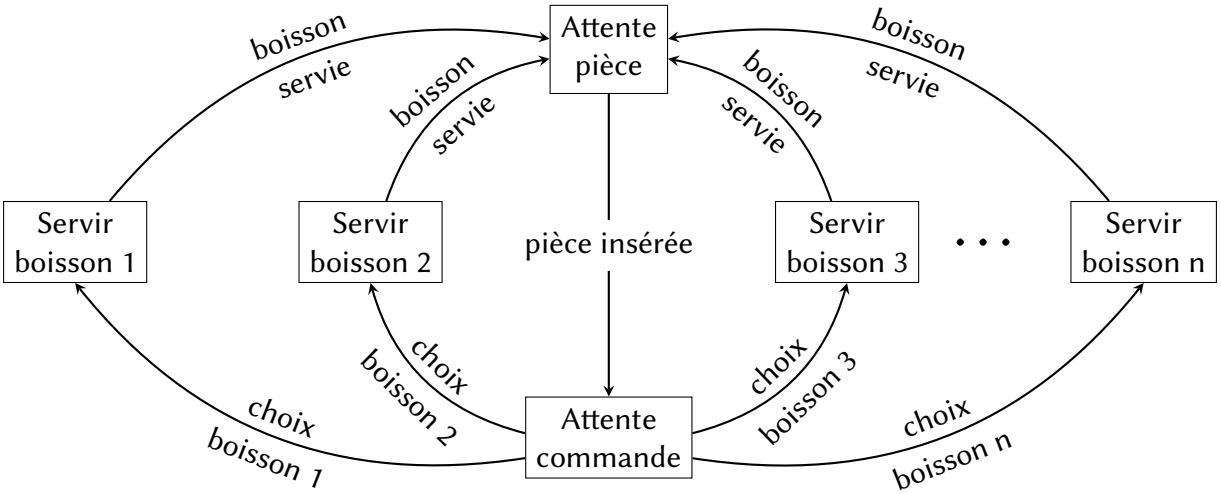


FIGURE 3.2 – Modélisation du distributeur avec un nombre infini de boissons

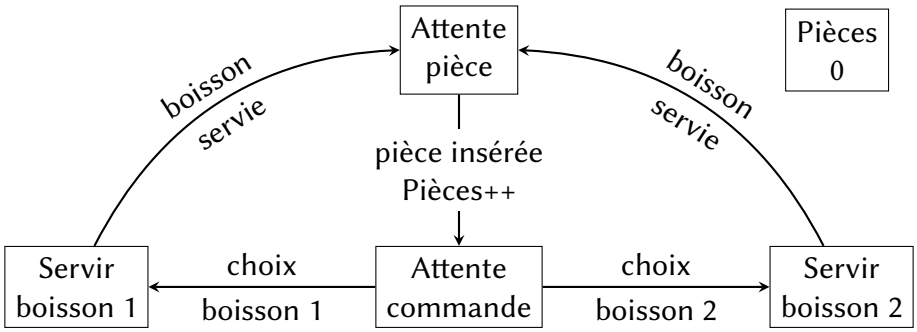


FIGURE 3.3 – Modélisation du distributeur avec mémorisation du nombre de pièces insérées

Les états atteignables seraient alors, dans le cas de la Figure 3.2 :

— en attente d'une pièce
— en attente de commande
— servant la première boisson
— servant la seconde boisson
— servant la troisième boisson
  ...
— servant la énième boisson

Et dans le cas de la Figure 3.3 :

— en attente d'une pièce, pièces = 0
— en attente de commande, pièces = 1
— servant la première boisson, pièces = 1
— servant la seconde boisson, pièces = 1
— en attente d'une pièce, pièces = 1

— en attente de commande, pièces = 2
— servant la première boisson, pièces = 2
— servant la seconde boisson, pièces = 2
...

Ces deux listes pourraient être continuées indéfiniment. Il est intéressant de remarquer que le fait que l'automate soit fini comme dans la Figure 3.3 ou la Figure 3.1 ou infini comme dans la Figure 3.2 n'indique pas que notre système a un nombre d'états atteignables fini ou infini. L'infini dans un système peut donc venir de ses variables prenant leurs valeurs dans un domaine infini ($\mathbb{N}$ dans le cas de notre distributeur comptant les pièces) ou d'un nombre inconnu de composants.

Je reparlerai plus tard du premier cas. Gérer l'infini découlant du domaine des variables est une tâche compliquée et très intéressante mais qui n'a pas été traitée lors de cette thèse.

Le deuxième cas, en revanche, m'intéresse bien plus car c'est sur cela que Cubicle se concentre : un nombre inconnu de composant. Une façon de représenter cet infini est de paramétrer le nombre de composants. Plutôt que de dire qu'on en a un nombre inconnu, on dit qu'on en a n sans spécifier n. Comme on peut le voir dans la Figure 3.2, l'automate n'est pas réellement infini mais le nombre n est inconnu. On retrouve ces systèmes dans les protocoles de cohérence de cache, notamment, mais aussi dans l'algorithmique distribuée et bien d'autres domaines manipulant des systèmes impliquant un nombre inconnu de parties ayant toutes le même fonctionnement.

Nous voici donc arrivés aux systèmes gérés par Cubicle : les systèmes paramétrés. Comment les décrit-on en pratique ? Sur quelles théories cette description se base-t-elle ? Comment gère-t-on la preuve de ces systèmes ? Voilà ce à quoi je vais de ce pas m'atteler à répondre.

À Retenir

Cubicle est un logiciel vérifiant les propriétés de sûreté (*rien de mauvais n'arrive*) de systèmes paramétrés (*c'est-à-dire comprenant un nombre inconnu de composants ayant tous le même comportement*).

3.1.2 Langage d'entrée

3.1.2.a Types et déclarations

Avant de prouver les propriétés de sûreté de nos systèmes, il s'agit de pouvoir les exprimer dans un langage suffisamment riche. Le langage choisi pour Cubicle se base sur le formalisme des systèmes à tableaux conçu par Ghilardi et Ranise [34] que je présenterai brièvement dans la sous-section 3.1.3.

Comme tout système, les systèmes paramétrés sont composés de ressources sous formes de variables, tableaux etc et d'opérations permettant de modifier ces ressources (incrémentation, affectation etc). Dans le cadre de Cubicle, les ressources peuvent prendre deux formes uniquement :

— des variables globales
— des tableaux

Ces ressources sont typées statiquement grâce à quatre types internes

— les entiers : int
— les booléens : bool
— les réels : real
— les processus : proc

Les trois premiers types sont bien connus, le quatrième peut sembler étrange. Ce type représente justement les identifiants de nos composants. Cubicle ayant été construit historiquement sur des protocoles de cohérence de cache, il traitait des processus cherchant à accéder à des sections critiques et le nom proc a donc été choisi comme type pour représenter les éléments par lequel le système est paramétré. La paramétricité vient donc du caractère inconnu de la cardinalité du domaine de proc.

Afin de ne pas limiter les systèmes à des manipulations de nombres et de booléens, de nouveaux types peuvent être créés selon deux constructions :

— les types abstraits [5]

```
type abst
```

— les types énumérés

```
type enum = A | B | C
```

Avant de me lancer plus loin dans la description du langage de Cubicle, je préfère me baser sur un exemple trivial me permettant d'illustrer aisément mes propos. Un des premiers algorithmes venant en tête lorsqu'il s'agit de vérification de sûreté d'un protocole paramétré est l'algorithme d'exclusion mutuelle. Cet algorithme met chaque processus en attente active afin d'accéder à une même ressource. Chaque processus est identifié par un numéro unique et une variable à laquelle chacun à accès en lecture indique quel processus peut entrer en section critique. De façon schématique, cet algorithme se comporte comme illustré dans la Figure 3.4

Cette figure schématise le fait que tout processus peut entrer dans les états Idle ou Want sans condition mais que pour entrer dans l'état critique il faut que la variable globale `Turn` ait la valeur du processus. Nous allons donc voir comment ce simple algorithme est implémenté en Cubicle.

Forts des types énumérés et des types internes à Cubicle, il devient possible de créer de nouveaux tableaux et variables. Étant donné qu'ils représentent les valeurs propres à chaque composant, les tableaux sont indicés uniquement par des éléments de type proc et sont donc de taille inconnue. La possibilité de rendre le langage de Cubicle plus riche en ajoutant de nouveaux types et de nouvelles constructions se pose évidemment mais ne semble pas nécessaire dans l'état actuel des travaux faits dessus. Dans notre exemple, il a été décidé de modéliser cet algorithme grâce à un tableau représentant l'état de chaque processus et une variable globale `Turn`.

Le début du fichier Cubicle correspondant est donné dans la Figure 3.5.

5. Qui ne seront pas du tout traités dans ce document.

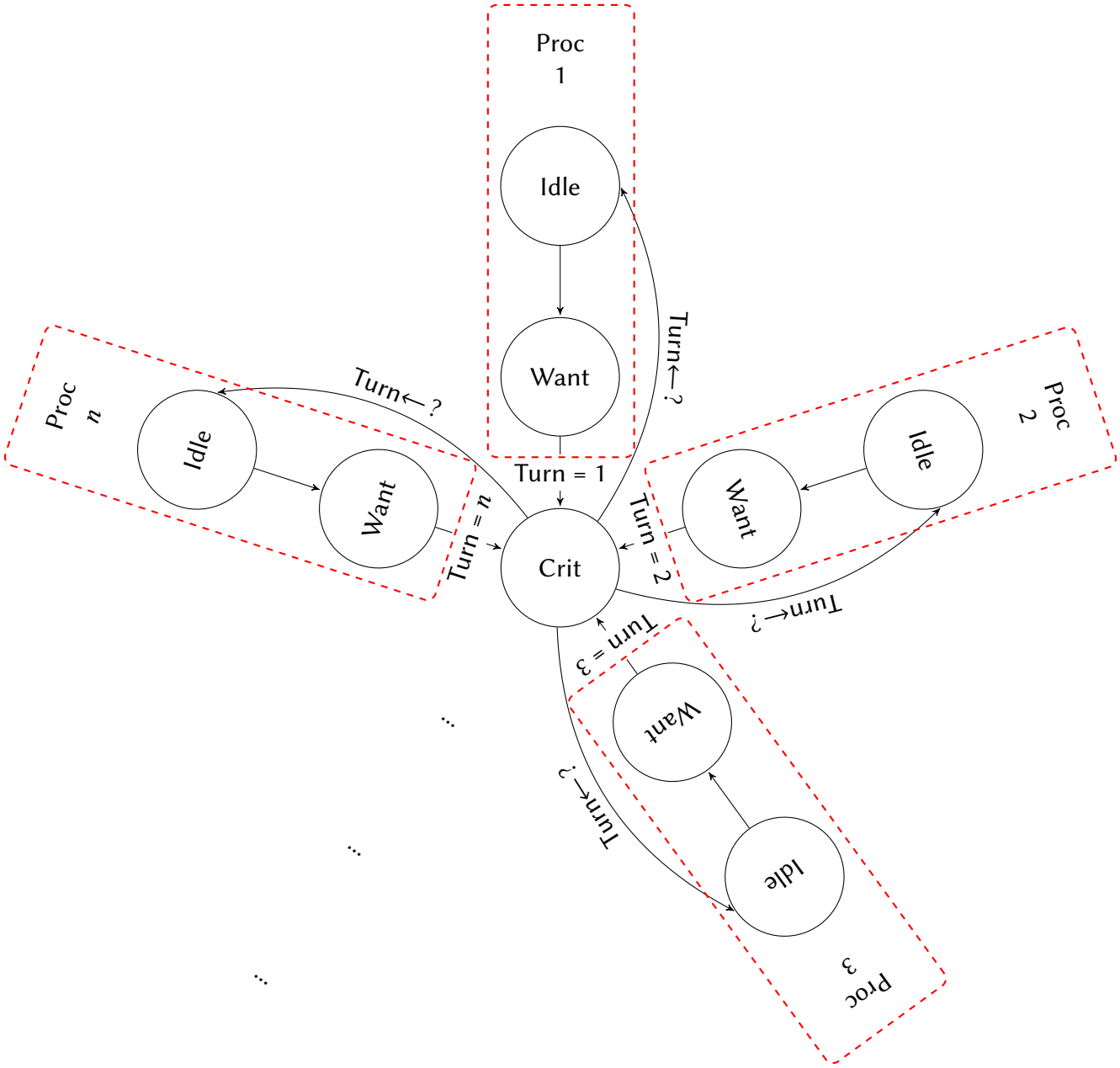


FIGURE 3.4 – Modélisation d'un algorithme simple d'exclusion mutuelle

À Retenir

Cubicle contient quatre types internes, int, bool, real et proc ainsi que la possibilité de définir des types énumérés avec la syntaxe type enum = A | B | C.

Il est possible de définir des variables avec la syntaxe var nom_var : type et des tableaux avec la syntaxe array nom_tab[proc (ou proc, proc, ...)] : type.

```
type state = Idle | Want | Crit

var Turn : proc
array State[proc] : state
```

FIGURE 3.5 – Déclaration des types, variables et tableaux pour l'algorithme d'exclusion mutuelle

3.1.2.b États initiaux des systèmes

Nous voici donc avec de nouveaux types, de nouvelles variables et de nouveaux tableaux. Il nous faut maintenant la possibilité de décrire notre système c'est-à-dire d'exprimer clairement dans quel état il commence et comment il se transforme. Le premier point d'entrée d'un programme Cubicle est la description de son état initial sous la forme d'une conjonction de littéraux. Dans le cadre de notre algorithme exemple, la variable `Turn` a une valeur arbitraire et chaque case du tableau `State` est initialisée à `Idle` comme on peut le voir dans la Figure 3.6

```
init (z) { State[z] = Idle }
```

FIGURE 3.6 – Formule `init` de l'algorithme d'exclusion mutuelle

Au vu de cette figure, il semble tout à fait normal de s'interroger sur la façon de la lire. Du fait des limitations de Cubicle en terme d'indexation des tableaux, la formule `init` est paramétrée par un unique identifiant de processus (ou aucun). Cet identifiant est implicitement quantifié universellement et permet donc de lire la formule précédente comme

$$init \equiv \forall z.State[z] = Idle$$

On remarque aussi que `Turn` n'apparaît pas dans cette formule. La formule `init` représente en effet l'ensemble des états possibles en en contraignant certaines variables et tableaux. Lorsqu'une variable ou un tableau ne sont pas contraints, cela signifie qu'ils peuvent prendre quelque valeur que ce soit dans le domaine de leur type. `Turn` étant un proc, les états initiaux du système sont donc les états où le tableau `State` est rempli exclusivement de `Idle` et où la variable `Turn` est égale à 0, 1, 2... Bien que cela n'apparaisse pas ici, il est aussi possible de mentionner simplement des relations entre les variables et/ou les tableaux. On remarquera l'absence de précision des types des paramètres, cela est dû au fait que les paramètres ne peuvent être que des proc.

À Retenir

La formule `init` est une formule universellement quantifiée contraignant la valeur de certaines variables et tableaux au moyen d'une conjonction de littéraux.

Tout état pour lequel les valeurs des variables et des tableaux respectent ces contraintes caractérise un état initial correct du système.

3.1.2.c États mauvais des systèmes

Les propriétés de sûreté de Cubicle indiquent quels sont les états *mauvais* du système. On indique donc dans notre programme quels états on ne veut surtout pas qu'il atteigne. Dû aux limitations du formalisme de MCMT, ces propriétés sont des conjonctions existentiellement quantifiées et impliquant donc un nombre fini (bien qu'on ne sache pas exactement lesquels) de processus.

Dans le cas de l'algorithme d'exclusion mutuelle la propriété de sûreté qu'on veut assurer en priorité est qu'aucun processus ne peut entrer en section critique si un autre processus y est déjà. Cette propriété se caractérise dans le langage de Cubicle comme montré dans la Figure 3.7.

```
unsafe (z1 z2) { State[z1] = Crit && State[z2] = Crit }
```

FIGURE 3.7 – Formule `unsafe` de l'algorithme d'exclusion mutuelle

Cette formule doit être lue comme

$$\exists z1 z2.z1 \neq z2 \wedge State[z1] = Crit \wedge State[z2] = Crit$$

On remarque notamment que chaque processus paramètre de la formule est implicitement distinct des autres sans qu'il soit nécessaire de le préciser.

À Retenir

Un fichier Cubicle peut contenir plusieurs formules `unsafe` exprimant les états mauvais du système.

Ces formules sont des conjonctions de littéraux existentiellement quantifiées par des processus distincts.

3.1.2.d Transformations des états des systèmes - Les transitions gardées

Arrivés à ce stade, nous avons les états initiaux de notre système, les états qu'il ne doit pas atteindre et il nous reste donc à décrire les transitions permettant de passer d'un état à un autre. Chaque transition est paramétrée par zéro ou plus processus distincts et ne peut s'exécuter que si sa garde est vérifiée par l'état courant du système, d'où l'appellation de *transition gardée*.

La troisième transition de l'algorithme d'exclusion mutuelle, par exemple, est présentée dans la Figure 3.8.

Cette transition, paramétrée par un unique processus `i` indique que si le système est dans un état où le tableau `State` contient la valeur `Crit` à l'indice `i` alors le nouvel état du système sera un état dans lequel la variable `Turn` contient une valeur arbitraire et où `State` contiendra `Idle` à l'indice `i` et la même valeur que précédemment dans toutes les autres cases.

```
transition exit (i)
requires { State[i] = Crit }
{
  Turn := . ;
  State[k] := case
              | k = i : Idle
              | _ : State[k];
}
```

FIGURE 3.8 – Troisième transition de l'algorithme d'exclusion mutuelle

La garde de chaque transition est précédée du mot clé `requires`. Cette garde contient une formule composée de conjonctions et disjonctions de littéraux portant sur les variables et tableaux du système ainsi que sur les paramètres de la transition. Il est en effet possible de comparer les paramètres dans la garde d'une transition du fait de l'existence d'un ordre total entre chaque identifiant de processus. Une transition peut donc parfaitement ressembler à transition t(i j) requires {i < j && ...}.

Il est aussi tout à fait possible de conditionner l'exécution d'une transition à l'état du reste du système. Afin de le faire, il suffit d'ajouter le mot clé forall_other suivi d'un identifiant de processus dans la garde. Si on voulait, par exemple, s'affranchir de la variable `Turn` dans notre exemple, on pourrait très bien écrire une nouvelle transition `enter_bis` de la forme suivante :

```
transition enter_bis (i)
requires { State[i] = Want && forall_other j. State[j] != Crit}
{ State[i] := Crit; }
```

FIGURE 3.9 – Transition avec garde globale pour l'algorithme d'exclusion mutuelle

Il est intéressant de remarquer que le filtrage par motif utilisé dans les actions des transitions pour les mises à jour de tableau a été remplacé par une affectation simple State[i] := Crit;. Ce sucre syntaxique permet d'alléger le code tout en gardant la même sémantique de non modification du reste des cases du tableau.

Nous verrons au Chapitre 6 pourquoi ces gardes contenant une partie globale peuvent s'avérer problématique en pratique.

Enfin, une fois que la garde a été vérifiée, la mise à jour du système peut avoir lieu. Cette mise à jour est considérée comme atomique, ce qui signifie que chaque mise à jour de variable ou tableau se fait simultanément. Ainsi, toute variable ou tableau apparaissant à droite du signe := décrit la valeur de ceux-ci avant que la transition ait été exécutée. Les tableaux peuvent être mis à jour de deux façons différentes bien que la seconde soit du sucre syntaxique pour la première :

— grâce à un filtrage avec la construction tab[variable fraîche] := case | cond1 : action_1 | cond2 :

action_2 | ... | _ : action_defaut (cf. Figure 3.8)
— avec une simple affectation grâce à la construction tab[variable liée à un paramètre] := nouvelle valeur (cf. Figure 3.9)

Enfin, il faut remarquer l'utilisation de la valeur spéciale . permettant de donner une nouvelle valeur arbitraire comme on le voit pour Turn := ..

Le fichier résultant est donné dans la Figure 3.10.

```
type state = Idle | Want | Crit

var Turn : proc
array State[proc] : state

init (z) { State[z] = Idle }

unsafe (z1 z2) { State[z1] = Crit && State[z2] = Crit }

transition req (i)
requires { State[i] = Idle }
{ State[i] := Want }

transition enter (i)
requires { State[i] = Want && Turn = i}
{ State[i] := Crit; }

transition exit (i)
requires { State[i] = Crit }
{
  Turn := . ;
  State[i] := Idle;
}
```

FIGURE 3.10 – Code Cubicle de l'algorithme d'exclusion mutuelle

À Retenir

Les transitions gardées de Cubicle se présentent sous la forme

```
transition nom (paramètres)
requires { Garde }
{ Action }
```

Il peut y avoir zéro, un ou plusieurs paramètres.

La garde est une formule logique portant sur les paramètres et les variables et tableaux du système ainsi qu'une partie globale paramétrée par les identifiants différents des paramètres.

L'action est un ensemble de mises à jour atomiques des tableaux et variables du système. Toutes les variables et tableaux non mentionnés demeurent inchangés.

3.1.3 Bref formalisme des systèmes à tableaux

Cubicle est basé sur un cadre théorique défini par Ghilardi et Ranise [34] appelé *model checking modulo théories* (MCMT). Ce cadre définit les systèmes comme des ensembles de transitions gardées portant sur des tableaux infinis. L'ensemble d'un système est intégralement défini grâce à des formules appartenant à un fragment restreint de la logique du premier ordre. Cette formalisation permet d'utiliser

la puissance des solveurs SMT actuels afin de décharger une grande partie du travail de preuve sur leur capacité à gérer de multiples théories.

Les théories impliquées dans MCMT sont les suivantes :

— $\mathcal{T}_P$, une théorie des processus admettant proc pour symbole de type. La satisfiabilité de $\mathcal{T}_P$ est décidable dans le fragment défini par MCMT
— $\mathcal{T}_{E_i}$, un ensemble de théories sur les éléments (les valeurs des systèmes). De même que pour $\mathcal{T}_P$, la satisfiabilité de $\mathcal{T}_{E_i}$ est décidable dans le fragment défini par MCMT
— $\mathcal{T}_A$, une théorie d'accès (l'ensemble des variables et tableaux des systèmes). La satisfiabilité de $\mathcal{T}_A$ est décidable si l'ensemble de ces symboles est de la forme proc $\times \cdots \times$ proc $\rightarrow$ $\texttt{elem}_i$ (le nombre de proc à gauche de la flèche pouvant être bien évidemment nul, ce qui correspond aux variables du système) c'est-à-dire lorsque les tableaux ne contiennent que des indices de type proc et ne peuvent admettre de valeur de type proc .

Ce fragment de logique est étendu d'un système de types permettant de distinguer les éléments selon leur type afin d'autoriser ou pas certaines opérations. Dans la pratique, cela revient à ranger les éléments des différentes théories dans un triplet (T, F, R) avec :

— T un ensemble d'identifiants de types
— F un ensemble de constantes ou de fonctions étant associés aux identifiants de types de T
— R un ensemble de relations entre les types (le premier prédicat étant l'égalité entre deux éléments du même type

Ce triplet, appelé *signature*, dénoté Σ, est décomposé en sous-triplets $(\Sigma_{\mathcal{T}_i})$ pour chaque théorie dont l'union donne la *signature* du système considéré.

Dans le cadre de l'algorithme d'exclusion mutuelle, par exemple, on a besoin de trois signatures

— $\Sigma_{\mathcal{T}_P} = (\{\text{proc}\}, \emptyset, \{=\})$
— $\Sigma_{\mathcal{T}_{state}} = (\{\texttt{state}\}, \{\texttt{Idle}, \texttt{Want}, \texttt{Crit}\}, \{=\})$
— $\Sigma_{\mathcal{T}_A} = (\emptyset, \{\texttt{State} : \text{proc} \rightarrow \texttt{state}, \texttt{Turn} : \text{proc}\}, \emptyset) \cup \Sigma_{\mathcal{T}_P} \cup \Sigma_{\mathcal{T}_{state}}$

Les systèmes à tableaux sont donc des ensembles logiques basés sur ses théories et leurs signatures décomposés en un quadruplet $\mathcal{S} = (Q, \mathcal{I}, \mathcal{T}, \mathcal{U})$ tel que :

— Q est décomposé en $Q_0, Q_1, \ldots, Q_n$ représentant les ensembles de symboles d'arité $0, 1, \ldots, n$. Q_0 représente donc l'ensemble des variables du système. Q_1, ..., Q_n représentent les tableaux à 1, ..., n dimensions (indicés, pour les raisons de décidabilité vues précédemment, par des proc)
— $\mathcal{I}$ est, comme on l'a vu dans dans la sous-sous-section 3.1.2.b, une formule quantifiée universellement sur une variable de processus caractérisant les états initiaux du système au moyen d'une conjonction de littéraux portant sur les éléments de Q
— $\mathcal{T}$, conformément à la sous-sous-section 3.1.2.d, représente l'ensemble des transitions du système c'est-à-dire des formules de la forme

$$t(Q^i, Q^{i+1}) = \exists \vec{i}\, \mathcal{G}(\vec{i}, Q^i) \wedge \forall \vec{j}\, \mathcal{A}(\vec{i}, \vec{j}, Q^i, Q^{i+1})$$

  — $\vec{i}$ est l'ensemble des processus servant de paramètres à la transition
  — $\mathcal{G}$ représente la garde de la transition c'est-à-dire l'ensemble des conditions nécessaires à son

exécution. Ces conditions portent sur $\vec{i}$ et l'état actuel du système donc l'ensemble de ses symboles Q.
- $\mathcal{A}$ représente l'ensemble des modifications du système effectuées par la transition (autrement appelé *action*) et porte sur les paramètres $\vec{i}$, l'ensemble des processus du système $\vec{j}$, l'état actuel du système Q et l'état futur du système (donc après exécution) Q'. Par convention on admet que toute variable (ou état) étant suffixée d'un prime représente cette même variable (ou état) après exécution de l'action d'une transition.

— $\mathcal{U}$ représente une disjonction de conjonctions de littéraux portant sur des symboles de Q quantifiées existentiellement sur des processus et caractérisant les états mauvais du système dont la sémantique a été donnée dans la sous-sous-section 3.1.2.c

Du point de vue logique, donc, notre algorithme d'exclusion mutuelle est représenté de la façon suivante :

— $Q = Q_0 \cup Q_1$ avec :
  — $Q_0 = \{\texttt{Turn}\}$
  — $Q_1 = \{\texttt{State}\}$
— $\mathcal{I} = \forall i.\texttt{State}[i] = \texttt{Idle}$
— $\mathcal{T} = t_{req} \vee t_{enter} \vee t_{exit}$ avec :
  — t_{req} : $\exists p.$ $\texttt{State}[p] = \texttt{Idle} \wedge$ $\forall q.\texttt{State}'[q] = \texttt{if } p = q \texttt{ then Want else State}[q] \wedge \texttt{Turn}' = \texttt{Turn}$
  — t_{enter} : $\exists p.$ $\texttt{State}[p] = \texttt{Want} \wedge \texttt{Turn} = p$ $\forall q.\texttt{State}'[q] = \texttt{if } p = q \texttt{ then Crit else State}[q] \wedge \texttt{Turn}' = \texttt{Turn}$
  — t_{exit} : $\exists p.$ $\texttt{State}[p] = \texttt{Crit} \wedge$ $\forall q.\texttt{State}'[q] = \texttt{if } p = q \texttt{ then Idle else State}[q]$
— $\mathcal{U} = \exists p, q.p \neq q \wedge \texttt{State}[p] = \texttt{Crit} \wedge \texttt{State}[q] = \texttt{Crit}$

3.1.3.a Comment vérifier la sûreté des systèmes à tableaux ?

On remarquera qu'à la différence de Cubicle qui, lorsqu'on ne mentionne pas une variable (ou une case de tableau) l'interprète comme si elle n'avait pas été modifiée, dans la logique permettant de le formaliser le contraire a lieu, toute variable (ou case de tableau) non mentionnée aura une valeur arbitraire après une transition. C'est pourquoi les mises à jour de tableaux se font avec des instructions conditionnelles quantifiées universellement sur l'ensemble des processus. C'est aussi la raison pour laquelle dans la formule de t_{exit}, Turn' n'est pas mentionnée afin d'assurer que sa nouvelle valeur sera arbitraire.

Le problème de sûreté revient donc à vérifier qu'il n'existe aucune suite de transitions t_1, t_2, ..., t_n de $\mathcal{T}$ telle que :

$$\mathcal{I}(Q^0) \wedge t_1(Q^1, Q^2) \wedge \cdots \wedge t_n(Q^{n-1}, Q^n) \wedge \mathcal{U}(Q^n)$$ [6]

Pour vérifier si une telle suite existe ou pas un moteur de recherche d'atteignabilité par chaînage arrière a été implémenté dans Cubicle. L'idée derrière le chaînage arrière est de construire progressivement l'ensemble des états pouvant atteindre $\mathcal{U}$ en partant de cet état.

6. $\mathcal{U}(Q^n)$ signifiant que Q^n est un modèle de $\mathcal{U}$

La pré-image $\text{Pre}_{\mathcal{T}}(Q^i)$ d'un état Q^i par $\mathcal{T}$, l'ensemble des transitions du système, est un ensemble d'états Q^{i-1} tels que pour chacun il existe $t \in \mathcal{T}$ telle que $t(Q^{i-1}, Q^i)$. Dans le cadre de l'algorithme d'exclusion mutuelle, les premières pré-images de $\mathcal{U}$ sont données dans la Figure 3.11 (on instancie les processus par des constantes de processus $\#_i$ afin de se débarrasser des quantificateurs existentiels mais cela ne change rien au fonctionnement de l'algorithme).

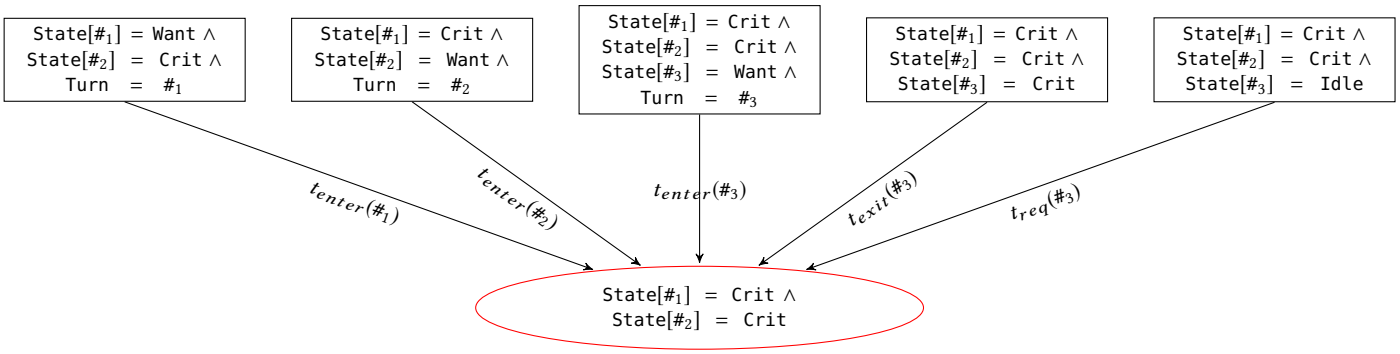


FIGURE 3.11 – Première rangée de pré-images de $\mathcal{U}$ pour l'algorithme d'exclusion mutuelle

La condition de terminaison de l'algorithme d'atteignabilité arrière est soit de trouver un état qui correspond à un état initial soit de trouver une clôture $\mathcal{V}$ de $\text{Pre}_{\mathcal{T}}$ contenant $\mathcal{U}$ c'est-à-dire de calculer un ensemble clôt par la relation $\text{Pre}_{\mathcal{T}}$ contenant $\mathcal{U}$ et tel que pour toute transition appliquée à n'importe quel état de $\mathcal{V}$, l'état résultant appartiendra à $\mathcal{V}$. Pour ce faire il faut pouvoir supprimer des nœuds qu'on ne veut pas traiter lors du calcul de la pré-image.

On remarque immédiatement que, du fait de la paramétricité du système, il est possible de voir apparaître de nouveaux processus lors du calcul de la pré-image. On pourrait continuer à calculer des pré-images avec $\#_4$, $\#_5$ etc mais ses variables étant sémantiquement liées à des quantificateurs existentiels, il suffit de les renommer pour retomber sur des états déjà générés. C'est pourquoi dans la Figure 3.11 on se limite à $\#_3$. Cela permet d'éviter une génération infinie d'états à chaque étape correspondant au nombre infinis de processus possibles.

Pour la même raison de renommage on remarque que la formule du deuxième état

$$\texttt{State}[\#_1] = \texttt{Crit} \wedge \texttt{State}[\#_2] = \texttt{Want} \wedge \texttt{Turn} = \#_2$$

correspond à la formule du premier état où $\#_1$ et $\#_2$ ont été intervertis. Le deuxième état peut donc être éliminé sans calculer ses pré-images car on obtiendra les mêmes, à renommage prêt, que pour le premier.

Enfin, les trois derniers états sont $\mathcal{U}$ à laquelle on a rajouté des littéraux. Ces états sont donc subsumés par $\mathcal{U}$ et calculer leur pré-image reviendrait à augmenter le nombre de processus impliqués sans rien apprendre de nouveau et avec le risque de provoquer une explosion du nombre d'états calculés. Ainsi, après la première étape de calcul de pré-image et le filtrage des nœuds redondants, il ne reste qu'un seul nœud comme illustré dans la Figure 3.12.

L'algorithme d'atteignabilité arrière est donné dans l'Algorithme 1. Cet algorithme commence par créer $\mathcal{V}$, l'ensemble des nœuds visités (la clôture, donc, de Pre), initialement vide, ainsi que Q, la file d'attente des nœuds à traiter, contenant initialement $\mathcal{U}$ (étant donné que $\mathcal{U}$ est une disjonction de conjonctions existentiellement quantifiées, Q peut contenir plusieurs nœuds, chacun correspondant à

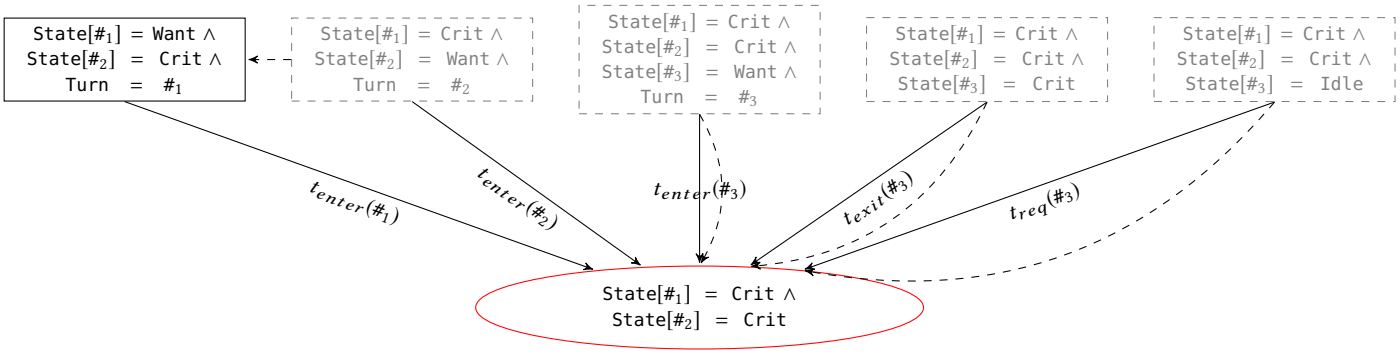


FIGURE 3.12 – Première rangée de pré-images de $\mathcal{U}$ pour l'algorithme d'exclusion mutuelle après filtrage des nœuds redondants

une des conjonctions). Tant que la file n'est pas vide, le nœud suivant est sélectionné, on vérifie qu'il ne correspond pas à un état initial possible auquel cas le système serait mauvais (lignes 6 et 7) et, si ce n'est pas le cas et que le nœud n'est pas redondant par rapport aux nœuds déjà traités (ligne 8) alors l'ensemble de ses pré-images est ajouté à la file. Lorsque la file est vide, on a réussi à trouver la clôture de PRE et donc un point fixe du système qui n'a aucune intersection avec les états initiaux ce qui garantit la sûreté de celui-ci (ligne 13). L'aspect SMT est utilisé aux lignes 6 et 8 bien qu'en pratique un test ensembliste fonctionne dans la plupart des cas pour le test de redondance.

Algorithme 1: Atteignabilité arrière

```
   Input: S = (Q, I, T, U)
   Variables:
       V : nœuds visités
       Q : file d'attente
1  function BWD(S) : begin
2      V := ∅;
3      push(Q, U);
4      while not_empty(Q) do
5          φ := pop(Q);
6          if (I ∧ φ sat) then
7              renvoyer unsafe
8          else if (φ ⊭ V) then
9              V := V ∪ {φ};
10             push(Q, Pre(φ));
11         end
12     end
13     renvoyer safe
14 end
```

Pour l'algorithme d'exclusion mutuelle, à l'issue de son exécution, le graphe des nœuds visités est donné dans la Figure 3.13. Il est intéressant de remarquer que l'invariant qui viendrait à l'esprit de toute

personne rencontrant cet algorithme :

$$\forall i.\texttt{State}[i] = \texttt{Crit} \implies \texttt{Turn} = i$$

Formulé de façon négative :

$$\exists i.\texttt{State}[i] = \texttt{Crit} \wedge \texttt{Turn} \neq i$$

n'a pas été découvert par Cubicle. Il apparaît bien dans les deux nœuds traités lors de l'analyse d'atteignabilité mais dans les deux cas avec un littéral supplémentaire. Bien que dans notre cas ceci ne soit pas dérangeant vu le peu de nœuds nécessaires à la preuve de sûreté il peut s'avérer problématique pour des algorithmes bien plus complexes. On verra dans la section suivante ainsi que dans le Chapitre 5 comment ce problème a été réglé et comment j'ai tenté d'améliorer ce nouvel algorithme.

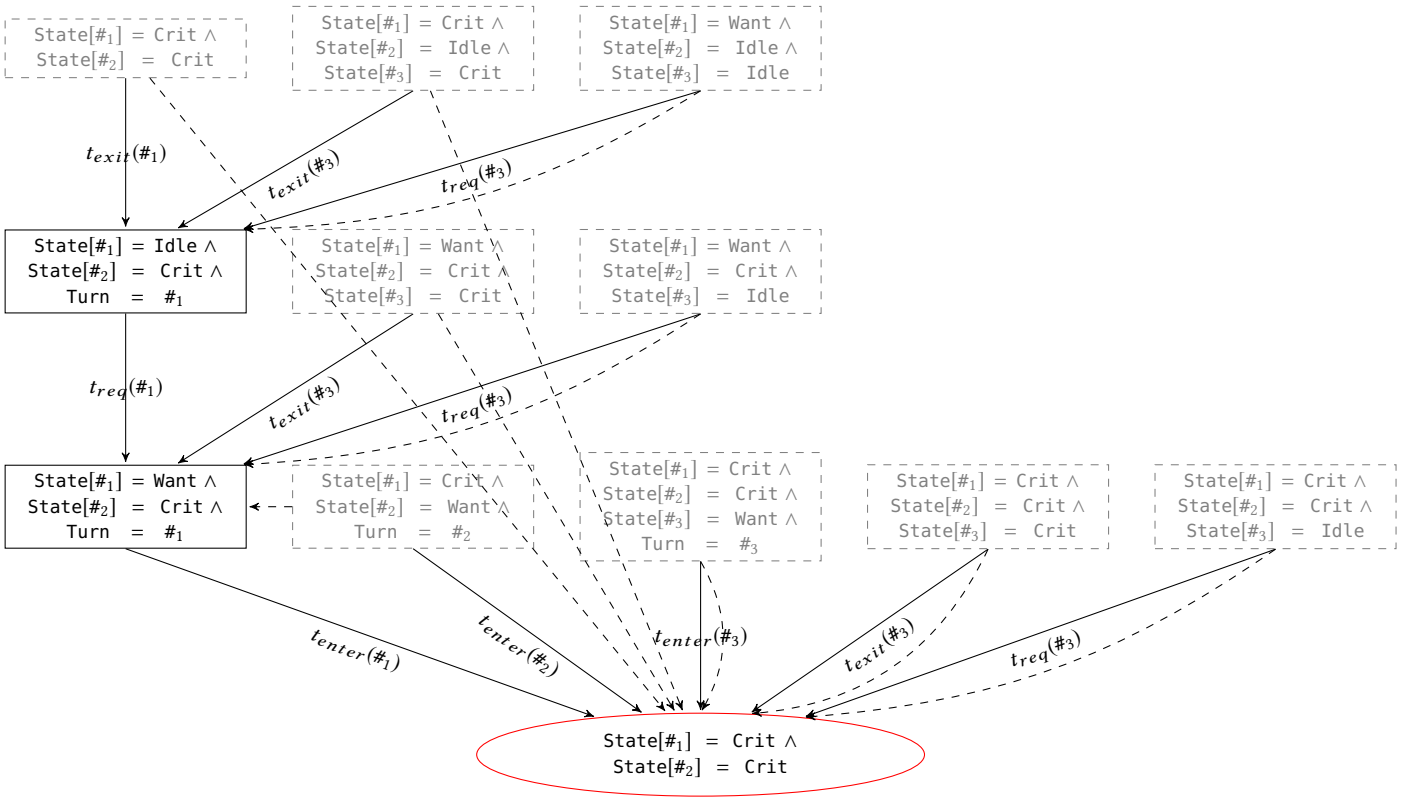


FIGURE 3.13 – Clôture de PRE pour l'algorithme d'exclusion mutuelle

3.2 Que veut-on y ajouter

On l'aura compris, arrivés ici, Cubicle est un vérificateur de modèles. Chercher à améliorer cet outil peut se faire de deux façons différentes :

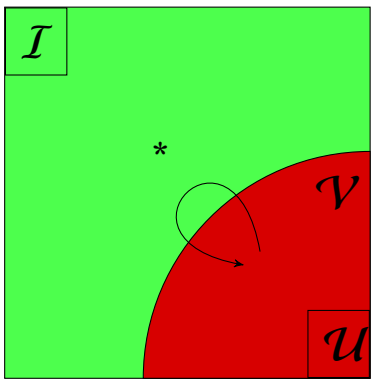
— En améliorant son *efficacité* donc sa rapidité de traitement des fichiers et ainsi permettre de prouver des fichiers bien plus complexes. Pour ce faire on peut donc travailler sur des améliorations ou des modifications de l'algorithme de point fixe comme le verra dans le Chapitre 4 ou sur des améliorations ou modifications de l'oracle pour l'algorithme actuel, ce qui sera expliqué dans le Chapitre 5

— En améliorant son *expressivité* donc le type de formule qu'on peut prouver ce qui sera montré dans le Chapitre 6 avec la vérification de propriétés de validité donc de propriétés mentionnant l'ensemble des processus du système telles que "*à la fin de l'exécution, mon système est mauvais s'il existe un processus ayant décidé une valeur qu'aucun autre processus n'a proposé*". Ce genre de propriété sortant du cadre expressif de Cubicle et, donc, du fragment décidable de celui-ci, il introduit de nouvelles problématiques de résolution que l'on tente de résoudre dans le Chapitre 6

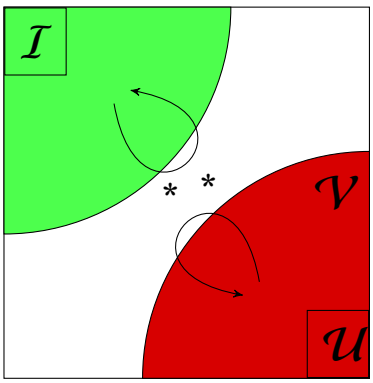
Chapitre 4

Atteignabilité avant abstraite - l'algorithme FAR

L'algorithme actuel utilisé dans Cubicle n'est pas guidé par autre chose que la formule qu'on souhaite prouver. C'est-à-dire que l'on part de la formule $\mathcal{U}$ et on cherche l'ensemble des nœuds permettant d'y accéder en espérant atteindre un point fixe ne contenant pas l'état initial. L'inconvénient de cet algorithme est que pour rester correct il ne permet pas une grande marge de manœuvre. En effet, à chaque étape on calcule des pré-images qu'on peut sur-approximer. Ces sur-approximations ne sont malheureusement basées que sur les pré-images déjà visitées et celles actuellement traitées alors qu'il existe toute une partie du monde qu'on ne prend pas en compte : l'ensemble des états atteignables depuis l'état initial. On peut voir la vision actuelle implémentée dans Cubicle dans la Figure 4.1a et la vision qu'on va chercher à implémenter dans la Figure 4.1b



(a) Du point de vue de l'analyse d'atteignabilité arrière, on ne connaît que les états mauvais



(b) Dans l'algorithme qu'on cherche à implémenter, on veut aussi connaître certains états atteignables et renforcer cette connaissance

FIGURE 4.1 – Deux visions du monde différentes

À l'origine de cette nouvelle approche on trouve IC3 [15] et son implémentation par Eén et al. [28] que l'on va adapter ici à nos besoins. L'idée derrière cette classe d'algorithme est d'être dirigé à la fois par la propriété que l'on souhaite vérifier (comme les algorithmes d'atteignabilité arrière) et par les états que l'on peut atteindre (comme les algorithmes d'atteignabilité avant). Le but à très haut niveau de cet

algorithme est de vérifier que, partant d'une propriété $\mathcal{P}$ décrivant les bons états d'un système, pour un système de transition ayant pour état initial $\mathcal{I}$ et pour ensemble de transitions $\mathcal{T}$, tout état X atteignable vérifie $\mathcal{P}(X)$.

Dans cette classe d'algorithme, on essaye de renforcer une propriété $\mathcal{F}$ en vérifiant d'une part qu'elle est bien une sur-approximation des états atteignables et d'autre part en supprimant ses parties conduisant aux états mauvais.

4.1 Algorithme

L'algorithme que je vais présenter, basé sur les travaux effectués sur IC3/PDR, a été appelé FAR (pour Forward Abstracted Reachability).

Pour rappel, on se base sur le cadre théorique des systèmes de transitions à tableaux. Ces systèmes sont définis comme $\mathcal{S} = (\mathcal{Q}, \mathcal{I}, \mathcal{T}, \mathcal{U})$ avec
- $\mathcal{Q}$: l'ensemble des variables et tableaux du système,
- $\mathcal{I}$: une formule quantifiée universellement sur une variable de processus caractérisant les états initiaux du système au moyen d'une conjonction de littéraux portant sur les éléments de $\mathcal{Q}$,
- $\mathcal{T}$: l'ensemble des transitions du système
- $\mathcal{U}$: l'ensemble des états mauvais du système

L'objectif de cet algorithme est de construire un graphe étiqueté dont chaque sommet v_i est composé de deux parties :
- La première, $\mathcal{W}_i$ (appelée "monde" du sommet) représente une sur-approximation des états atteignables depuis l'état initial en au moins i transitions.
- la deuxième, $\mathcal{B}_i$ (appelée "partie mauvaise" du sommet), représente une sur-approximation des états appartenant à $\mathcal{W}_i$ et atteignant $\mathcal{U}$ en au plus i transitions.

Les arêtes de ce graphe sont étiquetées par des identifiants de transition de $\mathcal{T}$.

Le graphe qu'on tente de construire est donc de la forme $\mathcal{G} = (V, E)$ avec V l'ensemble des sommets et E l'ensemble des arêtes dans $V \times \mathcal{T} \times V$.

Initialement, V contient trois sommets :

- $v_0 = v_{\mathcal{I}}$ représentant le sommet initial tel que :
  - $\mathcal{W}_{v_{\mathcal{I}}} = \mathcal{I}$: à part l'état initial, aucun état n'est censé être atteignable en 0 étape
  - $\mathcal{B}_{v_{\mathcal{I}}} = \bot$: l'état initial n'est pas censé contenir d'état atteignant $\mathcal{U}$ en ne prenant aucune transition)
- $v_1 = v_{\mathcal{U}}$ représentant le sommet mauvais tel que :
  - $\mathcal{W}_{v_{\mathcal{U}}} = \top$: a priori tout état est atteignable en au moins une étape
  - $\mathcal{B}_{v_{\mathcal{U}}} = \mathcal{U}$: si tous les états sont atteignables en au moins une étape alors $\mathcal{U}$ en fait logiquement partie et s'atteint lui-même en au plus une transition.
- v_∞ vers lequel on envoie les arêtes dont la transition ne peut être prise par aucun état représenté par le sommet dont elle part :
  - $\mathcal{W}_\infty = \bot$: puisque toute transition impossible mène à cet état, son monde est vide
  - $\mathcal{B}_\infty = \bot$: le monde de ce sommet étant vide il ne contient aucun sous-état permettant d'at-

teindre $\mathcal{U}$ quel que soit le nombre de transitions prises

Par définition, le sous-ensemble $V^{\mathcal{I}}$ représente l'ensemble des sommets du graphe atteignables depuis le sommet initial donc

$$V^{\mathcal{I}} = \{\mathrm{v} \in V . \mathrm{v}_{\mathcal{I}} \xrightarrow{\tau_i *} \mathrm{v}\}$$

On note aussi $F \models_\tau G$ le fait que $F \wedge \tau \models G$ donc que la mise-à-jour de la formule F par τ est un modèle G.

Définition 4.1.1 *Un graphe est* bien-étiqueté *si*

$$\forall v_1 \xrightarrow{\tau} v_2 \in E.\ \mathcal{W}_{v_1} \models_\tau \mathcal{W}_{v_2}$$
$$\forall v \in V^{\mathcal{I}}.\ \mathcal{B}_v = \bot$$

Définition 4.1.2 *Un graphe est* complet *si*

$$\forall v_1 \in V, \tau \in \mathcal{T}.\ v_1 \neq v_{\mathcal{U}} \wedge \exists v_2 \in V.\ v_1 \xrightarrow{\tau} v_2 \in E$$

Définition 4.1.3 *Une formule Φ est appelée* invariant *du graphe si et seulement si*

$$\forall v \in V^{\mathcal{I}}.\ \mathcal{W}_v \models \Phi$$

4.1.1 FAR non déterministe

L'avantage de cet algorithme est qu'il se base sur un quintuplet de règles qu'on peut appliquer de façon non déterministe jusqu'à atteindre un point où plus une seule n'est applicable ou jusqu'à ce que $\mathcal{B}_{\mathrm{v}_{\mathcal{I}}}$ soit différente de $\bot$. Les règles sont donc les suivantes :

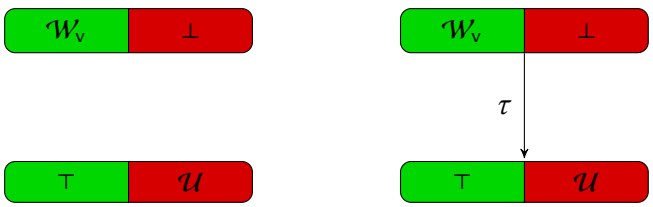
Règle 1 (Étendre)

On choisit un sommet v et une transition τ qui n'a pas encore été considérée pour v telle que $\mathcal{W}_v \models_\tau \top$ et on fait pointer v vers $v_{\mathcal{U}}$.

Paramètre(s) $\begin{cases} \mathrm{v} \in V \\ \tau \in \mathcal{T} \end{cases}$

Condition(s) $\begin{cases} \nexists\, \mathrm{v} \xrightarrow{\tau} \mathrm{u} \in E \\ \mathcal{W}_{\mathrm{v}} \models_\tau \top \end{cases}$

Action(s) $E = E \cup \{\mathrm{v} \xrightarrow{\tau} \mathrm{v}_{\mathcal{U}}\}$



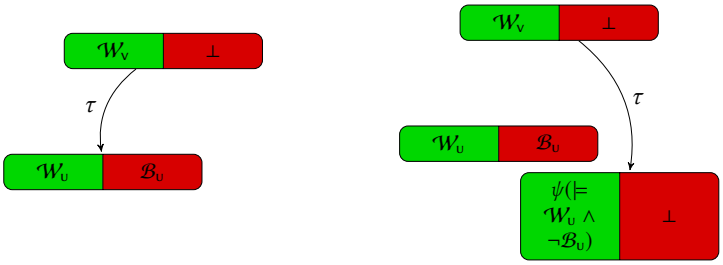
Règle 2 (Raffiner)

On part d'un nœud v *qui pointe par* τ *vers un nœud* u *dont la* partie mauvaise $\mathcal{B}_u$ *est non-vide* ($\neq \bot$). *On vérifie si le* monde $\mathcal{W}_v$ *de* v *implique réellement cette* partie mauvaise *par* τ. *Si ce n'est pas le cas, on crée un nouveau nœud dont le nouveau monde est le monde de* v *qu'on raffine en supprimant la* partie mauvaise *de* v.

Paramètre(s) $\begin{cases} v \xrightarrow{\tau} u \in E \\ \psi \text{ une formule logique} \end{cases}$

Condition(s) $\begin{cases} \mathcal{B}_u \neq \bot \\ \mathcal{W}_v \models_\tau \psi \\ \psi \models \neg \mathcal{B}_u \end{cases}$

Action(s) $\begin{cases} \text{On crée un nouveau nœud } w \\ \hookrightarrow \mathcal{W}_w = \mathcal{W}_u \wedge \neg \mathcal{B}_u \\ E = E \cup \{v \xrightarrow{\tau} w\} \setminus \{v \xrightarrow{\tau} u\} \end{cases}$



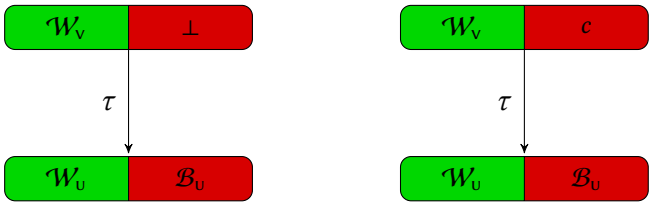
Règle 3 (Propager)

Considérant la transition $v \xrightarrow{\tau} u$ *on cherche à savoir s'il existe une partie du monde représenté par* v *qui, par* τ, *atteint la partie mauvaise de son fils* u. *Cette partie est elle-même mauvaise et on met donc* v *à jour. Comme dans la Règle 2, on part d'un nœud* v *qui pointe par* τ *vers un nœud* u *dont la* partie mauvaise $\mathcal{B}_u$ *est non-vide* ($\neq \bot$). *On vérifie si le* monde $\mathcal{W}_v$ *de* v *contient une formule* c *telle qu'en prenant la transition* τ *depuis* c *on atteint bien la* partie mauvaise *de* u. *Si c'est le cas, on met à jour la* partie mauvaise *de* v *avec* c.

Paramètre(s) $\begin{cases} v \xrightarrow{\tau} u \in E \\ c \text{ une formule logique} \end{cases}$

Condition(s) $\begin{cases} \mathcal{B}_u \neq \bot \\ c \models \mathcal{W}_v \\ c \models_\tau \mathcal{B}_u \end{cases}$

Action(s) $\mathcal{B}_v = c$



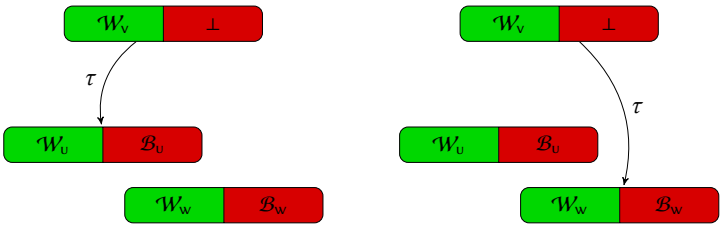
Règle 4 (Couvrir)

Considérant la transition $v \xrightarrow{\tau} u$ *on cherche à savoir s'il existe un nœud* w *dont le monde est inclus dans celui de* u *(représentant moins d'états donc réduisant les possibilités d'en contenir des mauvais) qui contienne toujours tous les états atteignables depuis* v *par* τ.

Paramètre(s) $\begin{cases} v \xrightarrow{\tau} u \in E \\ w \in V \end{cases}$

Condition(s) $\begin{cases} \mathcal{W}_w \models \mathcal{W}_u \\ \mathcal{W}_v \models_\tau \mathcal{W}_w \end{cases}$

Action(s) $E = E \cup \{v \xrightarrow{\tau} w\} \setminus \{v \xrightarrow{\tau} u\}$



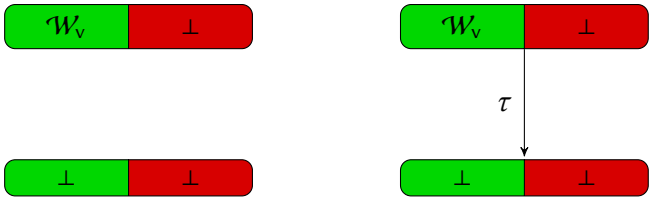
Règle 5 (Puit)

On choisit un nœud v et une transition τ qui n'a pas encore été considérée pour v. Si la garde de τ est incohérente avec le monde décrit par v, on fait pointer v vers v_∞ (qui, par sa construction, est un puit ne permettant d'activer aucune règle).

Paramètre(s) $\begin{cases} v \in V \\ \tau \in \mathcal{T} \end{cases}$

Condition(s) $\begin{cases} \nexists\, v \xrightarrow{\tau} u \in E \\ \quad \mathcal{W}_v \not\models_\tau \top \end{cases}$

Action(s) $E = E \cup \{v \xrightarrow{\tau} v_\infty\}$



L'Algorithme 2 est l'algorithme de base non déterministe qui applique les règles précédentes jusqu'à épuisement ou découverte d'un chemin menant de $\mathcal{I}$ à $\mathcal{U}$. Pour des raisons de clarté et de lisibilité, la cinquième règle n'est pas prise en compte. Bien qu'utile d'un point de vue théorique pour assurer la complétude de la méthode, d'un point de vue pratique une fois que les quatre autres règles ne peuvent plus être appliquées on a épuisé l'ensemble des transitions applicables depuis chaque sommet et le reste sera nécessairement envoyé vers v_∞.

4.2 FAR déterministe

En pratique il est bien évidemment impossible de garder un algorithme non déterministe en espérant obtenir un résultat intéressant en terme de rapidité d'exécution et de terminaison. Dans le cas de FAR il faut donc déterminiser les quatre points suivants :

1. Quelle règle doit-on choisir ?
2. Lorsque la Règle 1 (Étendre) est choisie, avec quel sommet et quelle transition ?
3. Quelle formule ψ choisir pour la Règle 2 (Raffiner) ?
4. Quel sommet doit-on choisir pour la Règle 4 (Couvrir) ?

Je propose donc ci-après un algorithme déterministe correspondant aux quatre premières règles explicitées précédemment.

Algorithme 2: Version non déterministe de FAR

Input: $\mathcal{S}$= $(Q, \mathcal{I}, \mathcal{T}, \mathcal{U})$
Variables:
V : sommets du graphe
E : arêtes du graphe
$v_{\mathcal{I}}$: sommet représentant les états initiaux
$(\mathcal{W}_{v_{\mathcal{I}}} = \mathcal{I}, \mathcal{B}_{v_{\mathcal{I}}} = \bot)$
$v_{\mathcal{U}}$: sommet représentant les états mauvais
$(\mathcal{W}_{v_{\mathcal{U}}} = \bot, \mathcal{B}_{v_{\mathcal{U}}} = \mathcal{U})$

```
1 function FAR(S) : begin
2     V := {v_I, v_U};
3     while ∃Règle(V, E) ∈ {Étendre, Raffiner, Couvrir, Propager} exécutable do
4         Règle(V, E);
5         if B_{v_I} ≠ ⊥ then
6             renvoyer unsafe
7     end
8     renvoyer safe
9 end
```

La fonction principale de cet algorithme initialise le graphe qu'on cherche à construire avec les deux sommets $v_{\mathcal{I}}$ et $v_{\mathcal{U}}$ dans V et aucune arête dans E et une file d'attente avec $v_{\mathcal{I}}$. La boucle principale se contente de sortir un sommet v de la file Q et applique pour chaque transition τ de $\mathcal{T}$ la Règle 1 donc ajoute l'arête $v \xrightarrow{\tau} v_{\mathcal{U}}$ à l'ensemble des arêtes. Avant de passer à la transition suivante, la fonction `déroule` (dont le fonctionnement est décrit dans l'Algorithme 4) est appelée sur l'arête nouvellement créée. S'il ne reste plus aucun sommet à traiter dans la file, le graphe a été intégralement construit et le système est sûr car aucun sommet connecté à $v_{\mathcal{I}}$ ne contient de partie mauvaise non vide.

La fonction `déroule` prend en paramètre une arête $v_1 \xrightarrow{\tau} v_2$ et n'est exécutée que si la partie mauvaise de v_1 est vide et la partie mauvaise de v_2 ne l'est pas. Si tel est le cas, la fonction `ferme` est appelée sur l'arête et renvoie trois résultats possibles :

- Soit il existe déjà un sommet v_3 permettant d'appliquer la Règle 4 c'est-à-dire un sommet dont le monde est plus restrictif que v_2 et tel que $\mathcal{W}_{v_1} \models_\tau \mathcal{W}_{v_3}$. Dans ce cas, l'arête entre v_1 et v_2 est supprimée et une nouvelle arête entre v_1 et v_3 étiquetée par τ est ajoutée. v_3 étant dans la file d'attente ou ayant déjà été traité, il n'est rien besoin de faire pour ce sommet. En revanche, la fonction `déroule()` est rappelée sur cette nouvelle arête, la précédente ayant été supprimée
- Soit il existe une formule $c \in \mathcal{W}_{v_1}$ c'est-à-dire qu'il existe une formule du monde de v_1 qui, par τ, touche la partie mauvaise de v_2. Dans ce cas, soit v_1 est le sommet initial $v_{\mathcal{I}}$ et le système étudié est donc mauvais, soit ce n'est pas le cas et la partie mauvaise de v_1 est mise à jour pour contenir c. Comme on l'avait vu, `déroule()` n'était exécutée que si la partie mauvaise de v_2 n'était pas vide, la partie mauvaise de v_1 étant devenue non vide on appelle la fonction `déroule()` sur toutes les arêtes menant d'un sommet quelconque à v_1
- Soit, dernier cas possible, il s'avère que le monde de v_1 ne touche pas réellement la partie mauvaise de v_2 par τ (donc le monde dans lequel v_1 plonge par τ représenté pour le moment par v_2 est trop général). Un nouveau sommet v_3 a donc été créé pour représenter un monde plus restreint

Algorithme 3: Boucle principale

Input: $\mathcal{S} = (Q, \mathcal{I}, \mathcal{T}, \mathcal{U})$
Variables:
V : sommets du graphe
E : arêtes du graphe étiquetés par une transition de $\mathcal{T}$
Q : file de travail à priorité
$\mathsf{v}_{\mathcal{I}}$: sommet représentant les états initiaux
$(\mathcal{W}_{\mathsf{v}_{\mathcal{I}}} = \mathcal{I}, \mathcal{B}_{\mathsf{v}_{\mathcal{I}}} = \bot)$
$\mathsf{v}_{\mathcal{U}}$: sommet représentant les états mauvais
$(\mathcal{W}_{\mathsf{v}_{\mathcal{U}}} = \bot, \mathcal{B}_{\mathsf{v}_{\mathcal{U}}} = \mathcal{U})$

```
1  fonction FAR(S) : begin
2      V := {v_I, v_U};
3      E := ∅;
4      push(Q, v_I);
5      while not_empty(Q) do
6          v ← pop(Q);
7          foreach τ ∈ T do
8              if W_v ∧ τ satisfiable then
9                  E = E ∪ {v --τ--> v_U};
10                 déroule(v --τ--> v_U)
11         end
12     end
13     renvoyer safe
14 end
```

atteignable depuis v_1 par τ. Comme lors de la couverture (premier cas) l'arête entre v_1 et v_2 est supprimée et une nouvelle arête entre v_1 et v_3 étiquetée par τ est ajoutée. Contrairement au cas de la couverture, en revanche, étant donné que v_3 vient d'être créé, il est ajouté à la file afin d'être traité plus tard.

La dernière fonction à présenter est donc la fonction `ferme` qui est celle traitant directement les arêtes. Cette fonction se contente de vérifier quelles conditions présentes dans les règles 2, 3 et 4 sont vérifiées par l'arête actuelle. Priorité est donnée à la Règle 4 car cette règle permet potentiellement de créer une arête vers un sommet sans partie mauvaise. Si on ne trouve pas de sommet permettant la couverture, on vérifie si le monde de v_1 atteint bien l'état mauvais de v_2 par τ et si tel est le cas on renvoie la sous-formule c de $\mathcal{W}_{\mathsf{v}_1}$ qui permet d'atteindre $\mathcal{B}_{\mathsf{v}_2}$. Sinon, cela signifie que $\mathcal{W}_{\mathsf{v}_2}$ était trop large et on le raffine en créant un nouveau sommet v_3 tel que $\mathcal{W}_{\mathsf{v}_3}$ est $\mathcal{W}_{\mathsf{v}_2}$ amputé de la partie mauvaise de v_2, $\mathcal{B}_{\mathsf{v}_2}$.

4.2.1 Implémentation

Afin de comprendre les choix d'implémentation il est nécessaire de comprendre ce qui distingue cet algorithme de l'analyse d'atteignabilité arrière déjà implémentée dans Cubicle. Dans l'algorithme

Algorithme 4: Construction du graphe

```
1  fonction déroule(v1 -τ-> v2) : begin
2      if B_{v1} = ⊥ ∧ B_{v2} ≠ ⊥ then
3          switch ferme(v1 -τ-> v2) do
4              case Couvert v3 do
5                  E = E ∪ {v1 -τ-> v3} \ {v1 -τ-> v2};
6                  déroule(v1 -τ-> v3)
7              end
8              case Mauvais c do
9                  if v1 = v_I then renvoyer unsafe ;
10                 else
11                     B_{v1} = c;
12                     foreach v -τ'-> v1 do déroule(v -τ'-> v1) ;
13                 end
14             end
15             case Raffiné v3 do
16                 E = E ∪ {v1 -τ-> v3} \ {v1 -τ-> v2};
17                 push (Q, v3)
18             end
19         end
20     end
21     else renvoyer;
22 end
```

Algorithme 5: Règles de base

```
1  fonction ferme(v1 -τ-> v2) : begin
2      if ∃v3.W_{v3} ⊨ W_{v2} ∧ W_{v1} ⊨_τ W_{v3} then
3          renvoyer Couvert v3
4      else if ∃c ∈ W_{v1}.c ∧ τ ∧ B_{v2} satisfiable then
5          renvoyer Mauvais c
6      else
7          create v3 such that
8          W_{v3} = W_{v2} ∧ ¬B(v2);
9          renvoyer Raffiné v3
10     end
11 end
```

d'atteignabilité arrière, le monde est séparé en deux parties, les nœuds mauvais et les autres. A priori, donc, tous les autres nœuds (dans le cadre d'un système sûr) sont considérés comme atteignables depuis l'état initial.

Si, par malheur, on approxime trop les nœuds constituant $\mathcal{V}$ on n'a aucune garantie sur le fait qu'on

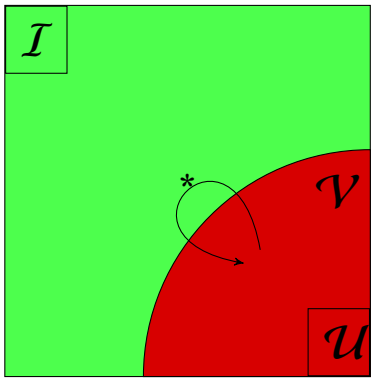


FIGURE 4.7 – Le monde vu du point de vue de l'analyse d'atteignabilité arrière

ne créera pas de faux positifs. Il faudra donc continuer l'exécution de l'algorithme jusqu'à trouver un chemin de $\mathcal{I}$ à $\mathcal{U}$ et se rendre compte qu'il est fallacieux pour recommencer l'exploration. De plus, dans l'analyse d'atteignabilité arrière, l'algorithme fait évoluer au fur et à mesure un monde. Toute erreur devient très difficile à anticiper ou à corriger au moment de sa découverte à cause des problèmes de subsomption. On renvoie les lecteurs voulant une explication précise à la thèse d'Alain Mebsout [51] expliquant très clairement les problèmes liés à la sur-approximation des nœuds . En résumé, lorsqu'une approximation est faite rien n'impose de fermer le nœud approximé avant de fermer les autres et il est donc tout à fait possible que des nœuds d'autres branches soient subsumés par celui-ci et donc pas explorés. Si on supprime le nœud coupable d'une trace fallacieuse sans rien faire d'autre, les nœuds subsumés ne le sont plus et doivent donc être explorés et cela peut entraîner des complications supplémentaires avec des cascades de subsomptions. La suppression pure et simple de ces nœuds n'est donc pas suffisante et il faut relancer l'algorithme depuis $\mathcal{U}$ en tentant de ne pas refaire cette approximation.

Au contraire, avec FAR, le monde de chaque sommet est une approximation des nœuds atteignables qui peut contenir ou non une partie mauvaise. Chaque sommet est donc un monde en soi (voir Figure 4.8) et les conséquences liés au fait de mal approximer sont d'une importance moindre par rapport à l'atteignabilité arrière simple.

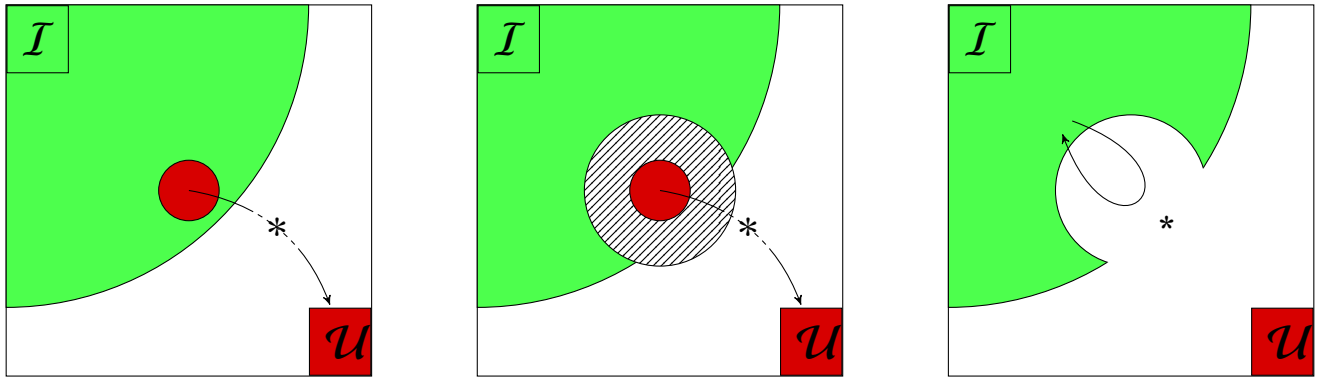
Deux cas peuvent survenir :

— Sur-approximer directement les états mauvais du monde au risque de se retrouver dans le cas de l'atteignabilité arrière avec des sommets du graphe inutilisables et l'obligation de revenir au point de départ
— Approximer lorsque la Règle 2 est choisie donc supprimer du monde une sur-approximation des états atteignant des états mauvais

Pour ces raisons, les deux choix d'implémentation suivant ont été pris :

— On le remarque dans la Règle 2, s'il existe une arête étiquetée par τ entre v et u avec $\mathcal{B}_u \neq \bot$ et qu'il s'avère que v par τ n'atteint pas réellement $\mathcal{B}_u$ alors on choisit une formule ψ ne contenant pas $\mathcal{B}_u$ qui devient le monde d'un nouveau sommet vers lequel on envoie v par τ. En pratique, deux version de FAR ont été implémentées :
  — La première, FAR, où cette formule est simplement $\mathcal{W}_u \wedge \neg\mathcal{B}_u$
  — La seconde, FAR-α, où on crée une formule ψ, généralisation de $\neg\mathcal{B}_v$ obtenue soit en choisissant

le plus petit sous-ensemble de littéraux qui vérifie les conditions liées à la formule ψ de la Règle 2 avant de créer une nouvelle formule $\mathcal{W}_U \wedge \psi$ soit en utilisant BRAB. BRAB est un algorithme de génération d'invariants implémenté à l'origine dans Cubicle par Alain Mebsout [51] et que je décris dans le Chapitre 5.



(a) Sommet contenant une partie mauvaise

(b) Sommet auquel on a supprimé la partie mauvaise

(c) Sommet sûr dont chaque transition reste dans le monde

FIGURE 4.8 – Le monde vu du point de vue de sommets de FAR

4.2.2 Exemple

On rappelle le code Cubicle de l'algorithme d'exclusion mutuelle dans la Figure 4.9 afin de se remémorer les transitions et différents états du système.

```
type state = Idle | Want | Crit

var Turn : proc
array State[proc] : state

init (z) { State[z] = Idle }

unsafe (z1 z2) { State[z1] = Crit && State[z2] = Crit }

transition req (i)
requires { State[i] = Idle }
{ State[i] := Want }

transition enter (i)
requires { State[i] = Want && Turn = i}
{ State[i] := Crit; }

transition exit (i)
requires { State[i] = Crit }
{
  Turn := . ;
  State[i] := Idle;
}
```

FIGURE 4.9 – Code Cubicle de l'algorithme d'exclusion mutuelle

L'exemple donné dans la Figure 4.10 permet de comprendre le fonctionnement de l'algorithme. À chaque nouveau sommet créé, une arête par transition activable est ajoutée entre ce sommet et $\mathcal{U}$ puis celui-ci est fermé soit en créant une arête vers un sommet sans partie mauvaise soit en découvrant qu'il est mauvais et en remontant à son sommet parent. Ainsi, la première étape consiste à partir du sommet initial et à se rendre compte qu'une seule transition est directement activalble, la transition req. On crée donc une arête entre ce sommet et le sommet v_u. Cette arête étant trop générale on crée un nouveau

sommet contenant la négation de la partie mauvaise de $v_{\mathcal{U}}$ (ici $\neg\mathcal{U}$) puis on part de ce nouveau sommet pour chaque transition cohérente avec son monde en cherchant à savoir si on atteint réellement un état mauvais (par la transition `enter` dans ce cas) ou pas (la transition `req`). On continue ainsi jusqu'à trouver un chemin entre le sommet initial et $v_{\mathcal{U}}$ ou un ensemble de sommets rattachés au sommet initial pour lesquels toutes les transitions ont été considérées et n'ayant aucune arête avec des sommets contenant des formmules mauvaises.

4.3 Benchmarks

L'implémentation très naïve et peu optimisée de FAR a été comparée à l'algorithme d'atteignabilité arrière déjà implémenté dans Cubicle sans inférence d'invariants. Comme on peut le voir dans le tableau suivant, les résultats sont tout à fait prometteurs.

Protocol	BR	FAR	FAR-α
`dekker`	0.04s	0.04s	0.03s
`mux_sem`	0.04s	0.05s	0.03s
`german-ish`	0.06s	0.1s	0.55s
`german-ish2`	0.13s	0.11s	0.65s
`german-ish3`	1.2s	8.3s	0.65s
`german-ish4`	3.5s	2.5s	0.75s
`german-ish5`	1.9s	8.2s	0.60s
`german`	18s	5.8s	4.25s
`szymanski_at`	TO	13s	2.60s
`szymanski_na`	TO	TO	16s

Bien que Cubicle avec l'inférence d'invariant permette de prouver ces protocoles bien plus rapidement, il est intéressant de remarquer que même dans sa version naïve FAR permet de prouver plus rapidement certains fichiers et devient extrêmement compétitif une fois qu'un mécanisme de généralisation est mis à l'œuvre.

4.4 Conclusion

J'ai présenté un nouvel algorithme de vérification de modèle dans le cadre des systèmes de transitions à tableaux.

Cet algorithme consiste à allier la puissance de l'analyse d'atteignabilité arrière avec une approche d'abstraction de l'analyse d'atteignabilité avant afin d'être à la fois dirigé par les propriétés à prouver et par les états atteignables. Une implémentation naïve m'a permis de prouver qu'il pouvait être tout à fait compétitif par rapport aux algorithmes actuels et qu'il mériterait qu'on y accorde de l'importance. Cet algorithme a été à l'origine d'une publication [24].

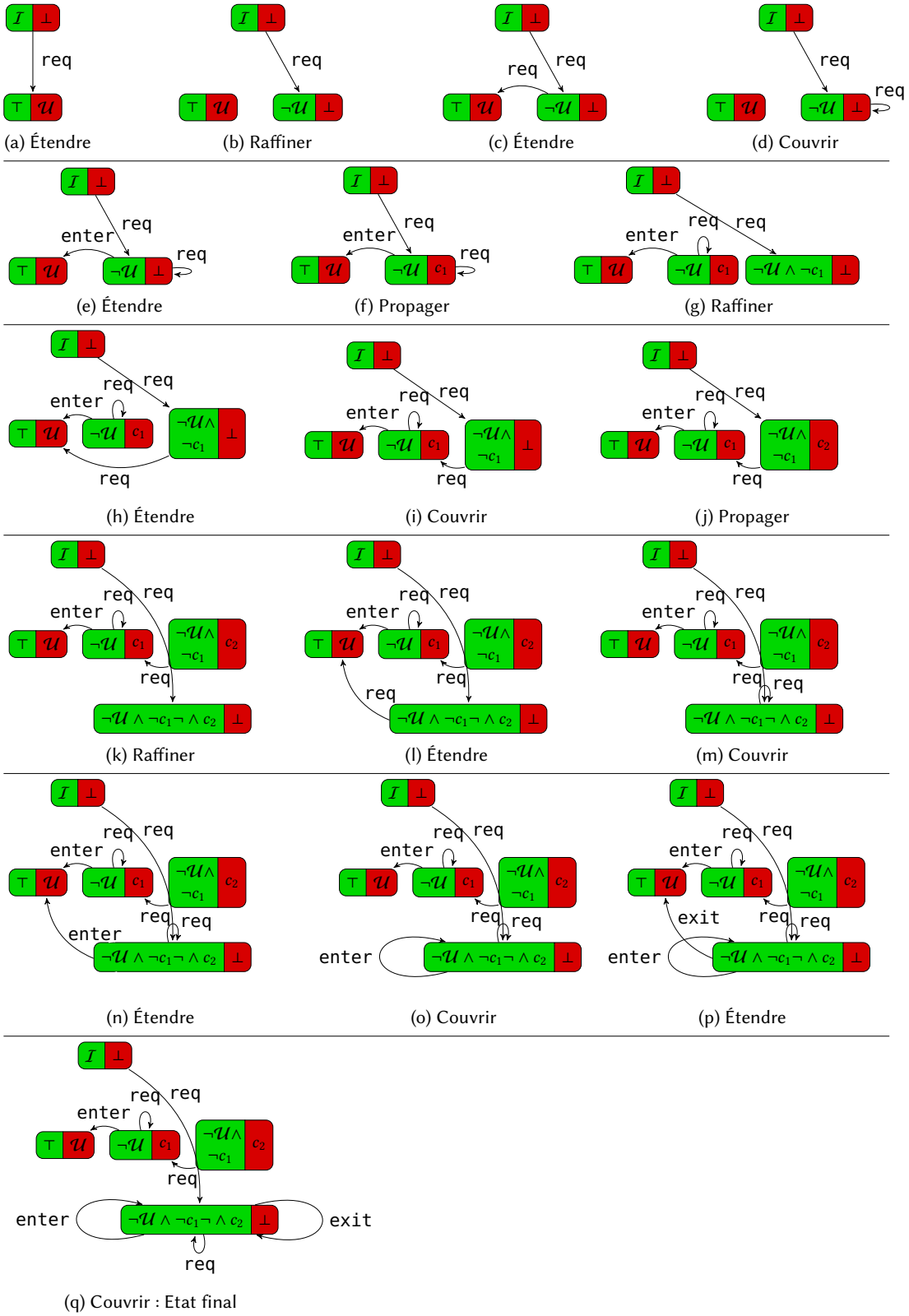


FIGURE 4.10 – Exécution complète sur l'algorithme d'exclusion mutuelle.
Le dernier sommet du graphe est un sommet sûr, il représente une formule étant un invariant du système conformément à la Définition 4.1.3 donc le protocole est sûr.

L'optimisation principale que j'aimerais lui ajouter est le fait de sauvegarder dans chaque sommet du graphe l'état actuel du solveur SMT afin de pouvoir avoir une approche bien plus incrémentale que l'approche actuelle consistant à relancer le solveur SMT de zéro à chaque appel. Dans le même ordre d'idée, en remarquant que chaque nouveau sommet créé représente le monde de son sommet parent auquel on a ajouté des contraintes, il serait intéressant d'améliorer les structures de données représentant les sommets pour qu'ils occupent moins d'espace en mémoire et pour éviter d'avoir à recopier dans chaque sommet les mondes des sommets précédents.

D'autres optimisations intéressantes sont surtout celles qui touchent à la généralisation des contre-exemples afin de supprimer rapidement des grands ensembles d'états non atteignables.

Chapitre 5

Améliorer l'efficacité de Cubicle

Comme présenté dans l'introduction, un autre moyen d'améliorer l'efficacité de Cubicle plutôt que d'essayer d'y ajouter un nouvel algorithme est d'améliorer l'algorithme existant qui a déjà démontré [26, 51] sa grande puissance et sa capacité à vérifier des protocoles de taille industrielle. Je vais donc, dans ce chapitre, m'intéresser à présenter succinctement cet algorithme afin d'expliquer les pistes d'améliorations qui ont été explorées lors de ma thèse et qui, bien qu'elles n'aient pas abouti à des changements majeurs, sont tout à fait prometteuses si tant est qu'on y investisse un peu plus de temps.

5.1 Contexte - BRAB

On l'a vu à la fin du Chapitre 3, Cubicle dans sa version basique calcule des pré-images et vérifie qu'elles n'intersectent pas les états initiaux du système et qu'elles ne sont pas redondantes avec des nœuds précédemment calculés. En théorie, cet algorithme fonctionne parfaitement. En pratique, en revanche, beaucoup de protocoles de cohérence de cache ou d'algorithmes d'exclusion mutuelle contiennent une telle quantité d'états possibles qu'il devient impossible de les prouver en un temps humainement raisonnable.

L'algorithme d'exclusion mutuelle étudié dans le Chapitre 3 contient un tableau, une variable globale, un type énuméré à trois constructeurs et trois transitions. Pour trois processus, douze états sont atteignables depuis l'état initial. Pour le résoudre, Cubicle a besoin, en moyenne, de 0,10 secondes et parcourt uniquement trois nœuds.

Il existe, dans la littérature, un protocole de cohérence de cache fourni par Steven German (travaillant pour IBM) qui a été implémenté en cubicle. Ce protocole est composé de six tableaux, de trois variables globales, de deux types énumérés contenant respectivement trois et sept constructeurs et de treize transitions. Depuis l'état initial, avec uniquement trois processus impliqués, on génère 8712 états différents. Cubicle doit parcourir 2384 nœuds en 6,85 secondes en moyenne avant de le prouver.

Le protocole Flash est un protocole développé dans les années 90 à Stanford qui se veut proche des protocoles de taille industrielle. Le passage à l'échelle est immédiat, 29 variables et 12 tableaux, neuf

types énumérés contenant entre 2 et 6 constructeurs et 71 transitions. Il y a plus de deux millions d'états atteignables depuis l'état initial avec seulement trois processus et Cubicle ne parvient pas à prouver sa sûreté en un temps raisonnable.

Afin de prouver ces gros protocoles, un nouvel algorithme a été implémenté dans Cubicle, BRAB [26]. BRAB correspond à *Backward Reachability with Approximation and Backtracking*. L'idée fondamentale derrière cet algorithme est de simplifier les pré-images trouvées en les approximant, c'est-à-dire en les expurgeant de leurs littéraux inutiles. Comme on l'avait vu dans le Chapitre 3, pour prouver l'algorithme d'exclusion mutuelle Cubicle avait exploré deux pré-images

$$\texttt{State}[\#_1] = \texttt{Want} \wedge \texttt{State}[\#_2] = \texttt{Crit} \wedge \texttt{Turn} = \#_1$$

et

$$\texttt{State}[\#_1] = \texttt{Idle} \wedge \texttt{State}[\#_2] = \texttt{Crit} \wedge \texttt{Turn} = \#_1$$

Alors que ces deux nœuds représentaient fondamentalement le même état mauvais,

$$\texttt{State}[\#_2] = \texttt{Crit} \wedge \texttt{Turn} = \#_1$$

C'est-à-dire tout état où la variable `Turn` ne contient pas l'identifiant du processus actuellement en état critique. Il s'avère donc que le fait que $\texttt{State}[\#_1]$ soit à `Want` ou `Idle` rajoute une information inutile à l'information essentielle. L'algorithme de BRAB est donc le même algorithme que pour l'analyse d'atteignabilité arrière si ce n'est que Pre a été transformé en Pre_approx afin de correspondre à cette nouvelle approche comme on le voit à la ligne 10 de l'Algorithme 6.

Le fonctionnement de Pre_approx est relativement simple : avant de calculer la pré-image d'un nœud on tente de l'approximer (en en supprimant des littéraux) et si une approximation est trouvée c'est celle-ci qui est utilisée pour calculer les pré-images à la prochaine étape. La variable $\mathcal{B}$ contient l'ensemble des approximations ayant conduit à une erreur et ne pouvant donc pas être choisies à nouveau comme on le verra immédiatement après. Chaque nœud a un champ supplémentaire, $Kind$, indiquant s'il n'a jamais été modifié ($Kind = Orig$) ou s'il a été approximé ou vient d'un nœud approximé ($Kind = Approx$). Chaque nœud résultant du calcul de la pré-image hérite du type de son parent. Cette fonction est donnée dans l'Algorithme 7. La fonction $Approx()$ renvoie, en accord avec ce que je viens de décrire, soit le nœud à l'identique si aucune approximation n'a été trouvée et $Kind$ sera égal à $Orig$ soit une approximation et $Kind$ sera égal à $Approx$. Le fonctionnement de cette fonction est décrit plus précisément dans la Section 5.2. Dans sa version la plus triviale, on se contente de renvoyer le premier sous-ensemble des littéraux composant la formule donnée en paramètre qui ne soit pas déjà marqué comme étant un mauvais candidat.

Le principal problème pouvant découler de cette approche est qu'il est possible d'approximer à l'excès une formule et de se retrouver avec un chemin jusqu'à l'état initial alors même que le protocole considéré est sûr. Dans le cas de l'algorithme d'exclusion mutuelle, on voit bien qu'en supprimant les littéraux mentionnant $\texttt{State}[\#_1]$ on a bien des formules représentant des états mauvais mais si on supprime, par

Algorithme 6: BRAB - Atteignabilité arrière avec approximation et retour en arrière

Input:
$\mathcal{S} = (Q, \mathcal{I}, \mathcal{T}, \mathcal{U})$
$\mathcal{B}$: nœuds approximés ayant entraînés des faux-positifs
Variables:
$\mathcal{V}$: nœuds visités
Q : file d'attente

```
1  function BWDA(S, B) : begin
2      V := ∅;
3      push(Q, U);
4      while not_empty(Q) do
5          φ := pop(Q);
6          if (I ∧ φ sat) then
7              renvoyer unsafe from φ
8          else if (φ ⊭ V) then
9              V := V ∪ {φ};
10             push(Q, Pre_Approx(φ, B));
11         end
12     end
13     renvoyer safe
14 end
```

Algorithme 7: Calcul de la pré-image approximée

Input:
φ : formule qu'on veut approximer avant d'en calculer la pré-image
$\mathcal{B}$: nœuds approximés ayant entraînés des faux-positifs

```
1  function Pre_Approx(φ, B) : begin
2      φ_A = Approx(φ);
3      P = Pre(φ_A);
4      foreach ψ ∈ P do
5          ψ.Kind := φ_A.Kind
6      end
7      renvoyer P
8  end
```

exemple, les littéraux mentionnant `Turn` on se retrouve avec des formules tout à fait correctes. L'algorithme d'atteignabilité arrière est donc encapsulé dans une boucle qui relance l'analyse d'atteignabilité arrière à chaque fois que celle-ci renvoie *unsafe* et que la trace menant de $\mathcal{I}$ à $\mathcal{U}$ passe par au moins un nœud ayant été expurgé (c'est à dire un nœud pour lequel $Kind$ est égal à $Approx$). Ce genre de cas est communément appelé *faux-positif*. Cette boucle de vérification est donnée dans l'Algorithme 8. La partie réellement important est $\mathcal{B}$ et sa mise à jour. À chaque faux-positif trouvé, on parcourt la trace ayant mené de $\mathcal{U}$ à $\mathcal{I}$ jusqu'à trouver le premier nœud ayant été approximé. Ce nœud est ajouté à $\mathcal{B}$ afin d'être écarté de façon définitive de toute nouvelle exécution de `BWDA`. Si cette mémoire des faux-positifs n'était pas mise en place il serait possible de boucler indéfiniment et de ne jamais terminer l'analyse.

Algorithme 8: Boucle de vérification des faux-positifs

```
   Input: S = (Q, I, T, U)
1  function BRAB(S) : begin
2  |  U.Kind := Orig ;
3  |  B := ∅;
4  |  while BWDA(S, B) = unsafe from φ do
5  |  |  if φ.Kind = Orig then renvoyer unsafe;
6  |  |  B := B ∪ {φ}
7  |  end
8  |  renvoyer safe
9  end
```

La question qu'il est donc tout à fait naturel de se poser maintenant est la suivante : mais comment fait-on pour savoir quels littéraux expurger des nœuds traités ? Une version complètement naïve est simplement de générer l'ensemble des sous-formules et de laisser la boucle BRAB les filtrer une à une. Cette méthode, bien que triviale, peut être améliorée suffisamment pour obtenir des résultats tout à fait satisfaisant, ce que je vais montrer de ce pas.

5.2 Améliorer l'approximation grâce à un oracle trivial

La méthode triviale d'approximation peut clairement introduire une explosion combinatoire dont on se passerait volontiers étant donné qu'on cherche à éviter celle dû aux nombres d'états. La méthode actuellement implémentée dans Cubicle l'a été pour éliminer rapidement une grande quantité de nœuds. Afin de procéder à cette élimination préalable, on cherche à vérifier que les approximations proposées ne représentent pas des états trivialement atteignables depuis l'état initial. Pour ce faire on effectue une exploration en avant depuis l'état initial sur un nombre de processus fini et on vérifie pour chaque approximation qu'elle n'est pas contenue dans un des nœuds que l'on vient de calculer.

Pour illustrer cette méthode on reprend l'algorithme d'exclusion mutuelle. La formule initiale de cet algorithme impose que `State` ne contient que la valeur `Idle`. Les trois transitions possibles sont, on le rappelle [1] :

$$
\begin{aligned}
t_{req} : \quad & \exists p. \quad \texttt{State}[p] = \texttt{Idle} \wedge \texttt{State'}[p] = \texttt{Want} \\
t_{enter} : \quad & \exists p. \quad \texttt{State}[p] = \texttt{Want} \wedge \texttt{Turn} = p \\
& \qquad\qquad \texttt{State'}[p] = \texttt{Crit} \\
t_{exit} : \quad & \exists p. \quad \texttt{State}[p] = \texttt{Crit} \wedge \texttt{State'}[p] = \texttt{Idle} \wedge \texttt{Turn'} = .
\end{aligned}
$$

1. On écrit sous forme logique les transitions avec la sémantique de Cubicle. Ainsi quand une valeur n'est pas mentionnée dans l'état futur elle garde la même valeur. Cela permet de s'affranchir des instructions conditionnelles. Je rappelle à toute fin utile que la valeur spéciale . permet d'exprimer une valeur aléatoire.

L'exploration avant pour deux processus donne donc le graphe d'états $\mathcal{F}$ appelé aussi "*oracle*" donné dans la Figure 5.1. Il est important de remarquer que dans ce graphe chaque état est total dans le sens où il mentionne toutes les variables et l'intégralité de chaque tableau du système. Lorsqu'une variable ou case de tableau n'a pas de valeur connue, on lui attribue la valeur unique `.`. L'intérêt de cette approche sera clairement compréhensible dans la partie suivante.

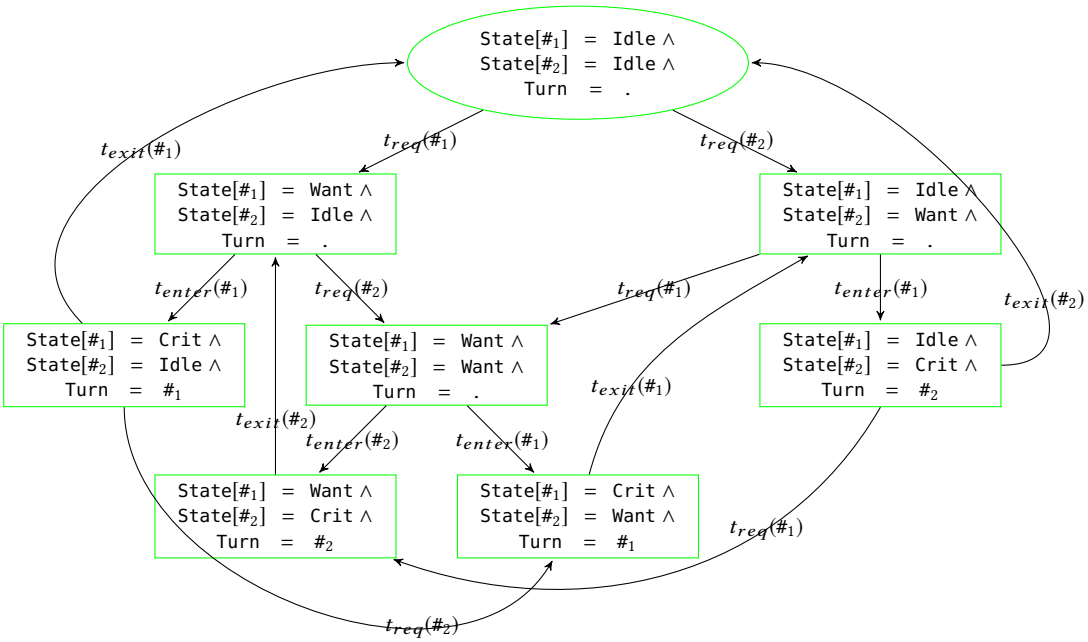


FIGURE 5.1 – Clôture de l'exploration avant avec deux processus pour l'algorithme d'exclusion mutuelle

Le nombre d'états possible est fini étant donné que le nombre de processus est limité et nous permet alors de vérifier que les approximations qu'on voudrait utiliser ne correspondent pas à des formules calculées. Les formules approximés attendant d'être filtrées sont appelées "candidats". Pour chaque candidat c, on vérifie que pour tout état $e \in \mathcal{F}$, $c \not\subseteq e$. Si tel n'est pas le cas, c remplace le nœud dont il est l'approximation sinon on garde le nœud tel quel. Ainsi, lors de l'utilisation de BRAB sur l'algorithme d'exclusion mutuelle on se retrouve avec le graphe d'exploration finale donné dans la Figure 5.2 [2]

Il est intéressant de remarquer que si les nœuds générés lors de l'exploration en avant n'avaient pas été totaux, le deuxième candidat

$$\texttt{State}[\#_1] = \texttt{Want} \wedge \texttt{Turn} = \#_1$$

n'aurait pas été éliminé alors même qu'il représente un état tout à fait sûr. De plus, on remarque aussi qu'aucun des candidats ne contient un seul littéral car il est évident que ce candidat serait éliminé immédiatement.

Certaines optimisations ont été rajoutées à l'exploration en avant telle que la suppression de nœuds redondants à renommage près, des structures de données intelligentes pour stocker le plus grand nombre d'états possibles etc.

2. On a supprimé, pour simplifier la lecture, les arêtes superficielles, c'est-à-dire celles allant d'un nœud à un nœud déjà obtenu autrement.

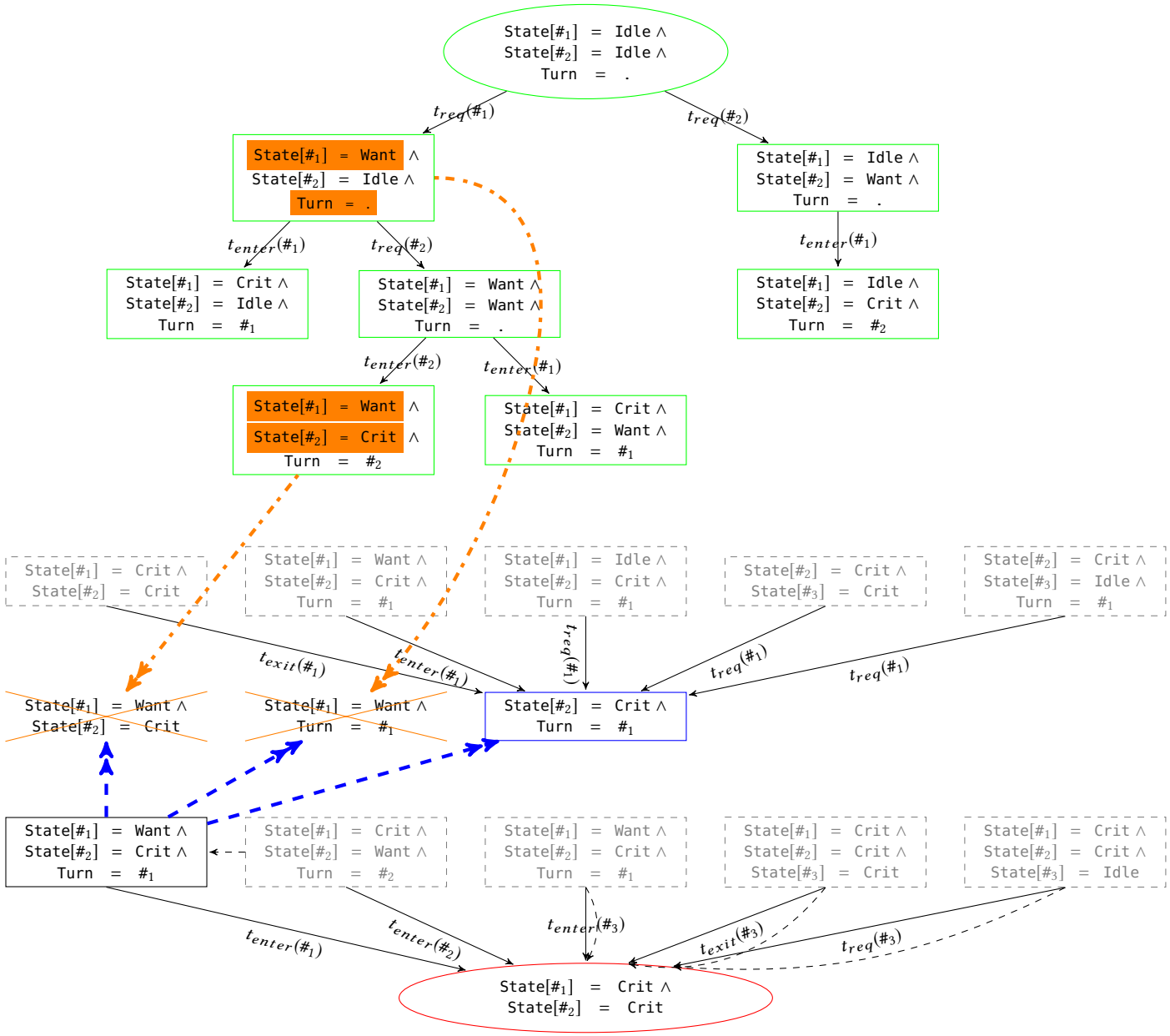


FIGURE 5.2 – Exécution complète de BRAB sur l'algorithme d'exclusion mutuelle

Cette méthode novatrice a permis de prouver des protocoles d'une grande complexité et notamment le protocole FLASH qu'on a vu précédemment en 1m30s.

Néanmoins, un problème émerge. Afin de prouver ce protocole il a fallu limiter la profondeur du graphe de parcours en avant car, comme je l'ai déjà dit auparavant, pour seulement trois processus ce graphe serait constitué de plus de deux millions de nœuds. Au delà du fait que la génération de plus de deux millions de nœuds prend un temps non négligeable, le temps passé à vérifier pour chaque candidat invariant qu'il ne correspond pas à un état de l'oracle serait démesurément long. C'est pourquoi il a été décidé, pour chaque nœud, de ne pas continuer l'exploration en avant si sa profondeur dépassait une certaine valeur choisie à l'exécution. Dans le cadre de FLASH la profondeur est par exemple limitée à 14 et après avoir généré plus de 400 000 nœuds Cubicle réussit à prouver ce protocole.

La limitation de la profondeur du graphe, même si elle permet de prouver certains protocoles très complexes, crée un nouveau problème lié à la pauvreté informationnelle de l'oracle pour des protocoles bien plus importants. Bien que les expériences suivantes n'aient pas permis de prouver de plus gros protocoles n'ayant jamais été prouvés dans le cas général, elles ont permis d'accélérer ,pour les protocoles déjà prouvés, le temps de preuve. Dans les sections suivantes je vais donc m'intéresser à décrire les expériences que j'ai effectuées sur l'oracle et les résultats que j'ai obtenus.

5.3 Frange et k-moyennes

5.3.1 Problématique

Comme décrit précédemment, le fonctionnement de BRAB implique la possibilité de recommencer à 0 (en retenant les mauvais candidats ayant provoqués les faux positifs) pour chaque faux positif. Cette situation a généralement lieu lorsqu'un candidat invariant a été validé par l'oracle et qu'il s'avère que ce candidat considéré comme représentant des états mauvais représente aussi des états atteignables depuis l'état initial. En réduisant le nombre d'états explorés par l'oracle on réduit à la fois le temps nécessaire pour les générer et le temps nécessaire à chaque requête de l'algorithme pour valider ou invalider un candidat invariant. On serait donc tenté de réduire l'oracle le plus possible mais, inéluctablement, la situation présentée dans la Table 5.1 apparaît, c'est-à-dire l'explosion du nombre de redémarrages et des temps de preuves bien plus importants.

Protocole	Forward		BRAB		
	Prof. max.	Card. de l'oracle	Redémarrages	nœuds visités	Temps
`german-ish4`	**12**	256	**0**	25	**0,02s**
`german-ish4`	**9**	211	**3**	205	**0,56s**
`german`	**14**	668	**0**	46	**0,07s**
`german`	**12**	550	**2**	307	**1,91s**
`german_pfs`	**17**	1497	**0**	52	**0,08s**
`german_pfs`	**14**	1151	**3**	707	**21,10s**
`szymanski_at`	**12**	**41**	**0**	32	**0,03s**
`szymanski_at`	**10**	37	**3**	207	**0,61s**
`szymanski_na`	**15**	72	**0**	39	**0,05s**
`szymanski_na`	**12**	64	**5**	475	**10,36s**

TABLE 5.1 – Illustration des conséquences liées à des profondeurs de recherche limitées

Le problème est donc de taille car le nombre de nœuds générés pour le protocole `german` par exemple, en faisant passer la limite de profondeur de 14 à 12 passe de 668 à 550 pour un temps de preuve passant de 0,07s à 1,91s [3]. On a donc réduit de 18% le nombre de nœuds de l'oracle pour une augmentation de temps

3. Bien que ces différences de temps puissent sembler dérisoires, de telles différences de performances sur des protocoles

de preuve de 2728%. La solution évidente serait bien sûr de fixer une profondeur maximale ne provoquant aucun redémarrage mais cette solution s'avère impossible pour des protocoles trop complexes.

5.3.2 La méthode des k-moyennes

La première idée explorée a donc été de refaire BRAB...mais en avant. C'est-à-dire d'approximer à la fois les pré-images générées lors de l'atteignabilité arrière mais et les post-images générées lors du parcours en avant. Le problème qui émerge immédiatement est : mais avec quel oracle ? Et c'est en effet un problème. À cette époque, n'en déplaise aux contempteurs des pauses cafés, la mienne se moquait régulièrement du fait que les logiciels de reconnaissance d'image actuels avaient osé confondre une photo de chat avec du guacamole. Attiré depuis longtemps par les évolutions liées au domaine de l'intelligence artificielle je me décidai donc à attaquer la problématique de l'abstraction des états de l'oracle par le biais du *partitionnement de données*.

Le **partitionnement de données** (aussi appelé *data clustering*, plus simplement *clustering* ou parfois *regroupement*) consiste à faire émerger d'un ensemble de données un ensemble de paquets (ou grappes, *clusters*, d'où le nom anglais) dans lesquels tous les éléments partagent un ensemble de caractéristiques communes.

À Retenir

Le partitionnement de données consiste à organiser des objets en ensembles dont les membres partagent des similitudes.

Un bon partitionnement doit garantir les deux propriétés suivantes :

— Les différences au sein de chaque paquet sont minimes
— Les différences entre chaque paquet sont maximales

Plusieurs méthodes de partitionnement sont possibles et la méthode choisie dépend généralement du type de paquet que l'on veut créer.

Dans le cadre de Cubicle on n'a aucune autre information sur les états que la formule qui les représente et on veut créer des paquets devant faire émerger un représentant chacun qui corresponde à la sous-formule contenant chacun des états du paquet.

Le fait de ne bénéficier d'aucune information nous permettant de guider l'algorithme nous impose de chercher dans la catégorie des algorithmes d'apprentissage non supervisé. Succinctement, l'apprentissage supervisé consiste à entraîner son algorithme sur des exemples annotés afin de le faire travailler ensuite sur des problèmes nouveaux mais similaires quand l'apprentissage non supervisé consiste à faire travailler l'algorithme directement sur les problèmes sans annotations préalables et donc sans possibilité d'entraîner l'algorithme au préalable.

Le fait de vouloir faire émerger un représentant qui soit en quelque sorte une moyenne des états de chaque paquet lié au fait de chercher un algorithme d'apprentissage non supervisé m'a donc orienté

industriels peuvent faire passer le temps de leur vérification de plusieurs minutes à plusieurs heures ou même jours.

vers la méthode des k-moyennes (connue dans la littérature sous sa version anglaise, k-means).

La méthodes des k-moyennes est une méthode qui étant donné un entier k et un ensemble de points (ou d'états, dans notre cas), essaye de regrouper ces points en k groupes de façon à minimiser la distance de chaque point d'un paquet à la moyenne de ce paquet. Cette méthode a été initialement décrite par MacQueen [47] et a fait l'objet de nombreuses améliorations au cours des années notamment en ce qui concerne la détermination de k [39, 56].

L'idée derrière cette méthode est relativement simple. Initialement, on choisit k points puis on associe chaque point au paquet dont il est le plus proche et on met à jour la moyenne de chaque paquet. Tant qu'une modification est faite (un point est changé de paquet ou une moyenne change), les deux dernières opérations sont répétées. Cette méthode ne garantit ni optimalité ni temps de calcul polynomial mais s'avère assez efficace en pratique.

5.3.3 Choix d'implémentation

5.3.3.a Initialisation

On s'en doute, l'initialisation est bien évidemment l'étape cruciale de cette méthode. Prendre les k points de façon aléatoire peut entraîner une différence entre les paquets bien trop faibles et un partitionnement très peu intéressant. Une autre façon de faire décrite dans [9] consiste à choisir le premier représentant de façon aléatoire puis à choisir les représentants suivant en fonction de leur distance aux points déjà choisis. Pour ce faire chaque point a une probabilité d'être choisi proportionnelle au carré de la distance entre lui et le représentant le plus proche. Ainsi les points les plus éloignés ont plus de chance d'être choisis. Mon implémentation s'est basée sur deux méthodes. Une première, consistant à choisir les points de façon complètement aléatoire, une seconde, relativement différente, consistant à déterminer une distance minimale entre représentants et à ne sélectionner aléatoirement que des points éloignés d'au moins cette distance des autres représentants.

5.3.3.b Fonction de distance

Reste alors à choisir la fonction de distance entre les points. Dans notre cas nous manipulons des états composés de littéraux qui ont l'inconvénient d'être des sous-ensembles de l'ensemble des variables et tableaux du système. L'avantage de l'exploration en avant est qu'elle se fait sur un nombre limité de processus impliquant donc une taille fixe pour le système. Cette limitation nous permet donc de représenter les états du parcours en avant sous forme de tableaux de taille finie en faisant correspondre à chaque indice du tableau une variable ou une case de tableau de Cubicle du système. Un exemple valant mieux qu'un long discours, voici la représentation de plusieurs état de l'algorithme d'exclusion mutuelle pour une exploration en avant limitée à deux processus.

Tout état de l'algorithme contient une variable `Turn` pouvant prendre les valeurs $\#_1$ ou $\#_2$ ainsi que deux cases du tableau `State` pour les processus $\#_1$ et $\#_2$ pouvant prendre les valeurs `Idle`, `Want` et `Crit`. On fait donc correspondre ces états à des tableaux de taille trois tout en faisant correspondre chaque valeur possible à un entier unique. On remarque dans la Figure 5.3 la possibilité d'une valeur

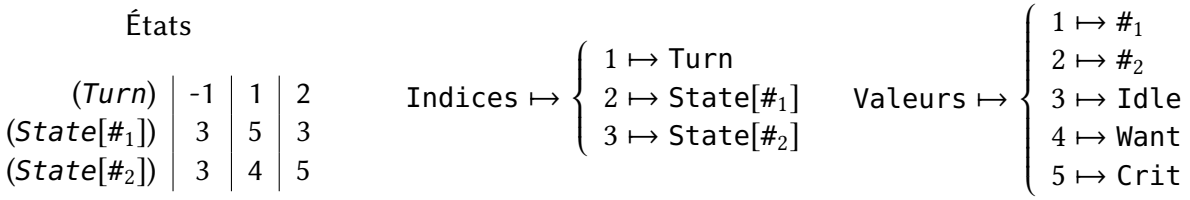


FIGURE 5.3 – États sous forme de tableaux et dictionnaires de correspondance

-1 qui n'apparaît pas dans le dictionnaire des valeurs. Cette valeur est extrêmement importante car elle signifie que la variable ou la case de tableau correspondante n'a pas de valeur définie dans l'état présent.

La fonction de distance que j'ai choisie a donc été la distance de Hamming qui correspond au nombre de cases où deux tableaux diffèrent. La seule particularité de notre fonction est qu'une case ayant pour valeur -1 est considérée comme ayant toutes les valeurs possibles et donc ne fait pas augmenter la distance entre deux tableaux. Ainsi, la distance entre les deux tableaux suivant :

-1	2
3	3
3	5

est égale à 1.

5.4 Implémentation de l'algorithme dans Cubicle

J'ai implémenté cet algorithme dans Cubicle plus dans la perspective de savoir s'il pourrait être intéressant de se pencher sur le fait de faire communiquer les domaines liés à l'intelligence artificielle et la vérification de modèle.

L'Algorithme 9 décrit le fonctionnement de la fonction principale de l'algorithme des k-moyennes. Cet algorithme, comme je l'ai décrit, se contente d'initialiser un ensemble $\mathcal{S}_C$ selon une fonction de sélection des représentants décrite dans l'Algorithme 10 puis, pour chaque tableau a_i appartenant à l'ensemble des tableaux donnés en paramètre, ce tableau est ajouté au paquet pour lequel la distance est minimale entre son représentant et a_i. Après que chaque tableau ait été attribué à un paquet, les nouveaux représentants de chaque paquets sont déterminés selon la fonction `moyenne` décrite plus tard et si un représentant a changé par rapport au représentant précédent, la phase d'attribution et de calcul des représentants est relancée. À l'issue de cette fonction, l'ensemble des représentants de chaque paquet est renvoyé.

Comme on le voit dans l'Algorithme 10, la première version s'arrête une fois que k représentants ont été trouvés alors que la seconde continue à créer de nouveaux paquets tant qu'il existe des tableaux ayant une distance supérieure à la valeur d donnée en paramètre de la fonction. Les deux versions ont leurs avantages et inconvénients bien que la seconde permette d'assurer une plus grande diversité des représentants au risque d'avoir beaucoup trop de paquets ou pas assez en fonction des tableaux donnés en paramètre. Il serait intéressant pour des travaux futurs de se pencher sur cette initialisation afin de

Algorithme 9: Algorithme des k-moyennes

Input: $\mathcal{S}_\mathcal{A} = \{a_1, a_2, ..., a_n\}$ un ensemble de tableaux à une dimension de même taille
Variables: *changement* $\mathcal{S}_C$

```
1  fonction k-moyennes(S_A) : begin
2      S_C := sélectionne_cluster(S_A);
3      changement := false;
4      repeat
5          foreach a_i ∈ S_A do
6              sélectionner c_j ∈ S_C tel que ∀c_k ≠ c_i ∈ S_C distance(a_i, c_k) ≤ distance(a_i, c_j);
7              S_C := replace(a_i, c_j);        /* Retire a_i du paquet dans lequel il était
                   présent et l'ajoute au paquet représenté par c_j */
8          end
9          foreach p_i ∈ S_C do
10             c'_i := c_i;          /* On garde en mémoire le représentant actuel de p_i */
11             c_i := moyenne(p_i);
12             if c_i ≠ c'_i then changement := true;
13         end
14     until changement = false;
15     renvoyer tous les c_i ∈ S_C
16 end
```

Algorithme 10: Sélection des premiers représentants de paquets

Input: $\mathcal{S}_\mathcal{A} = \{a_1, a_2, ..., a_n\}, k$
Variables: $\mathcal{S}_C$

```
1  fonction sélectionne_cluster_aléatoire(S_A, k) : begin
2      S_C := ∅;
3      while |S_C| ≤ k ∧ S_A ≠ ∅ do
4          a_i := choisir(S_A);
5          if ∀c_i ∈ S_C. distance(c_i, a_i) > 0 then S_C := S_C ∪ a_i ↦ ∅;
6          S_A := S_A \{a_i};
7      end
8      renvoyer tous les c_i ∈ S_C
9  end
10 fonction sélectionne_cluster_déterministe(S_A, d) : begin
11     S_C := ∅;
12     while S_A ≠ ∅ do
13         a_i := choisir(S_A);
14         if ∀c_i ∈ S_C. distance(c_i, a_i) > d then S_C := S_C ∪ a_i ↦ ∅;
15         S_A := S_A \{a_i};
16     end
17     renvoyer tous les c_i ∈ S_C
18 end
```

la rendre plus pertinente.

La fonction `moyenne` se contente de comparer chaque case des tableaux d'un paquet. Si toutes les cases correspondant à un même indice ont la même valeur alors le représentant reçoit cette valeur à cet indice sinon il reçoit la valeur -1.

La seule optimisation ayant été ajoutée à cet algorithme a été de coupler d'ajouter la possibilité de partitionner les états en cours d'exploration en avant dans l'idée d'accélérer la découverte d'états intéressants pour l'oracle. Il est en effet important de remarquer que la plupart des protocoles et algorithmes étudiés ont une phase d'initialisation qui est généralement la même pour tous les processus, les états critiques arrivant après un certain nombre de transitions. L'idée derrière cette optimisation était donc de générer des états suffisamment larges pour qu'ils représentent des états qui auraient été obtenus en beaucoup trop de transitions sinon.

5.5 Exemple

L'exemple suivant sert à expliciter le fonctionnement de l'algorithme sur des tableaux d'entiers.

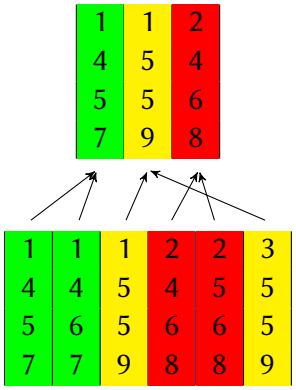
Supposons qu'on ait les tableaux suivants :

1	1	1	2	2	3
4	4	5	4	5	5
5	6	5	6	6	5
7	7	9	8	8	9

Selon la deuxième méthode, avec une distance minimale de 2 entre deux représentants les trois représentants choisis seront :

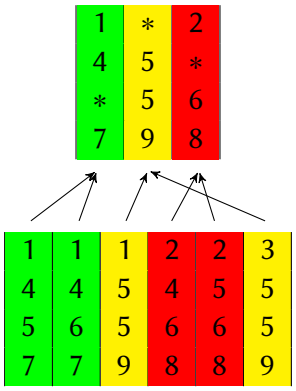
1	1	2
4	5	4
5	5	6
7	9	8

La première phase d'attribution se conclut dans l'état suivant :



Suite à cette phase d'attribution, les nouveaux représentants sont calculés en effectuant la moyenne, pour chaque paquet, des tableaux les composants [4] :

Étant donné qu'au moins un représentant a été changé, la phase d'attribution et de calcul des moyennes est relancée :



Les représentants n'ayant pas changé, l'algorithme a terminé et les trois représentants

sont renvoyés. On passe donc d'un ensemble de six tableaux contenant potentiellement du bruit à un ensemble de trois tableaux moins stricts.

4. À la place de la valeur -1 je préfère mettre * pour une meilleure lisibilité

5.5.1 Résultats

L'implémentation de cet algorithme très simple dans Cubicle a permis de prouver les protocoles dans des temps bien plus compétitifs mais sans atteindre les temps obtenus en ayant une profondeur de recherche plus grande. On observe notamment des cas où le partitionnement ne permet pas de faire émerger suffisamment de représentants tout en permettant de ne plus provoquer de redémarrages dus aux faux positifs. La colonne "Paquets" indique à quelle profondeur les partitionnements intermédiaires ont été faits.

Protocole	Forward			BRAB		
	k	Paquets	Card	R	V	Time
german-ish4	9		211	3	211	0.56s
german-ish4	9	8/9	**153**	**0**	**155**	**0.09s**
german	12		550	2	307	1.91s
german	12	12	**502**	**0**	**216**	**0.81s**
german_pfs	14		**1151**	3	707	21.10s
german_pfs	14	6/14	2101	**0**	**680**	**5.61s**
szymanski_at	10		37	3	207	0.61s
szymanski_at	10	10	**36**	**0**	**74**	**0.20s**
szymanski_na	12		64	5	**475**	10.36s
szymanski_na	12	4/12	**63**	**2**	664	**5.65s**

TABLE 5.2 – Illustration des résultats obtenus avec l'algorithme des k-moyennes

5.6 Regrouper pour mieux régner

Le défaut principal de la méthode précédente est malheureusement assez évident : elle ne permet pas de créer de nouveaux états. Cette méthode, bien qu'excellente pour réduire la taille de l'ensemble des nœuds générés par l'exploration en avant ne peut pas permettre de découvrir des états si les transitions permettant d'y arriver n'ont jamais été prises. Un protocole comme le protocole hiérarchique, par exemple, est composé de plus de 50 transitions. A moins de limiter la profondeur à 50 en espérant prendre chaque transition au moins une fois, il est tout à fait possible que certaines ne seront jamais prises.

Cette section sera une espèce de fourre-tout de ce qu'il est possible de faire pour transformer un oracle assez naïf en un algorithme bien plus fin. L'avantage à travailler sur l'oracle est qu'il a le droit de se tromper la correction de l'atteignabilité arrière avec approximations ne dépendant pas des erreurs de l'oracle.

5.6.1 Copie

L'idée est toute simple mais permet d'obtenir des résultats tout à fait encourageants. Initialement, à chaque transition on vérifie que seules des variables locales ont été modifiées. Cela se traduit par le fait que seule les cases de tableaux correspondants au paramètre d'une transition paramétrée par un seul processus ont été modifiées. Étant donné qu'aucune variable globale n'a été modifiée il est tout à fait naturel de penser qu'il existe un état du système dans lequel tous les processus ont pris cette transition. Plutôt que d'y arriver transition par transition au risque de dépasser la profondeur limite donnée en paramètre, on décide de modifier l'ensemble des tableaux ayant été modifiés par cette transition afin qu'ils contiennent tous les nouvelles valeurs. Dans un algorithme d'exclusion mutuelle, par exemple, au premier processus passant en attente d'entrée en section critique on crée un nouvel état où tous les processus sont en attente d'entrée en section critique.

Encore une fois, la méthode naïve montre assez rapidement ses limites comme on peut le voir dans le tableau de la Table 5.3.

Protocole	Forward			BRAB			
	Copie	Prof. max.	Card. de l'oracle	↺	nœuds visités	Approx	Temps
`german-ish4`	Non	**7**	135	**6**	358	9	**1,03s**
`german-ish4`	Oui	**7**	289 (211 copies)	**0**	51	7	**0,07s**
`german`	Non	**9**	335	**7**	1019	30	**6,83s**
`german`	Oui	**9**	1227 (1145)	**1**	491	18	**1,31s**
`szymanski_na`	Non	**9**	55	**7**	615	25	**14,89s**
`szymanski_na`	Oui	**9**	461 (347)	**0**	474	11	**2,37s**
`sense_barrier`	Non	**6**	21	**8**	369	5	**0,44s**
`sense_barrier`	Oui	**6**	150 (128)	**0**	203	0	**0,56s**
`german-ish4`	Non	**9**	211	**4**	204	9	**0,62s**
`german-ish4`	Oui	**9**	761 (710)	**0**	639	1	**1,52s**

TABLE 5.3 – Illustration des résultats obtenus avec une copie naïve des modifications locales

À l'issue de ces expérimentations, deux observations émergent immédiatement. Premièrement, alors qu'on cherchait à réduire le nombre de nœuds générés par l'oracle afin d'avoir une vérification des candidats plus rapides, dans la plupart des cas le nombre de nœuds générés grâce à la copie est bien trop élevé par rapport au nombre généré sans copie. On se satisferait de cet inconvénient si, comme on le voit sur les deux derniers exemples, quand bien même on n'a plus aucun redémarrage (colonne ↺) grâce aux copies, le temps nécessaire à la vérification de ces deux protocoles dépasse le temps nécessaire à leur vérification sans copie et avec redémarrage. J'ai alors remarqué que les temps supérieurs étaient corrélés au fait que très peu d'approximations aient été trouvées. Il m'est donc apparu que lors de la copie il était malheureusement possible de créer des états mauvais et donc de rejeter des approximations tout à fait correctes. Une première expérience a consisté à faire tourner l'algorithme d'atteignabilité arrière sans approximation jusqu'à une profondeur maximale afin de vérifier que les nœuds copiés ne représentaient pas des états mauvais mais, au delà du fait que cela allait à l'encontre du principe de l'oracle, il était

assez vain de penser qu'une copie ayant lieu après une ou deux transitions depuis l'état initial puisse être invalidée par l'algorithme d'atteignabilité arrière en peu d'étapes.

J'ai donc décidé de me concentrer sur une problématique différente et s'inscrivant dans un questionnement ouvert que j'aimerais tant voir étudié : comment transformer la lecture humaine que nous faisons des protocoles en Cubicle en une lecture automatique faite par le programme ?

Je m'explique. À la lecture d'un protocole assez complexe, en tant qu'être humain, nous aurons tendance à compartimenter les différentes transitions. Un premier groupe, par exemple, correspondra aux transitions internes à un processus pour passer de l'état inerte à l'état en attente, un autre groupe correspondra aux actions du système pour donner la main à tel ou tel processus et ainsi de suite. Plutôt que de tenter de tout prouver d'un seul coup il pourrait être intéressant de voir ce qu'il est possible de faire en vérification modulaire donc de vérifier chaque groupe séparément puis de vérifier que ces groupes, combinés ensemble, continuent d'être corrects. Le meilleur moyen pour vérifier qu'une idée fonctionne automatiquement est de la faire manuellement puis de l'automatiser.

La première étape dans cette perspective fut donc de tenter de regrouper des ensembles de transitions dans des expressions régulières afin de communiquer à Cubicle le fait que tout état résultant d'une suite de transition vérifiant une des expressions régulières lui ayant été fournies peut être généralisé à l'ensemble des processus. Ainsi, plutôt que de copier naïvement tous les états pour lesquels les variables globales n'ont pas été modifiées on ne copie que certains états de notre choix en essayant de voir si des comportements émergeront de ces expériences.

Si on prend des situations pathologique de la copie naïve, on obtient, en ajoutant la copie contrainte par expression régulière, les résultats suivant :

Protocole	Forward			BRAB			
	Copie	Prof. max.	Card. de l'oracle	↻	nœuds visités	Approx	Temps
german-ish4	Non	**9**	211	**4**	204	9	**0,62s**
german-ish4	Oui	**9**	299 (12 copies)	**0**	93	9	**0,17s**
german	Non	**12**	550	**2**	306	30	**2,16s**
german	Oui	**12**	714 (3)	**0**	90	28	**0,20s**

Les résultats obtenus, comme on le voit, sont bien meilleurs que la simple copie naïve. Le nombre de nœuds supplémentaires créées par l'oracle reste dans le même ordre de grandeur pour des temps de vérification bien plus courts. J'ai alors tenté d'aller encore plus loin en créant des super transitions et des transitions universelles.

Les super transitions sont des transitions qui regroupent dans leur garde et leur action l'action de plusieurs transitions (généralement, donc, les transitions qu'on a utilisées dans les expressions régulières) et qui peuvent être utilisées à la fois par la recherche en avant pour l'oracle et par l'analyse d'atteignabilité arrière.

Protocole	Forward			BRAB			
	Super Trans.	Prof. max.	Card. de l'oracle	↺	nœuds visités	Approx	Temps
german-ish4	Non	**9**	211	**4**	204	9	**0,62s**
german-ish4	Oui	**9**	275	**0**	24	9	**0,04s**
german	Non	**12**	550	**2**	306	30	**2,16s**
german	Oui	**12**	793	**0**	45	28	**0,08s**

Les transitions universelles, elles, sont similaires aux super transitions mais ne sont utilisées que par la recherche en avant car elles sont utilisées pour mettre directement à jour l'ensemble de chaque tableau. Elles sont donc équivalentes à l'utilisation des expressions régulières et offrent des résultats strictement équivalents et ne méritent donc pas plus d'explication.

Mon plus grand regret aura été, durant cette thèse, de ne pas aller plus loin dans ce domaine de la vérification modulaire étant donnés les résultats tout à fait prometteurs que mes expériences manuelles avaient permis de mettre en avant. Étant donné qu'il fallait choisir un sujet sur lequel se concentrer, le dévolu a été jeté sur la problématique que j'aborde dans le chapitre suivant mais il serait tout à fait intéressant de voir ce qu'il est possible de faire avec cette partie inachevée.

Chapitre 6

Exprimer de nouvelles propriétés

6.1 La problématique des formules universellement quantifiées

On a vu dans les trois précédents chapitres que les cubes utilisés par Cubicle permettaient de caractériser un grand nombre de propriétés de sûreté. Il arrive néanmoins que certaines propriétés nécessitent de mentionner l'état global du système. Il n'est pas rare, en effet, dans le domaine de l'algorithmique distribuée, de trouver des propriétés mentionnant l'état du système à la fin de son exécution. Prenons, par exemple, un algorithme de consensus. Cette classe d'algorithme demande à ce qu'un certain nombre de processus s'accordent sur une unique valeur. Les propriétés demandées à ce genre d'algorithme sont que tous les processus doivent décider une valeur (*terminaison*), décider une valeur ayant été proposée par soi ou un autre (*intégrité*) et décider la même valeur (*accord*).

Dans le cadre de Cubicle, ces propriétés sont exprimées sous leur forme négative :

— la *terminaison* n'a pas lieu s'il existe un processus qui ne décide pas tout en ayant terminé (aisément exprimable sous forme de cube)

— l'*accord* n'a pas lieu s'il existe deux processus n'ayant pas décidé la même valeur (aisément exprimable sous forme de cube)

— l'*intégrité* n'a pas lieu s'il existe un processus ayant décidé une valeur qu'aucun processus n'a proposée (non exprimable sous forme de cube)

Pour pouvoir exprimer l'intégrité des algorithmes de consensus il faut que Cubicle puisse gérer des formules du type $\exists \vec{i} \forall j.C(\vec{i}, j)$, $C(\vec{i}, j)$ étant une conjonction de littéraux paramétrés par un vecteur $\vec{i}$ de variables de processus distinctes et j une variable de processus distincte des variables de $\vec{i}$. Nous avons décidé d'appeler ces nouveaux types de formules *cubes universels*.

En Pratique

Comme on l'a vu dans la Figure 3.9, il est tout à fait possible d'ajouter des gardes universelles à Cubicle.

Le mot clé forall_other permettant déjà d'exprimer des propriétés universelles dans les gardes des transitions, d'un point de vue sémantique il ne faut rien rajouter à Cubicle pour traiter ces nouveaux types de propriété. La façon dont elles ont donc été ajoutées s'est donc faite en les compilant vers des transitions menant à un état mauvais.

Ainsi, on ajoute à Cubicle une nouvelle construction :

```
universal_unsafe (p1 p2 ... pn, pm) {C(p1, p2, ..., pn, pm)}
```

Qui en pratique est traduite comme une transition menant vers un état mauvais créé pour l'occasion :

```
var Error : bool

init (p1) { I(p1) && Error = False }

unsafe { Error = True }

transition error(p1 p2 .... pn)
requires {C'(p1, p2, ..., pn) && forall_other pm. C''(p1, p2, ..., pn, pm)}
{ Error := True}
```

Avec
— `I(p1)` la formule init originelle
— `C' ∧ C" ≡ C`

La question qu'il est donc naturel de se poser est la suivante : si la possibilité était déjà offerte dans Cubicle, sa résolution n'était-elle pas déjà implémentée ? Il me faut, pour répondre à cela, expliquer maintenant la façon dont Cubicle gérait jusqu'alors les gardes universelles en commençant pas présenter le fichier exemple qui me servira tout au long de ce chapitre.

6.1.1 Les Splitters

Les splitters ont été introduits initialement par Lamport [45] afin d'implémenter des algorithmes rapides d'exclusion mutuelle. Ils ont ensuite été réutilisés par Moir et Anderson [52] afin de résoudre le problème de renommage dans les mémoires partagées. Un schéma du fonctionnement de base d'un splitter est donné dans la Figure 6.1. Le splitter est un item fonctionnant de manière concurrente afin de distinguer un nombre arbitraire (ici n) de processus appelants. Chacun de ces processus appelle le splitter pour obtenir une décision pouvant être de s'arrêter, de descendre ou d'aller à droite. Ces décisions doivent obéir à quatre règles précises :

— Il n'y a que trois décisions possibles : `Stop`, `Right` et `Down`
— Au plus un processus reçoit la décision `Stop`
— Au plus $n-1$ processus reçoivent la décision `Down`
— Au plus $n-1$ processus reçoivent la décision `Right`

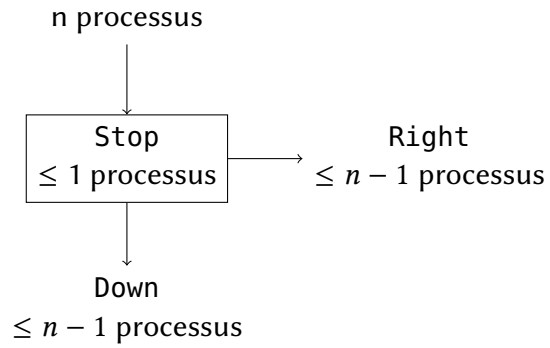


FIGURE 6.1 – Schéma d'un splitter

En Pratique

En mettant des splitters en grille et en attribuant à chaque splitter un identifiant unique on peut assurer qu'au bout d'un nombre d'étapes finies chaque processus ce sera arrêté sur un splitter et aura pour identifiant celui du splitter sur lequel il s'est arrêté et que celui-ci sera donc unique. Cette méthode permet donc de réduire l'espace des noms sans bloquer les processus.

L'algorithme pour le splitter est présenté dans la Figure 6.2 sous la forme d'un automate à sept états (de PC_0 à PC_3 et les trois valeurs de décision `Stop`, `Down` et `Right`) ainsi que deux variables internes au splitter X et Y. Chaque processus p est initialement à l'état PC_0. Le splitter démarre avec Y à $\bot$ et X indéterminée. Lorsqu'un processus demande une décision au splitter, il passe de l'état PC_0 à l'état PC_1 tout en assignant à X son identifiant. À l'étape suivante, si Y est à $\top$, la valeur `Right` est décidée par le splitter pour p sinon p est passé dans l'état PC_2. Au premier processus qui arrive en PC_3, Y se voit assigner $\top$ et pour chaque processus p dans l'état PC_3, le splitter décide la valeur `Down` si X est différent de p et `Stop` sinon.

La modélisation de cet algorithme se fait de façon immédiate en utilisant les systèmes de transitions à tableaux. On définit pour cela un type énuméré state contenant les compteurs de programme PC_0, PC_1, PC_2 et PC_3 ainsi que les valeurs de décision `Stop`, `Down` et `Right`. On définit ensuite tout naturellement les deux variables X et Y ainsi qu'un tableau PC associant à chaque processus p une valeur de type state. Initialement, comme on l'a vu dans la description de l'automate correspondant, la formule $\mathcal{I}nit$ est la suivante :

$$\mathcal{I}nit : \forall p.\mathsf{PC}[p] = \mathsf{PC}_0 \wedge \mathsf{Y} = \bot$$

Les six transitions sous leur forme logique sont décrites dans la Figure 6.3 par les six formules $\mathtt{spl}_{xxx}$. Chacune d'elle mentionne à la fois l'état précédant la transition et l'état la suivant. Toute variable (ou tableau) suffixée d'un prime (X'...) indique l'état de cette variable (ou tableau) après exécution de la transition. La transition $\mathtt{spl}_0$ doit donc être lue comme : s'il existe un processus p tel que $\mathsf{PC}[p] = \mathsf{PC}_0$ alors PC[p] reçoit PC_1 et X reçoit p.

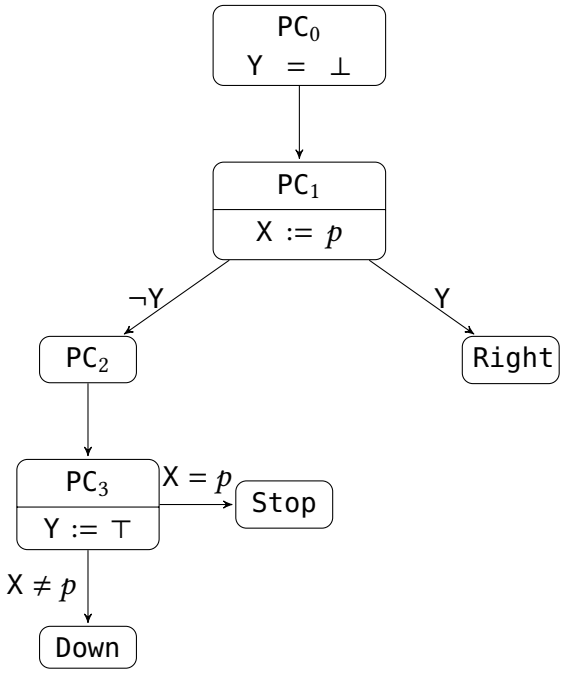


FIGURE 6.2 – Automate du splitter. Les mises-à-jour des états des processus et des variables apparaissent dans les nœuds, les conditions pour passer d'un nœud à un autre sont étiquetées aux arêtes

$$
\begin{array}{lll}
spl_0 : & \exists p. & \mathtt{PC}[p] = \mathtt{PC}_0 \wedge \\
 & & \mathtt{PC'}[p] = \mathtt{PC}_1 \ \wedge \ \mathtt{X'} = p \\
spl_{right} : & \exists p. & \mathtt{PC}[p] = \mathtt{PC}_1 \ \wedge \ \mathtt{Y} \wedge \\
 & & \mathtt{PC'}[p] = \mathtt{Right} \\
spl_1 : & \exists p. & \mathtt{PC}[p] = \mathtt{PC}_1 \ \wedge \ \neg\mathtt{Y} \wedge \\
 & & \mathtt{PC'}[p] = \mathtt{PC}_2 \\
spl_2 : & \exists p. & \mathtt{PC}[p] = \mathtt{PC}_2 \wedge \\
 & & \mathtt{PC'}[p] = \mathtt{PC}_3 \ \wedge \ \mathtt{Y'} = \top \\
spl_{stop} : & \exists p. & \mathtt{PC}[p] = \mathtt{PC}_3 \ \wedge \ \mathtt{X} = p \wedge \\
 & & \mathtt{PC'}[p] = \mathtt{Stop} \\
spl_{down} : & \exists p. & \mathtt{PC}[p] = \mathtt{PC}_3 \ \wedge \ \mathtt{X} \neq p \wedge \\
 & & \mathtt{PC'}[p] = \mathtt{Down}
\end{array}
$$

FIGURE 6.3 – Système de transition sous forme logique du Splitter

Le code Cubicle correspondant au splitter est donné dans la Figure 6.4.

D'après les propriétés énoncées précédemment, prouver la correction de cet algorithme revient à

```
type state = PC0 | PC1 | PC2 | PC3 | Stop | Down | Right

var X : proc
var Y : bool
array PC[proc] : pc

init(i) { Y = False && PC[i] = PC0 }

universal_unsafe (i) { PC[i] = Down }

transition spl0(i)
requires { PC[i] = PC0 }
{  X := i;
   PC[i] := PC1; }

transition splright(i)
requires { PC[i] = PC1 && Y = True }
{  PC[i] := Right; }

transition spl1(i)
requires { PC[i] = PC1 && Y = False }
{  PC[i] := PC2; }

transition spl2(i)
requires { PC[i] = PC2 }
{  Y := True;
   PC[i] := PC3; }

transition splstop(i)
requires { PC[i] = PC3 && X = i }
{  PC[i] := Stop;}

transition splDown(i)
requires { PC[i] = PC3 && X <> i }
{  PC[i] := Down; }
```

FIGURE 6.4 – Code Cubicle du splitter

prouver que les trois formules suivantes ne sont pas atteignables :

$$\begin{aligned}\varphi_1 &: \exists ij.\ i \neq j \ \wedge\ \mathtt{PC}[i] = \mathtt{Stop} \ \wedge\ \mathtt{PC}[j] = \mathtt{Stop}\\ \varphi_2 &: \forall i.\mathtt{PC}[i] = \mathtt{Down}\\ \varphi_3 &: \forall i.\mathtt{PC}[i] = \mathtt{Right}\end{aligned}$$

φ_1 représentant un cube, l'analyse d'atteignabilité par Cubicle de cette formule sera automatique et terminera.

Les formules φ_2 et φ_3 en revanche, ne sont pas des cubes car elles contiennent un quantificateur universel. Nous sommes donc en présence de cubes universels et il m'est maintenant possible de montrer comment Cubicle les gère pour le moment.

6.1.1.a Traitement actuel des cubes universels

On peut se demander ce qui se passerait si on appliquait la même mécanique de pré-image aux cubes universels. La pré-image de φ_2 par $\mathtt{spl}_{down}(j)$ (j étant une nouvelle variable de processus fraîche) est ainsi la formule suivante :

$$\varphi'_2 : \exists j \forall i.\ i \neq j \implies \mathtt{X} \neq j \ \wedge\ \mathtt{PC}[j] = \mathtt{PC}_3 \ \wedge\ \mathtt{PC}[i] = \mathtt{Down}$$

Cette formule est lue comme "un état est mauvais s'il existe un processus j tel que pour tout i, si i est différent de j alors X est différent de j, PC[j] est à l'état $\mathtt{PC}_3$ et PC[i] a reçu la décision Down". La pré-image de cette nouvelle formule par la même transtion donne la formule suivante :

$$\varphi''_2 : \exists jk \forall i.\quad \begin{array}{l} i \neq j \neq k \implies \mathtt{X} \neq j \ \wedge\ \mathtt{X} \neq k \ \wedge \\ \mathtt{PC}[j] = \mathtt{PC}_3 \ \wedge\ \mathtt{PC}[k] = \mathtt{PC}_3 \ \wedge\ \mathtt{PC}[i] = \mathtt{Down} \end{array}$$

Il n'apparaît pas nécessaire de continuer à calculer les pré-images successives de ces états pour montrer qu'on arrive face au gros problème que pose les cubes universels : il peut ne pas exister de point fixe donc de condition de terminaison lorsqu'on tente de calculer les pré-images de formules contenant des quantificateurs universels. Cela est dû au fait que les transitions rajoutent des quantificateurs universels et donc de nouvelles variables de processus tous distincts menant à la possibilité de trouver une suite infinie de formules n'étant pas subsumées par les précédentes.

Le framework MCMT a, pour cette raison, toujours été limité à l'analyse de cubes afin d'assurer la terminaison de cette analyse. Néanmoins, il arrive forcément un moment où l'on rencontre de nouvelles propriétés qui ne peuvent être réduites à des cubes comme, dans notre cas, φ_2 et φ_3.

Il existe des techniques pour étendre MCMT aux quantificateurs universels. Ghilard et al. [7] proposent une transformation syntaxique dans MCMT. Cette transformation peut être interprétée comme l'ajout d'un modèle de panne franche [1] au système considéré. En pratique, cela revient à voir la quantification universelle comme une conjonction infinie et à sur-approximer cette conjonction comme une conjonction finie existentiellement quantifiée. On considère donc que tous les processus ne faisant pas partie de cette conjonction finie sont tombés en panne avant d'atteindre l'état décrit par la formule sans pour autant les mettre réellement dans un état de panne.

Dans le cas de φ_2, celle-ci serait donc transformée (en considérant que le nombre de processus n'étant pas tombés en panne est limité à un seul) en

$$\psi_2 : \exists i.\mathtt{PC}[i] = \mathtt{Down}$$

Il ne semble pas nécessaire de montrer ici pourquoi cette transformation rend un état réellement mauvais (tous les processus reçoivent la décision `Down`) en un état visiblement bon (un processus a reçu la décision `Down` sans qu'on ne sache rien des autres).

Il s'avère que cette transformation ne fonctionne qu'en présence de système tolérants aux pannes ce qui n'est pas le cas du splitter, par exemple. Le temps est donc venu de présenter la nouvelle solution que je propose pour traiter les cubes universels dans le cadre de systèmes non tolérants aux pannes.

6.2 Algorithme

Je vais présenter, dans cette section, $\mathcal{BRWP}$ (pour Backward Reachability with Weakest Precondition ou Algorithme d'atteignabilité arrière avec plus faibles préconditions), une version améliorée de l'algorithme d'atteignabilité arrière de Cubicle (et de MCMT) permettant de traiter les cubes universels (u-cubes). Pour rappel, les u-cubes sont de la forme $\exists \vec{i} \forall j.C(\vec{i}, j)$, $C(\vec{i}, j)$ étant une conjonction de littéraux paramétrés par un vecteur $\vec{i}$ de variables de processus distinctes et j une variable de processus distincte des variables de $\vec{i}$. Vérifier les u-cubes est fait en trois étapes.

Étape 1 : Analyse d'atteignabilité arrière dans un domaine fini Afin de supprimer les quantificateurs de nos u-cubes on restreint le domaine des processus à un ensemble fini d'identifiants représentés

1. Une panne franche survient quand le composant en panne cesse de fonctionner de façon immédiate et permanente et n'a plus aucune influence sur le système.

par les symboles $\#_1$, $\#_2$... La cardinalité de ce domaine est déterminée en prenant un nombre supérieur strictement au nombre de variables de processus impliquées dans les formules (ce qui, en pratique, limite généralement la cardinalité à trois ou quatre processus). Le fait de bénéficier d'un domaine fini permet d'instantier nos u-cubes en cubes. La formule φ_2, dans un domaine réduit à trois processus $\#_1$, $\#_2$ et $\#_3$ est ainsi transformée en un cube $\varphi_2^{\#3}$:

$$\varphi_2^{\#3} : \mathtt{PC}[\#_1] = \mathtt{Down} \ \wedge \ \mathtt{PC}[\#_2] = \mathtt{Down} \ \wedge \ \mathtt{PC}[\#_3] = \mathtt{Down}$$

Il est ensuite aisé de lancer l'analyse d'atteignabilité arrière usuelle utilisée par Cubicle tout en la restreignant au domaine fini précédemment fixé. La pré-image de $\varphi_2^{\#3}$ par $\mathtt{spl}_{down}(\#_1)$ est ainsi la formule ${\varphi'}_2^{\#3}$:

$${\varphi'}_2^{\#3} : \mathtt{PC}[\#_1] = \mathtt{PC}_3 \ \wedge \ \mathtt{X} \neq \#_1 \ \wedge \ \mathtt{PC}[\#_2] = \mathtt{Down} \ \wedge \ \mathtt{PC}[\#_3] = \mathtt{Down}$$

Remarque

Il est important de remarquer qu'il serait impossible, par exemple, de calculer la pré-image de $\varphi_2^{\#3}$ par $\mathtt{spl}_{down}(\#_4)$ du fait des contraintes imposées par le domaine.

Cette étape ne diffère en rien de l'analyse d'atteignabilité habituelle. Si l'algorithme d'atteignabilité termine avec une pré-image atteignant l'état initial alors le système sera aussi faillible dans le cadre paramétré. Au contraire, si l'algorithme termine en ayant atteint un point fixe il nous est seulement permis de conclure sur le fait que le système est sûr pour notre domaine fini mais sans pouvoir généraliser au cadre paramétré. Pour cela, on poursuit en passant à l'étape 2.

Étape 2 : Généralisation des invariants Afin de pouvoir prouver les u-cubes donc prouver le système dans un domaine infini, nous décidons de travailler sur les pré-images calculées à l'étape 1. Pour cela nous tentons de généraliser ces formules à un domaine infini en abstrayant les identifiants de processus par des variables de processus quantifiées universellement ou existentiellement. Avant d'effectuer ces abstractions, un problème demeure. Certaines pré-images calculées dans l'étape précédente peuvent caractériser des états inatteignables dans le domaine fini mais atteignables dans le domaine infini [2]. Prenons, par exemple, la pré-image par $\mathtt{spl}_{down}(\#_1)$ de $\varphi_2^{\#3}$, ${\varphi'}_2^{\#3}$. Comme on peut le voir dans la Figure 6.5, cette formule caractérise des états inatteignables dans un domaine limité à trois processus depuis l'état initial (donc des états mauvais pour le système) mais qui deviennent atteignables (et donc sûrs) dès qu'un quatrième processus sur lequel on n'a aucune information donnée par la formule apparaît. Dans le cas de ${\varphi'}_2^{\#3}$, par exemple, on sait que X est différent de $\#_1$ donc il est tout à fait possible que X soit égal à $\#_4$ et que $\#_4$ soit dans l'état $\mathtt{PC}_3$ et prêt à recevoir la décision Down ce qui caractérise bien un état sûr du point de vue des règles du splitter.

On vérifie que chaque pré-image calculée dans le domaine fini représente bien un état mauvais dans le domaine infini en les transformant en cubes quantifiés existentiellement par des variables de processus qui viennent remplacer les constantes de processus $\#_1$, $\#_2$... et en relançant l'algorithme d'atteignabilité arrière sur ses cubes paramétrés. A l'issue de cette nouvelle analyse, tous les cubes qui sont effectivement mauvais sont gardés pour l'étape 3. Les cubes ayant été rejetés car étant sûrs dans le domaine infini sont

2. Ce qui semble tout à fait normal étant donné que ces algorithmes ne sont pas tolérants aux pannes.

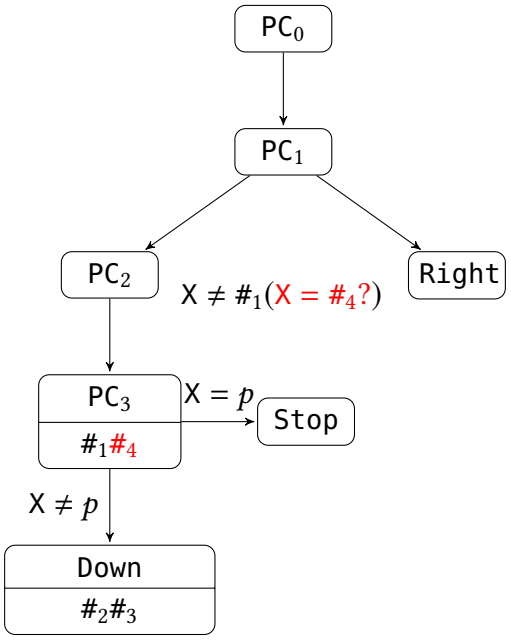


FIGURE 6.5 – Cas d'une formule mauvaise dans un domaine fini et bonne dans un domaine infini - $\varphi'^{\#3}_2$: Les processus $\#_2$ et $\#_3$ ont reçu la décision Down tandis que le processus $\#_1$ est dans l'état $\mathtt{PC}_3$, prêt à recevoir la décision Down mais un processus supplémentaire $\#_4$ peut être dans n'importe quel état

retransformés en u-cubes [3]. Pour transformer un cube en u-cube on abstrait les constantes de processus en variables de processus qui sont

— existentiellement quantifiées si elles ont été touchées par une transition
— universellement quantifiées si elles n'ont jamais été touchées

Un exemple valant souvent mieux que beaucoup d'explications, reprenons la formule $\varphi'^{\#3}_2$. Cette formule est transformée, comme décrit précédemment, en un cube $\varphi'^{\exists}_2$ en abstrayant les trois constantes de processus $\#_1$, $\#_2$ et $\#_3$ en trois variables de processus $\mathtt{p}_1$, $\mathtt{p}_2$ et $\mathtt{p}_3$ existentiellement quantifiées :

$$\varphi'^{\exists}_2 : \exists p_1 p_2 p_3 . p_1 \neq p_2 \neq p_3 \ \wedge\ \mathtt{X} \neq p_1 \ \wedge\ \mathtt{PC}[p_1] = \mathtt{PC}_3 \ \wedge\ \mathtt{PC}[p_2] = \mathtt{Down} \ \wedge\ \mathtt{PC}[p_3] = \mathtt{Down}$$

Comme on l'a vu dans la Figure 6.5, en exécutant l'algorithme d'atteignabilité arrière depuis $\varphi'^{\exists}_2$ on se rend compte qu'elle décrit des états sûrs. En tant que telle cette formule ne peut donc pas être gardée afin de servir d'invariant du système. Il faut donc la transformer en u-cube. On remarque pour cela que la suite de transitions ayant mené à $\varphi'^{\#3}_2$ n'est composée que de $\mathtt{spl}_{down}(\#_1)$ et que cette transition n'implique (que ce soit dans ses paramètres, sa garde ou son action) que $\#_1$ et donc ni $\#_2$ ni $\#_3$. Sémantiquement, cela peut être vu comme le fait d'avoir fait sortir un processus du domaine infini sans avoir touché aux autres ce qui se traduit par

1. Ajouter une variable fraîche de processus $\mathtt{p}_1$ quantifiée existentiellement
2. Regrouper toutes les autres constantes de processus sous une même variable fraîche de processus $\mathtt{p}_2$ quantifiée universellement

3. On remarque qu'il aurait été possible de transformer immédiatement toutes les pré-images du domaine fini en u-cubes mais il s'avère qu'en pratique les cubes permettent d'exprimer des propriétés beaucoup plus fortes pour prouver les systèmes.

Après avoir effectué ces changements on peut enfin transformer $\varphi'^{\#3}_2$ en $\varphi'^{\exists\forall}_2$:

$$\varphi'^{\exists\forall}_2 : \exists p_1.\forall p_2.p_1 \neq p_2 \implies \mathsf{X} \neq p_1 \ \wedge \ \mathsf{PC}[p_1] = \mathsf{PC}_3 \ \wedge \ \mathsf{PC}[p_2] = \mathtt{Down}$$

Cette formule représente bien des états mauvais étant donné qu'elle caractérise l'ensemble des états tels qu'il existe un processus p_1 dans l'état PC_3 tel que $\mathsf{X} \neq \mathsf{p}_1$ et tel que tout autre processus a reçu la décision Down [4] Une fois que toutes les formules ont été transformées en cubes, filtrées puis transformées en u-cubes on peut passer à l'étape 3.

Étape 3 : Vérification déductive A l'issue des étapes 1 et 2 nous nous retrouvons avec des propriétés φ_1, φ_2..., des cubes $\varphi_1^{\exists}$, $\varphi_2^{\exists}$... et des u-cubes $\varphi_1^{\exists\forall}$, $\varphi_2^{\exists\forall}$... nous permettant de créer une nouvelle formule ψ telle que

$$\psi = \varphi_1 \wedge \varphi_2 \wedge \cdots \wedge \varphi_1^{\exists} \wedge \varphi_2^{\exists} \wedge \cdots \wedge \varphi_1^{\exists\forall} \wedge \varphi_2^{\exists\forall} \wedge \ldots$$

Pour prouver que φ_1, φ_2... sont des propriétés du système, étant donné qu'il nous est impossible de le faire avec Cubicle, nous prouvons ψ au moyen des techniques de vérification déductive.

Contrairement à la vérification de modèle, la vérification déductive n'implique pas d'analyser l'ensemble des états possibles du système mais de vérifier que les propriétés qu'on a énoncé sur celui-ci restent vraies tout au long de l'exécution du système. Ainsi la vérification déductive est conduite par les propriétés à prouver. L'exhaustivité de la vérification de modèle permet, avec un outil suffisamment performant, de donner très peu d'indications autres que les propriétés qu'on souhaite prouver en contrepartie d'une plus faible résistance à des propriétés complexes (comme on l'a vu, dans le cadre de Cubicle nous sommes obligés de nous limiter à des conjonctions existentiellement quantifiées). Dans la vérification déductive, les formules qu'on peut prouver peuvent être bien plus complexes en contrepartie d'une plus grande charge de travail pour l'utilisateur qui est obligé de fournir bien plus d'informations à l'outil pour qu'elle arrive à un résultat.

Prenons comme exemple, pour illustrer véritablement la différence entre vérification déductive et vérification de modèle un système tout simple qui incrémente une variable X initialement égale à 0. On veut prouver que X ne peut jamais être négative. En vérification de modèle, l'outil part de $\mathsf{X} < 0$ (état mauvais) et par pré-image découvre que cet état est atteignable par tout état où $\mathsf{X} < -1$ et ainsi de suite. À moins d'avoir un outil de raisonnement arithmétique suffisament puissant pour se rendre compte que tout état où $\mathsf{X} < -1$ est forcément un état où $\mathsf{X} < 0$ le model checker continuera de calculer des pré-images indéfiniment. En revanche, un outil de vérification déductive prouvera que cette propriété est correcte immédiatement à condition d'ajouter des propriétés intermédiaires dans le programme. On sacrifie dans un cas l'expressivité au bénéfice de la simplicité et dans l'autre la simplicité au bénéfice de l'exhaustivité.

Face à cette dichotomie flagrante entre ces deux méthodes, l'idée nous est donc venue d'utiliser les invariants calculés automatiquement par Cubicle afin de prouver notre système dans un outil de vérification déductive. Nous avons choisi comme outil de vérification déductive, étant donné la quantité impressionnante de solveurs SMT ou TPTP qu'il permet d'utiliser, Why3 [5]. Il nous fallait donc traduire un fichier Cubicle ainsi que tous ses invariants en un fichier WhyML ce qui peut sembler trivial en apparence mais s'avère bien plus retors en pratique car, notamment

4. on peut se convaincre de l'état mauvais de cette formule en lui appliquant la transition $\mathtt{spl}_{down}(\mathsf{p}_1)$ qui fait passer le système à un état où tous les processus ont reçu la décision Down.

5. Un autre avantage certain était la présence à nos côtés de l'équipe de recherche travaillant dessus.

— il existe un vrai fossé sémantique entre les systèmes de transition à tableaux et WhyML
— la façon dont on représente les transitions de Cubicle en WhyML a une influence non négligeable sur son moteur de déduction

Une fois cette traduction effectuée on lance le moteur de déduction de Why3 avec les solveurs de notre choix. Il incombe donc à Why3 de prouver que ψ est maintenue par le système bien qu'une fois la traduction faite il soit complètement possible pour l'utilisateur d'ajouter de nouvelles propriétés à la main tout en ayant été déchargé du plus gros du travail.

6.3 Implémentation dans Cubicle

L'implémentation de $\mathcal{BRWP}$ est présentée dans Algorithme 11. Cette implémentation étend l'algorithme d'atteignabilité arrière de MCMT déjà implémentée dans Cubicle afin de pouvoir gérer les u-cubes.

Algorithme 11: Analyse d'atteignabilité arrière pour les u-cubes

Input:
φ : formule quantifiée (existentiellement et/ou universellement)
c : constante d'instantiation permettant de fixer le nombre de processus qu'on souhaite utiliser dans les formules
Variables:
$\mathcal{V}^{\#c}$: nœuds instanciés visités
Q : file d'attente

```
1  function BRWP(φ, c) : begin
2      φ^#c := Instantiate (φ, c);
3      V^#c := ∅;
4      push(Q, φ^#c);
5      while not_empty(Q) do
6          φ^#c := pop(Q);
7          if (I ∧ φ^#c sat) then
8              renvoyer unsafe
9          else if (φ^#c ⊭ V^#c) then
10             V^#c := V^#c ∪ {φ^#c};
11             push(Q, FinitePre(φ^#c, c));
12         end
13     end
14     S1, V'^#c := Generalize_and_filter(V^#c);
15     S2 := Universal_Generalization(V'^#c);
16     Check_inductive_invariant(φ ∧ S1 ∧ S2)
17 end
```

Cet algorithme, comme celui d'atteignabilité arrière vu dans l'Algorithme 1, prend en entrée une formule φ ainsi qu'une constante entière c. Après avoir instancié φ en $\varphi^{\#c}$ on crée l'ensemble $\mathcal{V}$, ensemble des nœuds visités, initialement vide (ligne 3) ainsi qu'une file Q de nœuds en attente d'être traités conte-

nant initialement $\varphi^{\#c}$. L'instanciation de φ en $\varphi_c^{\exists}$ se fait comme décrit à l'étape 6.2 en fixant la cardinalité du domaine fini à c. La boucle principale de cet algorithme ne diffère ensuite de l'algorithme originel que dans le calcul de la pré-image effectué par FinitePre$^*(\varphi^{\#})$. FinitePre calcule les pré-images sans pouvoir ajouter de nouveaux processus et en filtrant naturellement toute formule trivialement impossible dans le domaine fini. Une formule trivialement impossible est par exemple une formule contenant, dans le cadre du splitter, $\mathsf{X} \neq \#_1 \wedge \mathsf{X} \neq \#_2 \wedge \mathsf{X} \neq \#_3$ dans un domaine limité à trois constantes de processus. L'algorithme $\mathcal{BRWP}$ termine si une des deux conditions suivantes est remplie :

— Le test de sûreté (ligne 7) échoue indiquant qu'on est en présence d'un nœud correspondant à un état initial et donc à la faillibilité du système
— Il ne reste plus aucun nœud à traiter (ligne 5) dans Q ce qui signifie qu'un point fixe du système à été atteint et que, dans le domaine fini, le système est sûr.

Le système restreint à un domaine fini étant inclus dans le système paramétré originel, s'il est faillible alors le système paramétré le sera aussi. On s'intéresse donc au cas où le système restreint est sûr. Pour prouver que ce système est sûr dans le cas général, il faut dans ce cas, comme vu à l'étape 6.2, généraliser et filtrer les nœuds (ligne 14) comme décrit dans la sous-section 6.3.1, généraliser de façon universelle les nœuds ayant été filtrés (ligne 15) comme présenté dans la sous-section 6.3.2 puis vérifier que l'ensemble des invariants trouvés permet bien d'assurer la sûreté du système conformément à l'étape 6.2 (ligne 16). Les détails de cette vérification sont décrits dans la Section 6.4

6.3.1 Généralisation et filtrage des cubes

L'Algorithme 12 décrit le fonctionnement de la fonction `Generalize_and_filter`. Cette fonction prend ainsi en paramètre l'ensemble $\mathcal{V}^{\#c}$ des cubes instantiés obtenus lors de l'analyse d'atteignabilité arrière par application de FinitePre. Cette fonction se contente de renommer les constantes de processus en variables de processus liées à des quantificateurs existentiels. On remarque néanmoins à la ligne 13, la présence d'une fonction `Simplify`. Cette fonction est en effet très utile puisqu'elle permet, étant donné que le domaine des constantes de processus est fini, de simplifier certains cubes.

Reprenons, par exemple, la formule $\varphi'^{\#3}_2$ vue dans la Section 6.2. Pour rappel, la formule est la suivante :

$$\varphi'^{\#3}_2 : \mathsf{PC}[\#_1] = \mathsf{PC}_3 \ \wedge \ \mathsf{X} \neq \#_1 \ \wedge \ \mathsf{PC}[\#_2] = \mathsf{Down} \ \wedge \ \mathsf{PC}[\#_3] = \mathsf{Down}$$

Sa pré-image par $\mathtt{spl}_{down}(\#_2)$ permet d'obtenir la formule suivante :

$$\varphi''^{\#3}_2 : \mathsf{X} \neq \#_1 \ \wedge \ \mathsf{X} \neq \#_2 \ \wedge \ \mathsf{PC}[\#_1] = \mathsf{PC}_3 \ \wedge \ \mathsf{PC}[\#_2] = \mathsf{PC}_3 \ \wedge \ \mathsf{PC}[\#_3] = \mathsf{Down}$$

Du fait de la limitation du domaine à trois constantes, X étant différent de $\#_1$ et $\#_2$, il est forcément égal à $\#_3$. Avant d'être généralisée, la formule est donc transformée en

$$\varphi''^{\#3}_2 : \mathsf{X} = \#_3 \ \wedge \ \mathsf{PC}[\#_1] = \mathsf{PC}_3 \ \wedge \ \mathsf{PC}[\#_2] = \mathsf{PC}_3 \ \wedge \ \mathsf{PC}[\#_3] = \mathsf{Down}$$

Cette formule est ensuite généralisée en

$$\varphi''_2 : \exists \mathsf{p}_1, \mathsf{p}_2, \mathsf{p}_3 . \mathsf{p}_1 \neq \mathsf{p}_2 \neq \mathsf{p}_3 \ \wedge \ \mathsf{X} = \mathsf{p}_3 \ \wedge \ \mathsf{PC}[\mathsf{p}_1] = \mathsf{PC}_3 \ \wedge \ \mathsf{PC}[\mathsf{p}_2] = \mathsf{PC}_3 \ \wedge \ \mathsf{PC}[\#_3] = \mathsf{Down}$$

Algorithme 12: Filtrage et généralisation

Input:
$\mathcal{V}^{\#c}$: Ensemble des nœuds instanciés
Variables:
$\mathcal{V}'^{\#c}$: Ensemble des nœuds instanciés ne pouvant pas être généralisés trivialement
$\mathcal{S}_1$: nœuds généralisés trivialement

```
1  function Generalize_and_filter(V^#c) : begin
2      V'^#c := V^#c ;
3      S_1 := ∅;
4      forall φ^#c ∈ V^#c do
5          Δ_∃ := ⊤ ;
6          forall p ∈ V⃗^#c do
7              v := Fresh_Variable();
8              Δ_∃ := Δ_∃ ∧ ∃v;
9              replace(p, v, φ^#c);   /* Remplace toutes les occurrences du processus p
                par la variable de processus v */
10         end
11         D = Distinct(Δ_∃);            /* Toutes les variables sont distinctes */
12         φ = Δ_∃ ∧ D ∧ φ^#c;
13         Simplify(φ);
14         if BWD(φ) safe then
15             S_1 := S_1 ∪{φ};
16             V'^#c := V'^#c \{φ^#c}
17         end
18     end
19     renvoyer (S_1, V'^#c)
20 end
```

Suite à cette simplification suivie de la généralisation, chaque formule φ est fournie à l'algorithme d'atteignabilité arrière pour les systèmes paramétrés donc sans la contrainte de cardinalité finie. Le vérificateur de modèle termine, comme on l'a déjà vu, dans deux cas, soit lorsque la propriété de départ est sûre (pas d'intersection avec les états initiaux), en ce cas φ est ajoutée à l'ensemble $\mathcal{S}_1$ et sa version instanciée $\varphi^{\#}$ est supprimée de l'ensemble $\mathcal{V}'^{\#c}$. À l'issue de cette fonction, deux ensembles sont renvoyés, $\mathcal{S}_1$, l'ensemble des cubes envoyés tel quel au moteur de vérification déductive et $\mathcal{V}'^{\#c}$, l'ensemble des u-cubes instanciés nécessitant un traitement plus fin grâce à la fonction `Universal_Generalization`.

6.3.2 Généralisation universelle des cubes

Le fonctionnement de `Universal_Generalization` est présenté dans l'Algorithme 13. De même que pour l'Algorithme 12, cette fonction prend en paramètre un ensemble $\mathcal{V}^{\#c}$ de u-cubes instanciés.

Algorithme 13: Généralisation universelle

Input:
$\mathcal{V}'^{\#c}$: Ensemble des nœuds instanciés ne pouvant être généralisés trivialement
Variables:
$\mathcal{S}_2$: nœuds généralisés universellement

```
1  function Universal_Generalization(V'^{#c}) : begin
2      S_2 := ∅;
3      forall φ^{#c} (étiqueté par V^{#c} = (V^{#c}_∃, V^{#c}_∀)) ∈ V^{#c} do
4          if V^{#c}_∀ = ∅ then Filter_Out(φ);
5          else
6              Δ_∃ := ⊤ ;
7              Δ_∀ := ⊤ ;
8              forall p ∈ V^{#c}_∃ do
9                  v := Fresh_Variable();
10                 Δ_∃ := Δ_∃ ∧ ∃v;
11                 replace(p, v, φ^{#c});          /* Remplace toutes les occurrences de la
                    constante de processus p par la variable de processus fraîche v
                    */
12             end
13             v := Fresh_Variable();
14             Δ_∀ := ∀v;
15             Remove_and_Replace(p, v, φ^{#c});    /* Enlève tout littéral contenant des
                processus présents dans V^{#c}_∀ et en ajoute un nouveau équivalent
                contenant la variable de processus fraîche v */
16             D = Distinct(Δ_∃, Δ_∀)/* Toutes les variables doivent être distinctes
                   */
17             φ = Δ_∃ ∧ Δ_∀ ∧ D ⟹ φ^{#c};
18             S_2 := S_2 ∪{φ};
19         end
20     end
21 end
```

Cet algorithme relativement simple permet de séparer les constantes en deux parties afin de les traiter en fonction du quantificateur auquel elles sont attachées. Je vais l'expliquer pas à pas en repartant de la propriété initiale qu'on cherche à prouver

$$\varphi : \forall p.\mathtt{PC}[p] = \mathtt{Down}$$

Cette formule, comme vu précédemment, est instanciée pour devenir (avec $c = 3$) :

$$\varphi^{\#3} : \mathtt{PC}[\#_1] = \mathtt{Down} \wedge \ \mathtt{PC}[\#_2] = \mathtt{Down} \wedge \ \mathtt{PC}[\#_3] = \mathtt{Down}$$

A l'instanciation, chacune de ces formules est étiquetée par un vecteur de processus $\overrightarrow{V}^{\#c}$ (ici $\overrightarrow{V}^{\#3}$) permettant de distinguer à quel quantificateur chaque constante de processus est liée. Dans le cas de

$\varphi^{\#3}$, $\overrightarrow{V}^{\#3} = \{\#_1, \#_2, \#_3\}_\forall$, signifiant que les trois constantes de processus sont liées au quantificateur universel. Le calcul de la pré-image de $\varphi^{\#3}$ par $\mathtt{spl}_{down}(1)$ permet d'obtenir la formule

$$\varphi_1^{\#3} = \mathtt{PC}[\#_1] = \#_3 \ \wedge\ \mathtt{X} \neq \#_1 \ \wedge\ \mathtt{PC}[\#_2] = \mathtt{Down} \ \wedge\ \mathtt{PC}[\#_3] = \mathtt{Down}$$

On remarque que cette transition ne mentionne que $\#_1$ et ne modifie ou ajoute que des littéraux mentionnant $\#_1$. Le vecteur associé à cette nouvelle formule doit donc illustrer le fait que $\#_1$ a été sorti du giron du quantificateur universel en passant par un quantificateur existentiel. Le vecteur associé à $\varphi_1^{\#3}$ devient donc $\overrightarrow{V}^{\#3} = \{\#_1\}_\exists, \{\#_2, \#_3\}_\forall$. Chaque vecteur est décomposé en deux sous-vecteurs, $\overrightarrow{V}^{\#c}_\exists$ et $\overrightarrow{V}^{\#c}_\forall$ correspondants aux constantes liées à des quantificateurs existentiels et aux variables correspondants au quantificateur universel originel. Si $\overrightarrow{V}^{\#c}_\forall$ est vide (ligne 4, la formule n'est quantifiée que par des quantificateurs existentiels et correspond donc au cube ayant été filtré par le moteur d'atteignabilité arrière lors de l'exécution de `Generalize_and_filter()`, cette formule est donc définitivement écartée car résultant de la finitude du domaine mais ne pouvant être pertinente dans notre cas. Toutes les littéraux contenant une ou plusieurs constantes de processus appartenant à $\overrightarrow{V}^{\#c}_\exists$ sont maintenus en renommant chacune de ces constantes grâce à des variables fraîches de processus liées chacune à un quantificateur existentiel (ligne 8 à ligne 12). Les littéraux contenant, eux, une ou plusieurs constantes de processus appartenant à $\overrightarrow{V}^{\#c}_\forall$ sont fusionnés en un seul quantifié par des variables fraîches de processus liées à un quantificateur universel (ligne 13 à ligne 15). A l'issue de cette généralisation, on inscrit dans la formule le fait que chaque variable doit être distincte des autres avant d'ajouter la nouvelle formule obtenue à l'ensemble final des invariants. Dans le cas de $\varphi_1^{\#3}$, la formule finale est donc

$$\varphi_1 = \exists \mathtt{p}_1.\forall \mathtt{p}_2.\mathtt{p}_1 \neq \mathtt{p}_2 \implies \mathtt{PC}[\mathtt{p}_1] = \mathtt{PC}_3 \ \wedge\ \mathtt{X} \neq \mathtt{p}_1 \ \wedge\ \mathtt{PC}[\mathtt{p}_2] = \mathtt{Down}$$

$\mathtt{p}_2$ et $\mathtt{p}_3$ étant toujours liés au quantificateur universel, on a réduit $\mathtt{PC}[\#_2] = \mathtt{Down}$ et $\mathtt{PC}[\#_3] = \mathtt{Down}$ en $\mathtt{PC}[\mathtt{p}_2] = \mathtt{Down}$ lié à la variable de processus $\mathtt{p}_2$ universellement quantifiée. La formule représente bien des états mauvais puisqu'elle représente tous les états pour lesquels il existe un processus $\mathtt{p}_1$ tel que pour tout processus $\mathtt{p}_2$ différent de $\mathtt{p}_1$, $\mathtt{X} \neq \mathtt{p}_1$, $\mathtt{p}_1$ est dans l'état $\mathtt{PC}_3$ et chaque processus $\mathtt{p}_2$ a reçu la décision `Down`. Un aperçu des deux fonctions de généralisation est donné dans la Figure 6.6

6.4 Traduction vers Why3

La dernière fonction appelée dans l'Algorithme 11 est `Check_inductive_invariant`$(\varphi \wedge \mathcal{S}_1 \wedge \mathcal{S}_2)$ ce qui correspond au passage de Cubicle à Why3.

6.4.1 Qu'est-ce que Why3 ?

Why3 [68] est, comme l'indiquent ses créateurs, une plateforme pour la vérification déductive. C'est donc un outil prenant en entrée un programme $\mathcal{P}$ et un ensemble de spécifications $\mathcal{T}$(contenant donc les théories, invariants de programmes et propriétés nécessaires à la vérification de $\mathcal{P}$). Grâce à un moteur de plus faible pré-condition (WP pour Weakest Precondition), Why3 va tenter de vérifier que $\mathcal{T}$représente

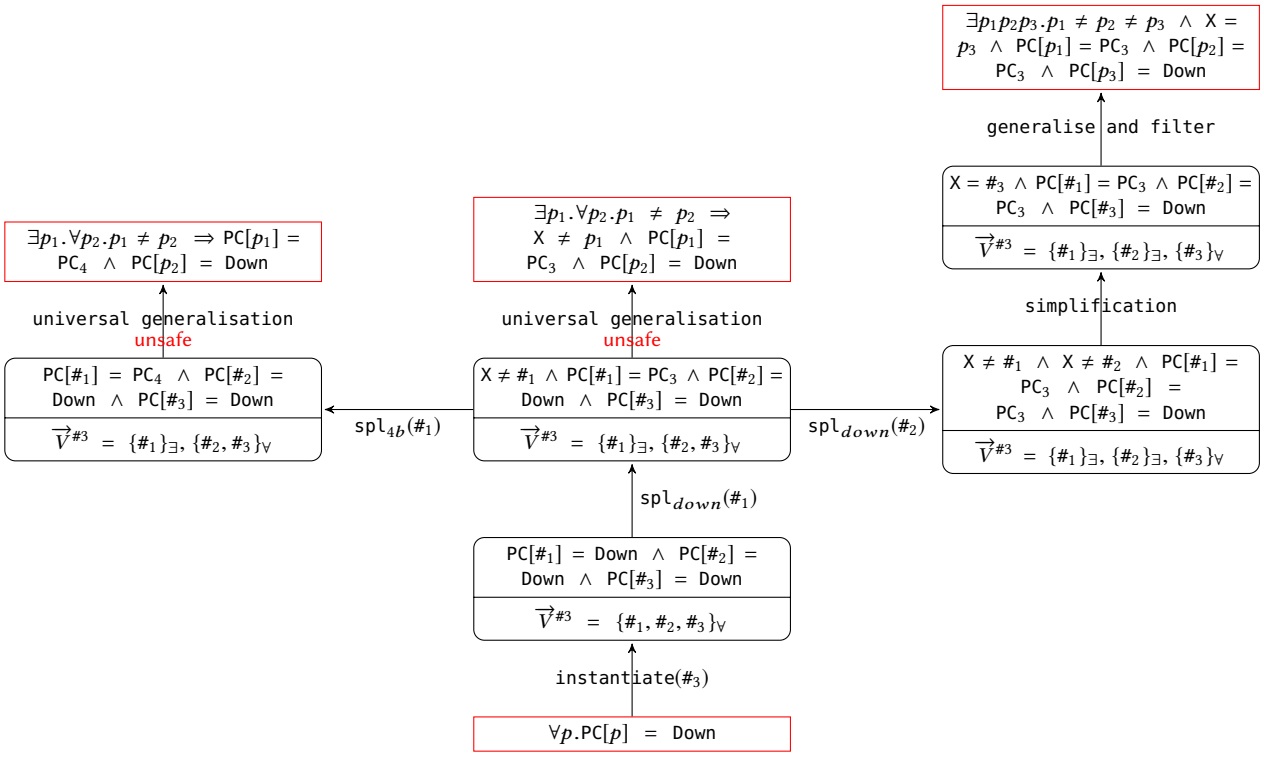


FIGURE 6.6 – Premiers nœuds du splitter suivis de leur simplification, filtrage et généralisation

bien un invariant inductif de $\mathcal{P}$. Why3 va donc générer un ensemble de buts à prouver qu'il enverra aux solveurs de notre choix (solveurs SMT comme Alt-Ergo, CVC4 ou Z3, solveurs TPTP comme Vampire ou SPASS et assistants de preuve comme Coq ou PVS).

Ce document ne s'intéresse nullement à expliquer le fonctionnement de Why3 mais le lecteur intéressé pourra trouver toutes les explications supplémentaires qu'il souhaitera sur [68].

La partie qui nous intéresse est donc le passage de Cubicle à Why3 que je vais donc présenter dans la sous-section suivante.

6.4.2 De Cubicle à Why3

Le principe de base de `Check_inductive_invariant()` est de traduire un fichier Cubicle avec ses invariants en un fichier WhyML avec ses spécifications et invariants puis de faire tourner Why3 sur ce fichier en espérant qu'il le prouvera sans autre intervention humaine. Avant de commencer il me faut néanmoins exprimer clairement la difficulté inhérente à une telle démarche. Cubicle et les systèmes de transitions à tableaux sont conçus pour prouver des systèmes concurrents, non déterministes et ne terminant pas. Why3 a été conçu pour prouver des systèmes séquentiels, déterministes qui terminent. La difficulté a donc été de surmonter ces différences fondamentales de paradigmes. Dans la suite de cette section, je vais donc illustrer cette traduction en continuant d'utiliser le splitter.

6.4.2.a Types et déclarations

Tout comme pour Cubicle, il faut commencer par déclarer les types, variables et tableaux du systèmes. Why3, du fait de sa grande richesse logique, se doit d'être un tant soit peu exigent et impose donc, afin de ne pas provoquer de conflits, de redéfinir l'égalité pour chaque type créé. On ne peut donc, comme dans Cubicle, se contenter d'écrire type state = PC0 | PC1 | ... mais il faut encapsuler cette déclaration dans un *scope* (Figure 6.7) avec une définition de l'égalité pour chaque nouveau type algébrique créé.

```
scope import State
  type state = PC0 | PC1 | PC2 | PC3 | Stop | Down | Right

  let (=) (a b: state): bool
    ensures {result <-> a = b}
    = match a, b with
      | PC0, PC0 | PC1, PC1 | PC2, PC2 | PC3, PC3 | Stop, Stop | Down, Down | Right, Right -> true
      | _ -> false
  end
end
```

FIGURE 6.7 – Déclaration du type state dans WhyML

Le type proc étant interne à Cubicle, on le déclare comme alias du type int afin de ne pas avoir à changer toutes les déclarations de type. On pourrait très bien s'en passer mais cela permet de garder une forme de cohérence syntaxique entre les deux fichiers. De plus, on remarque que les post-conditions dans Why3 mentionnent une variable `result`. Cette variable représente la valeur renvoyée par toute fonction de WhyML à l'issue de son exécution (dans le cas de la déclaration de l'égalité pour state on spécifie donc que `result` est vraie si les deux éléments comparés sont effectivement égaux et fausse sinon.)

Étant donné qu'on souhaite pouvoir exprimer des propriétés sur n'importe quel élément du système à l'issue de celui-ci et que le seul moyen de parler de l'état du système lorsque le programme termine est d'utiliser cette variable `result`, on représente l'état du système sous la forme d'un enregistrement contenant toutes les variables et tableaux le composant. Dans le cas du splitter, on le représente donc sous la forme d'un enregistrement contenant deux variables mutables `x` de type `proc` et `y` de type `bool` ainsi qu'un tableau `pc` de type `state`. Les tableaux sont des structures impératives (donc modifiables en place). On aurait pu choisir, étant donné que WhyML est un langage à la ML, de manipuler des structures de données fonctionnelles, le choix de manipuler des structures de données impératives découle de l'aspect impératif de Cubicle et de la volonté de rester le plus proche possible du fichier d'origine (Figure 6.8) afin de permettre à l'utilisateur utilisant ces deux outils de ne pas être perdu après la traduction de son fichier Cubicle vers Why3.

```
type system = {
  mutable x : proc;
  mutable y : bool;
  pc : array state;
}
```

FIGURE 6.8 – Déclaration de l'enregistrement correspondant à l'état du système en WhyML

6.4.2.b Exécution infinie

La sémantique derrière un système de transition à tableau consiste en l'exécution infinie de deux actions :

— Évaluer l'ensemble des transitions pouvant être exécutées étant donné l'état actuel du système
— Choisir l'une de ces transitions afin de mettre à jour le système

D'un point de vue logique, tout programme ne terminant pas permet de satisfaire n'importe quelle propriété. Why3 cherchant à vérifier que $\{\mathcal{T}\}\mathcal{P}\{\mathcal{T}\}$, si $\mathcal{P}$ ne termine pas, quel que soit $\mathcal{T}$, la réponse sera que $\mathcal{P}$ satisfait $\mathcal{T}$. Pour contourner ce problème, on paramètre aussi la terminaison de $\mathcal{P}$. Le programme résultant du fichier Cubicle en WhyML est donc un programme paramétré par le nombre de processus et par le nombre d'étapes d'exécution ce qui permet de cacher la non terminaison (Figure 6.9). La boucle n'est donc pas infinie mais s'arrête quand elle atteint le nombre maximal d'étapes autorisés, ce nombre étant un paramètre de la fonction et étant donc arbitraire. Cette construction permet d'assurer la terminaison de la boucle principale (puisqu'à chaque étape on se rapproche du nombre maximal d'étapes autorisées – `variant { maxsteps - !nbsteps }` –, Why3 est assuré que la boucle est finie et que le programme termine).

```
let splitter1 (maxprocs : int) (maxsteps : int) : system
  requires { 0 < maxprocs }
  (* ... *)
  =
  (* ... *)
  while ( !nbsteps < maxsteps ) do
    variant { maxsteps − !nbsteps }
    incr nbsteps;
    (* ... *)
  done;
  (* ... *)
end
```

FIGURE 6.9 – Boucle principale du programme en WhyML

6.4.2.c États initiaux

Pour rappel, la formule initiale correspondant au splitter est la suivante :

$$\mathcal{I}nit : \forall p.\mathsf{PC}[p] = \mathsf{PC}_0 \wedge \mathsf{Y} = \bot$$

La sémantique derrière cette formule est que tout état pour lequel Y est faux et PC ne contient que PC_0 est, quelle que soit la valeur de X, un état initial du système. Why3, comme la plupart des langages de programmation, n'accepte pas de manipuler des variables non initialisées dans un enregistrement. On contourne ce problème en initialisant X à une valeur positive aléatoire inférieure au nombre de processus (Figure 6.10).

```
let s = {
  y = false;
  pc = Array.make maxprocs PC0; (* Nouveau tableau de taille maxprocs initialisé à PC0 *)
  x = Random.random_int maxprocs; (* Valeur aléatoire comprise entre 0 et maxprocs−1 *)
} in
```

FIGURE 6.10 – Initialisation du système en WhyML

6.4.2.d Transitions et indéterminisme

Les systèmes de transitions à tableaux sont indéterministes de deux façons différentes. Pour rappel, la boucle principale choisit une transition parmi toutes celles applicables sur l'état actuel du système. Supposons, par exemple, qu'un système contienne les deux transitions suivantes :

$$t_1 : \exists p.\mathsf{PC}[\mathsf{p}] = \mathsf{PC}_0 \wedge \mathsf{PC'}[\mathsf{p}] = \mathsf{PC}_1$$

$$t_2 : \exists p.\mathsf{PC}[\mathsf{p}] = \mathsf{PC}_0 \wedge \mathsf{PC'}[\mathsf{p}] = \mathsf{PC}_2$$

On remarque que si $\mathsf{PC}[\mathsf{p}] = \mathsf{PC}_0$ pour un p quelconque, les deux transitions peuvent être choisies avec, dans le premier cas, un nouvel état du système où $\mathsf{PC}[\mathsf{p}] = \mathsf{PC}_1$ et, dans le deuxième cas, un nouvel état du système où $\mathsf{PC}[\mathsf{p}] = \mathsf{PC}_2$. Ses deux transitions doivent pouvoir être choisies dans Why3 ce que ne permet pas une traduction triviale avec des instructions conditionnelles. En effet, comme on peut le voir dans la Figure 6.11, si on traduit chaque transition comme une instruction conditionnelle `si-alors` (c'est-à-dire sans `sinon`), la première branche sera toujours exécutée et modifiera PC[p], rendant impossible l'exécution de la seconde. L'autre problématique introduite par le fait de ne pas utiliser de `sinon` est qu'il devient possible d'exécuter plusieurs instructions conditionnelles dans le même tour de boucle si les conditions le permettent.

Afin de régler ce problème, on utilise une fonctionnalité très pratique de Why3, la définition de fonction sans besoin de la spécifier. On définit donc une fonction `coin` imitant un lancer de pièce. Chaque transition se voit donc augmentée d'un lancer de pièce pour savoir si elle peut être exécutée ou pas, ce

```
if pc.[p] = PC0
then pc.[p] <- PC1
else if pc.[p] = PC0
then pc.[p] <- PC2
```

FIGURE 6.11 – Mauvaise traduction de transitions en WhyML car introduisant du déterminisme

```
val coin () : bool
if coin () && pc.[p] = PC0
then pc.[p] <- PC1
if coin () && pc.[p] = PC0
then pc.[p] <- PC2
```

FIGURE 6.12 – Bonne traduction de transitions en WhyML car conservant l'indéterminisme

qui permet à Why3 d'explorer toutes les possibilités. Dans notre exemple précédent, les transitions sont donc traduites comme dans la Figure 6.12

Du fait que Cubicle soit sémantiquement dans le paradigme concurrentiel, à tout instant n'importe quel processus peut vouloir activer une transition. La deuxième cause d'indéterminisme vient donc du (ou des) processus choisi(s) à chaque tour de boucle. On doit, à tout instant, pouvoir exécuter n'importe quelle transition avec n'importe quel(s) processus unique(s) si tant est que la garde soit vérifiée. Il faut bien sûr que ce(s) processus choisi(s) soi(en)t arbitraire(s). On définit donc pour cela une fonction commune à toutes les traductions, `k_random` prenant en paramètres le nombre maximum de processus dans le système n ainsi que le nombre de processus distincts dont on a besoin k. k est déterminé en calculant le nombre maximum de processus nécessaires impliqués dans les paramètres, gardes et actions des transitions. Cette fonction doit donc garantir que chaque processus généré sera distinct des autres et compris dans le domaine des valeurs possibles de processus. Tout comme pour la fonction `coin`, il n'est nul besoin de spécifier `k_random`. Néanmoins, suite à mes expériences il est apparu que Why3 et les solveurs qu'il emploie avaient beaucoup de mal avec les listes et autres structures de données algébriques. C'est pourquoi cette fonction renvoie un tableau de taille k comme on peut le voir dans la Figure 6.13 (dans le cas du splitter, k est égal à 1)

6.4.2.e Invariants

Après avoir traduit l'ensemble du système, il faut ajouter la dernière couche, les invariants. On pourrait, soit dit en passant, tout à fait fournir le fichier en l'état à Why3 pour voir s'il le prouve seul mais il n'y arrivera pas. Il faut en effet ajouter les invariants de boucles qui sont en réalité des invariants du système. Comme on l'a vu dans la Section 6.2, ces invariants ont été trouvés en utilisant l'algorithme d'atteignabilité arrière de Cubicle. En pratique, on a utilisé une version plus efficace de cet algorithme, BRAB [25, 26]. Ces invariants sont donc automatiquement ajoutés au fichier WhyML sous forme néga-

```
val k_random (k:int) (n:int) : (result:array int)
  requires { 0 <= k }
  requires { k <= n }
  ensures { length result = k }
  ensures { forall i j:int. 0 <= i < n /\ 0 <= j < n /\ i <> j ->
            result[i] <> result[j] }
  ensures { forall i:int. 0 <= i < n -> 0 <= result[i] < n }
```

FIGURE 6.13 – Fonction k_random permettant de générer k processus arbitraires distincts

tive (ils représentent en effet des états mauvais dans Cubicle et doivent représenter des états sûrs dans Why3) comme on le voit dans la Figure 6.14

```
while ( !nbsteps < maxsteps ) do
  invariant { 0 <= s.x < maxprocs }
  invariant { forall _p1 : int. 0 <= _p1 < _n /\
                    s.x = _p1 -> s.pC[_p1] <> Down }
  invariant { exists _p1 : int. 0 <= _p1 < _n /\ s.pC[_p1] <> Down }
  (* ... *)
done
```

FIGURE 6.14 – Invariants de boucles ajoutés au fichier WhyML

L'œil attentif du lecteur remarquera immédiatement que le deuxième invariant ressemble à un sous-ensemble de φ''_2 vue à la Section 20 :

$$\varphi''_2 : \exists p_1, p_2, p_3 . p_1 \neq p_2 \neq p_3 \ \wedge\ X = p_3 \ \wedge\ PC[p_1] = PC_3 \ \wedge\ PC[p_2] = PC_3 \ \wedge\ PC[p_3] = Down$$

a été transformée en

$$\varphi''_2 : \exists p_1 . X = p_1 \ \wedge\ PC[p_1] = Down$$

Il s'avère que BRAB, en filtrant cette formule, permet de remarquer que certains littéraux ne changent rien au fait que cette formule caractérise des états mauvais et l'a donc simplifiée avant de la valider.

Le fichier final est fourni dans la Figure 6.15 et a été obtenu automatiquement grâce à Cubicle sans aucune intervention sur le fichier WhyML généré. Suite à ce résultat, cette méthode a été décrite dans une publication [27].

6.5 Résultats

Afin de vérifier que cette méthode fonctionnait effectivement je l'ai confrontée à d'autres exemples.

Le premier exemple, extrêmement simple, consiste à vérifier l'intégrité d'une procédure de décision toute simple (deux transitions) dans le cadre des problèmes de consensus qui ne pouvait être vérifiée par Cubicle. Le code de cette procédure est donné dans la Figure 6.16. Comme on le voit, la première transition choisit deux processus tels qu'aucun processus dans le système n'a décidé de valeur et attribue la proposition du second au premier. La deuxième se contente de donner comme valeur de décision à chaque processus n'ayant pas décidé la valeur de décision d'un processus ayant décidé.

Les deux propriétés fondamentales dans les problèmes de consensus, comme je l'ai dit dans l'introduction de ce chapitre, sont l'accord [6], spécifiant que tout processus ayant décidé doit avoir décidé la même valeur que les autres processus ayant décidé, qui est aisément vérifiée par Cubicle et l'intégrité [7], spécifiant que tout décision prise doit avoir été proposée par au moins un autre processus (impossible de créer des décisions ex nihilo), qui n'est pas vérifiée par Cubicle car intolérante, dans notre cas, aux pannes.

La difficulté provenant de cette procédure est bien évidemment la présence d'une garde globale pour la première transition mais sans aucune modification de la traduction obtenue avec Cubicle en WhyML, elle a été vérifiée immédiatement par Why3 avec Alt-Ergo 2.1.0. Le code étant fourni dans la Figure 6.17 permet surtout de remarquer l'utilisation d'une fonction externe créée automatiquement pour gérer les gardes globales des transitions ainsi que la possibilité de travailler sur plusieurs processus distincts sans problème.

Une version plus complexe du splitter avec neuf transitions contre six dans la version que l'on a étudiée précédemment ainsi que que quatre propriétés à prouver contre une seule et deux tableaux plutôt qu'un a aussi été prouvée automatiquement. Cette preuve n'a besoin que de cinq invariants issus de l'analyse d'atteignabilité arrière effectuée par Cubicle sur 3 processus et aucune intervention humaine n'est requise.

6. Que l'on peut voir comme les propriétés de sûreté de Cubicle dans le sens où elles mentionnent des états mauvais si deux processus ne sont pas d'accord sur la valeur de décision

7. Que l'on peut voir comme une propriété de vivacité car mentionnant l'ensemble des processus

```
module Splitter

use array.Array
use int.Int
use ref.Refint
use random.Random

type proc = int

scope import State
  type state = PC0 | PC1 | PC2 | PC3 | Stop |
    Down | Right

  let (=) (a b: state): bool
    ensures {result <-> a = b}
    = match a, b with
      | PC0, PC0 | PC1, PC1 | PC2, PC2 |
        PC3, PC3 | Stop, Stop | Down, Down |
        Right, Right -> true
      | _ -> false
  end
end

val k_random (k:int) (n:int) : (result:array
    int)
    requires { 0 <= k }
    requires { k <= n }
    ensures { length result = k }
    ensures { forall i j:int. 0 <= i < n /\
        0 <= j < n /\ i <> j ->
            result[i] <> result[j] }
    ensures { forall i:int. 0 <= i < n ->
        0 <= result[i] < n }

val coin () : bool

type system = {
  mutable x : proc;
  mutable y : bool;
  pc : array state;
}

let splitter (maxprocs : int) (maxsteps :
    int) : system
  requires { 0 < maxprocs }
  ensures { exists _p1 : int. 0 <= _p1 <
    maxprocs /\ result.pc[_p1] <> Down }
  =
    let s = {
      y = false;
      pc = Array.make maxprocs PC0;
      x = Random.random_int maxprocs;
    } in

    assert {s.pc[0] = PC0};

    let nbsteps = ref 0 in
    while ( !nbsteps < maxsteps ) do
      variant { maxsteps - !nbsteps }
      invariant { 0 <= s.x < maxprocs }
      invariant { forall _p1 : int.
        0 <= _p1 < maxprocs /\ s.x = _p1 ->
            s.pc[_p1] <> Down }
      invariant { exists _p1 : int.
        0 <= _p1 < maxprocs /\
            s.pc[_p1] <> Down }

      incr nbsteps;

      let procs = k_random 1 maxprocs in
      let _p0 = procs[0] in

      if coin () && s.pc[_p0] = PC0
      then begin
        label Spl0 in
        s.x <- _p0;
        s.pc[_p0] <- PC1;
      end

      else if coin () && s.pc[_p0] = PC2
      then begin
        label Spl2 in
        s.y <- true;
        s.pc[_p0] <- PC3;
      end

      else if coin () && s.y &&
            s.pc[_p0] = PC1
      then begin
        label Spl1a in
        s.pc[_p0] <- Right;
      end

      else if coin () && not s.y &&
            s.pc[_p0] = PC1
      then begin
        label Spl1b in
        s.pc[_p0] <- PC2;
      end

      else if coin () && s.x = _p0 &&
            s.pc[_p0] = PC3
      then begin
        label Spl4stop in
        s.pc[_p0] <- Stop;
      end

      else if coin () && s.x <> _p0 &&
            s.pc[_p0] = PC3
      then begin
        label Spl4Down in
        s.pc[_p0] <- Down;
      end

    done;
    s
  end
```

FIGURE 6.15 – Fichier WhyML correspondant au splitter

```
type state = Ok | Error

array Proposed[proc] : int
array Hasdecided[proc] : bool
array Decision[proc] : int
var State : state

init(i) { Hasdecided[i] = False && State = Ok }

unsafe(i j) { Hasdecided[i] = True && Hasdecided[j] = True && Decision[i] <> Decision[j] }

universal_unsafe(i, j) { Hasdecided[i] = True && Decision[i] <> Proposed[i] &&
                         Decision[i] <> Proposed[j] }

transition decide_first(i k)
requires { Hasdecided[i] = False && Hasdecided[k] = False &&
           forall_other j. (Hasdecided[j] = False) }
{
  Hasdecided[i] := True;
  Decision[i] := Proposed[k];
}

transition decide(i j)
requires { Hasdecided[i] = False && Hasdecided[j] = True }
{
  Hasdecided[i] := True;
  Decision[i] := Decision[j];
}
```

FIGURE 6.16 – Procédure de décision naïve en Cubicle

```
module Tentative_validity

use array.Array
use int.Int
use ref.Refint
use random.Random

scope import State
  type state = Ok | Error

  let (=) (a b: state): bool
    ensures {result <-> a = b}
    = match a, b with
      | Ok, Ok | Error, Error -> true
      | _ -> false
  end
end

val coin () : bool
val k_random (k:int) (n:int) : (result:array
    int)
  requires { 0 <= k }
  requires { k <= n }
  ensures { length result = k }
  ensures { forall i j:int. 0 <= i < n /\ 0
    <= j < n /\ i <> j -> result[i] <>
    result[j] }
  ensures { forall i:int. 0 <= i < n -> 0 <=
    result[i] < n }

type proc = int

type system = {
  mutable state : state;proposed : array int;
  hasdecided : array bool;
  decision : array int;
}

let tentative_validity (maxprocs : int)
    (maxsteps : int) : system
  requires { 1 < maxprocs }
  =
    let s = {
      state = Ok;
      hasdecided = Array.make maxprocs false;
      proposed = Array.make maxprocs
    (Random.random_int maxprocs);
      decision = Array.make maxprocs
    (Random.random_int maxprocs);
    } in

    assert {s.hasdecided[0] = false};
    assert {s.hasdecided[1] = false};

    let nbsteps = ref 0 in
    val forall_other_decide_first0 (_p0 _p1:
    proc) : (result : bool)
      requires { 0 <= _p0 < maxprocs && 0 <=
    _p1 < maxprocs }
      ensures { result = True <-> (forall
    _p1':proc.
                0 <= _p1' < maxprocs /\ _p0
    <> _p1' /\ _p1 <> _p1' ->
                not s.hasdecided[_p1']) }
    in

    while ( !nbsteps < maxsteps ) do
      variant { maxsteps - !nbsteps }

      invariant { forall _p1 _p2 : int. 0 <=
    _p1 < maxprocs /\
                    0 <= _p2 < maxprocs /\
                    _p1 <> _p2 /\
    s.hasdecided[_p1] /\ s.hasdecided[_p2] ->
                                s.decision[_p1]
    = s.decision[_p2] }
      invariant { forall _p1 _p2 : int. 0 <=
    _p1 < maxprocs /\
                    0 <= _p2 < maxprocs /\
                    _p1 <> _p2 /\
    s.hasdecided[_p1] /\
                                s.decision[_p1]
    <> s.proposed[_p1] ->
                                s.decision[_p1]
    <> s.proposed[_p2] }
      invariant { forall i j : int. 0 <= i <
    maxprocs /\ 0 <= j < maxprocs /\
                    i <> j /\ s.hasdecided[i]
    /\ s.hasdecided[j] ->
                           s.decision[i] =
    s.decision[j] }

      incr nbsteps;

      let procs = k_random 2 maxprocs in
      let _p0 = procs[0] in
      let _p1 = procs[1] in
      assert { _p0 <> _p1 };

      (*decide*)
      if coin () && not s.hasdecided[_p0] &&
    s.hasdecided[_p1]
      then begin
        label Decide in
        s.hasdecided[_p0] <- true;
        s.decision[_p0] <- s.decision[_p1];
      end

      (*decide_first*)
      else if coin () && not
    s.hasdecided[_p0] && not
    s.hasdecided[_p1] &&
        forall_other_decide_first0 (_p0 :
    proc)(_p1 : proc)
      then begin
        label Decide_first in
        s.hasdecided[_p0] <- true;
        s.decision[_p0] <- s.proposed[_p1];
      end

    done;
    s
  end
```

Chapitre 7

Conclusion et perspectives

7.1 Contributions

Nous voici arrivés à la fin de ce document. Au fur et à mesure de sa rédaction je me suis rendu compte à quel point travailler sur ce logiciel déjà implémenté m'a permis de me concentrer sur ce que je préfère : faire communiquer les choses entre elles. Ma première contribution a donc consisté à adapter un algorithme issu de la vérification de matériel en implémentatant d'un nouvel algorithme d'atteignabilité mélangeant les avantages de l'atteignabilité avant et ceux de l'atteignabilité arrière dans le Chapitre 4. Cette contribution a surtout servi à démontrer le potentiel de ce genre d'algorithme et son aspect tout à fait compétitif avec les méthodes actuelles. Ce chapitre a donc servi à montrer que, en accord avec ce que d'autres ont fait, des algorithmes implémentés à l'origine pour servir à résoudre des problèmes SAT ou des circuits matériels peuvent être modifiés légèrement afin d'être adaptés à des problèmes différents tels que, dans notre cas, la vérification automatique de systèmes de transitions paramétrés. Dans le Chapitre 5 j'ai décidé de voir ce qu'il était possible de faire avec les développements plus ou moins récents de l'IA et notamment avec l'algorithme des k-moyennes dans la Section 5.3 permettant de regrouper rapidement des nœuds partageant suffisamment de caractéristiques communes. L'implémentation de cet algorithme dans Cubicle a permis de prouver des protocoles beaucoup plus rapidement et en consommant dans certains cas beaucoup moins d'espace mémoire. J'ai donc poussé les expérimentations plus loin notamment en regardant du côté de la vérification modulaire qui, grossièrement, vérifie des sous-parties d'un grand ensemble avant de vérifier l'ensemble au complet. J'ai donc commencé par travailler sur une solution automatique mais très naïve qui malgré tout a fourni des résultats, comme on l'a vu dans la sous-section 5.6.1, tout à fait encourageants. Poussé par ces résultats j'ai donc cherché à contraindre ces copies par des techniques de vérification modulaires et ai obtenu, au final, des résultats équivalents aux meilleurs résultats obtenus sur Cubicle en occupant beaucoup moins d'espace mémoire.

La plus grosse partie de ma thèse a ensuite été occupée par ce que j'ai décrit dans le Chapitre 6, c'est à dire la vérification automatique d'algorithmes distribués. Après m'être attaqué à l'efficacité de Cubicle pour prouver de plus gros problèmes ou les prouver plus vite je m'attaquais donc à l'expressivité de Cubicle pour prouver de nouveaux types de problèmes. Il m'a donc fallu travailler dans un nouveau fragment hautement indécidable et donc très difficilement automatisable. Et c'est encore une fois en faisant communiquer les choses qu'une solution a émergé, la transformation d'un modèle infini en mo-

dèle fini puis à nouveau en modèle infini afin de vérifier ces nouveaux problèmes à l'aide d'outils de vérification déductive.

7.2 Perspectives

Comme je l'ai déjà exprimé, j'ai un grand regret, c'est le fait de ne pas avoir pu automatiser la vérification modulaire dans Cubicle. Il serait tout à fait intéressant de pouvoir ajouter à Cubicle la possibilité de décrire les systèmes non pas comme seulement des transitions permettant de mettre à jour des tableaux mais comme des modules ayant des transitions de communication entre eux et des transitions internes indépendantes de ce que font les autres modules. Il semble en effet que pour franchir une étape supplémentaire dans la complexité des systèmes étudiés et notamment pour les protocoles dits hiérarchiques il faille non pas se concentrer sur l'optimisation de l'algorithme déjà existant mais sur le fait de pouvoir diviser pour mieux vérifier. Les protocoles hiérarchiques faisant intervenir plusieurs protocoles pour chaque type de cache composant la dite hiérarchie (L1, L2, ...), l'approche modulaire leur semble parfaitement adaptée. Cette problématique me semble tout à fait intéressante et offre de multiples possibilités tout à fait passionnantes.

Du point de vue de l'expressivité, en ayant montré qu'il était possible d'utiliser Cubicle pour générer des invariants utilisables par Why3 pour prouver des systèmes plus expressifs (mais toujours exprimables dans Cubicle, naturellement), j'ai montré que l'utilisation de Cubicle non plus comme une solution en soi mais comme un outil pour alléger la recherche d'invariants dans la perspective de prouver des systèmes au moyen de la vérification déductive n'était pas du tout aberrante. Conséquemment, je pense qu'ajouter des constructions dans Cubicle qui ne peuvent être prouvées automatiquement dans un cadre paramétré en utilisant Why3 comme prouveur externe permettrait à Cubicle de franchir une nouvelle étape.

Bibliographie

[1] P. A. Abdulla, K. Cerans, B. Jonsson, and Y.-K. Tsay. General decidability theorems for infinite-state systems. In *LICS*, pages 313–321. IEEE, 1996.

[2] P. A. Abdulla, G. Delzanno, N. B. Henda, and A. Rezine. Regular model checking without transducers. In *TACAS*. Springer, 2007.

[3] P. A. Abdulla, G. Delzanno, and A. Rezine. Parameterized verification of infinite-state processes with global conditions. In *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, pages 145–157, 2007.

[4] P. A. Abdulla, G. Delzanno, and A. Rezine. Parameterized verification of infinite-state processes with global conditions. In *CAV 2007, Berlin, Germany, July 3-7*, 2007.

[5] P. A. Abdulla, G. Delzanno, and A. Rezine. Approximated parameterized verification of infinite-state processes with global conditions. *Formal Methods in System Design*, 34(2) :126–156, 2009.

[6] P. A. Abdulla, N. B. Henda, G. Delzanno, and A. Rezine. Handling parameterized systems with non-atomic global conditions. In *VMCAI 2008, San Francisco, CA, USA, Jan 7-9*, 2008.

[7] F. Alberti, S. Ghilardi, E. Pagani, S. Ranise, and G. P. Rossi. Universal guards, relativization of quantifiers, and failure models in model checking modulo theories. *JSAT*, 8(1/2) :29–61, 2012.

[8] K. R. Apt and D. Kozen. Limits for automatic verification of finite-state concurrent systems. *Inf. Process. Lett.*, 22(6) :307–309, 1986.

[9] D. Arthur and S. Vassilvitskii. K-means++ : the advantages of careful seeding. In *In Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms*, 2007.

[10] A. Blanchard, F. Loulergue, and N. Kosmatov. From concurrent programs to simulating sequential programs : Correctness of a transformation. In *Proceedings Fifth International Workshop on Verification and Program Transformation, VPT@ETAPS 2017, Uppsala, Sweden, 29th April 2017.*, pages 109–123, 2017.

[11] A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract regular tree model checking. *Electr. Notes Theor. Comput. Sci.*, 149(1) :37–48, 2006.

[12] A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract regular (tree) model checking. *STTT*, 14(2) :167–191, 2012.

[13] A. Bouajjani, P. Habermehl, and T. Vojnar. Abstract regular model checking. In R. Alur and D. A. Peled, editors, *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings*, volume 3114 of *Lecture Notes in Computer Science*, pages 372–386. Springer, 2004.

[14] A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular model checking. In E. A. Emerson and A. P. Sistla, editors, *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, volume 1855 of *Lecture Notes in Computer Science*, pages 403–418. Springer, 2000.

[15] A. R. Bradley. Sat-based model checking without unrolling. In *VMCAI*, pages 70–87, 2011.

[16] M. C. Browne, E. M. Clarke, and O. Grumberg. Reasoning about networks with many identical finite state processes. *Inf. Comput.*, 81(1) :13–31, 1989.

[17] A. Cimatti and A. Griggio. Software model checking via IC3. In *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, pages 277–293, 2012.

[18] A. Cimatti, A. Griggio, S. Mover, and S. Tonetta. IC3 modulo theories via implicit predicate abstraction. *CoRR*, abs/1310.6847, 2013.

[19] A. Cimatti, A. Griggio, S. Mover, and S. Tonetta. Parameter synthesis with IC3. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 165–168, 2013.

[20] A. Cimatti, A. Griggio, S. Mover, and S. Tonetta. IC3 modulo theories via implicit predicate abstraction. In *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, pages 46–61, 2014.

[21] A. Cimatti, A. Griggio, S. Mover, and S. Tonetta. Infinite-state invariant checking with IC3 and predicate abstraction. *Formal Methods in System Design*, 49(3) :190–218, 2016.

[22] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *25 Years of Model Checking - History, Achievements, Perspectives*, pages 196–215, 2008.

[23] E. M. Clarke, O. Grumberg, and M. C. Browne. Reasoning about networks with many identical finite-state processes. In *PODC'86*. ACM, 1986.

[24] S. Conchon, A. Goel, S. Krstic, R. Majumdar, and M. Roux. Far-cubicle - A new reachability algorithm for cubicle. In D. Stewart and G. Weissenbacher, editors, *2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017*, pages 172–175. IEEE, 2017.

[25] S. Conchon, A. Goel, S. Krstić, A. Mebsout, and F. Zaïdi. Cubicle : A Parallel SMT-based Model Checker for Parameterized Systems. In *CAV 2012, Berkeley, CA, USA, July 7-13*, 2012.

[26] S. Conchon, A. Goel, S. Krstic, A. Mebsout, and F. Zaïdi. Invariants for finite instances and beyond. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 61–68, 2013.

[27] S. Conchon and M. Roux. Reasoning about universal cubes in MCMT. In Y. A. Ameur and S. Qin, editors, *Formal Methods and Software Engineering - 21st International Conference on Formal Engineering Methods, ICFEM 2019, Shenzhen, China, November 5-9, 2019, Proceedings*, volume 11852 of *Lecture Notes in Computer Science*, pages 270–285. Springer, 2019.

[28] N. Eén, A. Mishchenko, and R. K. Brayton. Efficient implementation of property directed reachability. In *International Conference on Formal Methods in Computer-Aided Design, FMCAD '11, Austin, TX, USA, October 30 - November 02, 2011*, pages 125–134, 2011.

[29] E. A. Emerson and E. M. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In *Automata, Languages and Programming, 7th Colloquium, Noordweijkerhout, The Netherlands, July 14-18, 1980, Proceedings*, pages 169–181, 1980.

[30] P. Garg, C. Löding, P. Madhusudan, and D. Neider. ICE : A robust framework for learning invariants. In A. Biere and R. Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 69–87. Springer, 2014.

[31] P. Garg, D. Neider, P. Madhusudan, and D. Roth. Learning invariants using decision trees and implication counterexamples. In R. Bodík and R. Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 499–512. ACM, 2016.

[32] S. M. German and A. P. Sistla. Reasoning about systems with many processes. *J. ACM*, 39(3) :675–735, 1992.

[33] S. Ghilardi, E. Nicolini, S. Ranise, and D. Zucchelli. Towards SMT model checking of array-based systems. In *IJCAR 2008, Sydney, Australia, Aug 12-15*, 2008.

[34] S. Ghilardi and S. Ranise. Backward reachability of array-based systems by SMT solving : Termination and invariant synthesis. *LMCS*, 6(4), 2010.

[35] S. Ghilardi and S. Ranise. MCMT : A model checker modulo theories. In *Automated Reasoning, 5th International Joint Conference, IJCAR 2010, Edinburgh, UK, July 16-19, 2010. Proceedings*, pages 22–29, 2010.

[36] A. Gmeiner, I. Konnov, U. Schmid, H. Veith, and J. Widder. Tutorial on parameterized model checking of fault-tolerant distributed algorithms. In *Formal Methods for Executable Software Models - 14th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2014, Bertinoro, Italy, June 16-20, 2014, Advanced Lectures*, pages 122–171, 2014.

[37] R. P. Goldman, M. W. Boldt, and D. J. Musliner. Employing ai techniques in probabilistic model checking. *MOCHAP*, 2014.

[38] P. Z. Google. Reading privileged memory with a side-channel. https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html, 2018. Lien ; Accédé le 28-10-2019.

[39] G. Hamerly and C. Elkan. Learning the k in k-means. In *Proceedings of the 16th International Conference on Neural Information Processing Systems*, NIPS'03, pages 281–288, Cambridge, MA, USA, 2003. MIT Press.

[40] K. Hoder and N. Bjørner. Generalized property directed reachability. In *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference*, pages 157–171, 2012.

[41] Intel. Intel xeon processor 5500 series. https://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/xeon-5500-specification-update.pdf, 2015. Lien ; Accédé le 28-10-2019.

[42] N. Jansen, J. Katoen, P. Kohli, and J. Kretinsky. Machine learning and model checking join forces (dagstuhl seminar 18121). *Dagstuhl Reports*, 8(3) :74–93, 2018.

[43] I. Konnov and J. Widder. Bymc : Byzantine model checker. In *Leveraging Applications of Formal Methods, Verification and Validation. Distributed Systems - 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part III*, pages 327–342, 2018.

[44] M. Z. Kwiatkowska and D. Parker. Advances in probabilistic model checking. In T. Nipkow, O. Grumberg, and B. Hauptmann, editors, *Software Safety and Security - Tools for Analysis and Verification*, volume 33 of *NATO Science for Peace and Security Series - D : Information and Communication Security*, pages 126–151. IOS Press, 2012.

[45] L. Lamport. A fast mutual exclusion algorithm. *ACM Trans. Comput. Syst.*, 5(1) :1–11, Jan. 1987.

[46] L. Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3) :872–923, 1994.

[47] J. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1 : Statistics*, pages 281–297, Berkeley, Calif., 1967. University of California Press.

[48] K. L. McMillan. Lazy abstraction with interpolants. *CAV*, pages 123–126, 2006.

[49] K. L. McMillan. Quantified invariant generation using an interpolating saturation prover. In *TACAS*, pages 413–427. Springer, 2008.

[50] K. L. McMillan. Lazy annotation revisited. In *Computer Aided Verification - 26th International Conference, CAV 2014*, pages 243–259, 2014.

[51] A. Mebsout. *Inférence d'invariants pour le model checking de systèmes paramétrés. (Invariants inference for model checking of parameterized systems)*. PhD thesis, University of Paris-Sud, Orsay, France, 2014.

[52] M. Moir and J. H. Anderson. Wait-free algorithms for fast, long-lived renaming. *Sci. Comput. Program.*, 25(1) :1–39, Oct. 1995.

[53] M. Nilsson. *Regular Model Checking*. PhD thesis, University of Uppsala, Uppsala, Suède, 2005.

[54] S. Owicki and L. Lamport. Proving liveness properties of concurrent programs. *ACM Trans. Program. Lang. Syst.*, 4(3) :455–495, July 1982.

[55] Site web du projet anr pardi. http://pardi.enseeiht.fr/index.html, 2016. Lien ; Accédé le 28-10-2019.

[56] D. Pelleg and A. Moore. X-means : Extending k-means with efficient estimation of the number of clusters. In *In Proceedings of the 17th International Conf. on Machine Learning*, pages 727–734. Morgan Kaufmann, 2000.

[57] Site de pfs. http://www.it.uu.se/research/docs/fm/apv/tools/pfs/, 2007. Lien ; Accédé le 29-10-2019.

[58] A. Pnueli. The temporal semantics of concurrent programs. In *Proceedings of the International Symposium on Semantics of Concurrent Computation*, pages 1–20, London, UK, UK, 1979. Springer-Verlag.

[59] J. Queille. The CESAR system : An aided design and certification system. In *Proceedings of the 2nd International Conference on Distributed Computing Systems, Paris, France, 1981*, pages 149–161, 1981.

[60] J. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *International Symposium on Programming, 5th Colloquium, Torino, Italy, April 6-8, 1982, Proceedings*, pages 337–351, 1982.

[61] Reason - what & why. https://reasonml.github.io/docs/en/what-and-why, 2016. Lien ; Accédé le 02-11-2019.

[62] Rust programming language. https://www.rust-lang.org/, 2010. Lien ; Accédé le 02-11-2019.

[63] D. Silver, D. Hassabis, and G. DeepMind. Alphago : Mastering the ancient game of go with machine learning. https://ai.googleblog.com/2016/01/alphago-mastering-ancient-game-of-go.html, 2016. Lien ; Accédé le 29-10-2019.

[64] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis. Mastering the game of go without human knowledge. *Nature*, 550 :354–, Oct. 2017.

[65] M. R. Tuttle and A. Goel. Protocol proof checking simplified with SMT. In *11th IEEE International Symposium on Network Computing and Applications, NCA 2012, Cambridge, MA, USA, August 23-25, 2012*, pages 195–202, 2012.

[66] Site de undip. http://www.it.uu.se/research/docs/fm/apv/tools/undip/, 2007. Lien ; Accédé le 29-10-2019.

[67] Y. Vizel, A. Gurfinkel, S. Shoham, and S. Malik. IC3 - flipping the E in ICE. In A. Bouajjani and D. Monniaux, editors, *Verification, Model Checking, and Abstract Interpretation - 18th International Conference, VMCAI 2017, Paris, France, January 15-17, 2017, Proceedings*, volume 10145 of *Lecture Notes in Computer Science*, pages 521–538. Springer, 2017.

[68] Site de why3. http://why3.lri.fr/, 2012. Lien ; Accédé le 29-10-2019.

université PARIS-SACLAY
ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre : Extensions de l'algorithme d'atteignabilité arrière dans le cadre de la vérification de modèles modulo théories

Mots clés : Vérification de modèles, Vérification de modèles Modulo Theories, Vérification Déductive, Satisfiabilité Modulo Théories, Systèmes Distribués, Apprentissage non supervisé

Résumé : Cette thèse se propose de présenter plusieurs extensions ayant été ajoutées au vérificateur de modèles Cubicle.
Cubicle est un logiciel permettant de vérifier automatiquement la sûreté de systèmes paramétrés au moyen de techniques de vérification de modèles modulo théories.
La première contribution apportée par cette thèse consiste en l'implémentation d'un nouvel algorithme d'atteignabilité baptisé FAR (pour Forward Abstracted Reachabilty). FAR est un algorithme faisant intervenir à la fois des techniques de l'analyse d'atteignabilité en arrière déjà implémentée dans Cubicle ainsi que des techniques d'analyse d'atteignabilité en avant.
La seconde contribution est constituée de multiples ajouts inspirés de méthodes de l'intelligence artificielle afin d'améliorer la génération automatique d'invariants de Cubicle.
Enfin, la dernière contribution a permis d'augmenter l'expressivité de Cubicle afin de prouver des propriétés faisant intervenir des quantificateurs universels. Cette contribution a été mise en œuvre en associant Cubicle à Why3, une plateforme de vérification déductive.

Title : Extensions of the backward reachability algorithm in the model checking modulo theories framework

Keywords : Model Checking, Model Checking Modulo Theories, Deductive Verification, Satisfiability Modulo Theories, Distributed Systems, Unsupervised learning

Abstract : This thesis proposes to present several extensions that have been added to the Cubicle model checker.
Cubicle is a software allowing to automatically check the safety of parameterized systems using model checking modulo theory techniques.
The first contribution made by this thesis consists in the implementation of a new reachability algorithm called FAR (for Forward Abstracted Reachabilty). FAR is an algorithm involving both backward reachability analysis techniques already implemented in Cubicle as well as forward reachability analysis techniques.
The second contribution consists of multiple additions inspired by artificial intelligence methods to improve the automatic generation of Cubicle invariants.
Finally, the last contribution has increased Cubicle's expressiveness in order to prove properties involving universal quantifiers. This contribution was implemented by associating Cubicle with Why3, a deductive verification platform.

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France