Analyse de systèmes modulaires à l'aide de techniques d'abstractions hiérarchiques.

Yves-Stan Le Cornec

NNT : 2016SACLE019

THÈSE DE DOCTORAT
DE
L'UNIVERSITÉ PARIS-SACLAY
PRÉPARÉE A
L'UNIVERSITÉ D'EVRY VAL D'ESSONNE

ECOLE DOCTORALE N° 580
Sciences et technologies de l'information et de la communication

Spécialité Informatique

Par

M. Yves-Stan Le Cornec

Analyse de systèmes modulaires à l'aide de techniques d'abstractions hiérarchiques.

Thèse présentée et soutenue à Evry, le 11 juillet 2016 :

Composition du Jury :

M. Bouajjani Ahmed	Professeur de l'Université Paris Diderot	Président
Mme Bérard Béatrice	Professeur de l'Université P & M. Curie.	Rapporteur
M. Pradat-Peyre Jean-François	Professeur de l'Université Paris Ouest Nanterre la Défense	Rapporteur
M. Bouajjani Ahmed	Professeur de l'Université Paris Diderot	Examinateur
Mme Klaudel Hanna	Professeur de l'Université Paris-Saclay	Examinatrice
M. Pommereau Franck	Professeur de l'Université Paris-Saclay	Directeur de thèse

Sommaire

Chapitre 1

Introduction

Les outils logiciels et automatiques sont essentiels au fonctionnement de la société actuelle. Ils sont notamment responsables d'un nombre toujours croissant de services critiques, pour lesquels les conséquences d'une défaillance du système seraient désastreuses. En outre, les tailles et complexités de ces systèmes sont de plus en plus importantes et ils sont donc plus susceptibles de comporter des erreurs et des comportements non désirés. Il est par conséquent nécessaire d'être capable de raisonner formellement sur ces outils et d'assurer leur correction, c'est à ce besoin que répondent les méthodes formelles.

Parmi ces différentes méthodes, le model-checking [52, 108] occupe une place majeure comme en témoignent une riche littérature [108], une utilisation industrielle pour la vérification de certains systèmes critiques [17, 31] et le prix Turing décerné à Joseph Sifakis pour avoir posé les bases de cette technique (en parallèle des travaux de Edmund Clarke et Allen Emerson).

Un algorithme de model-checking considère un modèle du système à vérifier ainsi qu'une propriété de correction, tous deux exprimés dans des langages formels, et détermine automatiquement si le modèle vérifie ou non la propriété.

La principale limite de ces algorithmes est un problème d'explosion combinatoire. En effet, le principe de base du model-checking consiste à explorer les différents états possibles du système, qui peuvent être très nombreux en pratique. Par exemple, le simple fait d'ajouter une variable booléenne à un programme a pour conséquence de doubler l'ensemble des états possibles de ce programme (appelé espace d'états). Plusieurs techniques ont été développées dans le but de combattre cette explosion de l'espace d'états. Par exemple, les méthodes d'exploration à la volée permettent de n'explorer que les états

nécessaires en fonction des besoins de l'algorithme de vérification [50, 70, 82], et le model checking symbolique permet de représenter de façon concise de grands ensembles d'états [72, 20]. Il existe également différentes approches consistant à réduire l'espace d'état comme les réductions par ordre partiel [105, 55, 43] et par symétrie [29, 40] qui éliminent certains comportements redondants du système. On trouve également les réductions par abstraction consistent à réduire la taille des modèles considérées en décidant d'oublier leurs détails que l'on sait non pertinents pour la propriété à vérifier, on travaille alors sur des abstractions de ces modèles [30, 80, 74].

Il est parfois également possible d'exploiter la nature modulaire de certains systèmes afin d'améliorer l'efficacité des algorithmes : en considérant un modèle partie par partie (appelées modules), on évite de le manipuler dans sa globalité. L'abstraction hiérarchique [101, 30, 35] permet par exemple de calculer une abstraction du système de manière efficace, en utilisant une approche modulaire. Au lieu de combiner directement les modules du système pour construire le système complet, on peut les réduire au préalable : la composition des modules abstraits est alors une abstraction de la composition des modules. On peut même réduire les compositions intermédiaires à toute étape de la construction du système de façon à toujours manipuler des modèles réduits. Cette approche est également compatible avec de l'analyse partielle, qui consiste à analyser les différentes sous-parties du système au cours de la composition pour essayer de déterminer la valeur de vérité globale de la formule (ce qui peut parfois servir à guider la construction incrémentale du système). On note que les opérations d'abstraction classiques sont génériques dans le sens où elles préservent tout un ensemble de propriétés du système.

Dans ce mémoire on s'intéresse principalement à définir des opérations d'abstractions qui sont compatibles avec l'approche par abstraction hiérarchique et spécifiques à la propriété à vérifier (exprimée sous la forme d'une formule de μ-calcul).

Le manuscrit est organisé de la façon suivante, où les contributions principales sont les chapitres 5, 4 et 6 et 8.

Le chapitre 2 comporte l'état de l'art. On commence par présenter une vue d'ensemble des différents buts et techniques de modélisation et de vérification. On s'intéresse en particulier au model-checking et aux différentes techniques permettant de combattre l'explosion combinatoire. Parmi ces techniques on détaille les méthodes utilisant des réductions hiérarchiques qui est le cadre plus précis de la thèse.

Le chapitre 3 introduit les définitions formelles des formalismes de modé-

lisation et des logiques utilisés par la suite.

Le chapitre 4 présente un cas d'application des techniques de réductions hiérarchiques à un système biologique. Il s'agit d'un travail réalisé conjointement avec Mendes, Lang, Mateescu, Batt et Chaouiya [73]. Il comporte la définition des *réseaux de régulation modulaires* qui permettent de définir des systèmes biologiques composables et adaptés à une étude par abstraction hiérarchique. Ce cadre formel est ensuite appliqué au problème de l'accessibilité des états stables d'un modèle biologique de la littérature. Pour cela, on calcule de manière modulaire une abstraction du système à l'aide de la réduction SAFETY [16] qui préserve cette propriété d'accessibilité.

Le chapitre 5 définit deux nouvelles techniques permettant de réduire la taille du comportement d'une sous-partie d'un système, en préservant la valeur de vérité d'une formule de μ-calcul donnée sur le système global. Ces réductions peuvent donc être utilisées au cours d'une analyse hiérarchique. De plus, comme elles sont adaptées à une formule (quelconque mais connue au moment de la réduction) on peut espérer de meilleurs coefficients de réduction qu'avec des méthodes de réduction génériques.

Le chapitre 6 contient des expérimentations qui ont pour but de tester un prototype implémentant la première des deux réductions définie au chapitre précédent, et de constater des premiers résultats sur l'efficacité en terme de coefficients de réduction de la méthode. Ces expérimentations permettent également de fournir des données pour commencer à traiter le problème de la prédiction du coefficient de réduction ainsi que celui du meilleur ordre d'analyse pour l'abstraction hiérarchique, qui sont des problèmes difficiles et font partie des perspectives.

Le chapitre 7 est la conclusion du manuscrit.

Le chapitre 8 contient les preuves des différents résultats de la thèse.

Chapitre 2

État de l'art

Le but est ici d'exposer une vue d'ensemble du domaine de la vérification formelle afin d'y situer cette thèse. Les exemples de modèles ou de techniques de vérifications cités servent donc uniquement à illustrer certains concepts et ne constituent pas une liste exhaustive de l'existant.

2.1 Modélisation et vérification

Les modèles de calculs discrets nous permettent de décrire et d'étudier les dynamiques de différents systèmes à l'aide d'outils logiciels et mathématiques. On veut être capable de représenter les comportements de systèmes capables d'évoluer au cours du temps, c'est-à-dire de changer d'*état* en effectuant certaines *actions*. Le comportement d'une ampoule électrique qui ne grille jamais peut par exemple être modélisé par la *séquence d'exécution* suivante (elle est infinie car l'ampoule a toujours la possibilité de changer d'état).

$$\text{off} \xrightarrow{\text{allumer}} \text{on} \xrightarrow{\text{éteindre}} \text{off} \xrightarrow{\text{allumer}} \text{on} \xrightarrow{\text{éteindre}} \cdots$$

Certains systèmes peuvent avoir plusieurs actions possibles à certains points, car les résultats de certains évènements ne sont pas connus à l'avance (une commande de l'utilisateur, un appel à une fonction aléatoire, un paramètre physique comme la température ou encore un échec du système). Si l'on prend en compte le fait qu'une ampoule peut cesser de fonctionner à tout moment, il faut également considérer (en plus de la précédente) les séquences d'exécutions suivantes (qui sont elles finies) :

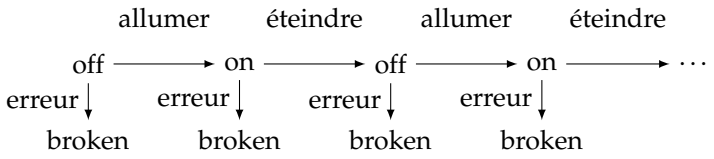
$$\text{off} \xrightarrow{\text{erreur}} \text{broken}$$

$$\text{off} \xrightarrow{\text{allumer}} \text{on} \xrightarrow{\text{erreur}} \text{broken}$$

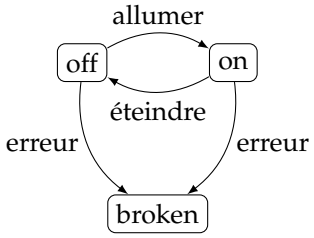
$$\text{off} \xrightarrow{\text{allumer}} \text{on} \xrightarrow{\text{éteindre}} \text{off} \xrightarrow{\text{erreur}} \text{broken}$$

$$\vdots$$

On peut représenter l'ensemble de ces exécutions sous la forme d'un *arbre d'exécution* :



On peut aussi utiliser une représentation plus compacte sous forme d'un *système de transition*, qui nous permet de ne dessiner qu'une fois les différents *états* du système (il y en a ici trois), les actions du système apparaissent sous forme de *transitions* entre ces états.

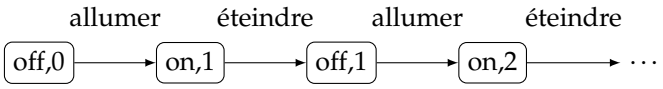


De tels modèles peuvent représenter des systèmes bien plus compliqués et relativement variés (circuits électroniques, programmes informatiques, protocoles de sécurité ou encore organismes biologiques) et sont utilisés dans plusieurs cadres :

- lors du développement d'un logiciel ou d'un circuit électronique, commencer par définir un modèle formel du système permet une première analyse en amont de l'implémentation qui peut mettre en avant certaines erreurs de conception et éviter du travail inutile ;
- dans le cadre d'une démarche scientifique où on cherche à comprendre et prédire un système existant ;

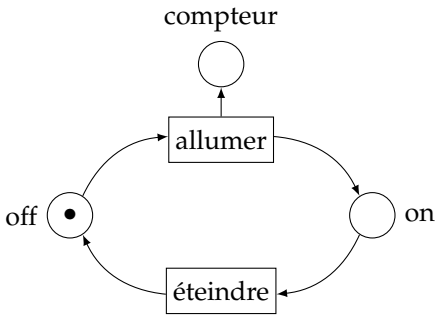
- pour améliorer notre confiance dans un système. On peut par exemple chercher à certifier qu'un système est correct sous réserve que le modèle étudié soit bien conforme à la réalité. De tels résultats sont essentiels dans le cas des systèmes dit critiques, où un échec du système peut avoir des conséquences désastreuses.

En pratique, on ne décrit pas directement le modèle sous la forme d'un système de transition, mais on utilise un *langage de modélisation* adapté au problème, qui permet de modéliser le système de façon plus concise. En effet, le nombre d'états possibles des systèmes auxquels on s'intéresse est souvent très élevé, et peut même être infini. C'est par exemple le cas du modèle suivant, qui retient le nombre de fois où une lampe a été allumée (dans le cas sans erreurs).



2.1.1 Les langages de modélisation

Les réseaux de Petri [83, 51, 84] sont un exemple d'un tel langage, et on pourrait utiliser le réseau suivant pour représenter de façon finie ce précédent système. Dans ce réseau, pour exécuter la *transition* « allumer », on doit consommer un *jeton* dans la *place* « off » et on en produit un dans chacune des autres places. Pour exécuter « éteindre », on consomme un jeton dans la place « on » et on en produit un dans la place « off ». Le nombre de jetons présents dans la place compteur reste toujours égal au nombre d'exécutions de l'action « allumer » et le réseau de Petri décrit bien le LTS précédent.



Les différents langages de modélisation peuvent être vus comme des outils permettant de représenter de façon compacte différents système de tran-

sitions. L'*expressivité* d'un langage est alors l'ensemble des systèmes de transitions qu'il permet de représenter.

Les langages de programmation ont pour objectifs de décrire de façon précise le comportement d'un ordinateur, et en effet ils possèdent souvent une *sémantique opérationnelle* permettant de définir un système de transition [77, 49]. On pourrait par exemple programmer l'exemple précèdent ainsi :

```
int etat=0;
int compteur=0;
while(1){
  compteur++;
  etat=!etat;
}
```

Les langages de programmation sont cependant très expressifs et produisent des systèmes très détaillés, ce qui les rend difficiles à utiliser en tant que langage de modélisation. Il est cependant possible d'analyser le code source d'un programme existant afin d'en tirer des conclusions sur son comportement [34, 7]. En règle générale on essaie de choisir un langage de modélisation le plus simple possible, tout en restant suffisamment expressif pour capturer les caractéristiques essentielles du système à modéliser (qui sont en lien direct avec les propriétés que l'on cherche à vérifier sur ce système).

On présente maintenant quelques attributs des langages de modélisation qu'il faut considérer lors du choix du langage.

2.1.1.1 Niveau d'abstraction

Ces langages ont un certain niveau d'abstraction, c'est à dire qu'il leur est impossible de décrire certains détails du système. Par exemple, un langage possédant uniquement un nombre fini de variables booléennes pourra permettre de décrire le système de l'ampoule, mais pas d'y ajouter le compteur infini. On remarque néanmoins que ce modèle est une *abstraction* du modèle avec compteur : si on décide d'ignorer le compteur, ils ont exactement le même arbre d'exécution.

Si la valeur du compteur est importante (par exemple si on veut détecter des dépassements d'entiers), le premier système est trop abstrait et non pertinent. Dans le cas contraire (par exemple si on teste l'absence d'états bloquants appelés *deadlocks*), on peut choisir de façon équivalente un des deux modèles.

Le modèle (et le langage) qui est le plus abstrait est dans ce cas le meilleur choix pour plusieurs raisons.

- Il est plus facile de décrire un système de façon abstraite car on n'a pas besoin de spécifier tous les détails de l'implémentation (ce qu'il est d'ailleur impossible de faire sur un système physique comme un organisme biologique).
- Toutes les implémentations cohérentes avec le modèle étudié, vérifieront ces propriétés de haut niveau, on n'a pas besoin de refaire l'analyse à chaque fois. Dans le cas d'un logiciel, si il y a une erreur quelque part, c'est quelque chose de spécifique à l'implémentation, et pas une erreur de conception.
- Plus un modèle est abstrait, plus sa taille est faible. Il peut même être fini alors qu'un modèle plus détaillé sera de taille infinie (comme c'est le cas pour l'exemple de l'ampoule). Cela permet souvent d'améliorer l'efficacité des méthodes d'analyses, et peut même rendre réalisables certaines méthodes (par exemple des méthodes uniquement applicables dans le cas fini).

2.1.1.2 Opérateurs adaptés

Les langages de modélisation disposent d'opérateurs de description permettant d'exprimer différents concepts qui peuvent être fondamentaux ou au contraire spécifiques à un domaine. Ces opérateurs peuvent augmenter l'expressivité du langage (les systèmes qu'il peut décrire) et sa concision (la facilité à décrire ces systèmes).

Concurrence Les systèmes concurrents sont constitués de plusieurs parties pouvant évoluer indépendamment mais qui collaborent pour effectuer une tâche donnée [110, 74]. Ils existent à toutes les échelles, des circuits électroniques aux algorithmes distribués sur plusieurs machines communiquant par internet, en passant par les programmes constitués de plusieurs threads utilisant plusieurs coeurs d'un processeur.

Raisonner sur ces systèmes est difficile et souvent non intuitif, et de nouveaux types de questions peuvent provenir de leur nature concurrente. Deux processus essaient-t-ils d'écrire dans le même fichier au même moment ? Un serveur web attend-t-il indéfiniment un message qui ne sera jamais envoyé ? Cela rend critique le besoin de les modéliser et les étudier de manière formelle et assistée par ordinateur.

Différents langages de modélisation disposent pour cette raison d'outils permettant de spécifier facilement de tels systèmes concurrents, ainsi que les techniques qu'ils utilisent pour communiquer comme l'accès à des ressources partagées ou l'envoi de messages.

Les réseaux de Petri sont par exemple des systèmes naturellement concurrents car chaque transition peut s'exécuter dès qu'elle a suffisamment de ressources. Les différentes transitions interagissent cependant car les ressources sont produites par d'autre transitions (et peuvent être utilisées par plusieurs transitions différentes). Il existe également des modèles permettant de déclarer un système composé de plusieurs réseaux de Petri évoluant concurremment [25, 88, 62, 24].

Les *algèbres de processus* [76, 48, 11] disposent d'opérateurs permettant de construire un processus à partir de plusieurs sous-processus s'exécutant en parallèle. On peut alors modéliser les interactions entre ces processus à l'aide de *synchronisations* sur certaines actions (ces actions constituent un rendez-vous entre les processus et doivent se produire au même moment).

À plus bas niveau cela revient à combiner les systèmes de transitions de tous les sous-processus à l'aide d'une opération de *produit synchronisé* dont on trouve un exemple figure 2.1 (page 26). Cette figure présente un *système de transition modulaire*, il est composé de deux systèmes de transitions L_1 et L_2 appelés *modules* qui évoluent de façon concurrente. Dans le cadre de ce modèle cela signifie qu'à tout moment on peut choisir de faire évoluer l'un ou l'autre. La seule restriction est qu'ils doivent se synchroniser pour exécuter l'action s, qui est une action commune aux deux modules.

Conception modulaire et hiérarchique Dans une optique de conception, un des avantages des systèmes concurrents est leur structure modulaire mais la concurrence n'est pas le seul moyen de construire un système en combinant l'ensemble de ses parties. Les algèbres de processus [76, 48, 11] et de réseaux de Petri [88, 12, 13] disposent par exemple d'opérateurs de séquentialité (on exécute un processus puis un autre quand le premier est terminé) ou de choix (on exécute soit l'un soit l'autre).

Travailler sur un système composé de plusieurs modules qui ont chacun différentes fonctions et interagissent les uns avec les autres présente en effet de nombreux avantages. Un module peut par exemple être utilisé dans plusieurs systèmes différents, ou même plusieurs fois au sein du même système de manière à éviter la duplication de code et d'ef-

fort. Un module peut également être facilement remplacé par un autre possédant les même fonctionnalités. Par ailleurs, il est aussi possible de fixer un certain niveau d'abstraction en fonction des parties du système que l'on considère. Si on raisonne sur les interactions et les communications entre les modules (par exemple pour vérifier qu'un message attendu sera toujours envoyé), on peut se placer a un niveau d'abstraction qui ignore comment les messages sont envoyés mais sait tout de même si un message a été envoyé ou non. À l'inverse, on peut également travailler sur un module en particulier indépendamment des autres, par exemple pour implémenter ou tester ses différentes fonctionnalités sans avoir à garder en tête le reste du système.

Quelques autres concepts Diffèrent opérateurs spécialisés permettent d'exprimer ou de faciliter la spécification de certains concepts. Ces concepts peuvent être très variés et on en cite ici quelques exemples. Si ces concepts sont essentiels aux propriétés que l'on veut étudier sur le système, il faut utiliser un langage qui permet de les capturer lors de la modélisation.

Temps Les modèles temporisés permettent de prendre en compte l'évolution du temps [4, 109]. C'est par exemple utile si on veut modéliser un *timeout* pour un système distribué (si un message n'arrive pas dans un intervalle de temps donné, on considère qu'il n'arrivera jamais et on redemande son envoi) ou on s'intéresse à des propriétés du système faisant intervenir le temps (si le système entre dans un état d'erreur, une alarme se déclenche dans la minute qui suit).

Hasard Les modèles probabilistes permettent d'associer des probabilités à certaines actions. On peut par exemple s'en servir pour modéliser les problèmes dus au réseau lors de la transmission d'un message (un packet a 1% de chance de ne pas arriver à destination). Les chaînes de Markov sont un exemple de tels modèles [57].

Les modèles stochastiques combinent les notions de temps et de hasard, en définissant de façon aléatoire le délai que prend une action pour être exécutée. On peut par exemple modéliser le fait que le temps de transition d'un message n'est pas constant mais suit une loi de probabilité connue. On peut alors obtenir des propriétés quantitatives de qualité de service en plus des propriétés de correction (le système retourne un résultat correct en un temps moyen

connu). On trouve dans cette famille les réseaux de Petri stochastiques et le π-calcul stochastique [109, 91].

Mobilité Certains langages permettent de décrire des systèmes mobiles ou reconfigurables dans lesquels les canaux de communication entre les différents modules peuvent apparaître ou disparaître. C'est par exemple le cas d'un système de téléphonie mobile où un téléphone ne peut plus communiquer avec une certaine antenne s'il s'en éloigne trop. Le π-calcul est une algèbre de processus possédant cette fonctionnalité, c'est-à-dire que l'ensemble des actions partagées entre deux modules peut varier au cours de l'exécution du système. [75]

Équité Intuitivement, une exécution du système n'est pas équitable si elle n'exécute jamais un comportement ou une action du système qui a suffisamment souvent la possibilité d'être exécuté. Si un serveur ignore indéfiniment la requête d'un client parce qu'il traite en priorité d'autres requêtes qui ne cessent jamais d'arriver, le système n'est pas équitable. C'est aussi le cas si l'ordonnanceur d'un système d'exploitation ne fait jamais évoluer un processus car d'autres processus sont prioritaires. Certains langages permettent donc de s'assurer que les systèmes décrits sont bien équitables. Soit à l'aide d'opérateurs spécialisés [33, 107, 81], soit parce que leur expressivité limitée ne leur permet pas de décrire des systèmes non équitables [22].

2.1.1.3 Expressivité

Choisir un certain niveau d'abstraction pour un langage limite l'ensemble des systèmes qu'il est capable de représenter, c'est-à-dire son expressivité. Parmi les langages offrant un même niveau d'abstraction, certains sont également plus expressifs que d'autres. Si c'est possible, on essaie alors de choisir un langage ayant une expressivité la plus faible possible car cela permet :

- de bénéficier de résultats de décidabilité, c'est-à-dire d'être sûr qu'il est possible de répondre à la question que l'on se pose sur le modèle. Si on considère par exemple un système décrit dans un langage de programmation Turing-complet, on ne peut pas toujours décider s'il termine [104]. On sait en revanche qu'on pourra toujours savoir s'il existe ou non un état de deadlock sur un système spécifié par un réseau de Petri [71, 23] ;

- d'utiliser des techniques d'analyses spécifiques car on connaît précisément l'ensemble des systèmes de transitions que le langage peut décrire. Ces techniques spécialisées sont donc potentiellement plus efficaces. On peut même dans certains cas savoir que tous les systèmes qu'il est possible de décrire dans un langage vérifient certaines propriétés (comme on l'a vu pour certaines propriétés d'équité). Parmi les méthodes spécifiques à un langage, on peut citer les méthodes statiques (sur lesquelles on revient brièvement par la suite) qui ne cherchent pas à étudier le système de transition sous-jacent, mais directement la description du système dans le langage de modélisation.

2.1.2 Exprimer des propriétés sur les modèles

Les propriétés que l'on cherche à vérifier doivent également être exprimées de manière formelle. C'est à la fois nécessaire pour obtenir des résultats mathématiques sur la correction du modèle et pour utiliser des outils informatiques lors de la vérification (par exemple pour l'automatiser).

Formellement, une *propriété* correspond à un sous-ensemble des modèles exprimables à l'aide du langage de modélisation choisi. Ce sont les comportements de tous les systèmes dits *vérifiant* la propriété. On peut par exemple considérer « les systèmes ne possédant pas de deadlocks » ou « les systèmes sans dépassement d'entiers ». Ces ensembles étant pour la plupart très grands ou infinis, on les décrit à l'aide de langages spécialisés appelés *logiques*. Une *formule* logique permet de décrire un tel ensemble de façon implicite (en intension) plutôt que de façon explicite (en extension).

En pratique les syntaxes des différentes logiques existent indépendamment du langage de modélisation choisi et une même formule logique peut représenter plusieurs propriétés différentes en fonction des types de modèles avec lesquels on travaille.

Si la syntaxe de la logique permet d'exprimer des concepts non disponibles dans le langage de modélisation, il est difficile de donner un sens à toutes les formules (en terme d'ensemble de modèles). Par exemple le hasard n'a pas de sens sur un système de transitions classique.

Le cas inverse ou le langage de modélisation permet d'exprimer des concepts non présents dans la logique est en revanche courant. Il est normal de vouloir vérifier des propriétés générales sur des systèmes spécialisés. Par exemple l'accessibilité d'un état est une propriété générale sur les LTS, on peut l'étendre simplement aux chaînes de Markov en décidant d'ignorer les probabilités apparaissant sur les arcs. Dans ce cas, on peut obtenir certains

modèles qui appartiendront exactement aux mêmes propriétés et seront donc indifférentiables par la logique (ici les systèmes dont seules les probabilités diffèrent).

2.1.2.1 Logique de Hoare

La logique de Hoare est historiquement un des premiers langages formels permettant d'exprimer certaines propriétés des programmes informatiques [47]. Dans cette logique, si φ_1 et φ_2 sont des propriétés sur les variables d'un programme comme par exemple « la variable x est positive » et P un programme ayant une exécution finie, la formule $\{\varphi_1\}P\{\varphi_2\}$ signifie que si l'état initial de P vérifie φ_1, alors son état final vérifie φ_2. La logique de Hoare ne considère que des algorithmes déterministes, c'est-à-dire que pour un état initial donné le système doit avoir une exécution unique et finie (il modélise une fonction).

2.1.2.2 Logiques Temporelles

Les logiques temporelles permettent d'exprimer des propriétés plus précises sur l'évolution des systèmes au cours du temps, et notamment de raisonner sur l'ordre dans lequel se produisent différents événements. Contrairement à la logique de Hoare, elles peuvent traiter de systèmes n'ayant pas d'états finaux ("le système possède ou ne possède pas d'états de deadlock") ou non déterministes ("certaines exécutions terminent mais pas toutes"). En conséquence, ces logiques constituent des outils particulièrement adaptés à la spécification des propriétés des systèmes concurrents [86, 85].

Parmi les logiques temporelles, on peut distinguer les logiques linéaires qui sont adaptées à des modèles déterministes (chaque état a au plus un successeur), des logiques de branchement qui permettent d'exprimer la notion de choix (un état peut avoir plusieurs successeurs). Il est cependant possible d'étendre une logique linéaire sur des modèles non déterministes, par exemple en déclarant qu'un système de transition vérifie une formule si et seulement si toutes ses séquences d'exécutions vérifient cette formule (chaque séquence est considérée comme un modèle déterministe).

LTL (Linear Temporal Logic [85]) est une logique linéaire capable de considérer des systèmes ne terminant pas. Sur l'exemple de l'ampoule sans erreurs, on peut par exemple exprimer le fait qu'à tout moment, si l'ampoule est éteinte alors elle sera allumée au moment suivant. Cette propriété s'écrit

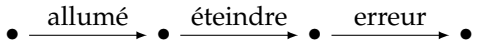
$$\Box(\text{off} \Rightarrow \bigcirc\text{on})$$

où $\Box$ signifie « à tout moment présent ou futur » et $\bigcirc$ signifie « au moment suivant ». Ici la notion de temps est modélisée par les changements d'états du système.

Les logiques temporelles de branchement permettent d'exprimer la notion de non déterminisme, et peuvent considérer des modèles qui peuvent avoir le choix entre plusieurs actions différentes dans un état donné. CTL (Computation Tree Logic [97]) par exemple reprend les opérateurs de LTL, mais y associe des quantificateurs pour exprimer le fait qu'un état du système peut avoir plusieurs successeurs possibles. Ainsi l'opérateur $\exists\bigcirc$ signifie « il existe un successeur » et l'opérateur $\forall\bigcirc$ signifie « pour tous les successeurs ».

Les logiques dynamiques [39, 45] introduisent des opérateurs dépendant des actions possibles d'un système donné. C'est notamment le cas de la logique HML (Hennessy-Milner Logique [46]) qui dispose des quantificateurs $\langle a \rangle$ et $[a]$ signifiant respectivement « il existe un successeur accessible par l'action a ... ». et « tous les successeurs accessibles par l'action a ... ».

On peut par exemple exprimer le fait que si l'ampoule est dans son état initial elle peut exécuter la séquence d'actions



à l'aide de la formule suivante :

$$\langle \text{allumer} \rangle \langle \text{éteindre} \rangle \langle \text{erreur} \rangle \top \ .$$

La formule $\top$ est ici une tautologie et est vérifiée par tous les arbres d'exécutions.

Le μ-calcul modal est une logique dynamique proposée par Kozen [60] qui étend la logique HML en y ajoutant un opérateur de point-fixe, noté μ, qui donne son nom à la logique.

Cet opérateur permet d'exprimer des formules récursives, par exemple pour exprimer que l'ampoule peut effectuer une séquence infinie en alternant les action « allumer » et « éteindre » :

$$\nu X. \langle \text{allumer} \rangle \langle \text{éteindre} \rangle X \ .$$

X représente ici l'appel récursif. Intuitivement, on peut effectuer les actions « allumer » et « éteindre », puis recommencer.

Grâce à cet opérateur de point fixe, le μ-calcul possède une forte expressivité, et nombre de logiques fréquemment utilisées possèdent une traduction vers le μ-calcul. En particulier LTL, CTL et HML [60, 99, 28, 90].

Ces logiques temporelles sont souvent étendues pour permettre d'exprimer différentes propriétés et concepts comme le temps ou les probabilités [3, 59, 44].

2.1.3 Analyse du modèle

Une fois qu'on dispose d'un modèle M de notre système exprimé dans un langage de modélisation, ainsi que d'une propriété de correction φ exprimée dans une logique, il faut choisir une méthode d'analyse adaptée pour déterminer si le modèle vérifie bien la propriété (noté $M \models \varphi$).

2.1.3.1 Approches par preuve

On a vu que les méthodes statiques fonctionnent en étudiant la description du programme dans le langage de modélisation et n'ont donc pas besoin de calculer les exécutions du système. La vérification d'une formule de la logique de Hoare sur un programme est un exemple d'une telle méthode statique, et est également l'une des premières méthode d'analyse formelle de programmes [47].

Cette logique est utilisée au coté d'un système de preuve permettant de représenter le code source d'un programme sous la forme d'une preuve de correction d'une formule en utilisant des règles de déduction du type

$$\{x \leq n\}\text{x++}\{x \leq n+1\}$$

signifiant que si la variable x est inférieure à n avant l'instruction « x++ », alors elle sera inférieure à $n+1$ après l'instruction.

Si on arrive à construire la formule $\{\varphi_1\}P\{\varphi_2\}$ à l'aide du système de preuve, on a la garantie que si l'état initial vérifie φ_1 et que le programme P termine alors l'état final vérifiera φ_2.

Les systèmes de types fonctionnent de façon similaires, dans la mesure où en même temps qu'on écrit un programme, on construit une preuve qu'il est bien typé. Par exemple si x est une variable entière et que le programme compile, on a la garantie qu'elle ne contiendra que des valeurs entières tout au long de l'exécution du programme. Certains systèmes de types peuvent cependant être encore plus expressifs et permettre de prouver des propriétés plus complexes sur les programmes [100, 69, 111].

2.1.3.2 Autres méthodes statiques

Différentes méthodes d'analyses statiques sont applicables sur les différents types de modèles existants. Par exemple, en étudiant la structure du réseau de Petri précédent (page 11), on peut montrer que le nombre de jetons total contenus dans les places « on » et « off » est constant. Puisque ce nombre est égal à 1 dans l'état initial, il y aura toujours un jeton dans une de ces deux places, et on pourra toujours exécuter soit la transition « allumer », soit la transition « éteindre ». Le système ne contient donc pas de deadlocks. On peut aussi montrer par exemple que le système ne revient jamais deux fois dans le même état, car le nombre de jetons dans la place compteur ne fait que croître [78].

Puisque les méthodes statiques ne nécessitent pas d'énumérer les différents états du systèmes pour obtenir des conclusions, elles permettent de prouver rapidement des propriétés sur des systèmes de grandes tailles ou infinis. En revanche, elles sont adaptés à des questions plus spécifiques que les méthodes dynamiques et nécessitent le plus souvent une intervention humaine.

2.1.3.3 Test et simulation

Le test est la méthode d'analyse la plus répandue pour les systèmes non critiques. En effet tester un programme est simple et est intuitivement la première chose que l'on fait pour vérifier si il marche. Concevoir de manière rigoureuse un ensemble de tests à faire passer à un programme permet de maximiser le nombre d'erreurs détectées [15, 89] mais cela n'est pas toujours suffisant. En effet, les systèmes non déterministes et notamment les systèmes concurrents peuvent par exemple comporter des erreurs qui ne se manifestent que dans des cas très rares et qui peuvent donc échapper aux procédure de test.

Ce problème est un cas d'utilisation des langages de modélisation spécialisés dans les systèmes concurrents (Réseaux de Petri, algèbre de processus) car en analysant de tels modèles, on peut considérer certaines exécutions réalisables mais très peu probables.

La simulation fonctionne de façon similaire au test, mais à partir d'un modèle formel et non du système réel. Les procédures de simulations génèrent plusieurs exécutions du modèle et cherchent à détecter des erreurs. Si on découvre une exécution (ou un ensemble d'exécutions) qui ne peuvent pas appartenir à un modèle vérifiant la propriété d'intérêt, le modèle a besoin d'être

corrigé. De plus, les exécutions incorrectes détectées permettent de localiser précisément les erreurs commises.

Il est également possible d'effectuer des analyses quantitatives sur un ensemble d'exécutions du modèle obtenues par simulation en utilisant des outils statistiques [56].

Le test et la simulation permettent en général de détecter de nombreuses erreurs, mais elles ne sont pas exhaustives. Comme on ne considère pas toutes les exécutions possibles on ne peut pas avoir de garanties de correction. De plus la probabilité de détecter des erreurs (et donc la rapidité de la méthode) diminue avec le nombre d'erreurs que contient le système.

2.1.3.4 Model checking

Dans le cas où le comportement du système est fini, les techniques de model checking [92, 27, 10] présentent l'avantage d'être totalement automatiques. On fournit un modèle du système et une propriété de correction à l'outil, qui peut alors considérer alors toutes les évolutions possibles du système et déterminer de manière automatique si la spécification vérifie la propriété. Si ce n'est pas le cas, il exhibe une ou plusieurs exécutions non valide du système (un *contre-exemple*) qui permet de comprendre et de corriger l'erreur trouvée. En réalité il est parfois possible d'utiliser des techniques de model checking sur des systèmes infinis. Par exemple, la vérification de propriétés s'intéressant uniquement au futur proche ne nécessite pas d'explorer tout l'espace d'état du système. Comme discuté par la suite, il existe également des techniques de réductions automatiques de l'espace d'état qui dans certains cas peuvent réduire un ensemble infini en fini.

2.2 Techniques classiques contre l'explosion de l'espace d'états

On a vu que les différent langages de modélisation disposent souvent d'opérateurs modulaires permettant de décrire un système comme la composition de ses parties. Cela facilite la spécification de systèmes de tailles très importantes (le nombre d'état du système global étant au pire égal au produit des nombres d'états de toutes ses parties). Cette *explosion combinatoire* rend en général impraticables les techniques de model checking qui nécessitent l'exploration de tous les états possibles du système. Dans certains cas, en fonction des types de modèles et de propriétés que l'on considère, il est cependant

possible d'obtenir une conclusion tout en évitant d'explorer l'espace d'états complet.

2.2.1 Méthodes à la volée

Un algorithme de model checking fonctionne *à la volée* si il génère l'espace d'état (à partir de la description du système dans un langage de modélisation de plus haut niveau) au fur et à mesure de la vérification. De cette façon, même si le système est de taille très importante (ou même infinie), on peut obtenir une réponse pour certaine propriétés en explorant uniquement une sous-partie de l'espace d'états global. Si on cherche par exemple à déterminer si un état particulier est accessible à partir de l'état initial, on teste tous les états au cours de l'exploration du comportement du système. Si on trouve un tel état, on peut arrêter la procédure et conclure sans explorer le système global [50, 70, 82].

2.2.2 Model checking symbolique

Les méthodes symboliques permettent de représenter des ensembles d'états de façon compacte. Cela permet de manipuler de très grands nombres d'états lors de la vérification, tout en évitant le coût en mémoire et en temps qui y seraient normalement associés. Les diagrammes de décision binaires [19] sont les représentations symboliques les plus classiques mais d'autre représentations existent, par exemple sous forme d'un problème de satisfiabilité [14].

Le model checking symbolique permet de prouver certaines propriétés sur des systèmes bien trop gros pour les méthodes explicites [72, 20]. En contrepartie, dans le cas où la propriété est fausse, il est en général plus difficile de fournir un contre-exemple (mais efficace dans certain cas [26]).

2.2.3 Techniques de réductions

2.2.3.1 Ordres partiels

Les méthodes de réductions consistent à remplacer un système de transition par un autre de plus petite taille, en s'assurant que cela n'affecte pas les propriétés que l'on cherche à vérifier. Les premières réductions sont les *réductions d'ordre partiel* [105, 55, 43]. Si on considère le graphe de la figure 2.1, les séquences d'actions $\bullet \xrightarrow{d} \bullet \xrightarrow{d} \bullet \xrightarrow{a} \bullet$ et $\bullet \xrightarrow{a} \bullet \xrightarrow{d} \bullet \xrightarrow{d}$ mènent vers le même état et correspondent toutes les deux au fait que L_1 exécute l'action a et que L_2

exécute la séquence $\bullet \xrightarrow{d} \bullet \xrightarrow{d} \bullet$. Si l'ordre ne nous intéresse pas, ces deux séquences sont équivalentes et on peut en supprimer une. En règle générale, on peut n'en garder qu'une parmi toutes les combinaisons possibles ce qui peut permettre des réductions importantes.

2.2.3.2 Symétries

Les réductions par symétrie identifient des comportements du système apparaissant plusieurs fois en plusieurs endroit différents, comme cela peut être le cas si un programme comporte plusieurs instances différentes d'un même processus (mis-à-part certaines caractéristiques comme l'identifiant du processus que l'on peut décider d'ignorer, les comportements seront identiques). On peut alors parfois réduire la taille du système à analyser en cherchant à éliminer ces comportements redondant. [29, 40]

2.2.3.3 Abstractions

D'autres d'opération de réduction, basées sur différentes relations d'équivalences entre les états d'un système de transitions sont abordées par la suite.

2.2.4 Modularité et abstraction

Dans une optique de conception, manipuler des modules interagissant le moins possible les uns avec les autres présente deux avantages importants :

- une partie importante de chaque module peut être implémentée ou testée sans connaître le reste du système. Il est plus simple de raisonner sur le comportement de cette partie car il n'y a pas de concurrence à prendre en compte (toute l'information nécessaire est locale au module), et si elle est valide elle le restera même si on modifie quelque chose dans un autre module.
- il est également plus simple de raisonner sur les communications entre les modules. D'une part, ils communiquent peu les uns avec les autres. D'autre part, la plus grande partie du comportement de chaque module est interne et n'est pas importante lorsque l'on s'intéresse aux interactions.

Ces deux concepts de décomposition et d'abstraction permettant de raisonner sur les systèmes modulaires sont aussi exploitables par les algorithmes de vérification. On trouve notamment des méthodes essayant de décomposer une propriété en plusieurs parties destinée à être vérifiées sur les diffé-

rents modules [87, 64, 53]. Ces méthodes sont proches des méthodes d'analyse partielles qui tentent d'obtenir une conclusion en observant uniquement une sous-partie du système [5, 66, 6].

Les méthodes par abstraction hiérarchique proposent une approche complémentaire qui consiste à réduire les comportement des différents modules avant de les composer ou de les étudier de manière à considérer efficacement une abstraction pertinente du comportement du système global [101, 30, 35].

2.3 Analyse partielle

Les méthodes d'analyses partielles permettent de cibler une sous-partie d'un système modulaire lors de la vérifications d'une propriété temporelle, dans l'espoir d'obtenir une réponse sans avoir à considérer le système global.

2.3.1 Quotient d'une formule sur un module

Le model checking partiel a été en premier proposé par Andersen [5, 6] pour la vérification d'une formule de μ-calcul sur un système constitué de plusieurs LTS composés par l'opérateur de parallélisation de l'algèbre de processus CCS [74]. L'approche a ensuite été étendue à une spécification arbitraire de CCS [9] puis adaptée aux réseaux de LTS [66].

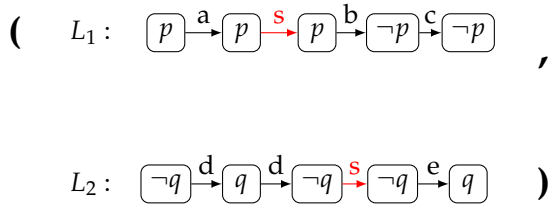
Cette approche a pour objectif d'atténuer l'effet de l'explosion combinatoire lors de l'analyse d'un système modulaire. Elle permet notamment de déterminer la valeur de vérité d'une propriété en analysant uniquement une sous-partie pertinente du système.

Considérons une propriété φ à vérifier sur un système modulaire $(S_1, S_2, \ldots, S_n)$. Le *quotientage* de φ sur une partie du système (par exemple S_1) renvoie une nouvelle formule (notée $\varphi /\!\!/ S_1$). Cette formule est destinée à être vérifiée sur le reste du système $(S_2, \ldots, S_n)$ et est construite telle que :

$$(S_1, S_2, \ldots, S_n) \models \varphi \Leftrightarrow (S_2, \ldots, S_n) \models \varphi /\!\!/ S_1.$$

On remarque que si la formule $\varphi /\!\!/ S_1$ est une tautologie, on a nécessairement $(S_2, \ldots, S_n) \models \varphi /\!\!/ S_1$. Cela permet de conclure immédiatement que le système global vérifie φ, en ayant seulement étudié le module S_1. À l'inverse, si la formule renvoyée est une contradiction, on sait que le système global ne satisfait pas la formule φ. Si on est dans un cas different, on peut réitérer la même technique pour vérifier la formule $\varphi /\!\!/ S_1$ sur le modèle (de plus petite taille) $(S_2, \ldots, S_n)$. Cette approche évite le calcul explicite du produit des modules, mais l'effet de l'explosion combinatoire se retrouve dans la taille

LTS modulaire :



Comportement du LTS modulaire :

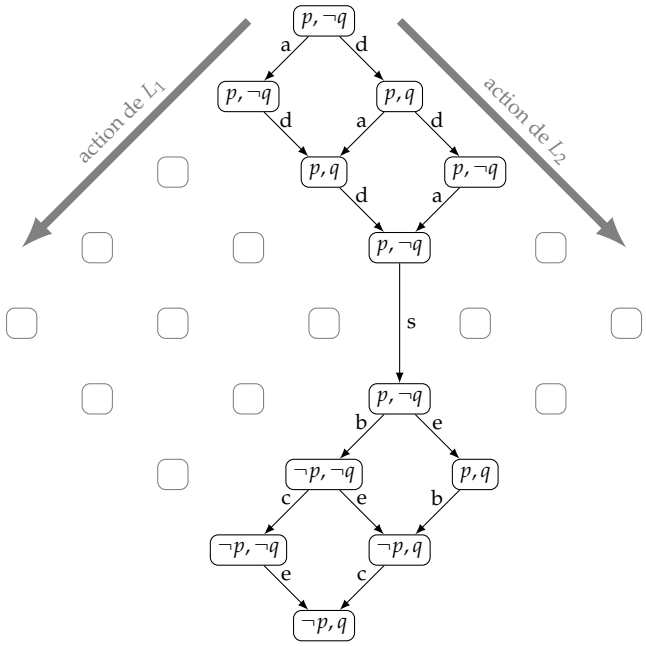


FIGURE 2.1 – Exemple d'un LTS modulaire et de son comportement. Le LTS modulaire est composé de deux modules L_1 et L_2. Les actions a, b et c sont locales à L_1, les actions d et e locales à L_2 et l'action s est partagée par les deux modules, ce qui signifie qu'ils doivent se synchroniser pour l'exécuter simultanément.

souvent importante des formules quotients construites. Pour cette raison les méthodes citées ci-dessus proposent également des opérations de réduction des formules de façon à limiter les tailles des formules intermédiaires, et dans le même temps de détecter les tautologies et les contractions.

2.3.2 Abstraction 3-valuée

Dans [96], les auteurs proposent une technique de vérification compositionnelle d'une formule de μ-calcul sur un modèle défini comme un produit de structures de Kripke [61] et expliquent comment il est possible de l'adapter aux LTS modulaires. Cette approche considère un module comme une abstraction du système global, puis tente de vérifier la formule sur cette abstraction à l'aide d'un graphe de jeux 3-valué. L'état initial (qui est une abstraction de l'état initial réel) est considéré gagnant si la formule y est vraie (quelque soit les raffinements possibles), perdant si elle est y fausse, et indéterminé si on le module ne dispose pas d'assez d'information pour conclure. Dans ce troisième cas, de l'information sur l'environnement du module est injectée dans le graphe, sous la forme d'un raffinement des parties du modèle sur lesquelles l'évaluation de la formule est indéterminée.

Mis-à-part les différences dues aux modèles, le graphe de jeux utilisé ici est proche du graphe d'accessibilité par une formule que l'on utilise sous-section 5.2.1 (qui est essentiellement un produit entre la formule à vérifier et un module) et la procédure d'évaluation locale de la formule est proche des calculs de Pass et Fail définis sous-section 5.1.1 (Nous avons pris connaissance de ces travaux durant l'étape de review de notre article [1]). Cependant la façon dont on utilise ces informations diffère car les objectifs poursuivis sont différents et complémentaires. En effet, comme le model checking partiel, cette approche a pour but objectif de retourner la valeur de vérité de la formule sur le système global, alors que dans la section 5.1, on s'intéresse à définir une opération de réduction d'un module similaire à celles présentée section 2.4, c'est-à-dire qu'elle est indépendante de l'environnement du module et peut être utilisée en conjonction d'une méthode particulière de model checking.

2.4 Abstractions hiérarchiques

Une manière d'améliorer les performances du model checking consiste à vérifier une propriété à un niveau d'abstraction plus élevé, c'est-à-dire en dissimulant certaines caractéristiques inutiles du système de manière à réduire la taille de son espace d'état. Classiquement cette opération de réduc-

tion consiste à remplacer un LTS par un autre (de plus petite taille) dont la structure est équivalente. De nombreuses notions différentes d'équivalence existent et préservent différentes caractéristiques des LTS.

L'une des plus fines relations (identifiant peu de systèmes comme équivalents), est par exemple la bisimulation forte [80], qui déclare deux systèmes équivalents si ils peuvent effectuer les mêmes séquences d'actions tout en passant par des états eux mêmes bisimilaires. À l'inverse, l'équivalence de trace requiert seulement que deux LTS puissent effectuer les même séquences d'actions. D'autres relations d'équivalences classiques sont définies en section 3.3. Certaines de ces relations (comme la bisimulation forte) considèrent les actions τ internes au système comme significatives, et sont capables de différentier deux LTS grâce à ces actions, alors que d'autres (comme la bisimulation faible [80, 74]) considèrent qu'elles sont invisibles. L'exemple donné pour la bisimulation faible dans la figure 3.2 (page 46) illustre ceci : les deux graphes sont équivalents pour la bisimulation faible qui ignore l'action τ, mais pas pour la bisimulation forte (qui détecte que le premier graphe peut effectuer l'action τ, mais pas le second).

Une seconde façon de définir une relation d'équivalence sur les systèmes de transitions consiste à considérer un ensemble de propriétés (par exemple une logique), et de déclarer deux LTS équivalents si et seulement si ils vérifient les mêmes propriétés de cet ensemble. Il est alors possible d'étudier les liens entre les équivalences définies par un ensemble de formules et celles définies sur la structure des LTS. En plus d'apporter une perspective sur leur nature, cela permet de définir exactement quelles propriétés sont préservées par une équivalence structurelle donnée, et donc de garantir une opération de réduction sans risque. On sait par exemple que si deux graphes sont fortement bisimilaires, ils vérifieront exactement les mêmes formules de μ-calcul [80].

Une autre propriété essentielle de ces relations d'équivalence sur les LTS, est que ce sont des congruences vis-à-vis du produit synchronisé, c'est-à-dire que si on remplace dans un LTS modulaire, un module par un autre module équivalent, on obtiendra un comportement global également équivalent à l'ancien. Cela permet de construire une abstraction du comportement d'un LTS modulaire en composant des abstractions de ses parties. Cette technique d'abstraction hiérarchique permet donc d'éviter de construire le comportement détaillé du système global si ce n'est pas nécessaire, dans le but d'atténuer ainsi l'effet de l'explosion combinatoire. En fait on peut même appliquer des étapes de réductions à n'importe quelle étape de la construction incrémentale du comportement global comme expliqué plus en détails par la suite.

2.4.1 Ordre d'analyse dans l'abstraction hiérarchique

Les différentes réductions vues précédemment permettent de construire une abstraction du comportement d'un système modulaire par une approche d'*agrégation compositionnelle*. Cette technique consiste à construire de manière incrémentale une abstraction du comportement du système global grâce à plusieurs étapes de compositions et de réductions des modules.

Le choix de l'ordre de construction est un problème difficile et a une forte influence sur l'efficacité de la méthode. De manière générale on ne peut pas être sûr qu'un ordre est meilleur qu'un autre sans les tester tous les deux. La spécification de cet ordre est donc souvent laissée à la charge de l'utilisateur, mais des heuristiques existent qui tentent de trouver un bon ordre de manière automatique. De telles méthodes peuvent se trouver dans [101] (sur des *communicating finite state machines*), [35] sur des LTS modulaires, et enfin [36] sur des réseaux de LTS.

Chapitre 3

Définitions générales

3.1 Modèles modulaires utilisés

On définit maintenant formellement les modèles utilisés par la suite. Les LTS modulaires et les réseaux de LTS. Les deux modèles sont constitués d'un ensemble de modules chacun défini par un système de transition. Pour un LTS modulaire, tous les modules partageant une action doivent se synchroniser pour pouvoir l'exécuter. Alors que les réseaux de LTS permettent de définir des règles de synchronisation plus précises.

C'est une représentation de bas niveau, ce qui permet de définir par la suite des méthodes d'analyse généralistes. On définit les LTS modulaire comme étant finis car ces méthodes fonctionnent sur des systèmes finis.

3.1.1 LTS modulaire

On définit un LTS (*labelled transition system*) modulaire comme une famille finie de LTS également finis. Ces LTS sont appelés des modules. Le comportement du LTS modulaire est un produit de ses modules, synchronisés sur leurs actions partagées.

Définition 1. Un *système de transition étiqueté* ou LTS (Labelled Transition System) est un graphe orienté $S \stackrel{\text{df}}{=} (Q, q_0, A, R, L)$ où Q est un ensemble fini d'*états* (sommets), $q_0 \in Q$ est l'*état initial*, A est l'ensemble des *actions* utilisés pour l'étiquetage des transitions, $R \subseteq Q \times A \times Q$ est l'ensemble des *transitions* (arcs). Soit $\mathbb{V}$ un ensemble de *variables d'état* de S, une *interprétation* de $\mathbb{V}$ associe a chacune de ses variables une valeur de vérité booléenne. L'étiquetage $L : Q \mapsto \mathbb{V} \mapsto \{0, 1\}$ des noeuds de S associe à chaque noeud une interprétation de $\mathbb{V}$.

Une transition $(q, a, q') \in R$ est couramment noté $q \xrightarrow{a} q'$.

Définition 2. Un *LTS modulaire* est une famille finie de LTS (modules), où les variables utilisées pour l'étiquetage des états sont uniques à chaque module.

Le comportement d'un LTS modulaires est le produit de ses modules, synchronisés sur les actions partagées. Plus précisément, les modules partageant une action s doivent se synchroniser pour tous l'exécuter ensemble. On note $x[i]$ le i-eme composant du n-uplet x.

Définition 3. Soit $S \stackrel{\text{df}}{=} (S_1, \ldots, S_n)$ un LTS modulaire avec $S_i \stackrel{\text{df}}{=} (Q_i, q_{0_i}, A_i, R_i, L_i)$. Le comportement $\mathsf{lts}(S)$ de ce LTS modulaire est le *produit synchronisé* des différents modules S_i. C'est le LTS (Q, q_0, A, R, L) défini par :

- $Q \stackrel{\text{df}}{=} \prod_{1 \leq i \leq n} Q_i$;
- $q_0 \stackrel{\text{df}}{=} (q_{0_1}, \ldots, q_{0_n})$;
- $A \stackrel{\text{df}}{=} \bigcup_{1 \leq i \leq n} A_i$;
- R est le plus petit sous-ensemble de $Q \times A \times Q$ tel que $p \xrightarrow{a} q \in R$ si pour tout $1 \leq i \leq n$, l'une des deux conditions suivantes est vérifiée.
  - $a \in A_i$ et $p[i] \xrightarrow{a} q[i] \in R_i$
  - $a \notin A_i$ et $p[i] = q[i]$
- pour tout état $p \in Q$, $L(p) \stackrel{\text{df}}{=} \bigcup_{1 \leq i \leq n} L_i(p[i])$.

On note $S_1 \otimes S_2$ le comportement du LTS modulaire à deux éléments (S_1, S_2). Ce produit est une opération binaire associative et commutative et on a pour un LTS modulaire $(S_1, \ldots, S_n)$:

$$\mathsf{lts}(S_1, \ldots, S_n) = S_1 \otimes S_2 \otimes \cdots \otimes S_n.$$

Il s'en suit qu'il est possible de remplacer n'importe quel sous-ensemble de modules (qu'on appelle parfois sous-système par la suite) par leur produit sans modifier le comportement du LTS modulaire. On peut donc itérer une telle opération afin d'obtenir un système constitué d'un unique module, de façon à construire le comportement du LTS modulaire de façon hiérarchique et incrémentale. Cette importante propriété des systèmes modulaire permet notamment d'appliquer des techniques de réduction hiérarchique comme celles détaillées section 2.4.

3.1.2 Réseau de LTS

Un réseau de LTS [65] est constitué d'une famille de LTS, associée à un ensemble de règles décrivant explicitement les synchronisations entre ces LTS et permettant de construire le comportement global du système. Les LTS modulaires présentés précédemment peuvent être vus comme des réseau de LTS possédant des règles de synchronisation implicites nécessitant que tous les modules partageant une action donnée se synchronisent sur cette action. Les règles de synchronisations permettent une définition fine du produit des différents LTS du réseau et permet d'exprimer de nombreux opérateurs provenant d'algèbres de processus comme le renommage, la suppression ou la dissimulation de certaines actions [76, 68].

Les règles de synchronisation suivantes permettent de décrire la sémantique du LTS modulaire de la figure 2.1 (page 26).

L_1	L_2	LTS produit
a	$\bullet$	a
b	$\bullet$	b
c	$\bullet$	c
$\bullet$	d	d
$\bullet$	e	e
s	s	s

Ici, les actions a, b et c sont locales au module L_1. Si celui-ci est capable de les exécuter, le système global le pourra également (et avec le même nom). De même pour les actions d et e qui sont locales à L_2. L'action s est elle synchronisée entre les deux modules, qui doivent l'exécuter simultanément pour construire une transition s du système global.

Définition 4 (Règles de synchronisations)**.** Soit $(S_i)_{1\leq i\leq n}$ un LTS modulaire. Soit A l'ensemble des actions du LTS produit. Une *règle de synchronisation* est de la forme (t, a) où $t \in \prod_i (A_i \cup \{\bullet\})$ et $a \in A$. Le LTS produit peut effectuer l'action a si et seulement si il existe une règle (t, a) du vecteur de transition telle que chaque composant S_i (tel que $t[i] \neq \bullet$) peut effectuer l'action $t[i]$. Formellement, le produit des S_i par l'ensemble de règles V est le LTS $(\prod_{1\leq i\leq n} Q_i, A, R, L)$ tel que :

- $p \xrightarrow{a} q$ appartient à R si et seulement si il existe une règle $(t, a) \in V$ telle que pour chaque composant S_i l'une des deux conditions suivantes est vérifiée :
  - $t[i] = \bullet$ et $p[i] = q[i]$;

- $t[i] \neq \bullet$ et $p[i] \xrightarrow{t[i]} q[i] \in R_i$;

- pour tout état q du LTS produit, $L(q) \stackrel{\text{df}}{=} \bigcup_i L_i(q[i])$.

3.2 μ-calcul

On définit maintenant formellement le μ-calcul.

Une formule φ du μ-calcul modal est destinée à être évaluée sur un LTS S. Cette évaluation retourne un sous-ensemble des états de S, qui sont les états *vérifiant* la formule φ.

3.2.1 Syntaxe

Une formule du μ-calcul modal est produite à l'aide de la grammaire suivante, où B est une formule de logique Booléenne sur un ensemble d'atomes $\mathbb{V}$ (correspondant aux labels des noeuds de S), α est un ensemble d'actions (effectuables par S) et X est une variable propositionnelle :

$$\varphi \quad ::= \quad B \mid \neg\varphi \mid \varphi \vee \varphi \mid \langle\alpha\rangle\varphi \mid \mu X.\varphi \mid X \ .$$

De plus, dans une formule du type $\mu X.\varphi$, φ doit être *positive* selon la variable X. C'est-à-dire que toutes les occurrences libres de X doivent se trouver sous un nombre pair de négations $\neg$.

Afin de définir formellement la notion d'occurrence libre, on commence par définir la relation de sous-formule de façon usuelle. C'est la plus petite relation d'ordre telle que, si φ et ψ sont des formules alors φ est une sous-formule de φ, $\neg\varphi$, $\langle\alpha\rangle\varphi$, $\mu X.\varphi$,$\varphi \vee \psi$ et de $\psi \vee \varphi$. On écrit $\varphi \leq \psi$ si φ est une sous-formule de ψ et $\varphi < \psi$ si φ est une sous-formule stricte de ψ (différente de ψ).

Une occurrence d'une variable X est dite *libre* (non-liée) dans une formule φ s'il n'existe pas de sous-formule de φ étant de la forme $\mu X.\psi$ et contenant cette occurrence. Une variable est dite libre (non-liée) dans une formule si cette formule contient une occurrence libre de cette variable. L'ensemble des variables libres de φ est noté $\mathsf{free}(\varphi)$.

$\varphi[X \leftarrow \psi]$ représente la formule φ dans laquelle on substitue les occurrences libres de X par la formule ψ. Si φ contient une unique variable libre X, on peut également écrire $\varphi(\psi) \stackrel{\text{df}}{=} \varphi[X \leftarrow \psi]$.

Définition 5 (Opérateurs non primitifs)**.** Afin de faciliter l'écriture des formules il est possible d'utiliser les opérateurs classiques suivants :

- $\varphi_1 \wedge \varphi_2 \stackrel{\text{df}}{=} \neg(\neg\varphi_1 \vee \neg\varphi_2)$;
- $[\alpha]\varphi \stackrel{\text{df}}{=} \neg\langle\alpha\rangle\neg\varphi$;
- $\nu X.\varphi \stackrel{\text{df}}{=} \neg\mu X.\neg\varphi[X \leftarrow \neg X]$;
- $\varphi \Rightarrow \psi \stackrel{\text{df}}{=} \neg\varphi \vee \psi$;

Une formule est dite sous *forme normale positive* si toutes les négations qu'elle contient sont appliquées à des propositions atomiques. Toute formule peux être réécrite sous forme normale positive en utilisant les opérateurs duaux ci-dessus.

Pour alléger la notation, si $\{a, b\}$ est un ensemble d'actions du LTS on utilise par la suite $\langle a, b\rangle\varphi$ au lieu de $\langle\{a, b\}\rangle\varphi$ si il n'y a pas d'ambiguïté.

On utilise également $\langle -\rangle\varphi$ et $[-]\varphi$ pour exprimer le fait que l'on peut utiliser n'importe quelle action et $\sigma X\varphi$ pour désigner une formule de plus grand ou de plus petit point fixe (σ peut représenter indifféremment μ ou ν).

On appelle *dépliage* d'une formule de point fixe $\sigma X.\varphi$ la formule $\varphi[X \leftarrow \sigma X.\varphi]$.

On note également par la suite $\mathcal{CL}(\varphi)$ la *fermeture* de la formule φ. C'est intuitivement l'ensemble des formules que l'on peut atteindre à l'aide d'opérations de décomposition et de dépliage de point fixe à partir de φ. Elle est définie comme suit :

- $\mathcal{CL}(\varphi) \stackrel{\text{df}}{=} \{\varphi\} \bigcup_{\varphi' < \varphi} \mathcal{CL}(\varphi')$ si φ n'est pas un point-fixe.
- $\mathcal{CL}(\sigma X.\varphi) \stackrel{\text{df}}{=} \{\sigma X.\varphi\} \cup \mathcal{CL}(\varphi[X \leftarrow \sigma X.\varphi])$.

3.2.2 Sémantique

L'évaluation d'une propriété sur un LTS a pour objectif d'identifier tous les états sur lesquels la propriété est vérifiée. Le LTS lui-même vérifie alors cette propriété si et seulement si son état initial appartient à cet ensemble.

Soient $S \stackrel{\text{df}}{=} (Q, q_0, A, R, L)$ un LTS et $\mathcal{X}$ un ensemble de variables propositionnelles. Une *valuation* de $\mathcal{X}$ associe à chacune de ces variables un ensemble d'états de Q.

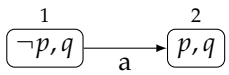
La sémantique d'une formule φ sur S (noté φ^S) est une fonction qui associe à une valuation des variables libres de φ, un ensemble d'états de Q :

$$\varphi^S : (\mathsf{free}(\varphi) \mapsto 2^Q) \mapsto 2^Q.$$

En particulier si la formule φ est close, φ^S est l'ensemble des états *vérifiant* la formule (φ^S est alors une fonction dont l'unique argument possible est $\emptyset$, et l'ensemble des fonctions de $\{\emptyset\}$ vers 2^Q est isomorphe à 2^Q).

Par exemple dans la formule $\mu X.\langle a\rangle X \vee B$, la sous formule $\langle a\rangle X$ définit une fonction qui associe à chaque ensemble $X \subseteq Q$, les états y vérifiant $y \xrightarrow{a} x$ avec $x \in X$.

Par ailleurs, si x est un état de S et B est une formule booléenne, on note $B@x$ la formule $B[v \leftarrow L(x,v) \mid v \in \mathbb{V}]$, où les variables d'état apparaissant dans B sont substituées par leurs valeurs en x. Par exemple dans le cas du LTS suivant, on a $(p \wedge q)@1 = \bot \wedge \top = \bot$ mais $(p \wedge q)@2 = \top \wedge \top = \top$.



Définition 6 (sémantique du μ-calcul)**.** Étant donnée une formule φ et une valuation $\mathcal{V}$ de $\mathsf{free}(\varphi)$, la sémantique φ^S de cette formule sur S est définie comme suit :

[1] $B^S \stackrel{\text{df}}{=} \{x \in Q \mid B@x = \top\}$**;**
B (en tant que formule de μ-calcul) est vérifiée sur tous les états dont le label est un modèle de B (en tant que formule booléenne).

[2] $(\neg\psi)^S(\mathcal{V}) \stackrel{\text{df}}{=} Q \setminus \psi^S(\mathcal{V})$**;**
La négation d'une formule est vérifiée dans tous les états où cette formule ne l'est pas.

[3] $(\psi_1 \vee \psi_2)^S(\mathcal{V}) \stackrel{\text{df}}{=} \psi_1^S(\mathcal{V}_{|\mathsf{free}(\varphi_1)}) \cup \psi_2^S(\mathcal{V}_{|\mathsf{free}(\varphi_2)})$**;**
La disjonction de deux formules est vérifiée si et seulement si au minimum l'une de ces formules l'est. L'opérateur dual $\wedge$ représente la conjonction.

[4] $(\langle\alpha\rangle\psi)^S(\mathcal{V}) \stackrel{\text{df}}{=} \{x \in Q \mid \exists y \in \psi^S(\mathcal{V}), \exists a \in \alpha, x \xrightarrow{a} y \in R\}$**;**
Un état x vérifie la formule $\langle\alpha\rangle\psi$ si et seulement si il est possible, à partir de x, d'atteindre un état vérifiant ψ en effectuant une action de l'ensemble α . L'opérateur dual $[\alpha]\psi$ exprime alors le fait que tous les successeurs par une action de α vérifient ψ.

[5] $(\mu X.\psi)^S(\mathcal{V}) \stackrel{\text{df}}{=} \bigcap\{\rho \in 2^Q \mid \psi^S(\mathcal{V} \cup (X,\rho)) \subseteq \rho\}$**;**
Ici, la variable X est libre dans ψ mais pas dans $\mu X.\psi$, elle n'apparaît donc pas dans la valuation $\mathcal{V}$. On peut considérer l'expression $\psi^S(\mathcal{V})$ comme une application partielle de la fonction ψ^S, obtenue en fixant toutes les variables libres différentes de X *i.e.* $\psi^S(\mathcal{V})(\rho) \stackrel{\text{df}}{=} \psi^S(\mathcal{V} \cup (X,\rho))$.
$(\mu X.\psi)^S(\mathcal{V})$ est alors le plus petit point fixe de cette fonction $\psi^S(\mathcal{V})$. La sémantique de l'opérateur dual $\nu X.\psi$ est le plus grand point fixe de cette même fonction.

Ce plus petit point fixe peut en pratique être calculé en itérant la fonction à partir de l'ensemble vide, comme expliqué pour l'exemple 2.

[6] $X^S(\mathcal{V}) \stackrel{\text{df}}{=} \mathcal{V}(X)$**;**
On retourne la valeur de X dans la valuation $\mathcal{V}$. X^S peut être vue comme la fonction identité de 2^Q vers 2^Q. En effet $\mathsf{free}(X)$ est égal à $\{X\}$ et donc $\mathsf{free}(X) \mapsto 2^Q$ est isomorphe à 2^Q.

On se permet parfois par la suite d'utiliser φ au lieu de φ^S si le contexte n'introduit pas d'ambiguïté.

Pour un état q d'un LTS S, on écrit $S, q \models \varphi$ ssi φ est une formule close et que q appartient à l'ensemble φ^S. On dit alors que l'état q vérifie la propriété φ. Le LTS S vérifie φ (noté $S \models \varphi$) ssi son état initial vérifie φ.

3.2.3 Profondeur d'alternance

Soit φ une formule du μ-calcul. φ contient une chaîne de points fixes alternants de taille k si il existe k formules $\psi_0 \ldots \psi_{k-1}$ telles que l'une des deux propriétés suivantes est vérifiée (les plus petits et plus grands points fixes alternent) ;

$$\text{(1) } \varphi \geq \mu X_0.\psi_0 > \nu X_1.\psi_1 > \cdots > \sigma X_{k-1}.\psi_{k-1}$$
$$\text{ou (2) } \varphi \geq \nu X_0.\psi_0 > \mu X_1.\psi_1 > \cdots > \sigma X_{k-1}.\psi_{k-1}$$

et telles que pour tout entier $i < k-1$ et toute formule ψ avec $\psi_i \geq \psi \geq \psi_{i+1}$, X_i est libre dans ψ.

La profondeur d'alternance d'une formule est la taille de la plus grande chaîne de ce type contenue dans cette formule, et représente le nombre d'opérations de points fixes imbriqués dans la formule.

Par exemple la formule $\mu X.(\neg p \vee \langle a \rangle X)$ de l'exemple 3.2 a une profondeur de 1, alors que la formule $\nu X.\mu Y.(\langle - \rangle Y \vee (p \wedge \langle - \rangle X))$ de l'exemple 3 a une profondeur de taille deux.

3.2.4 Sous-logiques

On présente maintenant les traductions en μ-calcul de plusieurs opérateurs de logiques temporelles classiques. En plus de témoigner de l'expressivité importante du μ-calcul, cette partie introduit des macros permettant de simplifier l'écriture de certaines formules.

On commence par traiter la logique CTL (Computational Tree Logic [28]) puis on s'intéresse par la suite à certains opérateur observationnels dont le but est d'ignorer certaines actions d'un LTS.

Une formule de CTL est générée à l'aide de la grammaire suivante.

$$\varphi ::= B \mid \neg\varphi \mid \varphi \vee \varphi \mid \exists \bigcirc \varphi \mid \forall \bigcirc \varphi \mid \exists(\varphi U \varphi) \mid \forall(\varphi U \varphi)$$

Cette logique s'interprète sur une *structure de Kripke* [61], que l'on peut définir comme un LTS vérifiant deux conditions :

- Son alphabet est constitué d'une action unique (il n'est alors pas nécessaire de nommer cette action ni d'étiqueter les arcs du graphes).
- Tout état doit avoir au minimum un successeur.

La traduction t suivante définit la sémantique de CTL en terme de μ-calcul sur un tel graphe. On peut également s'en servir comme macros afin de simplifier si possible l'écriture et la lecture de formules du μ-calcul.

[1] $t(\exists \bigcirc \varphi) = \langle - \rangle t(\varphi)$**;**
Il existe un successeur de l'état vérifiant φ.

[2] $t(\forall \bigcirc \varphi) = [-] t(\varphi)$**;**
Tous les successeurs de l'état vérifient φ.

[3] $t(\exists(\varphi U \psi)) = \mu X.t(\psi) \vee (t(\varphi) \wedge \langle - \rangle X)$**;**
Il existe un chemin constitué d'états vérifiant φ menant vers un état vérifiant ψ.

[4] $t(\forall(\varphi U \psi)) = \mu X.t(\psi) \vee (t(\varphi) \wedge [-] X)$**;**
Tout chemin est constitué d'états vérifiant φ et mène vers un état vérifiant ψ.

Les opérateurs non primitifs de CTL se traduisent alors de la manière suivante.

$t(\exists \Diamond \varphi) = t(\exists(\top U \varphi)) = \mu X.t(\varphi) \vee \langle - \rangle X$**.** Il existe un chemin menant à un état vérifiant φ.

$t(\forall \Diamond \varphi) = t(\forall(\top U \varphi)) = \mu X.t(\varphi) \vee [-] X$**.** Tout chemin infini atteint un état vérifiant φ. Sur un LTS possédant des états sans successeurs, il faut modifier légèrement la formule si l'on veut également considérer les chemins finis. Elle devient alors $\mu X.\varphi \vee ([-] X \wedge \langle - \rangle \top)$. On a également la formule $\forall \Diamond \bot = \mu X.[-] X$ qui signifie que tous les chemins existants sont finis.

$t(\exists\Box\varphi) = t(\neg\forall\Diamond\neg\varphi) = \nu X.t(\varphi) \wedge \langle - \rangle X$**.** Il existe un chemin infini sur lequel φ est toujours vrai. On remarque le cas particulier $\exists\Box\top = \nu X.\langle - \rangle X$ qui exprime l'existence d'un chemin infini.

$t(\forall\Box\varphi) = t(\neg\exists\Diamond\neg\varphi) = \nu X.t(\varphi) \wedge [-]X$**.** φ est vrai sur tous les chemins possibles (même les chemins finis).

Les opérateurs suivants sont utiles dans le cas où le modèle peut effectuer une action spéciale τ, qui représente une action interne que l'on veut parfois considérer comme invisible. Dans les définitions suivantes, a représente une action visible (différente de τ) et b représente une action quelconque.

$$\langle b \rangle_o \varphi = \begin{cases} \mu X.\varphi \vee \langle \tau \rangle X & \textbf{si } b = \tau, \\ \mu X.\langle \tau \rangle X \vee (\langle b \rangle (\mu Y.\langle \tau \rangle Y \vee \varphi)) & \textbf{si } b \neq \tau. \end{cases}$$

L'opérateur $\langle b \rangle_o$ considère différemment les actions observables de l'action invisible τ. Si l'action b est différente de τ, la formule $\langle b \rangle_o \varphi$ signifie que l'on peut atteindre un état vérifiant φ par une séquence d'actions $\tau^* b \tau^*$, on autorise donc le système à effectuer des actions invisibles avant et après l'action b. L'opérateur $\langle \tau \rangle_o \varphi$ autorise explicitement le système à effectuer des actions internes si nécessaire pour atteindre un état vérifiant φ [76].

$\langle a \rangle_s \varphi = \mu X.\langle \tau \rangle X \vee \langle a \rangle \varphi$**.**

Cet opérateur exprime l'accessibilité d'un état vérifiant φ à l'aide d'une séquence de transitions $\tau^* a$. C'est un cas particulier de l'opérateur du μ-calcul sélectif définie dans [8].

$$\varphi \langle b \rangle_u \psi = \begin{cases} \psi \vee (\mu X.(\varphi \wedge (\langle \tau \rangle \psi \vee \langle \tau \rangle X))) & \textbf{si } b = \tau, \\ \mu X.(\varphi \wedge (\langle b \rangle \psi \vee \langle \tau \rangle X)) & \textbf{si } b \neq \tau. \end{cases}$$

Cet opérateur se lit « φ until ψ » et exprime le fait qu'il existe une séquence de transitions $\tau^* b$ passant uniquement par des états vérifiant φ, et amenant à un état vérifiant ψ. Si b est l'action τ, on autorise cette séquence à être vide auquel cas l'état courant doit lui même vérifier ψ. Cet opérateur appartient à la logique « Hennessy-Milner avec until » définie dans [37].

3.2.5 Exemples de formules

Exemple 1. La table 3.1 présente quelques formules exprimant des propriétés ne nécessitant pas l'utilisation de points fixes (de profondeur d'alternance 0), ainsi que les ensembles d'états vérifiant ces formules sur un LTS donnée en exemple.

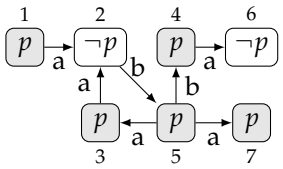
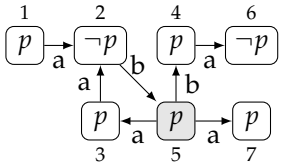
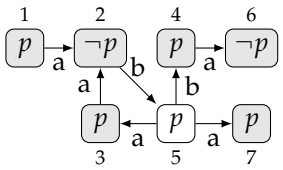
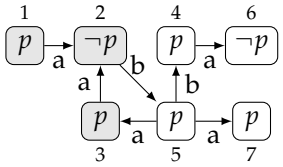
Propriété	Formule et sa sémantique sur un LTS donné. Les états grisés vérifient la formule.
La variable d'état p est vraie.	$\varphi = B = p$ 
On peut atteindre un état où p est vraie en effectuant l'action a exactement une fois.	$\varphi = \langle a \rangle p$ 
Quelle que soit l'action a choisie, le successeur atteint vérifie $\neg p$ (négation de la propriété précédente). On note que la propriété est notamment vérifiée si il n'existe pas d'action a possible à partir d'un état.	$\varphi = \neg\langle a \rangle p = [a]\neg p$ 
On peut atteindre p en exactement 2 actions (par les séquences aa, ab, ba ou bb).	$\varphi = \langle a, b \rangle \langle a, b \rangle p$ 

TABLE 3.1 – Quelques formules et leur sémantique

Exemple 2 (Plus petit point fixe). On considère la formule close $\varphi = \mu X.\psi$ et un LTS $S = (Q, q_0, A, R, L)$. ψ contient une unique variable libre (X) donc ψ^S est une fonction de 2^Q vers 2^Q. Le fait que la formule ψ soit positive selon X assure que ψ^S est monotone :

$$X_1 \subseteq X_2 \subseteq Q \Rightarrow \psi^S(X_1) \subseteq \psi^S(X_2).$$

On a donc notamment :

$$\varnothing \subseteq \psi^S(\varnothing) \subseteq \psi^S(\psi^S(\varnothing)) \subseteq \dots$$

L'ensemble des parties de Q étant fini, cette suite croissante atteint une valeur maximale qui est le plus le plus petit point fixe de ψ^S. En effet, si $F \subseteq Q$ est un point fixe de ψ^S, on a $\emptyset \subseteq F$. ψ^S étant monotone on a pour tout $n \in \mathbb{N}, \psi^{S^n}(\emptyset) \subseteq \psi^{S^n}(F) = F$. En particulier, la limite de la suite est incluse dans F.

La table 3.2 représente les différentes étapes du calcul des états vérifiant la formule $\mu X.\psi$ dans le cas où $\psi = \neg p \vee \langle a \rangle X$. Cette formule exprime le fait qu'il existe un chemin composé uniquement d'actions a menant à un état vérifiant $\neg p$. En effet ψ^S est une fonction qui à un ensemble d'états X associe l'union des antécédents par a de X et des états où la variable p est fausse.

$$\psi^S(X) = \{y \in Q \mid y \xrightarrow{a} x \text{ et } x \in X\} \cup \{y \in Q \mid (\neg p)@y = \top\}.$$

On a alors :

$$\begin{aligned}
\psi^S(\emptyset) =& \{y \in Q \mid (\neg p)@y = \top\} \\
=& \text{ les états où la variable } p \text{ est fausse} \\
\psi^S(\psi^S(\emptyset)) =& \{y \in Q \mid y \xrightarrow{a} x \text{ et } x \in \psi^S(\emptyset)\} \cup \{y \in Q \mid (\neg p)@y = \top\} \\
=& \left\{ y \in Q \mid \begin{array}{l} y \xrightarrow{a} x \text{ et } (\neg p)@x = \top \\ \text{ou } (\neg p)@y = \top \end{array} \right\} \\
=& \text{ les états à partir desquels } \neg p \text{ est accessible en une action } a \text{ ou moins.}
\end{aligned}$$

Plus généralement, si

$$x \in \psi^{S^n}(\emptyset) \Leftrightarrow \text{il existe un chemin composé d'actions } a \text{ de taille } n-1 \text{ ou moins vers } \neg p,$$

alors :

$$\begin{aligned}
x \in \psi^{S^{n+1}}(\emptyset) \Leftrightarrow& \text{Soit } x \text{ vérifie } \neg p, \text{ soit il existe } x \xrightarrow{a} y \text{ avec } y \in \psi^{S^n}(\emptyset) \\
\Leftrightarrow& \text{ il existe un chemin composé d'actions } a \text{ de taille } n \text{ ou moins vers } \neg p.
\end{aligned}$$

Le plus petit point fixe de la fonction ψ^S correspond alors à la propriété : il existe un chemin composé de a menant vers un état où la variable p est fausse.

Exemple 3 (Une formule de profondeur d'alternance 2). Considérons la formule de CTL* $\exists\Box\Diamond p$, qui exprime l'existence d'un chemin sur lequel p est

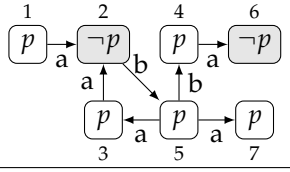
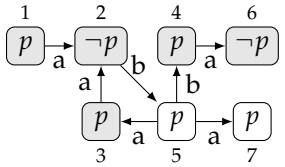
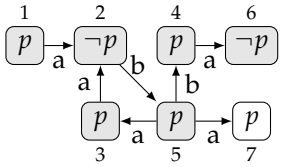
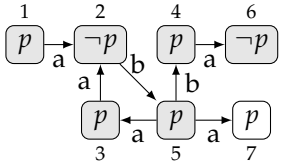
Propriété	Formule et sa sémantique sur un LTS donné. Les états grisés vérifient la formule.
Soit la formule $\psi = \neg p \vee \langle a \rangle X$, possédant une variable libre. On a $\psi(\bot) = (\neg p \vee \langle a \rangle \bot) = \neg p$, car $\langle a \rangle \bot = \bot$.	$\varphi = \psi(\bot) = \neg p$ 
On peut atteindre un état où p est faux en effectuant l'action a une ou zéro fois. $\psi^2(\bot) = \neg p \vee \langle a \rangle(\psi(\bot)) = \neg p \vee \langle a \rangle \neg p$	$\varphi = \psi^2(\bot) = \neg p \vee \langle a \rangle \neg p$ 
On peut atteindre un état où p est faux en effectuant l'action a deux fois ou moins. $\psi^3(\bot) = \neg p \vee \langle a \rangle(\psi^2(\bot)) = \neg p \vee \langle a \rangle \neg p \vee \langle a \rangle \langle a \rangle \neg p$	$\varphi = \psi^3(\bot)$ 
$\psi^4(\bot) = \psi^3(\bot) = \mu X.\psi$. On a atteint le plus petit point fixe.	$\varphi = \mu X.\psi$ 

TABLE 3.2 – Exemple de calcul d'un plus petit point fixe pour une formule d'accessibilité.

vraie infiniment souvent. Cette formule correspond à la formule de μ-calcul suivante :

$$\nu X.\mu Y.(\langle - \rangle Y \vee (p \wedge \langle - \rangle X)).$$

Une telle propriété nécessite l'utilisation de deux opérateurs de points fixes imbriqués. Elle est notamment différente de la formule de profondeur 1,

$$\nu X.(\langle - \rangle X \wedge (\mu Y.p \vee \langle - \rangle Y)),$$

qui est la traduction de la formule CTL $\exists\Box\exists\Diamond p$ et exprime le fait qu'il existe un

chemin infini sur lequel à tout moment, p est possible dans le future (mais pas nécessairement sur ce même chemin). La figure 3.3 exhibe un graphe illustrant la différence entre ces deux formules.

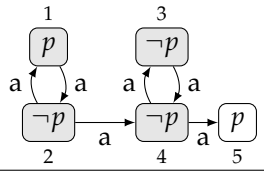
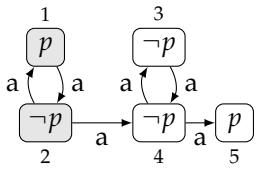
Propriété	Formule et sa sémantique sur un LTS donné. Les états grisés vérifient la formule.
$\exists\Box\exists\Diamond p$: Il existe un chemin infini sur lequel p est toujours possible dans le futur.	$\nu X.(\langle - \rangle X \wedge (\mu Y.p \vee \langle - \rangle Y))$ 
$\exists\Box\Diamond p$: Il existe un chemin sur lequel p est vrai infiniment souvent.	$\nu X.\mu Y.(\langle - \rangle Y \vee (p \wedge \langle - \rangle X)).$ 

TABLE 3.3 – Une formule de profondeur deux

3.3 Relations d'équivalence classiques

Maintenant que nous avons défini le μ-calcul, nous pouvons présenter plus en détails certaines des relations d'équivalences classiques mentionnées en section 2.4 ainsi que les différentes propriétés temporelles préservées par celles-ci.

Ces relations sont définies sur les états d'un LTS, on peut cependant les étendre à des relations sur les LTS comme suit : deux LTS sont équivalents si et seulement si leurs états initiaux sont équivalents lorsque l'on considère l'union disjointe des deux LTS. C'est-à-dire qu'on les représente côte à côte comme s'il s'agissait d'un unique graphe non connexe, sur lequel on sait tester l'équivalence des deux états initiaux.

La figure 3.1 comporte des diagrammes représentant visuellement les caractéristiques de quelques relations d'équivalences classiques, ainsi que les sous-logiques du μ-calcul coïncidant avec ces relations. Ces diagrammes dits de transferts représentent des conditions nécessaires pour qu'une relation

sur les états d'un LTS appartienne à une catégorie donnée. Dans cette figure, $x, y, \ldots$ représentent des états du LTS, b représente une action quelconque (possiblement τ) et a une action visible (différente de τ). La ligne correspondant à la bisimulation forte se lit ainsi : une relation $\mathcal{R}$ est une bisimulation forte si elle est symétrique et vérifie la propriété suivante ; pour tout couple (x, y) de $\mathcal{R}$ et toute transition $x \xrightarrow{b} x'$, il existe une transition $y \xrightarrow{b} y'$ telle que (x', y') appartient également à $\mathcal{R}$. Les autres diagrammes se lisent de façon similaire.

La figure 3.2 comporte des exemples caractéristiques de graphes équivalents pour certaines relations ainsi qu'un ordre représentant les inclusions entre ces différentes relations. Par exemple si deux LTS sont fortement bisimilaires, ils seront aussi équivalents pour toutes les autres relations présentées ici. Lors d'une réduction, on peut utiliser un tel ordre pour choisir une relation identifiant le plus de LTS possibles (et qui préserve les propriétés d'intérêt). En effet la taille d'un plus petit graphe d'une classe d'équivalence diminue lorsque la taille de la classe augmente.

Cette figure considère en outre trois relations qui ne sont pas mentionnée en figure 3.1. Informellement, l'équivalence de trace (ou de langage) considère deux états équivalents si il est possible d'effectuer les mêmes séquences d'actions (mots) à partir des ces états, et l'équivalence de trace faible [18] est similaire mais ignore les actions invisibles τ. L'équivalence SAFETY [16] est définie formellement au chapitre 4 (page 59) car on l'utilise par la suite.

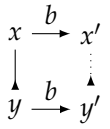
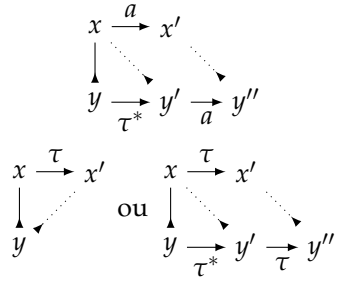
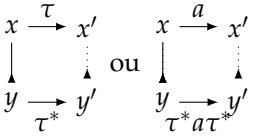
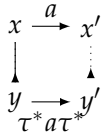
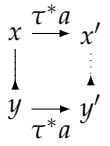
Bissimulation forte [80]	
μ-calcul [80]	$\varphi ::= \top \mid \neg\varphi \mid \varphi \vee \varphi \mid \langle b \rangle \varphi \mid \mu X.\varphi \mid X.$
Équivalence de branchement [106]	
μ-calcul avec Until [37]	$\varphi ::= \top \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \langle b \rangle_u \varphi \mid \mu X.\varphi \mid X.$
Équivalence observationnelle [74]	
	$\varphi ::= \top \mid \neg\varphi \mid \varphi \vee \varphi \mid \langle b \rangle_o \varphi \mid \mu X.\varphi \mid X.$
Bisimulation faible [37]	
	$\varphi ::= \top \mid \neg\varphi \mid \varphi \vee \varphi \mid \langle a \rangle_o \varphi \mid \mu X.\varphi \mid X.$
Équivalence $\tau^* a$ [42]	
μ-calcul sélectif [8]	$\varphi ::= \top \mid \neg\varphi \mid \varphi \vee \varphi \mid \langle a \rangle_s \varphi \mid \mu X.\varphi \mid X.$

FIGURE 3.1 – Quelques équivalences classiques et les logiques associées.

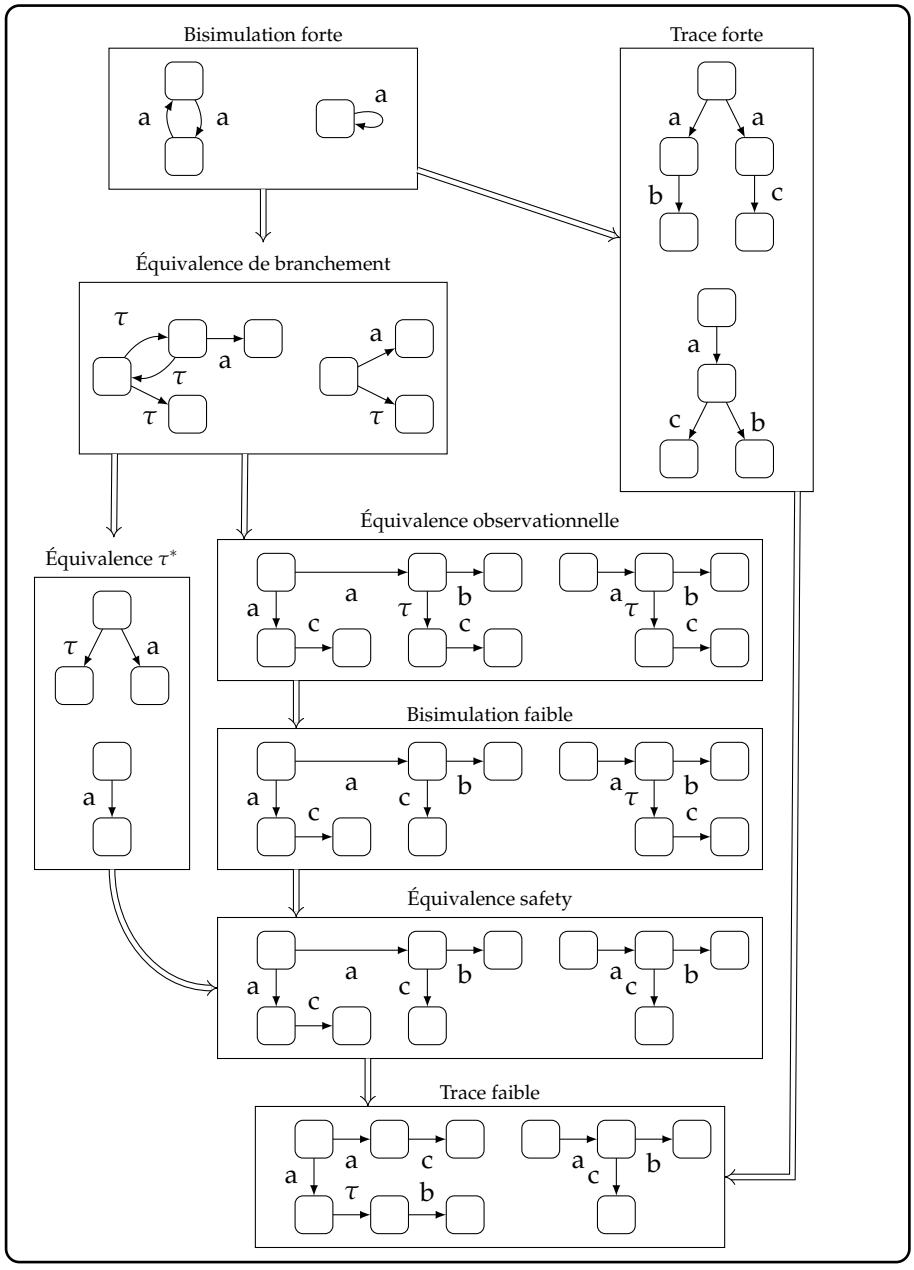


FIGURE 3.2 – Comparaison de différentes relations d'équivalences classiques et exemples représentatifs de ces relations. La bisimulation forte est ici l'équivalence la plus fine, c'est-à-dire que si deux graphes sont fortement bisimilaires, ils sont aussi équivalents pour toutes les relations représentées ici.

Chapitre 4

Application à l'analyse de réseaux de régulation biologique

La modélisation discrète des réseaux de régulation génétique et de signalisation cellulaire s'avère être très utile à l'analyse des systèmes biologiques comme en témoigne le nombre croissant de publications présentant de tels modèles [21, 41, 79]. On s'intéresse en particulier aux modèles définis par René Thomas [103, 102] dont on présente informellement un exemple jouet en figure 4. Ce réseau comporte trois composants (c_1, c_2 et c_3) qui décrivent l'état du système. Chacun prend des valeurs parmi un ensemble discret et fini et représente une caractéristique du système, par exemple la concentration d'une certaine protéine ou le fait qu'un gène soit exprimé ou non. On dessine un arc entre deux composants si le premier a une *influence* sur le deuxième, c'est-à-dire qu'il apparaît dans la fonction de régulation de celui-ci. La fonction de régulation d'un composant détermine la prochaine valeur de ce composant en fonction de l'état courant si on fait évoluer ce composant. De plus, le comportement du réseau considère une évolution asynchrone où les composant évoluent un à un. Sur l'exemple à partir de l'état (0,1,0), on peut faire évoluer c_2 et atteindre l'état (0,0,0) car selon sa fonction de régulation, la nouvelle valeur de c_2 est la valeur de c_3. On peut également modifier c_1 qui prend alors la valeur 1. Le composant c_3 est en revanche stable dans l'état (0,1,0) : si on note p cet état, on à $K_{c_3}(p) = p[c_3]$. Les états (0,0,0) et (1,1,1) sont appelés *états stables* car aucun composant ne peut évoluer à partir de ces états. Ils sont remarquables et souvent d'intérêts dans l'étude des systèmes biologiques. Par exemple si le réseau modélise un processus de différentiation cellulaire, différents états stables peuvent représenter les différents types de cellules pouvant

être obtenues à la fin de ce processus.

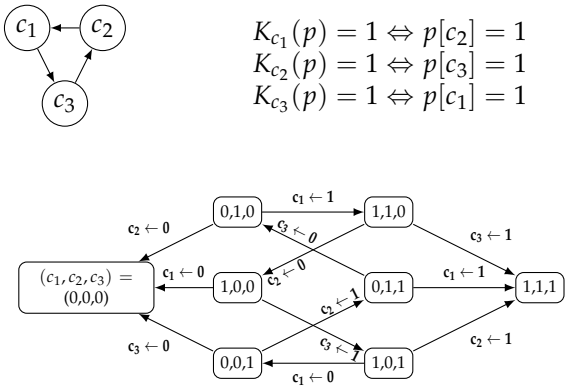


FIGURE 4.1 – Un réseau de régulation non modulaire et son comportement. En haut à gauche : la structure du réseau. En haut à droite : les fonctions de régulation définissant son comportement. En bas : le LTS représentant le comportement du réseau.

Malheureusement, le problème d'explosion combinatoire des espaces d'états est particulièrement présent dans ce domaine, ce qui rend difficile l'analyse de réseaux de grandes tailles et notamment de systèmes multicellulaires. En revanche, ces systèmes multicellulaires sont naturellement modulaires (on peut considérer une cellule comme un module) et une analyse par abstraction hiérarchique s'avère être une méthode efficace pour la vérification de certaines propriétés comme l'accessibilité des états stables du système. Dans ce chapitre, on commence par définir un formalisme adapté à la description de systèmes multicellulaires (un réseau de régulation modulaire), on montre ensuite comment construire le LTS représentant le comportement d'un tel système comme un produit du comportement de ses modules. Ce produit est adapté à une étude par abstraction hiérarchique, et on propose en particulier une méthode de vérification de la propriété d'accessibilité des état stables globaux du système à partir d'un état initial particulier. Cette approche est finalement appliquée à l'étude de la polarisation segmentaire lors du développement embryonnaire de la drosophile.

4.1 Modèles de régulation composable

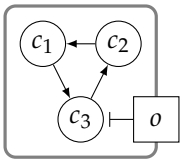
4.1.1 Réseaux ouverts

Dans le cas ouvert, on distingue les composants propres qui sont les composants réels du réseau et les composants d'entrées qui représentent des composants extérieurs au réseau ayant une influence sur des composants propres (comme la présence d'une protéine produite par les cellules voisines ou encore une variable globale comme la température). Le comportement du réseau est défini par un ensemble de fonctions de mise à jour décrivant comment peuvent évoluer les composants propres en fonction de l'état général du système. Les composants d'entrées peuvent évoluer librement pour simuler les changements possible de l'environnement.

Définition 7 (Module de régulation génétique)**.** $N \stackrel{\text{df}}{=} (C, K)$ où :

- C est l'ensemble des composants du module. Pour chaque $c \in C$ on note D_c l'ensemble fini des valeurs possibles de ce composant. On note également $D_C = \Pi_{c \in C} D_c$ les états possibles du module. C est partitionné en deux ensembles $C_p \uplus C_o$ où
  - C_P contient les composants propres qui composent effectivement le module,
  - C_O contient les composants ouverts représentant l'effet de l'environnement sur le module ;
- $K \stackrel{\text{df}}{=} \{K_c \mid c \in C_p\}$ est l'ensemble des fonctions de régulation. $K_c : D_C \to D_c$ définit l'évolution du composant c en fonction de l'état du module.

Un exemple jouet est présenté figure 4.2 dans lequel les composants propres du réseau s'activent les uns les autres, alors que le composant ouvert o inhibe le composant propre c_3. Le LTS représentant le comportement de ce module (définition 10) est dessiné figure 4.4.



$$K_{c_1}(p) = 1 \Leftrightarrow p[c_2] = 1$$
$$K_{c_2}(p) = 1 \Leftrightarrow p[c_3] = 1$$
$$K_{c_3}(p) = 1 \Leftrightarrow p[c_1] = 1 \wedge p[o] = 0$$

FIGURE 4.2 – Un module de régulation.

4.1.2 Composition des modules

Un réseau de régulation modulaire est constitué d'une part de l'ensemble de ses modules, et d'autre part d'un mapping (noté M) explicitant pour chaque composant ouvert, sa réelle valeur en fonction de l'environnement.

Définition 8 (Réseau de régulation modulaire)**.** Un réseau est un couple $((N^i)_{i\in I}, M)$. Comme précédemment, pour chaque composant $c \in \bigcup_I C^i$, on note D_c l'ensemble des valeurs possibles du composant c. Pour chaque module $j \in I$ et chaque composant ouvert $o \in C^j_O$ de ce module, on définit $M(o) = (Z_o, h_o)$ où :

- $Z_o \subseteq \bigcup_{I\setminus j} C^i_P$ est l'ensemble des composants propres de l'environnement servant à définir la valeur de o. Z_o peut être égal à l'ensemble vide, auquel cas o restera un composant ouvert. On considère également que si deux composants ouverts o et o' appartiennent au même module, les ensembles Z_o et $Z_{o'}$ sont disjoints : un même composant externe n'a pas d'influence sur deux composants ouverts d'un même module ;
- $h_o : \prod_{z\in Z_o} D_z \mapsto D_o$ précise (à tout moment) la valeur de o en fonction de la valeur des composants de Z_o.

Il serait possible d'autoriser un composant propre à avoir une influence sur deux composants ouverts d'un même module, mais cela compliquerait quelque peu les définitions et preuves par la suite. En effet, il faudrait alors autoriser ces composants ouverts à se mettre à jour de manière synchrone, pour représenter une modification du même composant externe, alors que les composants propres sont mis à jour de manière asynchrone. Ce choix n'est cependant pas une limitation en pratique : il nous empêche de définir la composition de modules de régulation partageant l'influence d'un même composant externe, mais pas de composer leurs comportements, ce qui permet tout de même l'analyse modulaire et incrémentale qui nous intéresse ici.

Définition 9 (Application du mapping)**.** Étant donné un réseau modulaire $((N^i)_{i\in I}, M)$, l'application du mapping M consiste à composer les différents modules du réseau en un, à l'aide de l'information fournie par M. L'opération consiste notamment à retirer les composant ouverts qui ne sont plus nécessaires car leur environnement est connu après composition. L'application du mapping M produit un module de régulation génétique (C, K) tel que :

- $C = C_P \cup C_O$ avec :
  - $C_P = \bigcup_I C^i_P$ est l'union des composants propres des modules.

- $C_O = \{o \in \bigcup_I C_o^i \mid Z_o = \emptyset\}$ est l'ensemble des composants ouverts des modules dont la valeur n'est pas spécifiée par M (car celle-ci dépend potentiellement de composants externes au réseau modulaire) ;

- pour tout $i \in I$, tout composant propre $p \in C_P^i$ et tout état $q \in D_C$, $K_p(q) = K_p^i(q^i)$ où $q^i \in D_{C^i}$ est tel que :
  - $q^i[c] = q[c]$ si c appartient à C
  - $q^i[o] = h_o(q_{|Z_o})$ sinon (o était un composant ouvert du module i qui a été supprimé).

Si l'on compose l'exemple jouet avec lui-même à l'aide du mapping M qui associe au composant ouvert o_1 la valeur du composant propre c_{22} (et qui laisse le composant o_2 ouvert), on obtient le module de régulation de la figure 4.3.

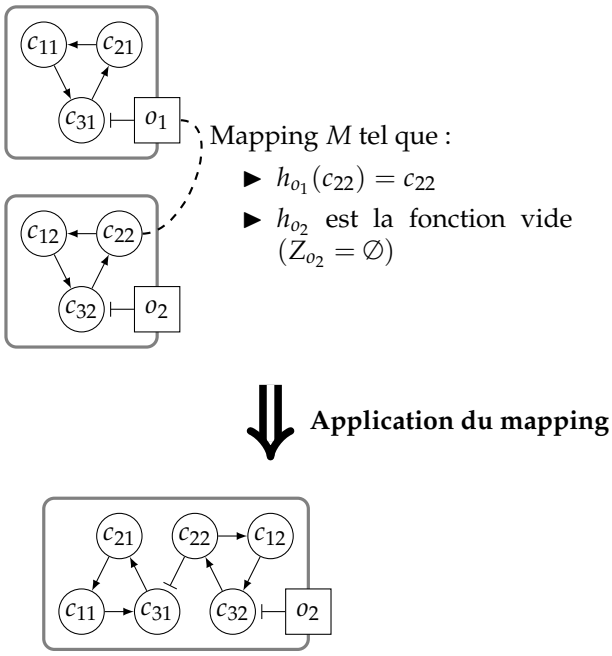


FIGURE 4.3 – composition de deux instances de l'exemple jouet

4.1.3 Comportement d'un module de régulation

Le comportement d'un module de régulation est un LTS représentant une évolution asynchrone des composants du module dans laquelle, à partir d'un état p, un composant propre c doit évoluer vers la valeur $K_c(p)$, et un composant ouvert peut évoluer vers toutes les valeurs de son domaine pour simuler l'évolution de l'environnement.

Définition 10 (Comportement d'un module de régulation)**.** Le comportement du module (C, K) est le LTS (D_C, A, R, L). Celui ci peut exécuter l'action (c, v) si et seulement si le composant c peut évoluer vers la valeur v. On a donc :

- l'ensemble des actions réalisables est $A = \{(a, v) \mid a \in C, v \in D_a\}$. Le couple (a, v) est parfois noté $a \leftarrow v$ par la suite (le composant a prend la valeur v) ;
- la transition $(p, (c, v), q)$ appartient à R si et seulement si les deux conditions suivantes sont vérifiées :
  - **[1]** $q[c] = v$ (le composant c prend la valeur v. De plus, si c est un composant propre il faut que $v = K_c(p)$ et si c est un composant ouvert il faut juste que $v \neq p[c]$.
  - **[2]** $q[c'] = p[c']$ pour $c' \neq c$ (les autres composants n'évoluent pas) ;
- notons $x_{c,v}$ la variable booléenne indiquant que v est la valeur du composant c et $\mathbb{V} = \{x_{c,v} \mid c \in C, v \in D_c\}$. $L : D_C \mapsto 2^{\mathbb{V}}$ associe à chaque état les variables booléennes actives dans cet état. Pour tout état q, $L(q) = \{x_{c,q[c]} \mid c \in C\}$.

Le graphe d'états obtenu pour le module jouet (figure 4.2) est dessiné figure 4.4. Ceux des modules de la figure 4.3 apparaissent figure 4.5.

4.1.4 Produit des comportements

Pour construire le LTS correspondant au comportement d'un réseau de régulation modulaire, on peut fusionner les différents modules le composant en appliquant le mapping associé puis calculer le comportement du module obtenu. On présente maintenant une seconde définition équivalente de ce comportement sous la forme d'un réseau de LTS.

On commence par définir la notion de module d'intégration. Pour chaque fonction h_o du mapping (représentant la valeur du composant ouvert o en fonction des composants de Z_o), on construit un nouveau LTS (noté $\mathsf{lts}(h_o)$)

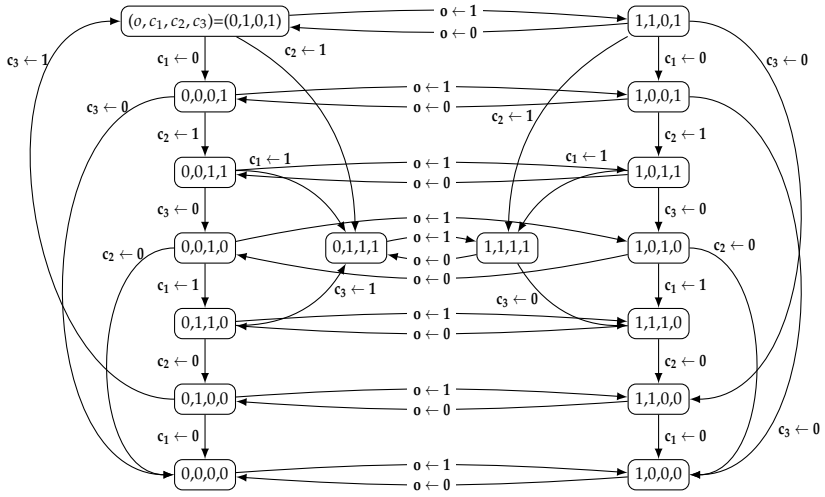


FIGURE 4.4 – Comportement du module jouet.

servant à assurer la synchronisation entre les valeurs de o et des éléments de Z_o.

Définition 11 (module d'intégration)**.** Étant donnée une fonction d'intégration h_o et son domaine Z_o, le LTS $\mathsf{lts}(h_o)$ assure la liaison entre le composant ouvert o et les composants externes qu'il représente (Z_o). $\mathsf{lts}(h_o) = (Q, A, R)$ où :

- $Q = \prod_{z \in Z_o} D_z$ est l'ensemble des états possibles des éléments de Z_o ;
- $A = \{p \leftarrow v_p \mid p \in Z_o, v_p \in D_p\} \times (\{o \leftarrow v_o \mid v \in D_o\} \cup \{\_\})$. La première partie de l'action représente la mise à jour d'un composant p de Z_o, la deuxième partie correspond à la mise à jour de la valeur de o si celle si est affectée par la modification de p (et _ si la valeur de o ne change pas) ;
- R est le plus petit sous-ensemble de $Q \times A \times Q$ tel que :
  - $p \overset{(c \leftarrow v_c, a)}{\rightarrow} q$ appartient à R si les conditions suivantes sont vérifiées :
    - $q[c] = v_c \neq p[c]$; le composant c prend la valeur v_c pour passer dans l'état q,
    - pour tout $c' \neq c, q[c'] = p[c']$; les autres composants ne changent pas,

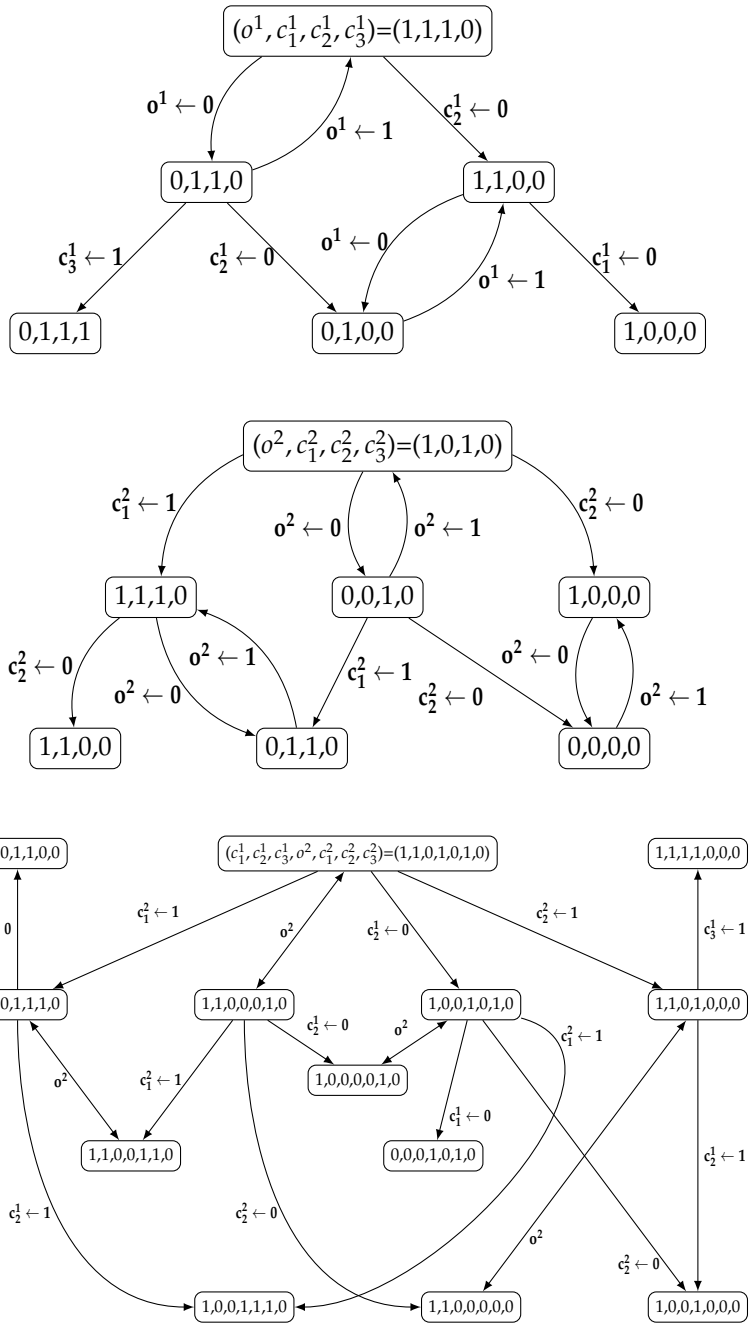
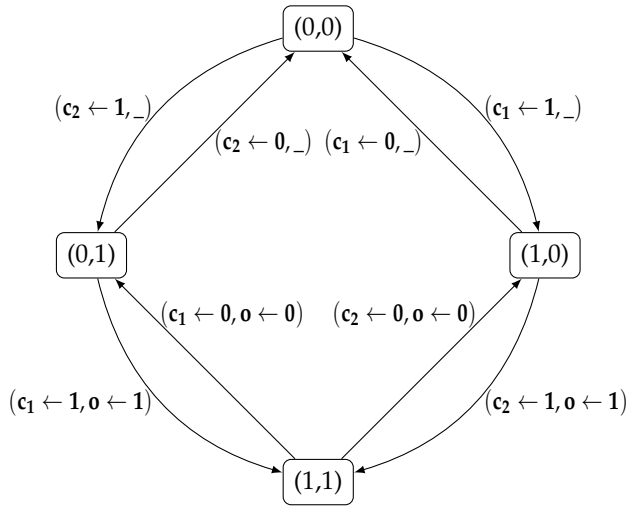


FIGURE 4.5 – Les comportements des modules de régulation de la figure 4.3. En haut : le comportement tronqué du premier module à partir de l'état $(1, 1, 1, 0)$. Au milieu : le comportement tronqué du second module à partir de l'état $(1, 0, 1, 0)$. En bas : le comportement du module fusionné à partir de l'état $(1, 1, 0, 1, 0, 1, 0)$.



L_1	L_2	L_3	$\mathsf{lts}(h_o)$	LTS produit
•	$c_2 \leftarrow 0$	•	$(c_2 \leftarrow 0, \_)$	$c_2 \leftarrow 0$
•	$c_2 \leftarrow 1$	•	$(c_2 \leftarrow 1, \_)$	$c_2 \leftarrow 1$
•	$c_2 \leftarrow 0$	$o \leftarrow 0$	$(c_2 \leftarrow 0, o \leftarrow 0)$	$c_2 \leftarrow 0$
•	$c_2 \leftarrow 1$	$o \leftarrow 1$	$(c_2 \leftarrow 1, o \leftarrow 1)$	$c_2 \leftarrow 1$
$c_1 \leftarrow 0$	•	•	$(c_1 \leftarrow 0, \_)$	$c_1 \leftarrow 0$
$c_1 \leftarrow 1$	•	•	$(c_1 \leftarrow 1, \_)$	$c_1 \leftarrow 1$
$c_1 \leftarrow 0$	•	$o \leftarrow 0$	$(c_1 \leftarrow 0, o \leftarrow 0)$	$c_1 \leftarrow 0$
$c_1 \leftarrow 1$	•	$o \leftarrow 1$	$(c_1 \leftarrow 1, o \leftarrow 1)$	$c_1 \leftarrow 1$

FIGURE 4.6 – Exemple de module d'intégration. Ce module sert à synchroniser les composants o, c_1 et c_2, de façon à avoir en permanence $o = c_1 \wedge c_2$. Les règles de synchronisation correspondant aux mises à jour de c_1 et c_2 sont données en supposant que c_1,c_2 et o appartiennent respectivement aux modules 1, 2 et 3.

- si $a = (o \leftarrow v_o)$ alors $h_o(q) = v_o \neq h_o(p)$; la valeur de o change entre les états p et q. La nouvelle valeur est indiquée dans le nom de l'action,
- si $a = \_$ alors $h_o(q) = h_o(p)$; la modification de c ne change pas la valeur de o.

La figure 4.6 est un exemple de module d'intégration permettant d'assurer que la valeur de o soit bien égale à la conjonction de c_1 et de c_2. On le synchronise avec le reste du système à l'aide d'un vecteur de synchronisation adapté comme indiqué dans la définition suivante.

Définition 12 (Réseau de LTS correspondant à un réseau modulaire)**.** Soit un réseau de régulation modulaire $((N^i)_{i\in I}, (Z, h))$, Soit $\overline{O}$ l'ensemble des composants ouverts $o \in \bigcup_{i\in I} C_O^i$ tels que $Z_o \neq \emptyset$. On note également par $\overline{HO}$ l'ensemble des fonctions d'intégrations h_o correspondantes. Soit $S = \{\mathsf{lts}(N^i) \mid i \in I\} \cup \{\mathsf{lts}(h_o) \mid o \in \overline{O}\}$ un LTS modulaire indexé par $I \cup \overline{HO}$, qui est constitué des comportements de chaque module de régulation génétique ainsi que des graphes d'intégration faisant la liaison entre ces modules. Soit V le vecteur de synchronisation définissant le comportement global du réseau. $V = \bigcup_{c\in C\setminus\overline{O}} V_c$ où $(v, c \leftarrow x) \in V_c$ si et seulement si les conditions suivantes sont vérifiées :

- $x \in D_c$
- $v[i] = \begin{cases} c \leftarrow x & \text{si } c \in C^i \setminus \overline{O} \\ o \leftarrow y & \text{si } o \in C_O^i, c \in Z_o \text{ et } v[h_o] = (c \leftarrow x, o \leftarrow y) \\ \bullet & \text{sinon} \end{cases}$
- $v[h_o] = \begin{cases} (c \leftarrow x, \_) \text{ ou } (c \leftarrow x, o \leftarrow y) & \text{si } c \in Z_o \text{ et } y \in D_o \\ \bullet & \text{sinon} \end{cases}$

  Le graphe d'intégration représentant h_o se synchronise avec la mise à jour de c dans le cas où c est un argument de la fonction h_o ($c \in Z_o$). Il se synchronise également avec la modification de o si la mise à jour de c entraîne un changement de la valeur de o.

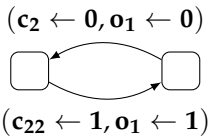
On peut maintenant calculer le comportement du module fusionné (figure 4.5) comme le comportement du réseau de LTS $((L_1, L_2, \mathsf{lts}(h_o)), V)$ où L_1 et L_2 sont les LTS représentés partiellement également figure 4.5. Le vecteur de synchronisation V et le module d'intégration $\mathsf{lts}(h_{o_1})$ sont dessinés figure 4.7.

Le théorème suivant confirme que ces deux méthodes de construction du comportement du système global sont bien équivalentes.

Théorème 1. Le comportement d'un réseau modulaire est isomorphe au comportement du réseau de LTS correspondant.

4.2 Analyse de propriétés d'accessibilité

On veut ici appliquer une approche hiérarchique et incrémentale dans le but d'identifier les états stables d'un réseau de régulation génétique, qui sont accessibles à partir d'un état initial donné. On travaille sur le réseau de LTS correspondant. Pour appliquer des réductions hiérarchiques lors de l'analyse du réseau, il faut déterminer quelle opération d'abstraction utiliser, ou plus



L_1	L_2	$\mathsf{lts}(h_{o_1})$	LTS produit
$c_{11} \leftarrow 0$	•	•	$c_{11} \leftarrow 0$
$c_{11} \leftarrow 1$	•	•	$c_{11} \leftarrow 1$
$c_{21} \leftarrow 0$	•	•	$c_{21} \leftarrow 0$
$c_{21} \leftarrow 1$	•	•	$c_{21} \leftarrow 1$
$c_{31} \leftarrow 0$	•	•	$c_{31} \leftarrow 0$
$c_{31} \leftarrow 1$	•	•	$c_{31} \leftarrow 1$
•	$c_{12} \leftarrow 0$	•	$c_{12} \leftarrow 0$
•	$c_{12} \leftarrow 1$	•	$c_{12} \leftarrow 1$
•	$c_{32} \leftarrow 0$	•	$c_{32} \leftarrow 0$
•	$c_{32} \leftarrow 1$	•	$c_{32} \leftarrow 1$
•	$o_2 \leftarrow 0$	•	$o_2 \leftarrow 0$
•	$o_2 \leftarrow 1$	•	$o_2 \leftarrow 1$
$o_1 \leftarrow 0$	$c_{22} \leftarrow 0$	$(o_1 \leftarrow 0, c_{22} \leftarrow 0)$	$c_{22} \leftarrow 0$
$o_1 \leftarrow 1$	$c_{22} \leftarrow 1$	$(o_1 \leftarrow 1, c_{22} \leftarrow 1)$	$c_{22} \leftarrow 1$

FIGURE 4.7 – Module d'intégration utilisé pour calculer le comportement globale de l'exemple en tant que reseau de LTS, ainsi que le vecteur de synchronisation associé.

généralement, quelles sont les informations présentes dans une sous-partie du réseau qu'il est nécessaire de conserver lors de la réduction.

On a en fait besoin de deux choses :

[1] Les actions du sous-système qui doivent se synchroniser avec son environnement, car elles sont essentielles au calcul du comportement global du réseau. Il s'agit ici des mises à jour des composants ayant une influence sur des modules extérieurs et des mises à jours des composants ouverts.

[2] On doit conserver l'accessibilité aux états du sous-système dans lesquels les composants propres ne peuvent pas évoluer. On les appelle des états faiblement stables car ils font possiblement partie d'un état stable global.

Comme vu précédemment, la première étape de la réduction d'un sous-système consiste à *cacher* un ensemble de composants dont les valeurs n'ont pas d'influence sur la propriété à vérifier. Cela consiste à remplacer dans les LTS les actions correspondant à leurs mises-à-jour par l'action invisible τ.

La seconde étape de la réduction d'un LTS consiste à construire un second LTS de plus petite taille et équivalent vis-à-vis d'une relation préservant la propriété d'intérêt. Ici, on veut identifier l'ensemble des états stables accessibles à partir d'un état initial donné. La réduction SAFETY (définie par la suite) est une bonne candidate car elle permet de supprimer toutes les transitions τ et les chemins redondants, tout en préservant les propriétés d'accessibilités. Elle ne préserve en revanche pas la stabilité des états, car il s'agit d'une propriété nécessitant de mentionner explicitement les actions invisibles (car un état où τ est possible n'est pas stable), et l'équivalence SAFETY n'est pas adaptée pour ces propriétés. En conséquence, la suppression de *self-loops invisibles* (les transitions de la forme $x \xrightarrow{\tau} x$) peut faire apparaître de nouveaux états stables et on perd également les états stables accessibles uniquement à l'aide de transitions invisibles.

On peut en revanche modifier légèrement le modèle en ajoutant d'une part des self-loops étiquetées $\perp$ sur les tous les états faiblement stables des modules, et d'autre part en synchronisant cette action $\perp$ entre tous les modules. Sur ce nouveau modèle, on doit préserver l'accessibilité de l'action $\perp$ sur le système global, ce qui est équivalent à préserver l'accessibilité des états stables du réseau mais ne fait pas intervenir l'action τ explicitement.

L'opération de réduction d'une sous partie du système s'effectue donc ainsi :

[1] On cache tous les composants propres n'ayant pas d'influence externe, c'est-à-dire qu'on remplace leurs actions de mise à jour par τ. Si on s'intéresse à la valeur de certains composants en particulier on peut les laisser visibles. Cela est notamment le cas si la valeur du composant sert à déterminer quel état stable en particulier est accessible.

[2] On réduit le LTS à l'aide de la réduction SAFETY, qui préserve l'accessibilité de ces états stables.

4.2.1 Équivalence SAFETY et minimisation

Une propriété de sûreté (safety) affirme qu'aucun évènement considéré négatif n'est susceptible de se produire dans le futur. Cette première définition intuitive est due à Lamport [63] mais les propriétés de sûreté ont par la suite été définies plus formellement et étudiées en tant qu'ensemble de séquence d'exécutions (*linear time semantics* [98] [2]) et en tant qu'ensemble d'arbre d'exécution (*branching time semantics* [16]). Dans [16], Bouajjani *et al*. définissent également l'équivalence SAFETY qui assure que deux LTS équi-

valents vérifient exactement les mêmes propriétés de sûreté. Dans notre cas notamment, l'accès aux états où l'action $\perp$ est disponible sera préservé par la réduction.

Définition 13 (équivalence SAFETY)**.** Un état p simule un état q pour la relation τ^* si et seulement si pour chaque séquence d'actions $q \stackrel{\tau^* a}{\rightarrow} q'$ (avec $a \neq \tau$), on peut trouver une séquence $p \stackrel{\tau^* a}{\rightarrow} p'$ telle que q' simule p'. Les états p et q sont alors équivalents pour la relation SAFETY si et seulement si p simule q et q simule p pour la relation τ^*. Comme précédemment, deux LTS sont dits équivalents si et seulement si leurs états initiaux sont équivalents.

La figure 4.8 présente un exemple de cette réduction appliqué au comportement du module jouet.

4.2.2 Application au module de polarité segmentaire

Le module de polarité segmentaire intervient lors du développement embryonnaire de la drosophile, qui a été beaucoup étudié en génétique. Il s'agit d'un système multicellulaire dont chaque cellule peut être modélisée par un module de régulation génétique. L'objectif étant d'étudier le système à l'aide de techniques d'abstractions hiérarchiques.

Au stade du développement qui nous intéresse l'embryon est organisé en segments (chacun constitué de plusieurs cellules) orientés selon l'axe antéro-postérieur de la drosophile. Les gènes de polarité segmentaire interviennent dans la différenciation des cellules situées à l'avant et à l'arrière des segments, phénomène qui a pour effet de consolider les bordures de ces segments. On utilise un modèle défini par Sánchez *et al*. [94] et dont un module est représenté en figure 4.9. Dans ce modèle, chaque cellule est constituée de douze composants, parmi lesquels deux peuvent influer sur les cellules voisines : Hedgehog (Hh) et Wingless (Wg). Un troisième composant dénommé Engrailed (En) nous intéresse particulièrement car, combiné avec Wg, il permet d'identifier les trois états stables possibles d'un module, qui correspondent à trois différents types possibles de la cellule. L'état W où Wingless est exprimé, l'état E où Engrailed est exprimé et enfin l'état T où l'on a ni l'un ni l'autre.

Pour réduire une sous-partie du système (à une étape donnée de l'analyse hiérarchique), on va donc ici dissimuler tous les composants différents de En et de Wg, dont la mise à jour est une action interne au système avant d'effectuer une réduction selon l'équivalence SAFETY. Si on considère par exemple un sous-système composé de trois cellules adjacentes, le composant Hh de la

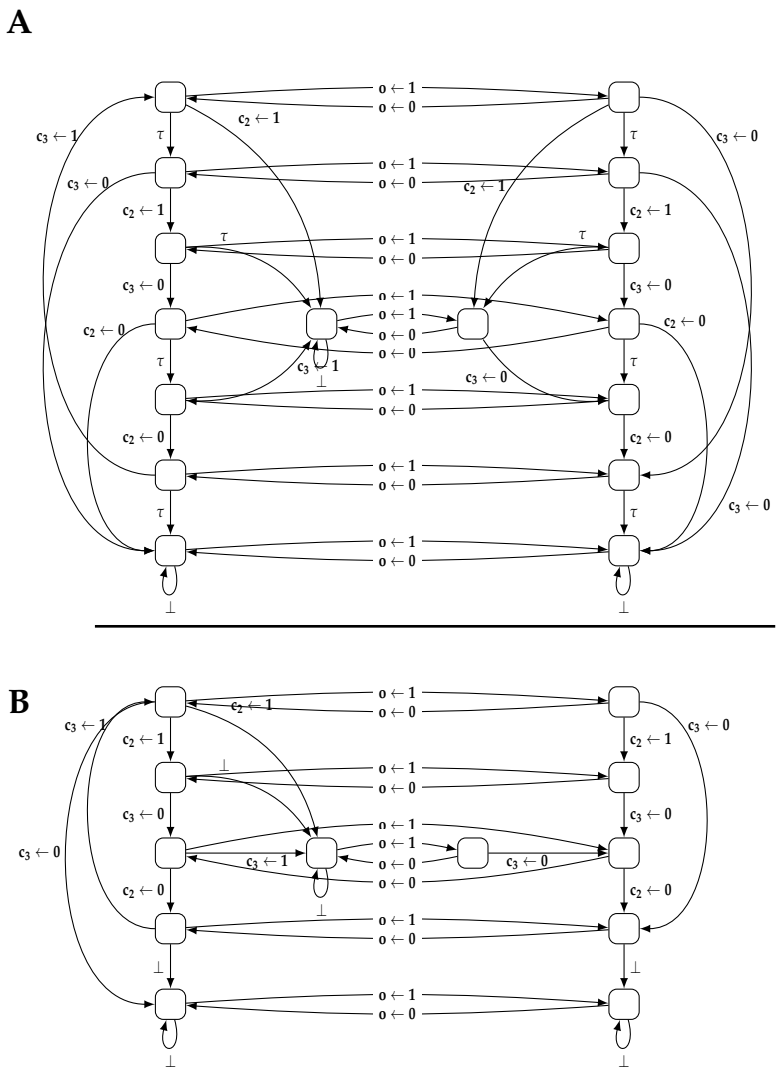


FIGURE 4.8 – Réduction à l'aide de l'équivalence SAFETY du comportement du module jouet de la figure 4.4. (A) Le LTS après la dissimulation du composant c_1. (B) Le LTS réduit équivalent pour la relation SAFETY.

cellule centrale peut être caché car il n'a d'influence que sur les cellules voisines et est donc local si l'on considère le produit des trois cellules. On ne cache par contre pas Wg car même si c'est un composant local, sa valeur est utile à l'identification des états stables trouvés.

Afin de tester la cohérence de la méthode on considère en premier lieu un

$$K_{W_g}(\{Slp,1\},\{Ci[act],1\}) = 1$$
$$K_{W_g}(\{Slp,1\},\{Ci[act],2\}) = 2$$
$$K_{W_g}(\{Slp,1\},\{Ci[act],2\},\{Nkd,2\}) = 2$$
$$K_{F_z}(\{W_g,[1,2]\},\{Wg_{input},[0,1,2]\}) = 1$$
$$K_{F_z}(\{Wg_{input},2\}) = 1$$
$$K_{Dsh}(\{F_z,1\}) = 1$$
$$K_{Slp}(\{Dsh,1\}) = 1$$
$$K_{En}(\{Dsh,1\}) = 1$$
$$K_{En}(\{Dsh,1\},\{En,1\}) = 1$$
$$K_{En}(\{Dsh,1\},\{Nkd,[1,2]\}) = 1$$
$$K_{En}(\{Dsh,1\},\{En,1\},\{Nkd,[1,2]\}) = 1$$

$$K_{Nkd}(\emptyset) = 1$$
$$K_{Nkd}(\{Dsh,1\}) = 2$$
$$K_{Hh}(\{En,1\}) = 1$$
$$K_{Ptc}(\emptyset) = 1$$
$$K_{Ptc}(\{Ci[act],[1,2]\}) = 2$$
$$K_{Pka}(\{Ptc,1\}) = 2$$
$$K_{Ci}(\emptyset) = 1$$
$$K_{Ci[act]}(\{C_i,1\}) = 1$$
$$K_{Ci[act]}(\{C_i,1\},\{Dsh,1\},\{Pka,[1,2]\}) = 1$$
$$K_{Ci[act]}(\{C_i,1\},\{Dsh,1\}) = 2$$
$$K_{Ci[rep]}(\{C_i,1\},\{Pka,2\}) = 1$$

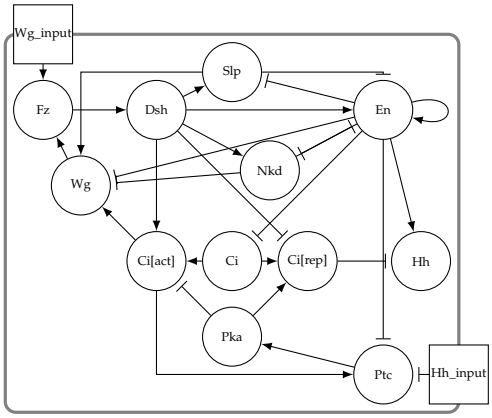


FIGURE 4.9 – Un module du modèle de polarité segmentaire. Les fonctions de régulations sont représentée à la manière de [94] pour des raisons de concisions. Les composants n'apparaissant pas dans une règle mais ayant une influence d'après le graphe sont de valeur nulle. Ainsi la première règle indique que dans le cas où les composants Slp et Ci[atc] sont égaux à 1, et En et Ndk sont égaux à 0, Wg peut prendre la valeur 1. De plus, si un état n'est pas pris en compte par ces règles, le composant peut prendre la valeur 0 (par exemple si En est actif, Wg prendra la valeur 0 s'il est mis à jour).

système composé de deux modules dont les états stables sont connus. Dans ce réseau, les composants Hh et Wg du premier module influencent le second, et réciproquement. Le LTS obtenu possède 9701 états et a nécessité presque sept heures de calcul et 28 GO de mémoire. On retrouve bien les motifs attendus (WE, TT et EW) apparaissant dans la littérature [94]. C'est en outre la première fois à notre connaissance que l'espace d'état est construit en entier (dans [93] la construction est interrompue après la découverte des états TT et EW).

Afin d'étudier l'impact de l'ordre de composition choisi lors de la construction hiérarchique, on considère un système constitué de six modules placés sur une ligne. Chaque module possède deux voisins à l'exception de ceux placés aux extrémités. On construit pour chaque module N^i, trois LTS différents : M^i le LTS représentant le comportement réduit du module, $H^i(Hh)$ et $H^i(Wg)$ les modules d'intégration correspondant aux effets respectifs de Hh et de Wg sur le module i. La figure 4.10 présente les résultats obtenus en terme de tailles de LTS, au cours de la construction hiérarchique du système selon trois ordres différents (dont les deux derniers sont également représentés graphiquement en figure 4.11). Pour le premier on calcule le produit global de tous les LTS en une étape. Le second ordre est celui proposé par l'heuristique "smart reduction" implémentée dans CADP [36]. La dernière compose les modules dans l'ordre physique des cellules et offre de bien meilleurs coefficients de réduction. En effet les modules restreignent fortement les comportements possibles de leurs voisins, et on a intérêt à les composer le plus tôt possible lors de la construction. Cette étude met en avant le fait que le choix de l'ordre d'analyse est un problème difficile et crucial dans les approches hiérarchique car l'efficacité de la technique en dépend grandement.

Composition	Taille du produit	Taille après réduction
Monolithique	548 208	864
Smart reduction		
$L_1 = H^3(Wg) \otimes H^5(Wg)$	27	27
$L_2 = H^2(Wg) \otimes H^4(Wg)$	27	27
$L_3 = H^3(Hh) \otimes H^5(Hh)$	27	27
$L_4 = H^2(Hh) \otimes H^4(Hh)$	27	27
$L_5 = M^1 \otimes M^2$	729	729
$L_6 = M^5 \otimes M^6$	747	747
$L_7 = L'_2 \otimes H^6(Wg)$	27	27
$L_8 = L'_1 \otimes H^1(Wg)$	27	27
$L_9 = L'_4 \otimes H^6(Hh)$	27	27
$L_{10} = L'_3 \otimes H^1(Hh)$	27	27
$L_{11} = M^3 \otimes M^4$	8019	8019
$L_{12} = L_{11'} \otimes L'_9 \otimes L'_5$	1 764 450	1 130 157
$L_{13} = L_{12'} \otimes L_{10'} \otimes L'_6$	25 999 469	Out of memory
Ordre physique		
$C_1 = M^1 \otimes H^1(Hh) \otimes H^1(Wg)$	81	81
$C_2 = M^2 \otimes H^2(Hh) \otimes H^2(Wg)$	729	729
$C_3 = M^3 \otimes H^3(Hh) \otimes H^3(Wg)$	729	729
$C_4 = M^4 \otimes H^4(Hh) \otimes H^4(Wg)$	891	891
$C_5 = M^5 \otimes H^5(Hh) \otimes H^5(Wg)$	891	891
$C_6 = M^6 \otimes H^6(Hh) \otimes H^6(Wg)$	81	81
$L_1 = C'_1 \otimes C'_2$	729	585
$L_2 = L'_1 \otimes C'_3$	5 265	2 691
$L_3 = L'_2 \otimes C'_4$	27 027	7 101
$L_4 = L'_3 \otimes C'_5$	75 852	32 409
$L_5 = L'_4 \otimes C'_6$	19 829	864

FIGURE 4.10 – Tailles des compositions intermédiaires pour le module de un système composé de six modules de polarité segmentaire et pour trois ordres hiérarchiques différents. Les L_i et C_i sont les noms donnés à ces compositions, les L'_i et C'_i sont leurs réductions respectives par l'équivalence SAFETY. Les M_i sont les comportements réduits des modules et les différents H_i les modules d'intégration. La figure 4.11 contient une représentation graphique des deux derniers ordres.

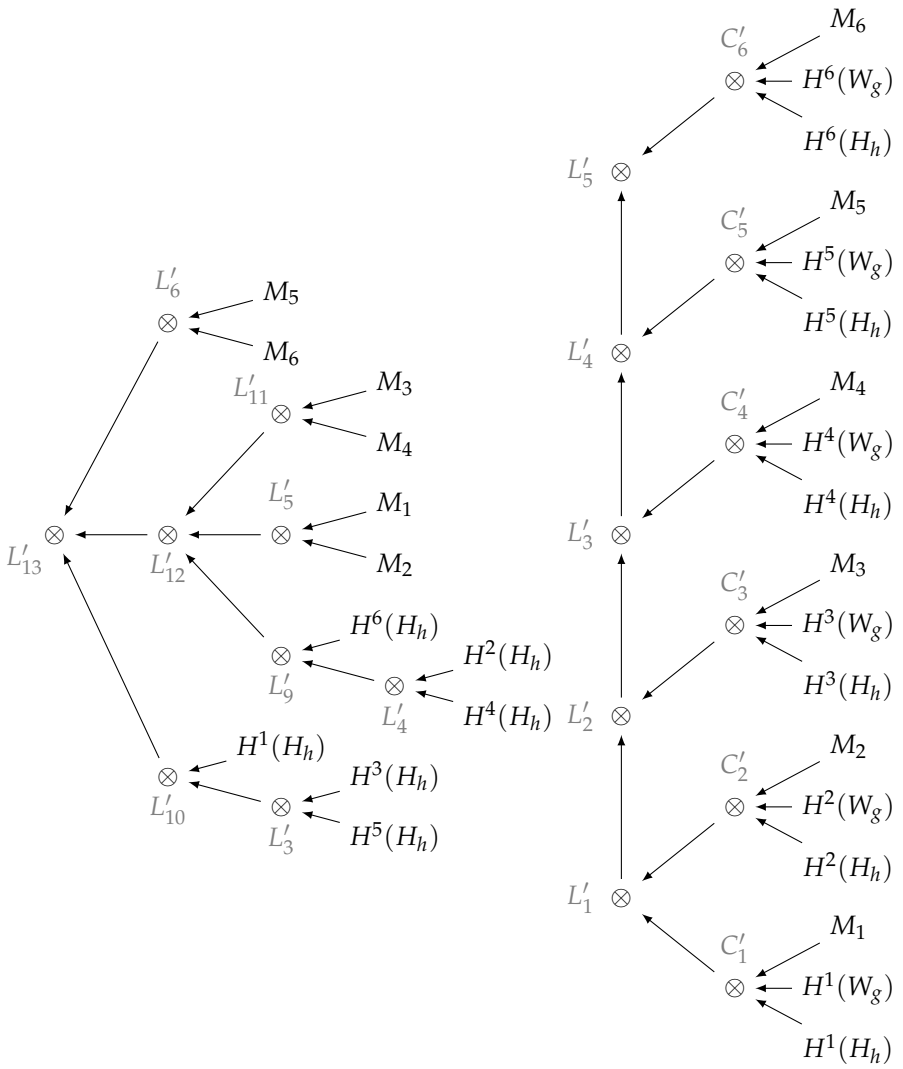


FIGURE 4.11 – Représentation graphique des ordres de compositions utilisés, que l'on retrouve sous une forme différente en figure 4.10. Chaque noeud $\otimes$ correspond à un produit synchronisé des LTS d'entrées suivis d'une réduction. À gauche : la partie du calcul proposé par l'heuristique "smart reduction" dont le calcul échoue par manque de mémoire. À droite : l'ordre de composition suivant la configuration physique du système.

Chapitre 5

Abstraction selon la formule

A partir d'une formule φ et d'un module S, on réduit ce module en préservant la valeur de φ sur le système global.

[1] En sous-section 5.1.1 On définit la fonction Pass^{φ} (resp. Fail^{φ}), qui retourne pour chaque état x d'un module une condition suffisante sur l'environnement Env (définit comme le comportement de l'ensemble des autres modules) pour que φ soit globalement vérifiée (resp. non vérifiée) lorsque le module est dans l'état x. Si Pass^{φ} (ou Fail^{φ}) est vraie sur l'état initial on peut immédiatement déterminer la valeur globale de φ.

[2] À partir des définitions de Pass^{φ} et Fail^{φ}, on construit une relation $\langle\langle\varphi\rangle\rangle$, telle que si deux états x et y d'un module sont équivalents, on peut changer l'état du module de l'un à l'autre sans modifier la valeur de vérité globale de φ.

[3] Section 5.2.1, grâce a la connaissance de l'état initial du module, on calcule un graphe produit, noté S^{φ}, entre le module et la formule. Cela permet d'une part de supprimer certaines parties du module qui ne sont pas nécessaires à la vérification de la propriété. D'autre part, cela permet de déterminer quelles sous-formules seront potentiellement vérifiées sur quels états et cette information est utile pour les réductions suivantes.

[4] La première de ces réductions est définie en sous-section 5.2.2, dans laquelle on montre qu'un état x de S^{φ} peut être remplacé par autre état y équivalent selon la relation $\langle\langle\varphi\rangle\rangle$, à condition que x ne soit pas accessible depuis y. Cela permet de réduire le graphe dans le cas où il y a moins d'états accessibles depuis y que depuis x. Par exemple si le comportement d'un module comporte une période d'initialisation qui n'est pas considérée dans la formule, on peut ainsi supprimer ces premiers états.

[5] Enfin, en section 5.2.4, on définie une seconde relation d'équivalence $\approx^{\varphi}$ qui à la différence de $\langle\langle\varphi\rangle\rangle$ est préservée par la fusion d'états équivalents, On peut donc réduire le module en calculant un graphe quotient par rapport à cette relation.

5.1 Équivalence dépendant de la formule

Le but de la section est d'identifier les états d'un module S, qui sont équivalents vis-à-vis d'une formule φ, c'est-à-dire les états qu'on ne peut jamais (quel que soit l'environnement) distinguer les uns des autres à l'aide de la valeur de vérité de φ sur le système global.

5.1.1 Pass **et** Fail

La première étape consiste à définir pour chaque état x, les formules $\mathsf{Pass}^{\varphi}(x)$ et $\mathsf{Fail}^{\varphi}(x)$ (φ est parfois omis par la suite si il n'y a pas d'ambiguïté). Celles-ci sont des formules Booléennes sur les variables d'états apparaissant dans φ et étant *externes* au module S (par opposition aux variables *locales* du module qui sont utilisées dans les labels des états de S). Elles peuvent être considérées comme des constantes propositionnelles du μ-calcul, qui peuvent être évaluées sur un état de l'environnement Env.

On veut que si un état $env \in Env$ satisfait la formule $\mathsf{Pass}^{\varphi}(x)$, (resp. $\mathsf{Fail}^{\varphi}(x)$), alors φ est vraie (resp. fausse) sur l'état global (x, env). Si l'une ou l'autre est évaluée à $\top$ sur l'état initial, on peut terminer l'analyse et donner une conclusion globale sur la valeur de vérité de φ (de façon similaire aux méthodes d'analyse partielles). Sinon ces deux fonctions sont utilisées dans les sections suivantes pour réduire la taille du module.

On note que c'est ici l'intérêt premier de ces fonctions, si on cherche principalement à savoir si le module contient suffisamment d'information pour conclure globalement sur la formule, il existe des méthodes de vérification partielles plus précises [66, 9, 74, 5, 6]. En projetant la formule sur un état x du module, une telle méthode permet d'obtenir une formule de μ-calcul $\varphi /\!/ x$, qui est une condition nécessaire et suffisante sur l'environnement pour que le système global vérifie φ. Cette formule peut néanmoins être de taille importante et déterminer si c'est ou non une tautologie est un problème de complexité exponentielle en la taille de cette formule [95]. Pass^{φ} retourne des conditions moins précises sous formes de formules booléennes sur les variables d'état de l'environnement et non pas de formules de μ-calcul générales, ce sont seulement des conditions suffisantes mais plus simples à calculer. On note que ce

choix de se limiter à des conditions Booléennes possède un coté un peu arbitraire et on pourrait choisir d'autres approximations de $\varphi /\!\!/ x$ (par exemple en se limitant aux formules ne dépassant pas une certaine « profondeur » en terme d'opérateurs d'actions de l'environnement).

La Définition 14 étend la relation d'ordre basée sur l'implication des formules Booléennes à l'ensemble des fonctions utilisées dans la construction de Pass et de Fail.

Définition 14. Soient Q l'ensemble des états d'un module S, φ une formule de μ-calcul, $\mathcal{V}_{env}$ l'ensemble des variables de φ qui sont externes à S, $\mathbb{B}(\mathcal{V}_{env})$ l'ensemble des fonctions Booléennes sur ces variables. Soient deux fonctions f et $g : Q \to \mathbb{B}(\mathcal{V}_{env})$ On définit $\overset{Q}{\Rightarrow}$ par : $f \overset{Q}{\Rightarrow} g$ ssi $f(x) \Rightarrow g(x)$ pour tout $x \in Q$.

L'ensemble des fonctions de Q dans $\mathbb{B}(\mathcal{V}_{env})$, muni de la relation $\overset{Q}{\Rightarrow}$ admet alors une structure de treillis. La borne supérieure (resp. inférieure) de deux fonctions f et g est la fonction qui à chaque état x associe $f(x) \vee g(x)$ (resp. $f(x) \wedge g(x)$). Comme vu précédemment, la sémantique habituelle d'une formule close est un sous-ensemble des états Q du LTS, que l'on peut voir comme une fonction qui associe une valeur de vérité à chaque état. Lors d'un calcul de point fixe, ces ensembles sont comparés à l'aide de la relation d'inclusion. On travaille donc en utilisant le treillis $(Q \to \{\top, \bot\}, \subseteq)$. Le calcul de Pass et de Fail se fait de façon similaire, mais dans le treillis $(Q \to \mathbb{B}(\mathcal{V}_{env}), \overset{Q}{\Rightarrow})$ qui est de taille à $(2^{2^{|\mathcal{V}_{env}|}})^{|Q|}$. En effet ses éléments sont des fonctions associant à chaque état de Q une fonction Booléenne à $|\mathcal{V}_{env}|$ variables, il y a $2^{2^{|\mathcal{V}_{env}|}}$ fonctions Booléennes possibles car celles-ci assignent une valeur de vérité à chaque valuation des variables de $\mathcal{V}_{env}$ (qui sont au nombre de $2^{\mathcal{V}_{env}}$). De plus, la taille de la plus grande chaîne possible est $Q \times 2^{|\mathcal{V}_{env}|}$ (ce qui correspond au pire des cas lors d'un calcul de point fixe). En effet l'incrément minimal possible d'un élément du treillis consiste à modifier un unique $\bot$ en $\top$ dans la table de vérité associée à un état de Q.

Définition 15 (Pass and Fail)**.** Considérons une formule φ sous forme normale positive, l'ensemble Q des états d'un module et x un de ces états. Les fonctions Pass^{φ} et Fail^{φ} sont définies par la figure 5.1. On écrit $x \in \mathsf{Pass}^{\varphi}$ (resp. $x \in \mathsf{Fail}^{\varphi}$) ssi $\mathsf{Pass}^{\varphi}(x) = \top$ (resp. $\mathsf{Fail}^{\varphi}(x) = \top$).

$$\mathsf{Pass}^{B}(x) \stackrel{\text{df}}{=} B@x$$
$$\mathsf{Pass}^{\varphi_1 \wedge \varphi_2}(x) \stackrel{\text{df}}{=} \mathsf{Pass}^{\varphi_1}(x) \wedge \mathsf{Pass}^{\varphi_2}(x)$$
$$\mathsf{Pass}^{\varphi_1 \vee \varphi_2}(x) \stackrel{\text{df}}{=} \mathsf{Pass}^{\varphi_1}(x) \vee \mathsf{Pass}^{\varphi_2}(x)$$
$$\mathsf{Pass}^{\langle l \rangle \varphi}(x) \stackrel{\text{df}}{=} \bigvee_{x \xrightarrow{l} x'} \mathsf{Pass}^{\varphi}(x')$$
$$\mathsf{Pass}^{\langle s \rangle \varphi}(x) \stackrel{\text{df}}{=} \bot$$
$$\mathsf{Pass}^{\langle e \rangle \varphi}(x) \stackrel{\text{df}}{=} \bot$$
$$\mathsf{Pass}^{[l] \varphi}(x) \stackrel{\text{df}}{=} \bigwedge_{x \xrightarrow{l} x'} \mathsf{Pass}^{\varphi}(x')$$
$$\mathsf{Pass}^{[s] \varphi}(x) \stackrel{\text{df}}{=} \begin{cases} \top & \text{si } \bigwedge_{x \xrightarrow{s} x'} \mathsf{Pass}^{\varphi}(x') = \top \\ \bot & \text{sinon} \end{cases}$$
$$\mathsf{Pass}^{[e] \varphi}(x) \stackrel{\text{df}}{=} \begin{cases} \top & \text{si } \mathsf{Pass}^{\varphi}(x) = \top \\ \bot & \text{sinon} \end{cases}$$
$$\mathsf{Pass}^{\mu X.\varphi}(x) \stackrel{\text{df}}{=} \bigwedge \{ f(x) \mid \mathsf{Pass}^{\varphi}(f) \stackrel{\text{Q}}{\Rightarrow} f \}$$
$$\mathsf{Pass}^{\nu X.\varphi}(x) \stackrel{\text{df}}{=} \bigvee \{ f(x) \mid \mathsf{Pass}^{\varphi}(f) \stackrel{\text{Q}}{\Leftarrow} f \}$$
$$\mathsf{Pass}^{X}(x) \stackrel{\text{df}}{=} X$$
$$\mathsf{Fail}^{\varphi}(x) \stackrel{\text{df}}{=} \mathsf{Pass}^{\neg \varphi}(x)$$

FIGURE 5.1 – Définition de Pass et de Fail. Les actions l, s et e sont respectivement locales, synchronisées et externes.

Intuitivement les règles définissant Pass et Fail peuvent être comprises ainsi :

[1] $\mathsf{Pass}^{B}(x) \stackrel{\text{df}}{=} B@x$**;**
On remplace les variables locales de B par leurs valeurs en x. La formule obtenue est une condition nécessaire et suffisante sur le label de l'état actuel env de l'environnement, pour avoir $(x, env) \models B$. Si cette condition est $\bot$, le système global ne peut jamais vérifier B tant que le module est dans l'état x (on a alors $x \in \mathsf{Fail}^{B}$). Par contre si cette condition est $\top$, le système global vérifie B quelque soit l'environnement ($x \in \mathsf{Pass}^{B}$). Considérons par exemple la formule $p \wedge w$ de la figure 5.2, où p est une variable locale au module et w une variable externe. Si p est fausse, $p \wedge w$ l'est également. C'est notamment le cas dans l'état 2, et donc $\mathsf{Pass}^{p \wedge w}(2) = \bot$. À l'inverse, si p est vraie il faut aussi avoir

$env \models w$ pour que $p \wedge w$ soit vérifiée sur le système global (c'est pour cela que $\mathsf{Pass}^{p \wedge w}(1) = w$).

[2] $\mathsf{Pass}^{\varphi_1 \wedge \varphi_2}(x) \stackrel{\text{df}}{=} \mathsf{Pass}^{\varphi_1}(x) \wedge \mathsf{Pass}^{\varphi_2}(x)$**;**
Si le label de l'environnement vérifie les deux conditions ($\mathsf{Pass}^{\varphi_1}(x)$ et $\mathsf{Pass}^{\varphi_2}(x)$), alors le système global vérifie $\varphi_1 \wedge \varphi_2$ à partir du moment où le module est dans l'état x.

[3] $\mathsf{Pass}^{\langle l \rangle \varphi}(x) \stackrel{\text{df}}{=} \bigvee_{x \xrightarrow{l} x'} \mathsf{Pass}^{\varphi}(x')$**;**
L'action l est locale au module, c'est à dire que s'il existe une transition $x \xrightarrow{l} x'$ dans le module, il existera une transition $(env, x) \xrightarrow{l} (env, x')$ dans l'environnement. Si le label de env vérifie $\mathsf{Pass}^{\varphi}(x')$ on a alors $(x, env) \models \langle l \rangle \varphi$.

[4] $\mathsf{Pass}^{\langle s \rangle \varphi}(x) \stackrel{\text{df}}{=} \bot$**;**
L'action s est partagée entre le module et l'environnement, ils doivent pouvoir l'exécuter de façon synchronisée pour qu'elle soit disponible dans le système globale. Pour avoir $(env, x) \models \langle s \rangle \varphi$ il faut que l'environnement puisse effectuer une action $env \xrightarrow{s} env'$ pendant que le module effectue $x \xrightarrow{s} x'$, et il faut que $(env', x') \models \varphi$. Autrement dit, il faut avoir $env \models \langle s \rangle (\bigvee_{x \xrightarrow{s} x'} \mathsf{Pass}^{\varphi}(x'))$. C'est cependant une propriété portant sur la dynamique de l'environnement et on choisit de ne pas en tenir compte comme discuté précédemment.

[5] $\mathsf{Pass}^{[s]\varphi}(x) \stackrel{\text{df}}{=} \begin{cases} \top & \textbf{si } \bigwedge_{x \xrightarrow{s} x'} \mathsf{Pass}^{\varphi}(x') = \top \\ \bot & \textbf{sinon} \end{cases}$**;**
L'environnement doit vérifier $[s]\mathsf{Pass}^{\varphi}(x')$ pour tous les successeurs x' de x. C'est a dire $\bigwedge_{x \xrightarrow{s} x'} [s]\mathsf{Pass}^{\varphi}(x') = [s] \bigwedge_{x \xrightarrow{s} x'} \mathsf{Pass}^{\varphi}(x')$. C'est une tautologie si l'on a $\bigwedge_{x \xrightarrow{s} x'} \mathsf{Pass}^{\varphi}(x') = \top$.

[6] $\mathsf{Pass}^{\mu X.\varphi}(x) \stackrel{\text{df}}{=} \bigwedge \{ f(x) \mid \mathsf{Pass}^{\varphi}(f) \stackrel{\text{Q}}{\Rightarrow} f \}$**;**
Si φ possède une seule variable libre, Pass^{φ} est une fonction de l'ensemble $(Q \mapsto \mathbb{B}(\mathcal{V}_{env}))$ vers lui même, elle représente une modification de la formule associée à chaque état. Elle est également monotone (au sens de la relation $\stackrel{\text{Q}}{\Rightarrow}$) donc son plus petit point fixe est bien défini au sein du treillis fini $(Q \to \mathbb{B}(\mathcal{V}_{env}), \stackrel{\text{Q}}{\Rightarrow})$. On peut également l'atteindre en itérant la fonction à partir de l'élément minimal, qui est la fonction qui associe $\bot$ à tous les états.

[7] $\mathsf{Fail}^{\varphi}(x) \stackrel{\text{df}}{=} \mathsf{Pass}^{\neg \varphi}(x)$**;**
On sait que la formule est fausse si et seulement si on sait que sa négation est vraie. La définition de Pass^{φ} considère que la formule est sous forme

normale positive, de telle sorte que les négations n'interviennent qu'au niveau des constantes propositionnelles. La définition de Fail^{φ} présuppose donc une représentation de $\neg\varphi$ sous forme normale positive.

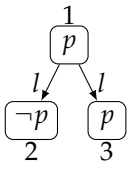
Module	Valeur de Pass			
		1	2	3
	$p \wedge w$	w	$\bot$	w
	$p \vee w$	$\top$	w	$\top$
	$p \Leftrightarrow w$	w	$\neg w$	w
	$\langle l \rangle (p \Leftrightarrow w)$	$\top$	$\bot$	$\bot$
	$[l](p \vee w)$	w	$\top$	$\top$

FIGURE 5.2 – Quelques exemples de calculs de Pass. La variable p est locale au module et w est externe.

Exemple 1. Considérons la formule $\Phi \stackrel{\text{df}}{=} \mu X.(p \Leftrightarrow v) \vee \langle l \rangle X$, qui signifie qu'il est possible d'atteindre un état où p a la même valeur que v, en utilisant uniquement l'action l, et calculons Pass^{Φ} sur le LTS S de la Figure 5.3.

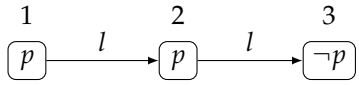


FIGURE 5.3 –

Par rapport à ce LTS, p est une variable locale, v une variable externe et l une action locale. Puisque Φ contient l'opérateur μ, on effectue un calcul de plus petit point fixe pour calculer Pass^{Φ}. L'initialisation P_0 est la fonction qui associe $\bot$ à chaque état, et on peut alors calculer le point fixe à l'aide de la relation de récurrence suivante. Pour tout état x de S et $n > 0$ on a

$$
\begin{aligned}
P_0(x) &\stackrel{\text{df}}{=} \bot \\
P_n(x) &\stackrel{\text{df}}{=} (\mathsf{Pass}^{a \Leftrightarrow v \vee \langle l \rangle \Phi}(x))[Pass^{\Phi} \leftarrow P_{n-1}] \\
&= (\mathsf{Pass}^{a \Leftrightarrow v}(x) \vee \mathsf{Pass}^{\langle l \rangle \Phi}(x))[Pass^{\Phi} \leftarrow P_{n-1}] \\
&= (a \Leftrightarrow v)@x \vee \bigvee_{x \xrightarrow{l} x'} P_{n-1}(x')
\end{aligned}
$$

Voici les étapes du calcul de Pass^{φ} sur le module S.

étape 0 : $P_0(x) = \bot \qquad \forall x$
étape 1 : $P_1(1) = v \vee P_0(2) = v$
$P_1(2) = v \vee P_0(3) = v$
$P_1(3) = \neg v$
étape 2 : $P_2(1) = v \vee P_1(2) = v \vee v = v$
$P_2(2) = v \vee P_1(3) = v \vee \neg v = \top$
$P_2(3) = \neg v$
étape 3 : $P_3(1) = v \vee P_2(2) = v \vee \top = \top$
$P_3(2) = v \vee P_2(3) = v \vee \neg v = \top$
$P_3(3) = \neg v$
étape 4 : $P_4 = P_3$ (le plus petit point fixe est atteint)

Savoir que le module S est dans l'état 1 ou 2, est donc suffisant pour conclure que la formule Φ est vérifiée sur le système global. Par contre si S est dans l'état 3, on sait seulement que Φ sera vraie si l'environnement est dans un état où v est faux.

Les fonctions Pass et Fail vérifient les propriétés suivantes.

Lemme 1. Les calculs de Pass et de Fail terminent.

Lemme 2 (Compositionnalité de Pass et de Fail)**.** Si S_1 et S_2 sont deux modules d'un LTS modulaire, x_1 un état de S_1 et x_2 un état de S_2, on a $\mathsf{Pass}^{\varphi}(x_1)@x_2 \Rightarrow \mathsf{Pass}^{\varphi}((x_1, x_2))$ et $\mathsf{Fail}^{\varphi}(x_1)@x_2 \Rightarrow \mathsf{Fail}^{\varphi}((x_1, x_2))$.

Lemme 3 (Pass et Fail sont corrects)**.** Pour tout état x d'un module et tout état env de son environnement,

$$\text{si } env \models \mathsf{Pass}^{\varphi}(x) \text{ alors } (x, env) \models \varphi.$$

Et de même pour Fail :

$$\text{si } env \models \mathsf{Fail}^{\varphi}(x) \text{ alors } (x, env) \not\models \varphi.$$

Théorème 2 (relation avec la sémantique du μ-calcul)**.** Si φ est une formule contenant uniquement des actions et des variables locales à un LTS S, on a bien

$$\varphi^S = \{x \mid \mathsf{Pass}^{\varphi}(x) = \top\}.$$

5.1.2 Implication locale

Le but de cette sous-section est d'identifier les états d'un module qu'il est impossible de différencier en connaissant uniquement la valeur de vérité d'une formule φ sur le système global. Autrement dit, on recherche les couples (x, y) tels que pour tout état possible de l'environnement $env \in Env$, la valeur de vérité de φ est identique dans les états globaux (x, env) et (y, env). Cette information sera utilisée par la suite pour réduire la taille du module (sous-section 5.2.2). La première étape consiste à rechercher des *relations d'implication locales* sur les états d'un module S : pour deux formules closes φ et ψ, on définit la fonction $\langle\langle \varphi, \psi \rangle\rangle$ qui associe à chaque couple d'états (x, y) une formule booléenne dont les propriétés atomiques sont des variables d'états des formules φ et ψ qui sont externes au module. L'objectif est que si le label d'un état $env \in Env$ vérifie la formule $\langle\langle \varphi, \psi \rangle\rangle(x, y)$, alors on aura la propriété suivante :

$$(x, env) \models \varphi \Rightarrow (y, env) \models \psi.$$

En particulier, si $\langle\langle \varphi, \varphi \rangle\rangle(x, y) = \langle\langle \varphi, \varphi \rangle\rangle(x, y) = \top$, les états x et y sont dits équivalents pour φ, et on a pour tout état env :

$$(x, env) \models \varphi \Leftrightarrow (y, env) \models \varphi.$$

De la même façon que $\mathsf{Pass}^{\varphi}(x)$ est une approximation de $\varphi /\!\!/ x$ comme discuté en sous-section 5.1.1, $\langle\langle \varphi, \psi \rangle\rangle(x, y)$ permet de calculer une approximation de la formule $\varphi /\!\!/ x \Rightarrow \psi /\!\!/ y$ en se limitant aux variables d'états de l'environnement et non au μ-calcul complet (pour les mêmes raisons).

Définition 16 (Conditions locales d'implication)**.** On considère l'ensemble Q des états d'un module S, et deux formules φ et ψ. Soit $\mathcal{V}$ l'ensemble des variables d'états de φ et de ψ qui sont externes à S. On définit les conditions d'implication locales grâce aux règles de la Figure 5.4. On écrit également $x \langle\langle \varphi \rangle\rangle y$ si et seulement si $\langle\langle \varphi, \varphi \rangle\rangle(x, y) = \langle\langle \varphi, \varphi \rangle\rangle(y, x) = \top$, auquel cas x et y sont dits localement équivalent.

$$\langle\langle B_1, B_2\rangle\rangle(\mathcal{V})(x,y) = B_1@x \Rightarrow B_2@y \tag{5.1}$$

$$\langle\langle \psi, \varphi_1 \vee \varphi_2\rangle\rangle(\mathcal{V})(x,y) = \langle\langle \psi, \varphi_1\rangle\rangle(\mathcal{V})(x,y) \vee \langle\langle \psi, \varphi_2\rangle\rangle(\mathcal{V})(x,y) \tag{5.2}$$

$$\langle\langle \psi, \langle l\rangle\varphi\rangle\rangle(\mathcal{V})(x,y) = \mathsf{Fail}^{\psi}(x) \vee \bigvee_{y \xrightarrow{l} y'} \langle\langle \psi, \varphi\rangle\rangle(\mathcal{V})(x,y') \tag{5.3}$$

$$\langle\langle \langle s\rangle\psi, \langle s\rangle\varphi\rangle\rangle(\mathcal{V})(x,y) = \mathsf{Fail}^{\langle s\rangle\psi}(x) \vee \begin{cases} \top & \text{si } \bigwedge_{x \xrightarrow{s} x'} \mathsf{Fail}^{\psi}(x') \vee \bigvee_{y \xrightarrow{s} y'} \langle\langle \psi, \varphi\rangle\rangle(\mathcal{V})(x',y') \\ \bot & \text{sinon} \end{cases} \tag{5.4}$$

$$\langle\langle \langle e\rangle\psi, \langle e\rangle\varphi\rangle\rangle(\mathcal{V})(x,y) = \begin{cases} \top & \text{si } \langle\langle \psi, \varphi\rangle\rangle(\mathcal{V})(x,y) \\ \bot & \text{sinon} \end{cases} \tag{5.5}$$

$$\langle\langle \mu X.\psi, \varphi\rangle\rangle(\mathcal{V})(x,y) = \begin{cases} \langle\langle \psi(\mu X.\psi), \varphi\rangle\rangle(\mathcal{V} \cup \{((\mu X.\psi, \varphi), \top_{Q^2})\})(x,y) \\ \quad \text{si } (\mu X.\psi, \varphi) \notin Dom(\mathcal{V}) \\ \mathcal{V}(\mu X.\psi, \varphi)(x,y) \\ \quad \text{sinon} \end{cases} \tag{5.6}$$

$$\langle\langle \psi, \varphi\rangle\rangle(\mathcal{V})(x,y) = \langle\langle \neg\varphi, \neg\psi\rangle\rangle(\mathcal{V})(y,x) \tag{5.7}$$

$$\langle\langle \psi, \varphi_1 \wedge \varphi_2\rangle\rangle(\mathcal{V})(x,y) = \langle\langle \psi, \varphi_1\rangle\rangle(\mathcal{V})(x,y) \wedge \langle\langle \psi, \varphi_2\rangle\rangle(\mathcal{V})(x,y) \tag{5.8}$$

$$\langle\langle \psi, [l]\varphi\rangle\rangle(\mathcal{V})(x,y) = \bigwedge_{y \xrightarrow{l} y'} \langle\langle \psi, \varphi\rangle\rangle(\mathcal{V})(x,y') \tag{5.9}$$

$$\langle\langle \nu X.\psi, \varphi\rangle\rangle(\mathcal{V})(x,y) = \begin{cases} \langle\langle \psi(\nu X.\psi), \varphi\rangle\rangle(\mathcal{V} \cup \{((\mu X.\psi, \varphi), \bot_{Q^2})\})(x,y) \\ \quad \text{si } (\mu X.\psi, \varphi) \notin Dom(\mathcal{V}) \\ \mathcal{V}(\mu X.\psi, \varphi)(x,y) \\ \quad \text{sinon} \end{cases} \tag{5.10}$$

$$\langle\langle \psi, \varphi\rangle\rangle(x,y) = \mathsf{Fail}^{\psi}(x) \vee \mathsf{Pass}^{\varphi}(y) \tag{5.11}$$

FIGURE 5.4 – Définition des conditions locales d'implications. Lorsque plusieurs règles sont applicables, on choisit celle qui apparaît en premier dans la liste ci-dessus. $\top_{Q^2}$ et $\bot_{Q^2}$ désignent les fonctions constantes qui associent respectivement $\top$ et $\bot$ à tous les couples d'états.

Intuitivement, les règles de la figure 5.4 peuvent être comprises de la ma-

nière suivante :

- cas $\langle\langle B_1, B_2\rangle\rangle$: B_1 et B_2 sont des propriétés portant sur les états du système global. On remarque que $\langle\langle B_1, B_2\rangle\rangle(x, y)$ est égal à la formule $\mathsf{Fail}^{B_1}(x) \vee \mathsf{Pass}^{B_2}(y)$. Si un état $env \in Env$ vérifie cette propriété alors on a $(x, env) \models B_1 \Rightarrow (y, env) \models B_2$;
- cas $\langle\langle \psi, \varphi_1 \vee \varphi_2\rangle\rangle$: si pour tout état env de l'environnement tel que $(x, env) \models \psi$, on a soit $(y, env) \models \varphi_1$ soit $(y, env) \models \varphi_2$, on a donc aussi $(x, env) \models \psi \Rightarrow (y, env) \models \varphi_1 \vee \varphi_2$;
- cas $\langle\langle \psi, \langle l\rangle \varphi\rangle\rangle$: si $env \models \langle\langle \psi, \langle l\rangle \varphi\rangle\rangle(x, y)$ alors soit $env \models \mathsf{Fail}^{\psi}(x)$ (auquel cas on sait que $(x, env) \not\models \psi$) où il existe une transition locale $y \xrightarrow{l} y'$ telle que $\langle\langle \psi, \varphi\rangle\rangle(x, y')$. Si la propriété $(x, env) \models \psi$ implique $(y', env) \models \varphi$ alors on sait qu'elle implique aussi $(y, env) \models \langle l\rangle \varphi$ (l'action l étant locale, il sera toujours possible de l'exécuter sur le système global).
- cas $\langle\langle \langle s\rangle \psi, \langle s\rangle \varphi\rangle\rangle$: l'action s est synchronisée entre le module et l'environnement. Sans connaissance du comportement de l'environnement on a besoin que, pour tout successeur x' de x, il existe un successeur y de y' tel que $\langle\langle \psi, \varphi\rangle\rangle(x', y')$. Si (x, env) vérifie $\langle s\rangle \psi$, il existe alors une transition $(x, env) \xrightarrow{s} (x', env')$ telle que (x', env') vérifie ψ. Si on a $\langle\langle \langle s\rangle \psi, \langle s\rangle \varphi\rangle\rangle(x, y)$, on peut trouver une transition $y \xrightarrow{s} y'$ tel que (y', env') vérifie ψ, ce qui signifie que l'on a également $(y, env) \models \langle s\rangle \varphi$.
- cas $\langle\langle \langle e\rangle \psi, \langle e\rangle \varphi\rangle\rangle$: e est une action externe. Si il existe une transition $env \xrightarrow{e} env'$ telle que env' vérifie $\langle\langle \psi, \varphi\rangle\rangle(x, y)$, on a bien $(x, env) \models \langle e\rangle \psi \Rightarrow (y, env) \models \langle e\rangle \varphi$. On est sûr que c'est le cas si $\langle\langle \psi, \varphi\rangle\rangle(x, y)$ est une tautologie.
- cas $\langle\langle \mu X \psi, \varphi\rangle\rangle$: si le calcul de $\langle\langle \psi(\mu X \psi), \varphi\rangle\rangle(\mathcal{V})$ nécessite d'évaluer l'expression $\langle\langle \mu X \psi, \varphi\rangle\rangle(\mathcal{V})$ (comme montré dans la figure 5.5), on a une relation de récurrence où $\langle\langle \mu X \psi, \varphi\rangle\rangle$ est définie en fonction d'elle même. On note $\langle\langle \psi, \varphi\rangle\rangle$ cette fonction, on a donc $\langle\langle \mu X \psi, \varphi\rangle\rangle = \langle\langle \psi, \varphi\rangle\rangle(\langle\langle \mu X \psi, \varphi\rangle\rangle)$. $\langle\langle \psi, \varphi\rangle\rangle$ est une fonction de l'ensemble de fonctions $Q \times Q \to \mathbb{B}(\mathcal{V})$ dans lui même, et on calcule son plus grand point fixe en un nombre d'itérations fini (cf. lemme 4).
- cas $\langle\langle \psi, \varphi\rangle\rangle(x, y) = \langle\langle \neg\varphi, \neg\psi\rangle\rangle(y, x)$: ce cas est un raccourci permettant d'exprimer les règles duales de celles ci-dessus.
- dernier cas : Si aucune des autres règles n'est applicable on regarde si env vérifie $\mathsf{Fail}^{\psi}(x)$ ou $\mathsf{Pass}^{\varphi}(y)$. Si c'est le cas on a alors soit $(x, env) \models \neg\psi$ soit $(y, env) \models \varphi$, et la propriété $(x, env) \models \psi \Rightarrow (y, env) \models \varphi$ est bien vérifié.

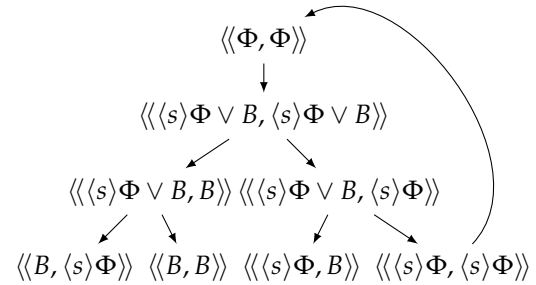


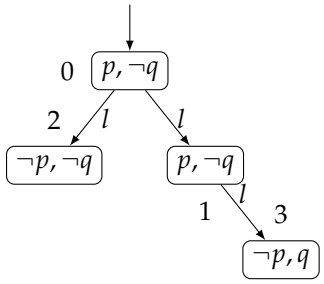
FIGURE 5.5 – Graphe de dépendance de $\langle\langle\Phi,\Phi\rangle\rangle$ (où $\Phi \stackrel{\text{df}}{=} \mu X.\langle s\rangle X \vee B$).

Lemme 4. Le calcul de $\langle\langle\varphi,\psi\rangle\rangle$ termine.

On appelle Φ la formule $\mu X.\langle l\rangle X \vee (p \wedge v) \vee (q \wedge w)$ signifiant qu'on peut atteindre un état vérifiant $(p \wedge v) \vee (q \wedge w)$ à l'aide d'actions l. La figure 5.6 comporte les valeurs de la fonction $\langle\langle\Phi,\Phi\rangle\rangle$ sur tous les couples d'états du graphe donné en exemple. Pour ce graphe, l est une action locale, p et q sont des variables locales, v et w sont des variables externes.

Dans l'état 2, p et q sont faux et le resteront toujours. Par conséquent, il appartient à Fail^{Φ} et on a $\langle\langle\Phi,\Phi\rangle\rangle(2,x) = \top$ pour tout état x. En effet, pour tout état env de l'environnement, Φ sera faux sur $(2,env)$ et donc l'implication $(2,env) \models \Phi \Rightarrow (x,env) \models \Phi$ sera vérifiée.

Par ailleurs, les états 0 et 1 sont équivalents car on a $\langle\langle\Phi,\Phi\rangle\rangle(0,1) = \langle\langle\Phi,\Phi\rangle\rangle(1,0) = \top$. La raison est qu'ils peuvent atteindre des valuations identiques de p et q, à l'exception de celle de l'état 2 qui est uniquement accessible depuis l'état 0. Mais comme l'état 2 appartient à Fail^{φ}, il n'apporte rien d'intéressant. On remarque que les colonnes (et lignes) 1 et 0 sont alors identiques (pour tout état x, on aura $\langle\langle\Phi,\Phi\rangle\rangle(0,x) = \langle\langle\Phi,\Phi\rangle\rangle(1,x)$ ainsi que $\langle\langle\Phi,\Phi\rangle\rangle(x,0) = \langle\langle\Phi,\Phi\rangle\rangle(x,1)$).



	0	1	2	3
0	$\top$	$\top$	$\neg v \wedge \neg w$	$\neg v \vee w$
1	$\top$	$\top$	$\neg v \wedge \neg w$	$\neg v \vee w$
2	$\top$	$\top$	$\top$	$\top$
3	$\top$	$\top$	$\neg w$	$\top$

FIGURE 5.6 – On considère la formule $\Phi = \mu X.\langle l \rangle X \vee (p \wedge v) \vee (q \wedge w)$ sur le graphe donné. les variables p et q et l'actions l sont locales alors que v et w sont externes. La case située ligne i et colonne j contient la valeur $\langle\langle \Phi, \Phi \rangle\rangle(i, j)$.

L'équivalence $\langle\langle \Phi \rangle\rangle$ est utilisée dans la sous-section 5.2.2 pour réduire la taille d'un module. Intuitivement, le théorème suivant permet d'utiliser un état x à la place d'un état y lors de la vérification d'une formule φ, à partir du moment où $x \langle\langle \varphi \rangle\rangle y$, car cette opération préserve la valeur de vérité de la formule.

Théorème 3 (préservation de la formule)**.** Soient x et y deux états d'un module tels que $x \langle\langle \varphi \rangle\rangle y$. On a alors $\mathsf{Pass}^{\varphi}(x) \Leftrightarrow \mathsf{Pass}^{\varphi}(y)$ et $\mathsf{Fail}^{\varphi}(x) \Leftrightarrow \mathsf{Fail}^{\varphi}(y)$.

Par ailleurs, on montre que cette relation est une congruence par rapport au produit synchronisé des modules. Cette propriété permet d'utiliser des graphes réduits pendant la construction incrémentale du comportement global, avec la garantie que l'on manipule toujours des graphes équivalents.

Théorème 4 (modularité)**.** Soient x_1 et y_1 deux états d'un module, x_2 et y_2 deux états d'un autre module, et φ une formule.

Si $x_1 \langle\langle \varphi \rangle\rangle y_1$ et $x_2 \langle\langle \varphi \rangle\rangle y_2$ alors on a également $(x_1, x_2) \langle\langle \varphi \rangle\rangle (y_1, y_2)$.

À l'aide de ces définitions et propriétés, on peut maintenant définir les opérations de réductions proprement dites.

5.2 Réductions

Cette section comporte les définitions de trois manières complémentaires de réduire un module S, sachant que l'on connaît la formule φ destinée à être vérifiée sur le système global ainsi que l'état initial du module. Ces réductions sont calculées sans connaissance de l'environnement et retournent un LTS que l'on peut utiliser à la place du module original quelle que soit la méthode de vérification choisie pour le système global.

5.2.1 Section des chemins inutiles

La première étape tire partie du fait que l'on connaît à la fois la formule et l'état initial du module. Elle consiste à construire le LTS S^φ de la définition 17 qui est un produit entre le module S et la formule φ, et dont on se sert de trois manières différentes.

[1] On retire d'une part les états et les transitions qui seront inutiles à la vérification de la formule. Les transitions étiquetée par des actions n'apparaissant pas dans la formule vont par exemple disparaître, ainsi que les états ainsi devenus inaccessibles. Certaines transitions peuvent également être supprimées si elle sont redondantes ou inutiles. Si on sait par exemple que $\langle l \rangle \varphi$ est la seule formule susceptible d'être vérifiée sur un état x, il est inutile de conserver les transitions $x \xrightarrow{l} x'$ telles que $x' \in \mathsf{Fail}^\varphi$.

[2] Le LTS S^φ permet également de connaître quelles sous formules peuvent potentiellement être vérifiées sur un état donné. Cela est utile pour les autres réductions des sous-sections 5.2.2 et 5.2.4. Intuitivement, on pourra utiliser un état à la place d'un autre si ils sont équivalents pour toutes les sous-formules potentiellement vérifiables sur ces états.

[3] On peut enfin l'utiliser pour savoir si un état *dépend* d'un autre pour le calcul d'une sous-formule. Ce dont on se sert pour la réduction de la sous-section 5.2.2.

Définition 17 (Accessibilité selon une formule)**.** Soit un LTS $S \stackrel{\text{df}}{=} (Q, i, A, R, L)$ et une formule φ. On construit le LTS $S^\varphi \stackrel{\text{df}}{=} (Q', i', A', R', L')$ dont les états sont des couples composés d'un état de S et d'une sous-formule de φ. Si l'état (x, ψ) appartient à Q', cela signifie que la sous-formule ψ est successible d'être vérifiée sur un état global où le module est dans l'état x.

Par la suite , $f(\varphi_1, \ldots, \varphi_k)$ est une formule de μ-calcul, où les $\varphi_1, \ldots, \varphi_k$ sont ses sous-formules de la forme $\varphi_j = \langle a \rangle \varphi'_j$ ou $\varphi_j = [a] \varphi'_j$ accessibles seule-

ment à l'aide de dépliages, de conjonctions ou de disjonctions. Le LTS S^φ est défini comme suit :

- l'ensemble des états est $Q' \stackrel{\text{df}}{=} \mathsf{St}(\varphi, i)$, avec

$$\mathsf{St}(f(\varphi_1, \ldots, \varphi_k), x) \stackrel{\text{df}}{=} \{(x, f(\varphi_1 \ldots \varphi_k))\} \cup \bigcup_{j=1\ldots k} \begin{cases} \bigcup_{x \xrightarrow{a} x', Fail^{\varphi'_j}(x') \neq \top} \mathsf{St}(\varphi'_j, x') & \text{si } \varphi_j = \langle l \rangle \varphi'_j \text{ ou } \varphi_j = \langle s \rangle \varphi'_j\,, \\ \bigcup_{x \xrightarrow{a} x', Pass^{\varphi'_j}(x') \neq \top} \mathsf{St}(\varphi'_j, x') & \text{si } \varphi_j = [l] \varphi'_j \text{ ou } \varphi_j = [s] \varphi'_j\,, \\ \mathsf{St}(\varphi'_j, x) & \text{si } \varphi_j = \langle e \rangle \varphi'_j \text{ ou } \varphi_j = [e] \varphi'_j\,; \end{cases}$$

- l'état initial est $i' \stackrel{\text{df}}{=} (i, \varphi)$;
- $A' \stackrel{\text{df}}{=} A \cap \{a \mid a \text{ apparaît dans dans } \varphi\}$
- l'ensemble des transitions R' est tel que si $(x, f(\varphi_1, \ldots, \varphi_k))$ et (x', ψ) appartiennent à St, alors $(x, f(\varphi_1, \ldots, \varphi_k)) \xrightarrow{a} (x', \psi) \in R'$ si et seulement si $x \xrightarrow{a} x' \in R$ et il existe $j \in 1 \ldots k$ tel que $\varphi_j = \langle a \rangle \psi$ or $\varphi_j = [a] \psi$;
- $L' \stackrel{\text{df}}{=} L \cap \{\lambda \mid \lambda \text{ apparaît dans } \varphi\}$.

Lemme 5. Le calcul de St termine.

Le LTS S^φ peut potentiellement avoir une taille supérieure à celle de S car chacun de ses états est constitué d'un état de S associé à une sous-formule de φ. Il est cependant possible de le réduire à un sous-graphe de S en fusionnant les états correspondants au même état de S. Si x est un état de S, on définit $\mathsf{Form}^\varphi(x) \stackrel{\text{df}}{=} \{\psi \mid (x, \psi) \in Q\}$.

Définition 18 (Réduction du graphe d'accessibilité par une formule)**.** Soient un module S et une formule φ tels que $S^\varphi \stackrel{\text{df}}{=} (Q, i, A, R, L)$. Le sous-graphe de S accessible selon φ est $\mathsf{Reduced}(S^\varphi) \stackrel{\text{df}}{=} (Q', i', A, R', L)$ où :

- $Q' \stackrel{\text{df}}{=} \{x \mid \mathsf{Form}(x) \neq \emptyset\}$;
- $(i', \varphi) \stackrel{\text{df}}{=} i$;
- $x \xrightarrow{a} y \in R'$ si et seulement si il existe φ_1 et φ_2 tels que $(x, \varphi_1) \xrightarrow{a} (y, \varphi_2) \in R$.

Le théorème suivant nous indique que les LTS obtenus par les opérations précédentes sont bien équivalents au module original.

Théorème 5 (Préservation de l'équivalence pour la formule)**.** Soient S et S' deux LTS, et φ une formule. On écrit $S \langle\langle \varphi \rangle\rangle S'$ si les états initiaux de S et de S' sont équivalent par rapport à la relation $\langle\langle \varphi \rangle\rangle$. On a alors les deux propriétés suivantes :

$$S \langle\langle \varphi \rangle\rangle S^{\varphi} \text{ et } \mathsf{Reduced}(S^{\varphi}) \langle\langle \varphi \rangle\rangle S^{\varphi}.$$

Exemple 2. La figure 5.7 présente l'exploration d'un LTS grâce à la formule $\Phi \stackrel{\text{df}}{=} \mu X.\langle \mathcal{L} \rangle X \vee B$ où $\mathcal{L} \stackrel{\text{df}}{=} \{l1, l2, s1, s2\}$ et $B \stackrel{\text{df}}{=} ((a \wedge v) \vee (b \wedge w))$. L'action étiquetée par l_3 n'apparaît pas dans la formule, on peut donc supprimer la partie droite du graphe sans perturber la vérification de la formule sur le système global. Par ailleurs, puisque l'état 2 appartient à Fail^{Φ}, on peut supprimer la transition l_1 menant vers cet état et donc les états 2 et 3 qui n'apporteront rien au calcul de la valeur de vérité de Φ.

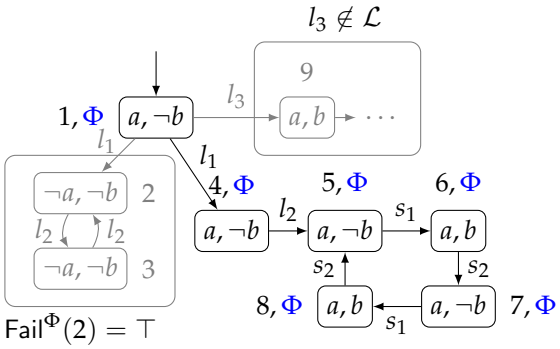


FIGURE 5.7 – Exploration dépendante de la formule avec section des chemins inutiles.

5.2.2 Raccourcissement de chemins

Soient x et y deux états d'un module tels que $x \langle\langle \varphi \rangle\rangle y$. On a montré que pour tout état *env* de l'environnement, $(x, env) \models \varphi \Leftrightarrow (y, env) \models \varphi$. L'idée est ici de choisir parmi ces deux états, celui qui va générer le plus petit graphe accessible par la formule φ (comme défini dans la section 5.2.1). Par exemple, quand une formule traite de l'exécution d'un système après une période d'initialisation, les états correspondants à cette initialisation pourront être retirés. Par contre dans d'autres cas, le graphe S^{φ} sera constitué d'une seule composante fortement connexe (SCC) et cette réduction n'aura dont aucun effet.

On montre dans le théorème 6 que l'on peut modifier le graphe S^{φ} en *remplaçant* un état (x, ψ) par un autre état (y, ψ), à partir du moment où l'on a

$(x,\psi)\langle\langle\psi\rangle\rangle(y,\psi)$ et qu'il n'y a pas de chemin de (y,ψ) à (x,ψ) dans S^{φ}. Cette opération de *remplacement* nous permet de "sauter par dessus" un état. La réduction du module commence par rechercher un état équivalent à l'état initial (mais à partir duquel on pourra atteindre moins d'états) que l'on pourra utiliser à la place. On réitère ensuite l'opération sur les successeurs de ce nouvel état.

L'opération de remplacement suivante peut être itérée sur le graphe d'accessibilité dépendant de la formule définie en section 5.2.1 avant d'appliquer l'opération Reduced.

Définition 19 (Remplacement d'un état)**.** Soit $S^{\psi} \stackrel{\text{df}}{=} (Q, i, A, R, L)$ un LTS obtenu d'après la définition 18, et (x,φ) et $(y,\varphi) \in Q$. $\mathsf{Replace}(S^{\psi},(x,\varphi),(y,\varphi))$ est construit en redirigeant les transitions d'entrée de (x,φ) vers (y,φ). Formellement, $\mathsf{Replace}(S^{\psi},(x,\varphi),(y,\varphi)) \stackrel{\text{df}}{=} (Q', i', A, R', L)$ où :

- $Q' \stackrel{\text{df}}{=} Q \setminus (x,\varphi)$;
- $i' \stackrel{\text{df}}{=} (y,\varphi)$ si $i = (x,\varphi)$, sinon $i' \stackrel{\text{df}}{=} i$;
- $R' \stackrel{\text{df}}{=} (R \setminus \{(x',\varphi') \xrightarrow{a} (x,\varphi)\})$
  $\cup \{(x',\varphi') \xrightarrow{a} (y,\varphi) \mid (x',\varphi') \xrightarrow{a} (x,\varphi) \in R\}$.

Le théorème suivant précise les conditions sous lesquelles un état peut être remplacé par un autre.

Théorème 6. Soit S^{ψ} un LTS, (x,φ) et (y,φ) deux états tels que $(x,\varphi)\langle\langle\varphi\rangle\rangle(y,\varphi)$. S'il n'y a pas de chemin de (y,φ) vers (x,φ) dans S^{ψ}, alors on a $S^{\psi}\langle\langle\psi\rangle\rangle\mathsf{Replace}(S^{\psi},(x,\varphi),(y,\varphi))$.

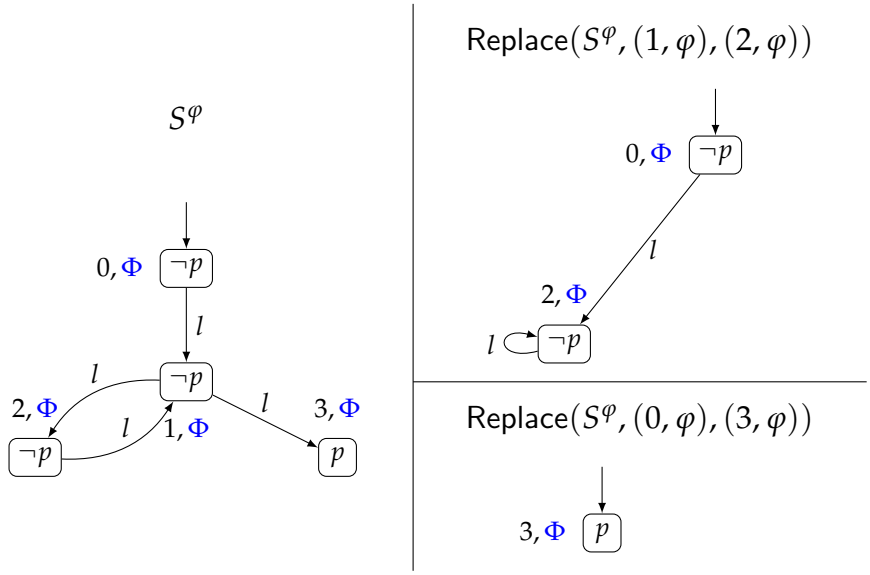


FIGURE 5.8 – Tous les états sont équivalents pour $\Phi = \mu X.\langle l \rangle X \vee p$ car ils appartiennent tous à Pass^Φ (il existe un chemin composé d'actions l menant vers p à partir de chaque état). On peut remplacer $(0,\varphi)$ par $(3,\varphi)$ car il n'y à pas de chemin de $(3,\varphi)$ vers $(0,\varphi)$, mais ce n'est pas le cas de $(1,\varphi)$ et de $(2,\varphi)$ pour lesquels le remplacement fait perdre l'accessibilité à p et modifie la valeur de la formule.

Cette opération nous permet d'éviter l'état (x,φ) et d'aller directement vers l'état (y,φ). Cela peut potentiellement amener à d'importantes réductions si (x,φ) et ses successeurs deviennent inaccessibles, auquel cas ils peuvent être supprimés.

Exemple 3. Le LTS de la figure 5.9 est obtenu à partir de celui de la figure 5.7 en remplaçant l'état initial $(1,\Phi)$ par $(5,\Phi)$. Cela est justifié car tous les états sont ici équivalents pour la relation $\langle\langle\Phi\rangle\rangle$ (et donc 1 and 5 en particulier) et parce qu'il n'y a pas de chemin de $(5,\Phi)$ vers $(1,\Phi)$. Les états 1 et 4 sont alors retirés car ils ne sont plus accessibles. On choisit ici l'état $(5,\Phi)$ car il appartient à l'unique composante fortement connexe terminale, afin d'obtenir la meilleure réduction possible. On peut en général avoir le choix entre plusieurs états équivalents amenant à des réductions différentes, et la recherche d'une stratégie optimale fait partie des perspectives.

L'algorithme du prototype utilisé dans le chapitre 6 explore le LTS S^φ, et remplace chaque état (x,ψ) découvert par un état (y,ψ) qui est équivalent, accessible depuis (x,ψ) et appartient à la composante fortement connexe la plus

éloignée possible de celle de (x, ψ) (pour un ordre topologique donné). Cette technique n'est cependant pas déterministe et peut être notamment améliorée grâce à un choix plus précis de la composante fortement connexe sélectionnée.

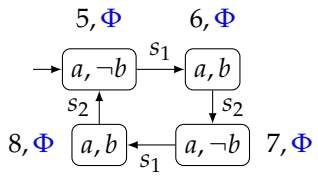


FIGURE 5.9 – Raccourcissement de chemin.

5.2.3 Suppression des arcs sortant inutiles

Considérons l'exemple de la figure 5.10. L'état 1 appartient à $\mathsf{Pass}^{\langle a\rangle p \vee \langle b\rangle \neg p}$ même si l'on supprime une de ses arêtes sortantes. De façon plus générale on peut essayer de supprimer des transitions sortantes (et donc les états uniquement accessibles par celles ci), pour les états (x, φ) tels que l'on a $\mathsf{Pass}^{\varphi}(x) = \neg\mathsf{Fail}^{\varphi}(x)$. On peut dans ce cas supprimer certains des arcs sortants de (x, φ) dans la mesure où cela ne modifie pas la valeur de Pass et de Fail.

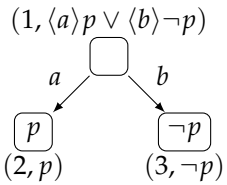


FIGURE 5.10 – Exemple d'arcs sortants redondants. Les actions a et b sont locales.

On cherche à supprimer certaines transitions sortantes d'un noeud (x, φ) avec $\varphi = f(\varphi_1, \ldots, \varphi_k)$. On construit dans un premier temps tous les ensembles de transitions suffisants à obtenir la bonne valeur pour $\mathsf{Pass}^{\varphi}(x)$. De façon similaire on obtient ceux suffisants pour le calcul de $\mathsf{Fail}^{\varphi}(x)$, puis on sélectionne un ensemble commun minimal. Il en existe au moins un qui est l'ensemble de toutes les transitions sortantes. Si il existe plusieurs ensembles valides et incomparables on en choisit un de façon arbitraire. Comme pour le choix d'une composante fortement connexe lors de la réduction précédente, on veux sélectionner l'ensemble qui permet d'enlever le maximum d'états du graphe mais on ne propose pas ici de stratégie.

Définition 20 (Transitions sortantes suffisantes)**.** Soit r un ensemble d'arêtes sortantes d'un noeud x, $x_{/r}$ représente le noeud x du LTS obtenu en ne conservant que celles ci (parmi les arêtes sortantes de x). Soient B_m et B_M des formules booléennes sur l'ensemble des variables d'états de φ externes au module. On construit l'ensemble $T_p(x, \varphi, [B_m, B_M])$ qui contient des ensembles d'arêtes sortantes de x, de telle sorte que pour tout r appartenant à $T_p(x, \varphi, [B_m, B_M])$, on ait $B_m \Rightarrow \mathsf{Pass}^{\varphi}(x_{/r}) \Rightarrow B_M$. On construit pour Fail, $T_f(x, \varphi, [B_m, B_M])$ de manière similaire.

[1] Cas $\varphi = \langle a \rangle \psi$ ou $\varphi = [a]\psi$;
r appartient à $T_p(x, \varphi, [B_m, B_M])$ si on a $B_m \Rightarrow \mathsf{Pass}^{\varphi}(x_{/r}) \Rightarrow B_M$. Il appartient à $T_f(x, \varphi, [B_m, B_M])$ si on a $B_m \Rightarrow \mathsf{Fail}^{\varphi}(x_{/r}) \Rightarrow B_M$.

[2] Cas $\varphi = \varphi_1 \vee \varphi_2$;
r appartient à $T_p(x, \varphi_1 \vee \varphi_2, [B_m, B_M])$ s'il existe une formule B' telle que $B' \Rightarrow B_m$ et telle que r est à la fois un élément de $T_p(x, \varphi_1, [B', B_M])$ et de $T_p(x, \varphi_2, [B_m \wedge \neg B', B_M])$. Le majorant B_M reste inchangé car pour avoir $\mathsf{Pass}^{\varphi_1 \vee \varphi_2}(x_{/r}) \Rightarrow B_M$ il faut avoir $\mathsf{Pass}^{\varphi_1}(x_{/r}) \Rightarrow B_M$ et $\mathsf{Pass}^{\varphi_2}(x_{/r}) \Rightarrow B_M$. Il suffit cependant de $B' \Rightarrow \mathsf{Pass}^{\varphi_1}(x_{/r})$ et $\neg B' \wedge B_m \Rightarrow \mathsf{Pass}^{\varphi_2}(x_{/r})$ pour avoir $B_m \Rightarrow \mathsf{Pass}^{\varphi_1 \vee \varphi_2}(x_{/r})$. On a aussi r appartient à $T_f(x, \varphi_1 \vee \varphi_2, [B_m, B_M])$ s'il existe une formule B' telle que $B_M \Rightarrow B'$ et telle que r est à la fois un élément de $T_f(x, \varphi_1, [B_m, B'])$ et de $T_f(x, \varphi_2, [B_m, B_M \vee \neg B'])$.

[3] Cas $\varphi = \varphi_1 \wedge \varphi_2$;
C'est le cas dual du précédent, r appartient à $T_p(x, \varphi_1 \wedge \varphi_2, [B_m, B_M])$ si il existe B' telle que $B_M \Rightarrow B$, et r appartient à la fois à $T_p(x, \varphi_1, [B_m, B'])$ et à $T_p(x, \varphi_2, [B_m, B_M \vee \neg B'])$. De plus, r appartient à $T_f(x, \varphi_1 \wedge \varphi_2, [B_m, B_M])$ si il existe B' telle que $B' \Rightarrow B_m$, et r appartient à la fois à $T_f(x, \varphi_1, [B', B_M])$ et à $T_f(x, \varphi_2, [B_m \wedge \neg B', B_M])$.

Sur un état (x, φ) tel que $\mathsf{Pass}^{\varphi}(x, \varphi) = \neg \mathsf{Fail}^{\varphi}(x, \varphi)$ on peut réduire le graphe en ne gardant qu'un ensemble d'arêtes minimal commun a $T_p((x, \varphi), \varphi, [\mathsf{Pass}^{\varphi}(x, \varphi), \mathsf{Pass}^{\varphi}(x, \varphi)])$ et $T_f((x, \varphi), \varphi, [\mathsf{Fail}^{\varphi}(x, \varphi), \mathsf{Fail}^{\varphi}(x, \varphi)])$. Cela ne modifie pas les valeurs de Pass et Fail, le nouvel état est alors équivalent à l'ancien.

5.2.4 Réduction par quotient

Si deux états équivalents appartiennent à la même composante fortement connexe du graphe S^{φ}, on ne peut pas *remplacer* l'un par l'autre (cf figure 5.8),

mais il est parfois possible de les fusionner. À cet effet, on définit la relation d'équivalence $\approx^{\varphi}$ qui est résistante à une réduction du graphe par quotient. On défini dans un premier temps une première relation $\sim^{\varphi}$ telle que deux états équivalents sont indifférentiables par φ (toujours par l'analyse du système global). On utilise alors le LTS S^{φ} de la sous-section 5.2.1 afin de savoir quelles sous-formules sont potentiellement vérifiées sur chaque état. Cela nous permet de fusionner les états étant au moins équivalents (pour $\sim$) pour ces formules. Sans cette information, il faudrait que les états fusionnés soient équivalents sur toutes les sous-formules possibles.

Étant donné un LTS S, deux états x, y, et une formule φ, on défini la fonction $\sim^{\varphi}_{S}$ grâce aux règles exposées en figure 5.11.

On écrit alors $x \sim^{\varphi} y$ si et seulement si on a $\sim^{\varphi}(x,y) = \top$.

$$\sim^{B}(x,y) \stackrel{\text{df}}{=} B@x \Leftrightarrow B@y \tag{5.12}$$

$$\sim^{\varphi_1 \wedge \varphi_2}(x,y) \stackrel{\text{df}}{=} \sim^{\varphi_1}(x,y) \wedge \sim^{\varphi_2}(x,y) \tag{5.13}$$

$$\sim^{\langle l \rangle \varphi}(x,y) \stackrel{\text{df}}{=} \mathsf{Fail}^{\langle l \rangle \varphi}(x) \vee \bigwedge_{x \xrightarrow{l} x'} \bigvee_{y \xrightarrow{l} y'} \sim^{\varphi}(x',y') \tag{5.14}$$
$$\bigwedge \mathsf{Fail}^{\langle l \rangle \varphi}(y) \vee \bigwedge_{y \xrightarrow{l} y'} \bigvee_{x \xrightarrow{l} x'} \sim^{\varphi}(y',x')$$

$$\sim^{\langle s \rangle \varphi}(x,y) \stackrel{\text{df}}{=} \mathsf{Fail}^{\langle s \rangle \varphi}(x) \tag{5.15}$$
$$\vee \begin{cases} \top & \text{si } \bigwedge_{x \xrightarrow{s} x'} \bigvee_{y \xrightarrow{s} y'} \sim^{\varphi}(x',y') \\ \bot & \text{sinon} \end{cases}$$
$$\bigwedge \mathsf{Fail}^{\langle s \rangle \varphi}(y)$$
$$\vee \begin{cases} \top & \text{si } \bigwedge_{y \xrightarrow{s} y'} \bigvee_{x \xrightarrow{s} x'} \sim^{\varphi}(y',x') \\ \bot & \text{sinon} \end{cases}$$

$$\sim^{\langle e \rangle \varphi}(x,y) \stackrel{\text{df}}{=} \begin{cases} \top & \text{si } \sim^{\varphi}(x,y) \\ \bot & \text{sinon} \end{cases} \tag{5.16}$$

$$\sim^{\mu X.\varphi}(x,y) \stackrel{\text{df}}{=} \bigvee \{ f(x,y) \mid \sim^{\varphi} f \stackrel{\mathcal{Q}^2}{\Leftarrow} f \} \tag{5.17}$$

$$\sim^{\neg \varphi}(x,y) \stackrel{\text{df}}{=} \sim^{\varphi}(x,y) \tag{5.18}$$

FIGURE 5.11 – Définition de la relation $\sim^{\varphi}$

Lemme 6. Pour toute formule φ, $\sim^{\varphi}$ est une relation d'équivalence calculable en un nombre d'itération fini.

La réduction du graphe consiste à fusionner les états équivalents par rapport à toutes les sous formules pouvant être vérifiées sur ces états.

Définition 21. Pour un LTS S et une formule ψ, on écrit $x \approx^{\psi} y$ si et seulement si on a $\mathsf{Form}^{\psi}(x) = \mathsf{Form}^{\psi}(y)$ et $x \sim^{\varphi} y$ pour chaque formule φ appartenant à $\mathsf{Form}^{\psi}(x)$.

On peut alors construire le graphe quotient de S par la relation $\approx^{\psi}$.

Définition 22. Soient $S \stackrel{\text{df}}{=} (Q, i, A, R, L)$ un LTS et φ une formule. La *réduction de S par rapport à* φ, est le LTS (Q', i', A', R', L') tel que :

- $Q' \stackrel{\text{df}}{=} Q/\approx^{\varphi}$ est l'ensemble des classes d'équivalence de la relation $\approx^{\varphi}$;
- $i' \stackrel{\text{df}}{=} [i]$;
- $A' = A$;
- $R' \stackrel{\text{df}}{=} \{(c_1, a, c_2) \in Q' \times A' \times Q' \mid \exists q_1 \in c_1, \exists q_2 \in c_2, (q_1, a, q_2) \in R\}$;
- $L'(c) \stackrel{\text{df}}{=} \bigvee_{q \in [c]} L(q)$.

Comme précédemment, les théorèmes suivants nous indiquent que cette opération de réduction préserve bien la valeur de vérité de la formule sur le système global.

Théorème 7 (Compositionnalité)**.** Soient x_1, y_1 deux états de S_1, x_2, y_2 deux états de S_2, et φ une formule. Si $x_1 \approx^{\varphi} y_1$ et $x_2 \approx^{\varphi} y_2$ alors $(x_1, x_2) \approx^{\varphi} (y_1, y_2)$.

Théorème 8 (Préservation de Pass et de Fail)**.** Soient x et y deux états de S, et ψ une formule. Si $x \approx^{\psi} y$ alors pour tout $\varphi \in \mathsf{Form}^{\psi}(x)$, $\mathsf{Pass}^{\varphi}(x) \Leftrightarrow \mathsf{Pass}^{\varphi}(y)$ et $\mathsf{Fail}^{\varphi}(x) \Leftrightarrow \mathsf{Fail}^{\varphi}(y)$.

Théorème 9 (Préservation de la relation par la réduction)**.** Pour tout état x et toute formule ψ, $x \approx^{\psi} [x]$.

Exemple 4. Le LTS de droite de la figure 5.12 est obtenu à partir de celui de gauche en appliquant la réduction quotient définie par la relation $\approx^{\Phi}$ (Φ étant définie dans l'exemple 2).

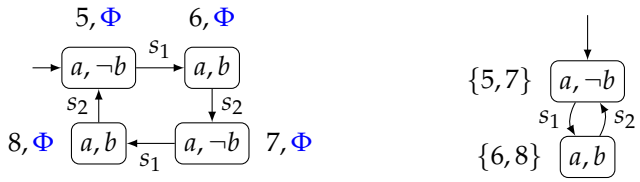


FIGURE 5.12 – Réduction par quotient.

Chapitre 6

Expérimentations

Ce chapitre présente quelques expérimentations réalisées à l'aide d'un prototype (comportant environ 2500 lignes de Haskell [54]) qui implémente les opérations d'accessibilité par une formule puis la réduction par raccourcissement de chemin (définies respectivement en sous-sections 5.2.1 et 5.2.2). Les autres réductions ne sont pas implémentées pour le moment.

Ces expérimentations ne constituent pas une campagne de benchmark rigoureuse (cela fait partie des perspectives) mais de premiers exemples relativement simples et artificiels ayant pour but principal de tester le prototype et de constater les réductions obtenues.

La section 6.1 comporte la définition formelle d'un réseau de Petri coloré, dont on se sert pour décrire de façon concise les différents modèles étudiés.

La section 6.2 étudie un module d'un système d'exclusion mutuelle, dont on peut augmenter la taille en modifiant le nombre de jetons initial du réseau de Petri. On compare les réductions obtenues avec la réduction SAFETY (cf 13) qui préserve également les propriétés considérées.

La section 6.3 considère un problème similaire au dîner des philosophes [38], où chaque agent a besoin de s'approprier un sous-ensemble donné des ressources communes pour pouvoir effectuer une action (puis il les libère). En modifiant les nombres d'agents et de ressources disponibles et les relations d'accessibilité des ressource pour les agents, on peut générer différents LTS modulaires. Pour ces différents systèmes, on s'intéresse alors aux coefficients de réduction obtenus pour toutes les agrégations possibles, dans l'optique de comparer ces résultats à la structure du système (en terme de ressources partagées) et d'obtenir des données pour étudier le problème de l'ordre d'analyse hiérarchique (cf. section 2.4.1).

6.1 Réseaux de Petri colorés

On rappelle brièvement la définition des réseaux de Petri colorés, qui vont nous permettre de décrire les différent modules des LTS modulaires utilisés dans nos expériences. Le comportement d'un tel LTS modulaire est également le comportement du réseau de Petri modulaire avec partage de transitions composé des réseaux de Petri décrivant les différents modules [88, 25, 24].

Le langage de coloration d'un réseau de Petri est décrit de manière abstraite par les ensembles suivants.

- $\mathbb{D}$ est l'ensemble des *valeurs* manipulables par le langage. Il peut contenir en particulier le jeton noir classique $\bullet$, les valeurs Booléennes True et False, et une valeur spéciale '**undefined**' ;
- $\mathbb{V}$ est un ensemble de *variables* ;
- $\mathbb{E}$ est l'ensemble des *expressions* du langage. Pour une expression donnée e, on note **vars**(e) l'ensemble des variables présentes dans e.

Une valuation est ici une fonction partielle $\beta : \mathbb{V} \to \mathbb{D}$ associant des valeurs à un ensemble de variables. Pour une expression $e \in \mathbb{E}$ on désigne par $\beta(e) \in \mathbb{D}$ l'évaluation de e pour la valuation β. Si cette évaluation échoue, par exemple si e est syntaxiquement incorrecte ou si β n'est pas définie pour toutes les variables de **vars**(e), on a alors $\beta(e) \stackrel{\text{df}}{=} \textbf{undefined}$.

Définition 23 (Réseau de Petri)**.** Étant donnés un ensemble de *types* $\mathbb{D}$ Un réseau de Petri est un triplet (P, T, l) où :

- P est un ensemble fini de *places* ;
- T est un ensemble fini de *transitions*, disjoint de P ;
- l est une fonction d'étiquetage telle que :
  - pour toute place p de $P, l(p) \subseteq \mathbb{D}$ est le *type* des jetons que peut contenir cette place,
  - pour toute transition t de $T, l(t) \in \mathbb{E}$ est la *garde* de la transition t, c'est-à-dire une condition permettant son exécution,
  - pour tout couple (x, y) de $(P \times T) \cup (T \times P), l(x, y)$ est un multiensemble de $\mathbb{E}$ définissant l'arc de x vers y.

Définition 24 (Comportement d'un réseau de Petri)**.** Un marquage M d'un réseau de Petri est une fonction qui associe à chaque place p un multiensemble fini de jetons qui appartiennent chacun à $l(p)$. Une transition t est dite franchissable pour un marquage M et une valuation β si et seulement si les conditions suivantes sont vérifiées :

- M possède suffisamment de jetons. C'est-à-dire que pour toute place p, on a $\beta(l(p,t)) \leq M(p)$;
- la garde est vérifiée : $\beta(l(t))$ = True ;
- les types des places sont respectés. C'est-à-dire que pour toute place p, $\beta(l(t,p))$ est un multiensemble d'éléments de $l(p)$.

Si une transition t du réseau de Petri est franchissable dans un marquage M à l'aide d'une valuation β, on peut tirer cette transition et atteindre le marquage M' où chaque place p contient le multiensemble de jetons $M(p) - \beta(l(p,t)) + \beta(l(t,p))$. On écrit alors $M[t,\beta\rangle M'$.

Le comportement du réseau accessible depuis le marquage M_0 est le plus petit système de transition G tel que :

- M_0 est un état de G
- si M est un noeud de G et $M[t,\beta\rangle M'$, alors M' est également un état de G et $M \xrightarrow{t,\beta} M'$ appartient à la relation de transition de G.

6.2 Système d'exclusion mutuel

On considère un système d'exclusion mutuelle, décrit par le réseau de Petri de la figure 6.1. Ce modèle permet de générer des LTS de tailles différentes en modifiant le nombre initial de jetons présents dans la place *i*. C'est un *réseau de Petri modulaire* avec partage de transitions et sa sémantique est le LTS modulaire dont les modules sont les comportements des différents réseaux de Petri [24]. On peut donc utiliser ces modèles afin de générer des LTS modulaires sur lesquelles nos techniques d'abstraction hiérarchique sont utilisables.

Les transitions *enter* et *leave* étant synchronisées entre les deux modules, le second module assure qu'il y a nécessairement une action *leave* entre deux actions *enter*, avec pour objectif de limiter le nombre de jetons dans la place *crit* du premier module à 1. Ce système n'est cependant pas correct car la transition locale $l3$ produit également des jetons dans la place *crit*.

On étudie la réduction du comportement du module situé en haut de la figure 6.1 à l'aide de la réduction SKIP avec des propriétés de sûreté et d'accessibilité. On compare les réductions obtenues avec la réduction SAFETY (cf définition 13), qui préserve également ces propriétés. Comme dans le chapitre 4 on utilise pour cela le logiciel CADP.

On appelle $A \stackrel{\text{df}}{=} \{l_1, l_2, l_3, enter, leave\}$ l'ensemble des transitions différentes de $loop$, et p une constante propositionnelle dont la valeur est vraie si et seulement si il y a au moins deux jetons dans la place *crit*.

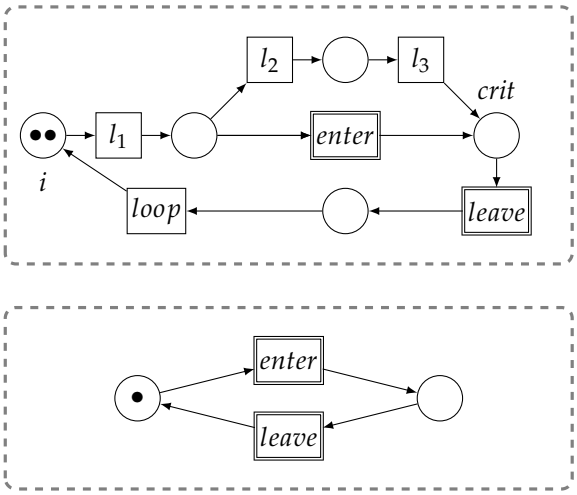


FIGURE 6.1 – Réseau de Petri modulaire modélisant un système d'exclusion mutuelle, chaque module est encadré en pointillés, les transitions partagées sont dessinées avec des bordures doubles.

On définit également H^{φ} comme l'ensemble maximal d'actions qu'il est possible de dissimuler avant la réduction par quotientage du module selon la relation d'équivalence SAFETY.

On considère ici trois formules différentes. La première est un exemple sur lequel la méthode SKIP n'offre aucune réduction, la seconde est un cas où la réduction obtenue est maximale car on retourne un graphe équivalent possédant un unique état, et la dernière permet d'obtenir une réduction plus équilibrée. Ces résultats sont présentés Figure 6.2

$$\varphi_1 \stackrel{\text{df}}{=} \mu X(\langle enter \rangle true \vee \langle A \cup \{loop\} \rangle X) \tag{6.1}$$

Cette formule signifie qu'on peut atteindre un état dans lequel *enter* est une action possible. SKIP ne permet de réduire ce graphe à l'aide de la formule φ_1 pour deux raisons. Premièrement, le graphe d'accessibilité $S_i^{\varphi_1}$ est composé d'une unique composante fortement connexe, ce qui implique qu'on ne pourra jamais trouver de remplaçants adaptés a un état. Deuxièmement, aucun état n'appartient à $\mathsf{Pass}^{\varphi_1}$ ou à $\mathsf{Fail}^{\varphi_1}$. Ils doivent donc tous être explorés lors de la construction de $S_i^{\varphi_1}$.

Afin d'obtenir un exemple sur lequel la réduction SKIP est applicable, on considère maintenant des formules ne contenant pas l'action *loop*, ce qui a pour conséquence de retirer les cycles du graphe $S_i^{\varphi_1}$.

$$\varphi_2 \stackrel{\text{df}}{=} \mu X(p \vee \langle A \rangle X) \tag{6.2}$$

La Formule φ_2 exprime l'accessibilité d'un état ayant deux jetons dans la place *crit*. Cette propriété est préservée par la réduction SAFETY si on dissimule toutes les actions locales, les actions synchronisées devant être préservées pour pouvoir composer les modules par la suite. On a donc $H^{\varphi_2} \stackrel{\text{df}}{=} \{l_1, l_2, l_3\}$.

En fait, il est ici possible de déterminer localement la valeur de vérité de φ_2, et la réduction SKIP retourne toujours un LTS comportant un unique état. Dans la cas où le réseau de Petri est initialisé avec un seul jeton, l'état initial appartient à $\mathsf{Fail}^{\varphi_2}$ car il n'est jamais possible d'avoir deux jetons en place *crit* (le nombre total de jetons restant constant). Le module peut alors être réduit à un seul état satisfaisant $\neg p$.

Dans le cas où la place i contient initialement deux jetons ou plus, il est possible de placer deux jetons dans la place *crit* en utilisant uniquement des transitions locales. L'état initial appartient à $\mathsf{Pass}^{\varphi_2}$ et le module est équivalent à celui composé d'un seul état vérifiant p.

$$\varphi_3 \stackrel{\text{df}}{=} \nu Y(\neg p \wedge [A \setminus l_2]Y) \tag{6.3}$$

La formule φ_3 signifie que tous les chemins menant à un état vérifiant p contiennent l'action l_2. Pour cette propriété, l'action l_2 doit rester visible car on doit pouvoir la différencier des autres actions locales. On a donc $H^{\varphi_3} \stackrel{\text{df}}{=} \{l_1, l_3\}$. Cette action locale est par contre ignorée lors des calculs du graphe d'accessibilité, et de la relation d'équivalence (possible car on a une connaissance parfaite de la formule et de l'état initial que la réduction SAFETY ne suppose pas).

Si le réseau de Petri comporte un unique jeton, comme précédemment le graphe est réduit à un unique état. Dans les autres cas, le graphe réduit obtenu conserve les actions synchronisées.

En pratique, même si la relation d'équivalence basée sur la formule est plus large que la relation SAFETY, on gagne en général à combiner les réductions SKIP et SAFETY pour deux raisons. D'une part la réduction SAFETY peut opérer au sein d'une composante fortement connexe où SKIP est inefficace , d'autre part elle est plus rapide et permet de calculer la réduction SKIP (qui prend plus de temps) sur un système réduit (en pratique la réduction SAFETY implémentée par CADP est instantanée sur les espaces d'états de tailles manipulables par notre prototype)

nombre initial de jetons		1	2	5	10
LTS complet (sans réduction)		8	18	128	1003
φ_1	SAFETY	4	7	22	67
	SKIP	8	18	128	1003
φ_2	SAFETY	4	7	22	67
	SKIP	1	1	1	1
φ_3	SAFETY	5	11	57	287
	SKIP	1	3	10	20

FIGURE 6.2 – Nombre d'états en fonction de la formule, du nombre initial de jetons dans la place i et de la réduction utilisée.

6.3 Le problème des philosophes

Dans l'optique de tester différents ordres d'analyse hiérarchique d'un système et de les comparer avec sa topologie en termes d'actions partagées, on considère le problème classique des philosophes. Les diffèrent agents (modules) du système ont des comportements similaires et on peut construire plusieurs expériences en définissant quels agents ont accès à quelles ressources.

6.3.1 Modélisation du système

Les agents peuvent prendre possession des ressources à leurs dispositions (action Take) puis les remettre à disposition des autres agents (action Free). Un agent ayant pris possession d'un nombre minimum de ressources peut effectuer l'action Eat et atteindre son but. À partir de ce moment il peut uniquement libérer les ressources dont il a déjà pris possession.

Ici, une topologie est un graphe biparti $T = (P_h, R_e, E)$ dans lequel P_h est l'ensemble des agents, R_e est l'ensemble des ressources et $E \subseteq P_h \times R_e$ est un ensemble d'arêtes spécifiant les accès qu'ont les différents agents aux ressources. On définit l'ensemble des ressources accessibles à un agent a de P_h, $\mathsf{Res}(a) \stackrel{\text{df}}{=} \{r \in R_e \mid (a, r) \in E\}$. La figure 6.3 présente un exemple possible de topologie, comportant trois agents et trois ressources, ainsi que le comportement d'un de ces agents. La figure 6.4 contient un réseau de Petri décrivant le comportement d'un agent i quelconque ayant besoin de trois ressources différentes au sein d'une topologie donnée. La place *Resources* contient initialement l'ensemble des ressources accessibles à cet agent. Il peut effectuer l'action « Take r i » pour s'approprier la ressource r et la déplacer dans la place

Taken. Si j est un second agent ayant accès à la ressource r, i peut également effectuer l'action « Take r j » qui a pour effet de déplacer la ressource dans la place *In use*. En se synchronisant sur les actions de mêmes noms, chaque agent est donc au courant lorsqu'une ressource devient indisponible. Le principe est similaire pour la libération des ressources. Par ailleurs, si l'agent i possède trois jetons dans la place *Taken* et un dans la place *Hungry*, il peut effectuer l'action "Eat i". À partir de ce moment le jeton de la place *Hungry* disparaît et l'agent ne peut plus s'approprier de ressources, seulement les libérer (et continuer les opérations de synchronisation avec ses voisins pour ne pas bloquer leurs exécutions).

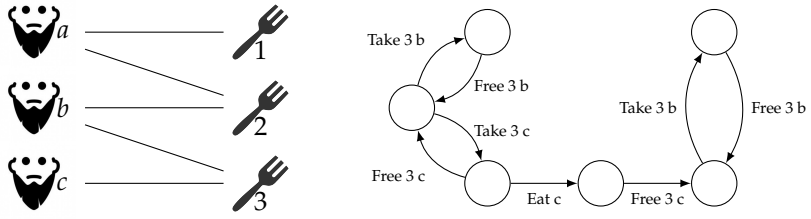


FIGURE 6.3 – A gauche : un exemple de topologie. $T = (\{a,b,c\},\{1,2,3\},\{(a,1),(a,2),(b,2),(b,3),(c,3)\})$. A droite : le comportement de l'agent c de cette topologie.

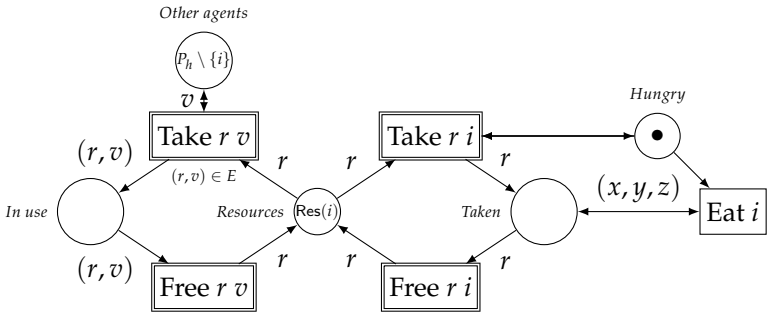


FIGURE 6.4 – Réseau de Petri décrivant le comportement d'un agent i, au sein d'une topologie $T = (P_h, R_e, E)$, ayant besoin d'acquérir trois ressources quelconques pour effectuer l'action Eat.

6.3.2 Méthode d'analyse

On considère maintenant une topologie particulière et on choisit le nombre de ressources nécessaires aux agents. Le système est alors décrit par le LTS modulaire $(S_i)_{i\in 1..n}$, où chaque agent est un module. Par la suite, on appelle *agrégations* les LTS modulaires composé de sous-ensembles de ces modules. Afin d'identifier l'influence des différents modules sur l'efficacité d'une analyse par abstraction hiérarchique, on compare les réductions obtenues sur toutes les agrégations possibles. Pour une agrégation J et une formule φ donnée on définit le *coefficient de réduction* obtenu, qui est le rapport entre la taille du produit des modules et la taille de ce produit réduit avec Skip, divisé par le nombre de module de l'agrégation :

$$\mathsf{Coef}^{\varphi}(J) = \frac{\mathsf{lts}(J)}{\mid \mathsf{Skip}^{\varphi}(\mathsf{lts}(J)) \mid \times \mid J \mid}.$$

Parfois, toutes les agrégations contenant une sous-agrégation particulière peuvent être réduites en un LTS de taille très faible, par exemple si l'état initial de cette sous-agrégation appartient à Pass. Dans ce cas les agrégations de plus grandes tailles auront un coefficient de réduction plus important que les autres mais ne sont pas nécessairement plus intéressantes. On divise donc le coefficient par le nombre de modules de l'agrégation pour atténuer cet effet. On définit ensuite une fonction d'efficacité qui accepte une agrégation si son coefficient de réduction est supérieur à un seuil donné :

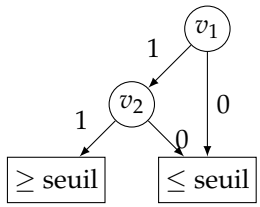
$$\mathsf{Eff}^{\varphi}_{\mathsf{seuil}}(J) = \mathsf{Coef}^{\varphi}(J) \geq \mathsf{seuil}.$$

En représentant une agrégation J par un valuation de n variables booléennes $v_1 \dots v_n$ (v_i est vrai si le module i appartient à l'agrégation), on peut exprimer la fonction $\mathsf{Eff}^{\varphi}_{\mathsf{seuil}}$ sous la forme d'un diagramme de décision binaire [67]. Cela permet d'une part de la visualiser et d'autre part d'identifier les modules ou ensembles de modules ayant une réelle influence sur l'efficacité de la réduction. Afin d'améliorer la lisibilité des diagrammes et de mettre en évidence les modules ayant des rôles symétriques, on peut parfois fusionner plusieurs noeuds en un.

- Le label du noeud créé correspond à l'ensemble des variables des noeuds supprimés.
- Les labels des arcs sortant de ce nouveau noeud représentent alors le nombre de variables étant vraies parmi celles apparaissant sur le noeud.

Cette opération est illustrée figure 6.5.

J	$\mathsf{Eff}^{\varphi}_{\mathsf{seuil}}(J)$
$\varnothing$	0
$\{1\}$	0
$\{2\}$	0
$\{1,2\}$	1



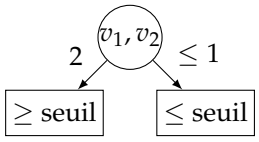


FIGURE 6.5 – Différentes représentations d'une fonction d'efficacité sur un système comportant deux modules. À gauche, les valeurs de la fonction pour chaque agrégation possible. À droite, le diagramme de décision correspondant. En bas, le diagramme simplifié : la fonction est vraie si et seulement si deux variables sont vraies parmi v_i et v_2.

On peut maintenant adapter la valeur du seuil pour obtenir différent types d'informations sur les rôles des modules. Si la valeur du seuil est 1, on accepte toutes les agrégations non vides comme efficaces. En augmentant la valeur du seuil les premières agrégations rejetées sont les moins efficaces. En étudiant le diagramme de décision correspondant, il peut par exemple être intéressant de constater que deux modules doivent nécessairement être composés ensembles pour obtenir un coefficient de réduction acceptable. En continuant à augmenter la valeur du seuil jusqu'à ne plus accepter qu'un petit nombre d'agrégations, on peut identifier les modules et combinaisons permettant les meilleurs réductions. On peut alors par exemple détecter que si deux modules appartiennent à une agrégation, celle-ci pourra toujours être réduite de façon importante.

6.3.3 Résultats

Pour une topologie comportant n modules, on définit les variables booléennes $\mathsf{end}_1, \ldots, \mathsf{end}_n$ telles que end_i signifie qu'il y a un jeton dans la place *En cours* de l'agent i, c'est-à-dire qu'il n'a pas encore effectué l'action *Eat*. La formule $\varphi^{\mathsf{ended}}_m$ signifiant qu'il est possible que m agents différents réussissent à effectuer l'action *Eat* s'exprime alors ainsi (en utilisant l'opérateur $\exists\Diamond$ de

CTL défini page 39) :

$$\varphi_m^{\mathsf{ended}} \stackrel{\text{df}}{=} \exists\Diamond \bigvee_{ps\subseteq P_h, |ps|=m} \bigwedge_{i\in ps} \mathsf{end}_i.$$

La formule $\varphi^{no\ loop}$ signifie que tous les chemins du systèmes sont finis (La propriété reste récursivement vraie pour tout successeur, quelque soit le label de la transition).

$$\varphi^{no\ loop} \stackrel{\text{df}}{=} \mu X[-]X$$

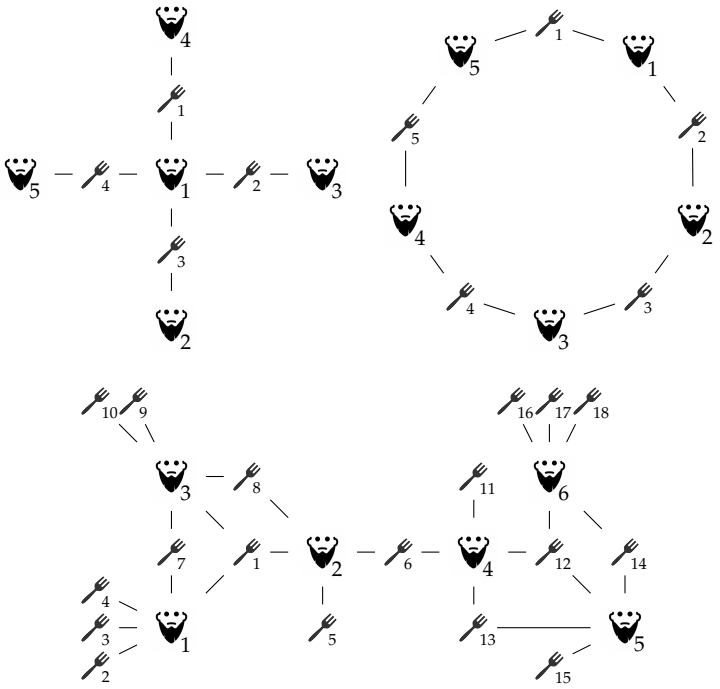


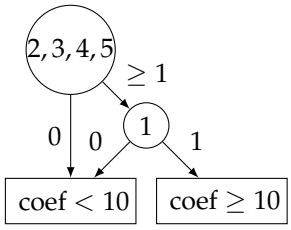
FIGURE 6.6 – Les différentes topologies considérées.

Présence d'un cycle pour la topologie en étoile.

On considère la propriété $\varphi^{no\ loop}$ exprimant que le système ne possède pas de cycle sur la topologie en étoile dans le cas où un agent a besoin de toutes ses ressources accessibles. Les coefficients de réduction obtenus sont présentés figure 6.7. Le diagramme de décision obtenu avec un seuil de 15 et correspondant aux meilleurs réductions possibles est le suivant :

Agrégation	Taille réelle	Taille avant réduction	Taille réduite	Coefficient
1	101	101	100	1.0
4	6	6	5	1.2
3	6	6	5	1.2
2	6	6	5	1.2
5	6	6	5	1.2
12	193	165	8	12.1
13	193	165	8	12.1
14	193	165	8	12.1
15	193	165	8	12.1
25	36	25	13	1.4
32	36	25	13	1.4
35	36	25	13	1.4
42	36	25	13	1.4
43	36	25	13	1.4
45	36	25	13	1.4
125	373	20	4	31.1
132	373	20	4	31.1
135	373	20	4	31.1
142	373	20	4	31.1
143	373	20	4	31.1
145	373	20	4	31.1
325	216	65	35	2.1
425	216	65	35	2.1
432	216	65	35	2.1
435	216	65	35	2.1
1325	727	10	1	181.8
1425	727	10	2	90.9
1432	727	10	1	181.8
1435	727	10	1	181.8
4325	1296	169	97	3.3
14325	1425	5	1	285

FIGURE 6.7 – Résultats pour la formule $\varphi^{no\ loop}$ sur la topologie en étoile. La taille réelle d'une agrégation est le nombre d'état du produit de ses différents modules. La taille avant réduction est la meilleur taille intermédiaire obtenue en appliquant le processus de réduction hiérarchique.



Pour obtenir une réduction importante, il est nécessaire que le module 1 soit présent dans l'agrégation, aux cotés d'au moins un autre module. En fait l'agrégation possède dans ce cas un cycle constitué d'actions locales, qui sera donc aussi présent sur le système total.

Topologie en cercle On considère la formule $\varphi_1^{\mathsf{ended}}$, sur la topologie circulaire de la figure 6.6. La figure 6.8 présente les résultats obtenus dans le cas où les agents ont besoin de deux ressources pour pouvoir terminer. On observe que les réductions les plus importantes (où le LTS réduit possède deux ou trois états) sont obtenues pour les agrégations contenant au moins trois modules voisins. Dans ce cas (et seulement dans ce cas) l'agrégation peut at-

Agrégation	Taille réelle	Taille avant réduction	Taille réduite	Coefficient
1	15	15	15	1
2	15	15	15	1
3	15	15	15	1
4	15	15	15	1
5	15	15	15	1
12	72	82	36	1.0
13	225	225	105	1.1
14	225	225	105	1.1
15	72	82	30	1.2
23	72	82	36	1.0
24	225	225	105	1.1
25	225	225	105	1.1
34	72	82	36	1.0
35	225	225	105	1.1
45	72	82	33	1.1
123	346	200	2	57.7
124	1080	540	333	1.1
125	346	167	3	38.4
134	1080	540	303	1.2
135	1080	450	273	1.3
145	346	220	3	38.4
234	346	207	2	57.7
235	1080	540	303	1.2
245	1080	495	305	1.2
345	346	183	2	57.7
1234	1664	15	2	208.0
1235	1664	44	3	138.7
1245	1664	75	3	138.7
1345	1664	15	2	208.0
2345	1664	15	2	208.0
12345	2575	9	2	257.5

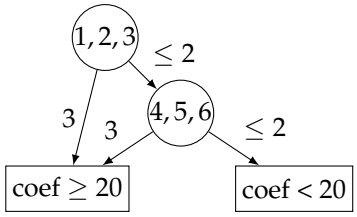
FIGURE 6.8 – Réductions obtenues pour la formule $\varphi_1^{\mathsf{ended}}$ sur la topologie circulaire. Deux ressources sont nécessaires à effectuer l'action Eat.

teindre un état où l'agent situé au milieu a terminé à l'aide d'actions locales, et on sait donc que la formule est globalement vraie. L'état initial de l'agrégation peut donc être directement remplacé par un état où l'action locale "Eat" est possible. La figure 6.9 présente des résultats similaires dans le cas où les modules ont seulement besoin d'une ressource pour pouvoir effectuer l'action "Eat". Ici il est nécessaire et suffisant qu'une agrégation possède deux modules voisins pour pouvoir conclure. La construction d'un diagramme de décision n'aide pas ici à l'interprétation des résultats. En effet, le fait qu'une agrégation possède des modules voisins ne s'exprime pas bien à l'aides des variables booléennes.

Un agent peut terminer sur la troisième topologie La figure 6.10 comporte les résultats obtenus sur la dernière topologie de la figure 6.6, dans le cas où un agents à besoin de toutes ses ressources pour terminer. La formule considérée est $\varphi_1^{\mathsf{ended}}$ (un état où un module a terminé est accessible) et le diagramme de décision obtenu est le suivant :

Agrégation	Taille réelle	Taille avant réduction	Taille réduite	Coefficient
1	15	15	11	1.4
2	15	15	11	1.4
3	15	15	11	1.4
4	15	15	11	1.4
5	15	15	11	1.4
12	72	48	3	12,0
13	225	121	53	2,1
14	225	121	53	2,1
15	72	48	3	12,0
23	72	48	3	12,0
24	225	121	53	2,1
25	225	121	53	2,1
34	72	48	3	12,0
35	225	121	53	2,1
45	72	48	3	12,0
123	346	33	3	38,4
124	1080	33	3	120,0
125	346	21	2	57,7
134	1080	33	3	120,0
135	1080	33	3	120,0
145	346	33	3	38,4
234	346	21	2	57,7
235	1080	33	3	120,0
245	1080	33	3	120,0
345	346	21	2	57,7
1234	1664	9	3	138,7
1235	1664	11	2	208,0
1245	1664	9	2	208,0
1345	1664	9	3	138,7
2345	1664	9	3	138,7
12345	2575	3	3	171,7

FIGURE 6.9 – Réductions obtenues pour la formule $\varphi_1^{\mathsf{ended}}$ sur la topologie circulaire. Une seule ressource suffit pour effectuer l'action Eat.



Encore une fois les agents sont regroupés en fonction de leur proximité en terme de ressources partagées. Les modules 1,2 et 3 partagent notamment l'action « F_1 », et il est nécessaire qu'ils fassent partie de la même agrégation pour que cette ressource soit locale, ce qui permet de trouver un chemin local permettant à tous ces modules de terminer. De même pour les modules 4,5 et 6 avec l'action « F_{12} ».

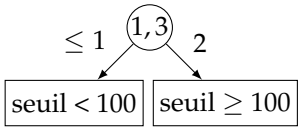
Libération des ressources sur la troisième topologie On considère maintenant la formule $\varphi_{1,F_7}^{liberee}$ sur le même système signifiant que si l'agent 1 prend possession de la ressource F_7, il a la possibilité de la libérer après un nombre fini d'actions. En notant $\mathcal{O}$ l'ensemble des actions différentes de Free F_7 1, cette propriété s'exprime ainsi :

Agrégation	Taille réelle	Taille avant réduction	Taille réduite	Coefficient
1	78	78	15	5.2
2	76	76	38	2
3	110	11	38	2.8
4	76	76	38	2
5	71	71	38	1.8
6	69	69	15	4.6
12	250	269	126	1
13	83	89	40	1
14	570	570	295	1
15	570	570	295	1
16	225	225	105	1.1
23	215	225	117	0.9
24	474	502	284	0.8
25	1444	1444	850	0.8
26	570	570	299	1
34	1444	1444	850	0.8
35	1444	1444	850	0.8
36	570	570	299	1
45	215	225	120	0.9
46	250	269	128	1
56	83	89	40	1
123	345	180	2	57.5
124	3122	1671	1188	0.9
125	9500	-	-	-
126	3750	1842	1233	1
134	3154	1520	1104	1
135	3154	1520	1104	1
136	1245	600	382	1.1
145	3225	1715	1175	0.9
146	3750	1820	1235	1
156	1245	690	375	1.1
234	2683	1541	1075	0.8
235	8170	-	-	-
236	3225	1755	1143	0.9
245	2683	1588	1134	0.8
246	3122	1691	1190	0.9
256	3154	1520	1082	1
345	8170	-	-	-
346	9500	-	-	-
356	3154	1520	1082	1
456	345	224	3	38
1234	4313	36	2	539
1235	13110	114	3	1 092
1236	5175	45	3	431
1245	17674	-	-	-
1246	20560	-	-	-
1256	20750	-	-	-
1345	17845	-	-	-
1346	20750	-	-	-
1356	6889	1600	1408	1.2
1456	5175	45	3	431
2345	15191	-	-	-
2346	17674	-	-	-
2356	17845	-	-	-
2456	4313	1288	2	539
3456	13110	114	3	1 092
12345	24425	36	2	2 442
12346	28414	36	3	1 894
12356	28635	45	3	1 909
12456	28414	165	3	1 894
13456	28635	120	3	1 909
23456	24425	234	2	2 442
123456	39271	9	3	2 181

FIGURE 6.10 – Réductions obtenues sur la troisième topologie pour la formule $\varphi_1^{\mathsf{ended}}$. Un résultat absent signifie que le calcul nécessite trop de mémoire (quelque soit l'ordre de réduction hiérarchique choisit).

$$\varphi^{liberee}_{1,F_7} \;=\; [\text{Take } F_7\ 1]\mu X.\langle\mathcal{O}\rangle X \vee \langle\text{Free } F_7\ 1\rangle\top \ .$$

Cette propriété est vérifiée car l'agent 1 a la possibilité de libérer la ressource juste après l'avoir acquise. Les résultats apparaissent figure 6.11. pour un seuil de 100 on obtient le diagramme :



Lorsque 1 et 3 appartiennent à la même agrégation, F_7 est une action locale et on sait que la propriété est vraie et la réduction est alors importante. Dans le cas contraire (notamment lorsque F_7) est une action externe a l'agrégation, le coefficient de réduction est très faible car même les actions locales sont importantes dans la mesure où elles comportent des cycles pouvant avoir un effet sur la valeur globale de la formule. Dans ce cas l'explosion combinatoire fait qu'on atteint très vite des tailles trop importantes et notre prototype échoue par manque de mémoire.

Agrégation	Taille réelle	Taille avant réduction	Taille réduite	Coefficient
1	93	93	9	10.3
2	83	83	66	1.3
3	139	139	43	3.2
4	83	83	66	1.3
5	79	79	66	1.2
6	88	88	61	1.4
12	3550	286	165	10.8
13	1926	142	4	240.8
14	7719	594	360	10.7
15	7347	594	348	10.6
16	8184	549	261	15.7
23	1700	428	301	2.8
24	2297	1425	1425	0.8
25	6557	-	-	-
26	7304	-	-	-
34	11537	-	-	-
35	10981	-	-	-
36	12232	-	-	-
45	968	647	647	0.7
46	3356	1925	1925	0.9
56	1031	602	602	0.9
123	17151	174	4	1429.3
124	98249	-	-	-
125	280450	-	-	-
126	312400	-	-	-
134	159858	264	18	2960.3
135	152154	264	18	2817.7
136	169488	244	10	5649.6
145	90024	-	-	-
146	312108	-	-	-
156	95883	-	-	-
234	47047	-	-	-
235	134300	-	-	-
236	149600	-	-	-
245	26790	-	-	-
246	92880	-	-	-
256	85573	-	-	-
345	134552	-	-	-
346	466484	-	-	-
356	143309	-	-	-
456	9385	-	-	-
1234	428572	175	10	10714.3
1235	428572	264	18	5952.4
1236	428572	244	10	10714.3
1245	428572	-	-	-
1246	-	-	-	-
1256	-	-	-	-
1345	-	423	66	∞
1346	-	571	66	∞
1356	-	319	34	∞
1456	-	-	-	-
2345	-	-	-	-
2346	-	-	-	-
2356	-	-	-	-
2456	-	-	-	-
3456	-	-	-	-
12345	-	742	10	∞
12346	-	879	130	∞
12356	-	1015	130	∞
12456	-	-	-	-
13456	-	976	66	∞
23456	-	-	-	-
123456	-	210	10	∞

FIGURE 6.11 – Réductions obtenues pour la formule $\varphi^{liberee}_{1,F_7}$ sur la troisième topologie. Les agents peuvent terminer si ils ont acquis plus de la moitié leurs ressources. Un résultat absent signifie que le calcul nécessite trop de mémoire (quelque soit l'ordre de réduction hiérarchique choisit). On assigne un coefficient de réduction infini si l'on échoue à déterminer le nombre d'états du LTS concret.

Chapitre 7

Conclusion

Dans cette thèse, on s'est intéressé à combattre l'explosion combinatoire du model-checking, dans le but de permettre l'analyse de systèmes modulaires de plus grandes tailles.

On exploite pour cela la structure modulaire des systèmes ainsi que notre connaissance de la formule à vérifier en appliquant une approche par réductions hiérarchiques. D'une part, on construit une abstraction spécifique à cette formule du système de manière incrémentale, en alternant des réductions et des compositions de ses différentes parties. D'autre part on détecte lors de la réduction d'un module ou d'un ensemble de modules, si ceux-ci comportent assez d'information pour déterminer la valeur de vérité de la formule sur le système complet, auquel cas on peut stopper l'analyse et conclure directement.

Dans cette optique on peut distinguer trois contributions principales :

- Le modèles des réseaux de régulation génétique modulaires, permettant de décrire des systèmes biologiques de façon adaptée à une analyse hiérarchique ainsi qu'une application au problème de l'accessibilité des états stables de ces systèmes.
- Deux techniques de réductions dépendant d'une formule de μ-calcul, permettant de réduire le comportement d'une sous-partie d'un système en ayant la garantie de préserver la formule d'intérêt sur le système global.
- Un prototype permettant de tester la première de ces deux technique de réduction ainsi que de premières expérimentations sur des modèles simples.

La première perspective théorique proche consiste à trouver des algorithmes ou techniques plus efficaces pour calculer les différentes réduction présentées ici (par exemple en utilisant des représentations symboliques si possible) et à obtenir des résultats sur la complexité du problème. On constate en effet que notre prototype atteint ses limites sur des systèmes de tailles relativement faibles, comme on le voit dans les expérimentations. Au besoin, on pourrait envisager de restreindre l'ensemble des systèmes considérés pour utiliser des algorithmes plus spécifiques.

La seconde perspective est de préciser certains critères de choix qui n'apparaissent pas dans les définitions. Dans la partie *raccourcissent de chemins* (sous-section 5.2.2), parmi un ensemble d'états équivalents, on en choisit un appartenant à une composante fortement connexe minimale, mais la stratégie pour le choix de cette composante n'est pas précisée. La situation est similaire dans la partie *suppression des arcs sortants inutiles* (sous-section 5.2.3). On y choisit un ensemble d'arcs sortants minimal, mais on ne précise pas comment distinguer ces ensembles minimaux. Dans les deux cas, le choix effectué a potentiellement une forte influence sur la taille du modèle réduit. Déterminer le choix optimal ou au moins de bonnes heuristiques est donc important.

Au niveau de l'implémentation, la poursuite du travail consistera à implémenter la réduction par quotient, puis les critères de choix mentionnés ci-dessus le cas échéant. Une autre branche consistera à transformer le prototype en un outil distribuable ainsi qu'a effectuer une campagne de benchmarks plus complète, par exemple avec des modèles issus du Model-Checking Contest [58] ou des exemples biologiques tels que celui de la drosophile [103, 102] utilisé au chapitre 4.

En perspective théorique à plus long terme, le problème du meilleur ordre d'analyse est critique car cet ordre a une influence drastique sur l'efficacité de la méthode. En étudiant les résultats des expérimentations dont on dispose et de celles à venir, on espère améliorer notre compréhension de ce problème et par exemple déterminer à quel point l'heuristique *smart reduction* de CADP [36] utilisée au chapitre 4 est pertinente pour les réductions définies ici.

Chapitre 8

Preuves

La plupart des preuves suivantes sont effectuées par induction sur la structure des formules. Lorsque les preuves correspondant à certains cas sont très similaires, on a choisit de n'en présenter qu'une seule. Par ailleurs, les cas correspondants à des opérateurs de points fixes ne sont pas traités directement. En effet comme montré dans les preuves 8.0.2 et 8.0.7, étant donné qu'on travaille sur des systèmes finis, on peut la plupart du temps remplacer une formule donnée par une autre formule équivalente qui ne comporte pas d'opérateurs de point fixe.

8.0.1 Théorème 1 : compositionnalité des modules de régulations

Soit $((N_i)_{i\in I}, (Z,h))$ un réseau modulaire et $((S_j)_{j\in I\cup\overline{HO}}, V)$ le réseau de LTS correspondant. On définit la fonction injective suivante des états de $S = \mathsf{lts}\,((N_i)_{i\in I}, (Z,h))$ vers ceux de $\tilde{S} = \mathsf{lts}\,((S_j)_{j\in I\cup\overline{HO}}, V)$.

On associe à chaque état q de S, l'état $\tilde{q}$ de la manière suivante, $\tilde{q} \stackrel{\text{df}}{=} (\tilde{q}^j)_{j\in I\cup\overline{HO}}$ avec :

- $\tilde{q}^i[c] = q[c]$ si $c \in C^i \setminus \overline{O}$
- $\tilde{q}^i[o] = h_o(q_{|Z_o})$ si $o \in C^i \cap \overline{O}$
- $\tilde{q}^{h_o} = q_{|Z_o}$.

Cette fonction est bien injective car tous les composants d'un état q se retrouvent à l'identique dans $\tilde{q}$ d'après la première règle. On montre maintenant que l'on peut également associer à chaque transition d'un graphe, une transition du second, et on en déduit la propriété suivante : le sous graphe de S accessible à partir d'un état l'état q_0, est isomorphe au sous graphe de $\tilde{S}$ accessible depuis $\tilde{q_0}$.

Montrons que si $p \xrightarrow{a} q$ appartient à R alors $\tilde{p} \xrightarrow{a} \tilde{q}$ appartient à $\tilde{R}$.
Supposons $p \xrightarrow{c \leftarrow x} q \in R$ (avec $c \in C^i$), c'est à dire :

- $q[c] = K_c^i[\tilde{p}^i] = x$ et
- $q[c'] = q[c]$ si $c' \neq c$.

Montrons qu'il existe une règle $(v, c \leftarrow x) \in V_c$ telle que pour tout $j \in I \cup \overline{HO}$, on ait soit $\tilde{p}^j \xrightarrow{v[j]} \tilde{q}^j$ soit $v[j] = \bullet$ et $\tilde{p}^j = \tilde{q}^j$. On construit v en précisant sa valeur pour tout j appartenant à $I \cup \overline{HO}$:

Pour $j = i$: pour $v[j] = c \leftarrow x$ on a bien $\tilde{p}^j \xrightarrow{v[j]} \tilde{q}^j$ car :

- $\tilde{q}^i[c] = x = K^i(\tilde{p}^i)$
- $\tilde{q}^i[c'] = \tilde{p}^i[c']$ pour $c' \neq c$

Pour $j \in \overline{HO}$: soit o le composant ouvert correspondant à j. Les trois cas suivants sont possibles.

- Si $c \in Z_o$ et $h_o(p_{|Z_o}) = h_o(q_{|Z_o})$ alors on choisit $v[h_o] = (c \leftarrow x, \_)$ et on a $\tilde{p}^{h_o} \xrightarrow{v[h_o]} \tilde{q}^{h_o} \in R^{h_o}$ par construction du module d'intégration.
- Si $c \in Z_o$ et $h_o(p_{|Z_o}) \neq h_o(q_{|Z_o})$ alors $v[h_o] = (c \leftarrow x, o \leftarrow h_o(q_{|Z_o}))$ et on a également $\tilde{p}^{h_o} \xrightarrow{v[h_o]} \tilde{q}^{h_o} \in R^{h_o}$.
- Si $c \notin Z_o$ alors $v[h_o] = \bullet$, on a donc $\tilde{p}^{h_o} = \tilde{q}^{h_o}$ car c n'a aucune influence sur o.

Pour $j \in I, j \neq i$ et $\tilde{p}^j = \tilde{q}^j$: on prend $v[j] = \bullet$

Pour $j \in I, j \neq i$ et $\tilde{p}^j \neq \tilde{q}^j$:
on prend $v[j] = \{o \leftarrow y \mid c \in C_O^j, c \in Z_o, v[h_o] = (c \leftarrow x, o \leftarrow y)\}$. On a bien $\tilde{p}^j \xrightarrow{v[j]} \tilde{q}^j \in R^j$ car les composants ouverts du module j sont non contraints.

Montrons que si $\tilde{p} \xrightarrow{a} p_2$ appartient à $\tilde{R}$ alors il existe q tel que $p_2 = \tilde{q}$ et $p \xrightarrow{a} q$ appartient à R.

- Cas $a = c \leftarrow x$ avec $c \in C^i$: Soit $p \in Q$ et $\tilde{p} \xrightarrow{c \leftarrow x} p_2 \in \tilde{R}$. On définit q de telle sorte que $\tilde{q} = p_2$, c'est à dire :
  - $q[c] = p_2^i[c]$ et
  - $q[c'] = p_2^j[c']$ si $c' \neq c$ appartient à C^j.

  $\tilde{p} \xrightarrow{c \leftarrow x} p_2 \in \tilde{R}$ donc (d'après le vecteur de synchronisation) on a $\tilde{p}^i \xrightarrow{c \leftarrow x} p_2^i \in R^i$ et puis $p_2^i[c] = q[c] = K_c^i(\tilde{p}^i)$. On a également $q[c'] = p_2^j[c'] = \tilde{p}^j[c'] = p[c']$ pour tout $c \neq c' \in C^j$. Ces deux conditions impliquent que $p \xrightarrow{c \leftarrow x} q$ est une transition de R.

8.0.2 Lemme 1 : le calcule de Pass termine

Comme vu après la définition 14, pour une formule close φ et l'ensemble des états Q d'un module, Pass^{φ} est une fonction qui associe à chaque état de Q, une formule booléenne sur les variables d'état de φ qui sont externes au module (une formule appartenant à l'ensemble $\mathbb{B}(\mathcal{V}_{env})$). De plus, l'ensemble des fonctions de Q vers $\mathbb{B}(\mathcal{V}_{env})$ munis de la relation d'ordre $\overset{Q}{\Rightarrow}$, forme un treillis fini si l'on choisit un représentant unique pour chaque classe de formules Booléennes équivalentes (par exemple la forme normale disjonctive).

Considérons le cas d'une formule de point fixe $\sigma.X\varphi$, où φ possède une unique variable libre. Pass^{φ} est alors une endofonction de l'ensemble $(Q \rightarrow \mathbb{B}(\mathcal{V}_{env}))$. Les seuls opérateurs de négation introduits par la définition de Pass (figure 5.1) apparaissent sur les cas de base. Cela implique que la fonction Pass^{φ} est une fonction monotone et que ses plus petit et plus grand points fixes sont bien définis.

Par ailleurs, on sait que pour un module donné, il existe un entier n tel que $\mathsf{Pass}^{\mu X\varphi} = \mathsf{Pass}^{\varphi^n(\bot)}$ (et $\mathsf{Pass}^{\nu X\varphi} = \mathsf{Pass}^{\varphi^n(\top)}$) car on travail sur des modules finis. Cette propriété nous permet de simplifier plusieurs des preuves suivantes en considérant que les formules étudiées ne comportent pas de points fixes. Ceci est justifié par le fait que pour toute formule φ, on peut construire une formule φ' sans point fixe (on dit aussi dépliée) équivalente.

8.0.3 Lemme 2 : Pass est compositionnel

Soient S_1 et S_2 deux modules d'un LTS modulaire, x_1 un état de S_1 et x_2 un état de S_2. Montrons que $\mathsf{Pass}^{\varphi}(x_1)@x_2$ implique $\mathsf{Pass}^{\varphi}(x_1, x_2)$. Cela est alors également le cas pour Fail car on a par définition $\mathsf{Fail}^{\varphi} = \mathsf{Pass}^{\neg\varphi}$.

Cas B **:** les variables d'état du module S_1 étant disjointes de celles de S_2, la formule $B@x_1@x_2$ est bien égale à $B@(x_1, x_2)$.

Cas $\varphi_1 \wedge \varphi_2$ **:** L'hypothèse d'induction nous indique que $\mathsf{Pass}^{\varphi_1}(x_1)@x_2$ implique $\mathsf{Pass}^{\varphi_1}((x_1, x_2))$, et que $\mathsf{Pass}^{\varphi_2}(x_1)@x_2$ implique $\mathsf{Pass}^{\varphi_2}((x_1, x_2))$. On en déduit que $\mathsf{Pass}^{\varphi_1}(x_1)@x_2 \wedge \mathsf{Pass}^{\varphi_2}(x_1)@x_2$ implique $\mathsf{Pass}^{\varphi_1}((x_1, x_2)) \wedge \mathsf{Pass}^{\varphi_2}((x_1, x_2))$, qui est égal à $\mathsf{Pass}^{\varphi_1 \wedge \varphi_2}((x_1, x_2))$ d'après la définition.

Cas $\langle l \rangle \varphi$ **:** De façon similaire, on a pour chaque transition $x_1 \xrightarrow{l} x_1'$, $\mathsf{Pass}^{\varphi}(x_1')@x2$ implique $\mathsf{Pass}^{\varphi}(x_1', x2)$ grâce à l'hypothèse d'induction. On en déduit que $\bigvee_{x_1 \xrightarrow{l} x_1'} \mathsf{Pass}^{\varphi}(x_1')@x_2$ implique $\bigvee_{x_1 \xrightarrow{l} x_1'} \mathsf{Pass}^{\varphi}((x_1', x_2))$. Ce qui correspond à $\bigvee_{(x_1,x_2) \xrightarrow{l} (x_1,x_2)'} \mathsf{Pass}^{\varphi}((x_1, x_2)')$ car l'action l est locale au module 1.

Cas $\langle s \rangle \varphi$ **and** $\langle e \rangle \varphi$ **:** Dans ces cas l'implication est vérifiée car la valeur de Pass est $\bot$ (on a $\bot \Rightarrow \bot$).

Cas $[e]\varphi$ **:** Dans ce cas on peut appliquer l'hypothèse d'induction telle quelle car la valeur de $\mathsf{Pass}^{[e]\varphi}$ est égale à celle de Pass^{φ}.

8.0.4 Théorème 2 : relation avec la sémantique du μ-calcul

Supposons que la formule close φ ne contienne que des actions et des variables locales au module S, la sémantique de Pass^{φ} correspond à la sémantique usuelle du μ-calcul (définition 6) et on a :

$$\varphi^S = \{x \mid \mathsf{Pass}^{\varphi}(x) = \top\}.$$

On remarque tout d'abord que si la formule φ ne contient que des actions et des variables locales, alors pour tout état x du module, $\mathsf{Pass}^{\varphi}(x)$ est évaluée soit à $\top$ soit à $\bot$.

Cas B **:** Si $B@x = \top$ alors x appartient à B^S, sinon on a $B@x = \bot$ et x appartient à $Q \setminus B^S$.

Cas $\varphi_1 \vee \varphi_2$ **:** On a $\mathsf{Pass}^{\varphi_1 \vee \varphi_2}(x) = \mathsf{Pass}^{\varphi_1}(x) \vee \mathsf{Pass}^{\varphi_2}(x)$. Si cette expression est égale à $\top$, on sait grâce à l'hypothèse d'induction que x appartient soit à φ_1^S soit à φ_2^S, et donc à $(\varphi_1 \vee \varphi_2)^S$ qui est définie comme étant l'union des deux.

Cas $\langle l \rangle \varphi$ **:** On a $\mathsf{Pass}^{\langle l \rangle \varphi}(x) = \bigvee_{x \xrightarrow{l} x'} (\mathsf{Pass}^{\varphi}(x'))$, ce qui signifie que $\mathsf{Pass}^{\langle l \rangle \varphi}(x)$ est égal à $\top$ si et seulement si il existe une transition $x \xrightarrow{l} x'$ telle que $x' \in \varphi^S$. Autrement dit si et seulement si x appartient à $(\langle l \rangle \varphi)^S$.

Cas $\mu X \varphi$ **:** Comme vu dans le lemme 8.0.2, il existe un entier n tel que $\mathsf{Pass}^{\mu X.\varphi} = \mathsf{Pass}^{n.X\varphi}(x)$. Or on pour tout entier $n \in \mathbb{N}, \mathsf{Pass}^{n.X\varphi}(x) = \top$ si et seulement si $x \in (n.X\varphi)^S$.

8.0.5 Lemme 3 : correction de Pass et de Fail

Montrons que pour tout état global (x, env), si env vérifie $\mathsf{Pass}^{\varphi}(x)$ alors (x, env) vérifie φ. Le même résultat est alors correct pour Fail. En effet, on a d'une part $\mathsf{Fail}^{\varphi} = \mathsf{Pass}^{\neg\varphi}$ et d'autre part $(x, env) \not\models \varphi \Leftrightarrow (x, env) \models \neg\varphi$. La propriété est prouvée par induction sur la structure de la formule.

Cas $\varphi = B$ **:** On a $(B@x)@env = B@(x, env)$, donc $env \models B@x$ est équivalent à $(x, env) \models B$.

Cas $\varphi = \varphi_1 \wedge \varphi_2$ **:** Si env vérifie $\mathsf{Pass}^{\varphi_1 \wedge \varphi_2}(x)$, on a $env \models \mathsf{Pass}^{\varphi_1}(x)$ et $env \models \mathsf{Pass}^{\varphi_2}(x)$. L'hypothèse d'induction nous permet de déduire $(x, env) \models \varphi_1$ et $(x, env) \models \varphi_2$, c'est à dire $(x, env) \models \varphi_1 \wedge \varphi_2$.

Cas $\varphi = \langle l \rangle \psi$: Par définition, $\mathsf{Pass}^{\varphi}(x) = \bigvee_{x \xrightarrow{l} x'} \mathsf{Pass}^{\psi}(x')$. On peut trouver un successeur x' de x tel que $env \models \mathsf{Pass}^{\psi}(x')$ et donc tel que $(x', env) \models \psi$ (grâce à l'hypothèse d'induction). On en déduit alors $(x, env) \models \varphi$.

Cas $\varphi = [s]\psi$: Si $env \models \mathsf{Pass}^{[s]\psi}(x)$, on a $\mathsf{Pass}^{\psi}(x') = \top$ pour toute transition $x \xrightarrow{s} x'$. Donc pour toute transition du système global $(x, env) \xrightarrow{s} (x, env)'$, on a $(x, env)' \models \psi$ par l'hypothèse d'induction. On a alors aussi $(x, env) \models \varphi$.

Cas $\varphi = \mu X \psi$ **:** Pour tout entier naturel n, $env \models \mathsf{Pass}^{nX.\psi}(x) \Rightarrow (x, env) \models nX.\psi$.

8.0.6 Lemme 4 : le calcul de $\langle\langle \varphi, \psi \rangle\rangle$ termine.

La preuve est similaire à celle du lemme 1, à la différence que l'on considère ici des fonctions de Q^2 vers $\mathbb{B}(\mathcal{V})$ et non plus de Q vers $\mathbb{B}(\mathcal{V})$. Pour un calcul de point fixe $\langle\langle \sigma X \varphi, \psi \rangle\rangle$, les règles de la Figure 5.4 utilisent uniquement des formules booléennes ne contenant pas de négations. Par conséquent $\langle\langle \varphi, \psi \rangle\rangle$ est une endofonction monotone du treillis fini $(Q^2 \to \mathbb{B}(\mathcal{V}), \overset{Q^2}{\Rightarrow})$. Ses plus grand et plus petit points fixes sont bien définis, et on peut les calculer en itérant la fonction à partir des extrema du treillis ($\top_{Q^2}$ et $\bot_{Q^2}$) qui sont les fonctions constantes retournant respectivement $\top$ et $\bot$.

8.0.7 Dépliage d'une formule de point fixe

D'après le Lemme 4, $\langle\langle \mu X.\varphi, \psi \rangle\rangle(x, y)$ est une formule booléenne finie construite en appliquant les règles de la figure 5.4. On l'appelle $F(\top, \ldots, \top) \overset{\text{df}}{=} \langle\langle \mu X.\varphi, \psi \rangle\rangle(x, y)$, où les arguments de F correspondent à l'initialisation du calcule du plus grand point fixe de $\langle\langle \varphi, \psi \rangle\rangle$. Soit m le plus grand nombre de dépliage de la formule $\mu X.\varphi$ lors du calcule de $\langle\langle \mu X.\varphi, \psi \rangle\rangle$. Il existe alors des entiers $n_1, \ldots, n_k$ tels que

$$\langle\langle \varphi^m(\bot), \psi \rangle\rangle = F(\langle\langle \varphi^{n_1}(\bot), \psi \rangle\rangle, \ldots, \langle\langle \varphi^{n_k}(\bot), \psi \rangle\rangle).$$

Puisque la formule F est monotone et que $\langle\langle \varphi^{n_1}(\bot), \psi \rangle\rangle$ est inférieur à $\top$ pour tout n, on a

$$\langle\langle \mu X.\varphi, \psi \rangle\rangle \overset{Q^2}{\Leftarrow} \langle\langle \varphi^m(\bot), \psi \rangle\rangle.$$

Définissons par ailleurs, la formule $G(\top, \ldots, \top) \overset{\text{df}}{=} \langle\langle \varphi^m(\bot), \psi \rangle\rangle$, dont les arguments correspondent aux composantes $\langle\langle \bot, \psi_1 \rangle\rangle \ldots \langle\langle \bot, \psi_k \rangle\rangle$ de la formule (où la constante $\bot$ de gauche est atteinte). On a alors

$$\langle\langle \mu X.\varphi, \psi \rangle\rangle \overset{Q^2}{\Leftrightarrow} \langle\langle \varphi^m(\mu X.\varphi), \psi \rangle\rangle \overset{Q^2}{\Leftrightarrow} G(\langle\langle \mu X.\varphi, \psi_1 \rangle\rangle, \ldots, \langle\langle \mu X.\varphi, \psi_2 \rangle\rangle).$$

Comme précédemment, G est monotone et on a pour tout ψ_i, $\langle\langle \mu X.\varphi, \psi_i \rangle\rangle \overset{Q^2}{\Rightarrow} \top$, donc

$$\langle\langle \mu X.\varphi, \psi \rangle\rangle \overset{Q^2}{\Rightarrow} \langle\langle \varphi^m(\bot), \psi \rangle\rangle.$$

Pour un calcul de plus grand point fixe ($\langle\langle \mu X.\varphi, \psi \rangle\rangle$) sur un modèle donné, on a montré que l'on peut trouver un entier m (correspondant au nombre maximum de dépliage de la formule $\mu X.\varphi$ lors du calcul), tel que $\langle\langle \mu X.\varphi, \psi \rangle\rangle$ est égal à $\langle\langle \varphi^m(\bot), \psi \rangle\rangle$. On peut montrer de façon similaire le résultat pour un calcul de plus petit point fixe. Cette propriété nous permet de nous limiter à des formules ne possédant pas de point fixes dans certaines des preuves suivantes.

8.0.8 Transitivité de $\langle\langle \varphi \rangle\rangle$

Soient x, y, z des états d'un module et φ, ψ, ξ des formules. Montrons que si on a à la fois $\langle\langle \varphi, \psi \rangle\rangle(x, y)$ et $\langle\langle \psi, \xi \rangle\rangle(y, z)$, alors on a aussi $\langle\langle \varphi, \xi \rangle\rangle(x, z)$.

Cas de base : On est dans le cas où $\langle\langle \varphi, \psi \rangle\rangle(x, y) = \mathsf{Fail}^{\varphi}(x) \vee \mathsf{Pass}^{\psi}(y)$ (On note que cela inclus également le cas B_1, B_2). Si on a $\mathsf{Fail}^{\varphi}(x)$ alors $\langle\langle \varphi, \xi \rangle\rangle(x, z)$ est vérifié. Si on a $\mathsf{Pass}^{\psi}(y)$, alors on a aussi $\mathsf{Pass}^{\xi}(z)$ d'après la preuve du théorème 3 et donc $\langle\langle \varphi, \xi \rangle\rangle(x, z)$.

Cas $\langle\langle \varphi_1 \wedge \varphi_2, \psi \rangle\rangle$ et $\langle\langle \psi, \xi \rangle\rangle$: Dans ce cas on a soit $\langle\langle \varphi_1, \psi \rangle\rangle(x, y) \wedge \langle\langle \psi, \xi \rangle\rangle(y, z)$ soit $\langle\langle \varphi_2, \psi \rangle\rangle(x, y) \wedge \langle\langle \psi, \xi \rangle\rangle(y, z)$. On sait donc à l'aide de l'hypothèse d'induction qu'on a soit $\langle\langle \varphi_1, \xi \rangle\rangle(x, z)$, soit $\langle\langle \varphi_2, \xi \rangle\rangle(x, z)$. C'est à dire $\langle\langle \varphi_1 \wedge \varphi_2, \xi \rangle\rangle(x, z)$.

Cas $\langle\langle \varphi, \psi_1 \wedge \psi_2 \rangle\rangle$ et $\langle\langle \psi_1 \wedge \psi_2, \xi \rangle\rangle$: On a $\langle\langle \varphi, \psi_1 \rangle\rangle(x, y)$,$\langle\langle \varphi, \psi_2 \rangle\rangle(x, y)$ et $\langle\langle \psi_1, \xi \rangle\rangle(y, z) \vee \langle\langle \psi_2, \xi \rangle\rangle(y, z)$. Si $\langle\langle \psi_1, \xi \rangle\rangle(y, z)$ est vraie, on peut utiliser l'hypothèse d'induction pour en déduire $\langle\langle \varphi, \xi \rangle\rangle(x, z)$. On procède de manière similaire pour le cas où $\langle\langle \psi_2, \xi \rangle\rangle(y, z)$ est vérifiée.

Cas $\langle\langle \langle l \rangle \varphi', \psi \rangle\rangle$ et $\langle\langle \psi, \xi \rangle\rangle$:

Supposons $\bigwedge_{x \xrightarrow{l} x'} \langle\langle \varphi', \psi \rangle\rangle(x', y)$ et $\langle\langle \psi, \xi \rangle\rangle(y, z)$.

L'hypothèse d'induction nous donne :

$$\bigwedge_{x \xrightarrow{l} x'} \langle\langle \varphi', \xi \rangle\rangle(x', z) = \langle\langle \langle l \rangle \varphi', \xi \rangle\rangle(x, z).$$

Cas $\langle\langle \varphi, \langle l \rangle \psi' \rangle\rangle$ et $\langle\langle \langle l \rangle \psi', \xi \rangle\rangle$:

Supposons $\bigvee_{y \xrightarrow{l} y'} \langle\langle \varphi, \psi' \rangle\rangle(x, y')$ et $\bigwedge_{y \xrightarrow{l} y'} \langle\langle \psi', \xi \rangle\rangle(y', z)$.

Cela implique : $\bigvee_{y \xrightarrow{l} y'} \langle\langle \varphi, \xi \rangle\rangle(x, z) = \langle\langle \varphi, \xi \rangle\rangle(x, z)$.

Cas $\langle\langle \langle s \rangle \varphi', \langle s \rangle \psi' \rangle\rangle$ et $\langle\langle \langle s \rangle \psi', \langle s \rangle \xi' \rangle\rangle$: On a les deux égalités suivantes :

$$\langle\langle \varphi, \psi \rangle\rangle(x, y) = \begin{cases} \top & \text{si } \bigwedge_{x \xrightarrow{s} x'} \bigvee_{y \xrightarrow{s} y'} \langle\langle \varphi', \psi' \rangle\rangle(x', y') \\ \bot & \text{sinon} \end{cases}$$

$$\langle\langle \psi, \xi \rangle\rangle(y, z) = \begin{cases} \top & \text{si } \bigwedge_{y \xrightarrow{s} y'} \bigvee_{z \xrightarrow{s} z'} \langle\langle \psi', \xi' \rangle\rangle(y', z') \\ \bot & \text{sinon.} \end{cases}$$

On a pour tout successeur x' de x, $\bigvee_{y \xrightarrow{s} y'} \langle\langle \varphi', \psi' \rangle\rangle(x', y') = \top$.

On sait également que pour tout successeur y' de y,

$\bigvee_{z \xrightarrow{s} z'} \langle\langle \psi', \xi' \rangle\rangle(y', z') = \top$ est vraie,

on a donc pour tout x',

$$\bigvee_{y \xrightarrow{s} y'} (\langle\langle \varphi', \psi' \rangle\rangle(x', y') \wedge \bigvee_{z \xrightarrow{s} z'} \langle\langle \psi', \xi' \rangle\rangle(y', z')) = \top$$

$$\Leftrightarrow \bigvee_{y \xrightarrow{s} y'} \bigvee_{z \xrightarrow{s} z'} \langle\langle \varphi', \psi' \rangle\rangle(x', y') \wedge \langle\langle \psi', \xi' \rangle\rangle(y', z') = \top$$

$$\Rightarrow \bigvee_{y \xrightarrow{s} y'} \bigvee_{z \xrightarrow{s} z'} \langle\langle \varphi', \xi' \rangle\rangle(x', z') = \top$$

$$\Rightarrow \bigvee_{z \xrightarrow{s} z'} \langle\langle \varphi', \xi' \rangle\rangle(x', z') = \top.$$

8.0.9 Théorème 3 : préservation de la formule par la relation $\langle\langle \varphi \rangle\rangle$

Montrons que si on a $\langle\langle \psi, \varphi \rangle\rangle(x, y)$ alors on a également $(\mathsf{Pass}^{\psi}(x) \Rightarrow \mathsf{Pass}^{\varphi}(y))$. Le résultat correspondant pour Fail s'ensuit car on a d'une part $\mathsf{Fail}^{\varphi} = \mathsf{Pass}^{\neg\varphi}$ et d'autre part $\langle\langle \psi, \varphi \rangle\rangle(x, y) = \langle\langle \neg\varphi, \neg\psi \rangle\rangle(y, x)$. On définie la formule $\langle\langle \varphi, \psi \rangle\rangle^{P}(x, y)$ grâce aux règles suivantes. Cela est en fait une autre

façon de définir la formule $\mathsf{Pass}^{\varphi}(x) \Rightarrow \mathsf{Pass}^{\psi}(y)$, dans le but de la comparer plus facilement avec la formule $\langle\langle \varphi, \psi \rangle\rangle(x, y)$.

$$
\begin{aligned}
&\langle\langle B_1, B_2 \rangle\rangle^P(x, y) = B_1@x \Rightarrow B_2@y \\
&\langle\langle \psi, \varphi_1 \vee \varphi_2 \rangle\rangle^P(x, y) = \langle\langle \psi, \varphi_1 \rangle\rangle^P(x, y) \vee \langle\langle \psi, \varphi_2 \rangle\rangle^P(x, y) \\
&\langle\langle \psi, \langle l \rangle \varphi \rangle\rangle^P(x, y) = \neg\mathsf{Pass}^{\psi}(x) \vee \bigvee_{y \xrightarrow{l} y'} \langle\langle \psi, \varphi \rangle\rangle^P(x, y') \\
&\langle\langle \langle s \rangle \psi, \langle s \rangle \varphi \rangle\rangle^P(x, y) = \top \\
&\langle\langle \langle e \rangle \psi, \langle e \rangle \varphi \rangle\rangle^P(x, y) = \top \\
&\langle\langle \mu X.\psi, \varphi \rangle\rangle^P(x, y) = \bigvee \{ f(x, y) \mid \langle\langle \psi, \varphi \rangle\rangle^P f \overset{\mathrm{Q}^2}{\Leftarrow} f \} \\
&\langle\langle \psi, \varphi \rangle\rangle^P(x, y) = \langle\langle \neg\varphi, \neg\psi \rangle\rangle^P(y, x) \\
&\langle\langle \nu X.\psi, \varphi \rangle\rangle^P(x, y) = \bigwedge \{ f(x, y) \mid \langle\langle \psi, \varphi \rangle\rangle^P f \overset{\mathrm{Q}^2}{\Rightarrow} f \} \\
&\langle\langle \psi, \varphi_1 \wedge \varphi_2 \rangle\rangle^P(x, y) = \langle\langle \psi, \varphi_1 \rangle\rangle^P(x, y) \wedge \langle\langle \psi, \varphi_2 \rangle\rangle^P(x, y) \\
&\langle\langle \psi, [l] \varphi \rangle\rangle^P(x, y) = \bigwedge_{y \xrightarrow{l} y'} \langle\langle \psi, \varphi \rangle\rangle^P(x, y') \\
&\langle\langle \psi, \varphi \rangle\rangle^P(x, y) = \mathsf{Pass}^{\psi}(x) \Rightarrow \mathsf{Pass}^{\varphi}(y)
\end{aligned}
$$

On prouve que $\langle\langle \varphi, \psi \rangle\rangle \overset{\mathrm{Q}^2}{\Rightarrow} \langle\langle \varphi, \psi \rangle\rangle^P$ par induction en comparant les règles une a une. En fait la formule Booléenne construite pour le calcul de $\langle\langle \varphi, \psi \rangle\rangle^P$ est similaire à celle construite pour $\langle\langle \varphi, \psi \rangle\rangle$ avec quelques différences. Les sous-formules correspondant aux cas concernant les actions non locales sont remplacées par $\top$ et les sous-formules correspondant à un $\mathsf{Fail}^{\varphi'}$ sont remplacée par $\neg\mathsf{Pass}^{\varphi'}$ (et on a $Fail^{\varphi'} \overset{\mathrm{Q}^2}{\Rightarrow} \neg\mathsf{Pass}^{\varphi'}$). Puisque la formule construite est monotone on en conclut que $\langle\langle \varphi, \psi \rangle\rangle \overset{\mathrm{Q}^2}{\Rightarrow} \langle\langle \varphi, \psi \rangle\rangle^P$.

8.0.10 Théorème 4 : modularité de la relation $\langle\langle \varphi \rangle\rangle$

Soient x_1 et y_1 deux états d'un module S_1, x_2 et y_2 deux états d'un module S_2, et φ, ψ, ξ trois formules. Montrons que si l'on a d'une part $\langle\langle \varphi, \psi \rangle\rangle(x_1, y_1)@x_2$ et d'autre part $\langle\langle \psi, \xi \rangle\rangle(x_2, y_2)@y_1$, alors la formule $\langle\langle \varphi, \xi \rangle\rangle((x_1, x_2), (y_1, y_2))$ est vérifiée. À partir de là on obtient que si $x_1 \langle\langle \varphi \rangle\rangle y_1$ et $x_2 \langle\langle \varphi \rangle\rangle y_2$, alors on a également $(x_1, y_1) \langle\langle \varphi \rangle\rangle (x_2, y_2)$.

Cas de base : On a

$$
\begin{aligned}
&\langle\langle \varphi, \psi \rangle\rangle(x_1, y_1) = \mathsf{Fail}^{\varphi}(x_1) \vee \mathsf{Pass}^{\psi}(y_1) \\
\text{et } &\langle\langle \psi, \xi \rangle\rangle(x_2, y_2) = \mathsf{Fail}^{\psi}(x_2) \vee \mathsf{Pass}^{\xi}(y_2).
\end{aligned}
$$

Si on est dans le cas où x_2 vérifie $\mathsf{Fail}^{\varphi}(x_1)$ alors $\mathsf{Fail}^{\varphi}(x_1, x_2)$ est vérifiée, ce qui implique $\langle\langle \varphi, \xi \rangle\rangle((x_1, x_2), (y_1, y_2))$. Dans le cas où x_2 vérifie $\mathsf{Pass}^{\psi}(y_1)$ alors y_1 ne peut pas vérifier $\mathsf{Fail}^{\psi}(x_2)$. Cela veut dire que l'on a $\mathsf{Pass}^{\xi}(y_2)@y_1$, et donc $\mathsf{Pass}^{\xi}(y_1, y_2)$ ce qui implique $\langle\langle \varphi, \xi \rangle\rangle((x_1, x_2), (y_1, y_2))$.

Cas $\langle\langle \varphi_1 \wedge \varphi_2, \psi \rangle\rangle$ **et** $\langle\langle \psi, \xi \rangle\rangle$ **:**
On a soit

$$\langle\langle \varphi_1, \psi \rangle\rangle(x_1, y_1)@x_2 \wedge \langle\langle \psi, \xi \rangle\rangle(x_2, y_2)@y_1,$$

soit

$$\langle\langle \varphi_2, \psi \rangle\rangle(x_1, y_1)@x_2 \wedge \langle\langle \psi, \xi \rangle\rangle(x_2, y_2)@y_1.$$

En appliquant l'hypothèse d'induction on obtient l'expression :

$$\langle\langle \varphi_1, \xi \rangle\rangle((x_1, x_2), (y_1, y_2)) \vee \langle\langle \varphi_2, \xi \rangle\rangle((x_1, x_2), (y_1, y_2)),$$

qui est égale à $\langle\langle \varphi_1 \wedge \varphi_2, \xi \rangle\rangle((x_1, x_2), (y_1, y_2))$.

Cas $\langle\langle \varphi_1 \vee \varphi_2, \psi \rangle\rangle$ **et** $\langle\langle \psi, \xi \rangle\rangle$ **:**
On a à la fois

$$\langle\langle \varphi_1, \psi \rangle\rangle(x_1, y_1)@x_2 \wedge \langle\langle \psi, \xi \rangle\rangle(x_2, y_2)@y_1$$

et

$$\langle\langle \varphi_2, \psi \rangle\rangle(x_1, y_1)@x_2 \wedge \langle\langle \psi, \xi \rangle\rangle(x_2, y_2)@y_1.$$

De façon similaire, on obtient $\langle\langle \varphi_1 \vee \varphi_2, \xi \rangle\rangle((x_1, x_2), (y_1, y_2))$ à l'aide de l'hypothèse d'induction.

Cas $\langle\langle \varphi, \psi_1 \wedge \psi_2 \rangle\rangle$ **et** $\langle\langle \psi_1 \wedge \psi_2, \xi \rangle\rangle$ **:**
On a à la fois

$$\langle\langle \varphi, \psi_1 \rangle\rangle(x_1, y_1)@x_2,$$
$$\langle\langle \varphi, \psi_2 \rangle\rangle(x_1, y_1)@x_2$$

et $$\langle\langle \psi_1, \xi \rangle\rangle(x_2, y_2)@y_1 \vee \langle\langle \psi_2, \xi \rangle\rangle(x_2, y_2)@y_1.$$

Si $\langle\langle \psi_1, \xi \rangle\rangle(x_2, y_2)@y_1$ est vérifiée on peut en déduire $\langle\langle \varphi, \xi \rangle\rangle((x_1, x_2), (y_1, y_2))$ grâce à l'hypothèse d'induction. On procède de manière identique pour le cas où $\langle\langle \psi_2, \xi \rangle\rangle(x_2, y_2)@y_1$.

Cas $\langle\langle \langle l \rangle \varphi', \psi \rangle\rangle$ **et** $\langle\langle \psi, \xi \rangle\rangle$ **:**

On a $\bigwedge_{x_1 \xrightarrow{l} x_1'} \langle\langle \varphi', \psi \rangle\rangle (x_1', y_1)@x_2$ et $\langle\langle \psi, \xi \rangle\rangle (x_2, y_2)@y_1$

L'hypothèse d'induction nous donne :

$$\begin{aligned}
&\bigwedge_{x_1 \xrightarrow{l} x_1'} \langle\langle \varphi', \xi \rangle\rangle ((x_1', x_2), (y_1, y_2)) \\
=&\bigwedge_{(x_1,x_2) \xrightarrow{l} (x_1,x_2)'} \langle\langle \varphi', \xi \rangle\rangle ((x_1, x_2)', (y_1, y_2)) \\
=&\langle\langle \langle l \rangle \varphi', \xi \rangle\rangle ((x_1, x_2), (y_1, y_2)).
\end{aligned}$$

car l est une action locale au module S1.

Cas $\langle\langle \langle s \rangle \varphi', \langle s \rangle \psi' \rangle\rangle$ **et** $\langle\langle \langle s \rangle \psi', \langle s \rangle \xi' \rangle\rangle$ **:**
On a

$$\langle\langle \varphi, \psi \rangle\rangle (x_1, y_1)@y_2 = \begin{cases} \top & \text{si } \bigwedge_{x_1 \xrightarrow{s} x_1'} \bigvee_{y_1 \xrightarrow{s} y_1'} \langle\langle \varphi', \psi' \rangle\rangle (x_1', y_1')@x_2 \\ \bot & \text{sinon} \end{cases}$$

$$\langle\langle \psi, \xi \rangle\rangle (x_2, y_2)@y_1 = \begin{cases} \top & \text{si } \bigwedge_{x_2 \xrightarrow{s} x_2'} \bigvee_{y_2 \xrightarrow{s} y_2'} \langle\langle \psi', \xi' \rangle\rangle (x_2', y_2')@y_1 \\ \bot & \text{sinon} \end{cases}$$

On a donc pour tout successeurs x_1' et x_2',

$$\begin{aligned}
&\bigvee_{y_1 \xrightarrow{s} y_1'} \langle\langle \varphi', \psi' \rangle\rangle (x_1', y_1')@x_2 \text{ et } \bigvee_{y_2 \xrightarrow{s} y_2'} \langle\langle \psi', \xi' \rangle\rangle (x_2', y_2')@y_1 \\
=&\bigwedge_{(x_1,x_2) \xrightarrow{s} (x_1,x_2)'} \bigvee_{(y_1,y_2) \xrightarrow{s} (y_1,y_2)'} \langle\langle \varphi', \psi' \rangle\rangle (x_1', y_1')@x_2 \wedge \langle\langle \psi', \xi' \rangle\rangle (x_2', y_2')@y_1
\end{aligned}$$

et on peut conclure en utilisant l'hypothèse d'induction.

8.0.11 Lemme 5 : la construction de S^φ termine

Soient Q l'ensemble des états de S et $i \in Q$ son état initial. L'ensemble $\mathrm{St}(\varphi, i)$ des états du LTS S^φ, est le sous-ensemble de $\mathcal{CL}(\varphi) \times Q$ accessible depuis l'état initial (φ, i). Comme $\mathcal{CL}(\varphi), Q$ est fini, la construction de $\mathrm{St}(\varphi, i)$ se termine en un nombre fini d'étapes.

8.0.12 Théorème 5 : $S^\varphi \langle\langle \varphi \rangle\rangle S$

Soient un module S, x un état de ce module et φ une formule. La première étape consiste a montrer par induction que si φ est une sous-formule

de f accessible à l'aide de dépliages, conjonctions et disjonctions, alors on a : $\mathsf{Pass}^{\varphi}(x) \Leftrightarrow \mathsf{Pass}^{\varphi}((x,f))$.

Cas B **:** les états x et (x,f) on le même label, donc $B@x$ est équivalent à $B@(x,f)$.

Cas $\varphi_1 \wedge \varphi_2$ **:** On a $\mathsf{Pass}^{\varphi_1}(x) \wedge \mathsf{Pass}^{\varphi_2}(x) = \mathsf{Pass}^{\varphi_1}(x,f) \wedge \mathsf{Pass}^{\varphi_2}(x,f)$ (On peut appliquer l'hypothèse d'induction car si f contient $\varphi_1 \wedge \varphi_2$, elle contient aussi φ_1 et φ_2).

Cas $\langle l \rangle \psi$ **:** On a $\bigvee_{x \xrightarrow{l} x'} \mathsf{Pass}^{\psi}(x') = \bigvee_{(x,f) \xrightarrow{l} (x',\psi)} \mathsf{Pass}^{\psi}(x',\psi)$. Les états successeurs x' qui n'apparaissent pas dans la seconde formule sont les états tels que $\mathsf{Fail}^{\psi}(x') = \top$, et on sait donc que $\mathsf{Pass}^{\psi}(x')$ est égal à $\bot$ pour ceux ci.

Montrons maintenant par induction que pour tout couple d'états x, y et de formules ψ_1, ψ_2 on a la propriété suivante :

$$\langle\langle \psi_1, \psi_2 \rangle\rangle(x,y) \Leftrightarrow \langle\langle \psi_1, \psi_2 \rangle\rangle((x,\psi_1),(y,\psi_2)).$$

Où (x,ψ_1) et (y,ψ_2) sont des états du LTS S^{φ}. On pourra ensuite écrire :

$$\begin{aligned}
\langle\langle \varphi, \varphi \rangle\rangle(x,(x,\varphi)) \Leftrightarrow & \langle\langle \varphi, \varphi \rangle\rangle((x,\varphi),((x,\varphi),\varphi)) \\
\Leftrightarrow & \langle\langle \varphi, \varphi \rangle\rangle((x,\varphi),(x,\varphi)) \\
\Leftrightarrow & \langle\langle \varphi, \varphi \rangle\rangle(x,x) \\
\Leftrightarrow & \top.
\end{aligned}$$

Cas de base : Fail et Pass sont préservés comme montré ci-dessus.

Cas $\psi, \varphi_1 \wedge \varphi_2$ **:** φ_1 et φ_2 sont des sous-formules de $\varphi_1 \wedge \varphi_2$ accessibles par une conjonction. On peut donc appliquer l'hypothèse d'induction sur $\langle\langle \psi, \varphi_1 \rangle\rangle(x,y)$ et $\langle\langle \psi, \varphi_2 \rangle\rangle(x,y)$. C'est à dire :

$$\begin{aligned}
\langle\langle \psi, \varphi_1 \wedge \varphi_2 \rangle\rangle(x,y) = & \langle\langle \psi, \varphi_1 \rangle\rangle(x,y) \wedge \langle\langle \psi, \varphi_2 \rangle\rangle(x,y) \\
= & \langle\langle \psi, \varphi_1 \rangle\rangle((x,\psi),(y,\varphi_1 \wedge \varphi_2)) \\
& \wedge \langle\langle \psi, \varphi_2 \rangle\rangle((x,\psi),(y,\varphi_1 \wedge \varphi_2)) \\
= & \langle\langle \psi, \varphi_1 \wedge \varphi_2 \rangle\rangle((x,\psi),(y,\varphi_1 \wedge \varphi_2)).
\end{aligned}$$

Cas $\psi, \langle l \rangle \varphi$ **:** On a

$$\begin{aligned}\langle\langle \psi, \langle l \rangle \varphi \rangle\rangle(x,y) =& \mathsf{Fail}^{\psi}(x) \vee \bigvee_{y \xrightarrow{l} y'} \langle\langle \psi, \varphi \rangle\rangle(x,y') \\ =& \mathsf{Fail}^{\psi}((x,\psi)) \vee \bigvee_{(y,\langle l \rangle \varphi) \xrightarrow{l} (y',\varphi)} \langle\langle \psi, \varphi \rangle\rangle((x,\psi),(y',\varphi)) \\ =& \langle\langle \psi, \langle l \rangle \varphi \rangle\rangle((x,\psi),(y,\langle l \rangle \varphi)).\end{aligned}$$

Cela est valide car pour tous les successeurs y' tels que $\mathsf{Fail}^{\varphi}(y') = \top$ (c'est à dire ceux auxquels ne correspond aucun état (y', φ) du graphe d'accessibilité), on a $\langle\langle \psi, \varphi \rangle\rangle(x,y') \Rightarrow \mathsf{Fail}^{\psi}(x)$, ceux qui nous permet de les ignorer dans la formule.

Cas $\langle s \rangle \psi, \langle s \rangle \varphi$ **:** On a

$$\langle\langle \langle s \rangle \psi, \langle s \rangle \varphi \rangle\rangle(x,y) = \mathsf{Fail}^{\langle s \rangle \psi}(x) \vee \begin{cases} \top & \text{si } \bigwedge_{x \xrightarrow{s} x'} \mathsf{Fail}^{\psi}(x') \vee \bigvee_{y \xrightarrow{s} y'} \langle\langle \psi, \varphi \rangle\rangle(x',y') \\ \bot & \text{sinon} \end{cases}.$$

La justification est similaire à celle précédente. Si la formule $\mathsf{Fail}^{\psi}(x')$ est vraie alors elle peut être retirée de la conjonction. Par ailleurs si $\mathsf{Fail}^{\varphi}(y')$ est vraie on a alors $\langle\langle \psi, \varphi \rangle\rangle(x',y') \Rightarrow \mathsf{Fail}^{\psi}(x')$, et le terme $\langle\langle \psi, \varphi \rangle\rangle(x',y')$ peut également être retiré de la conjonction.

8.0.13 Théorème 6 : l'opération de remplacement préserve $\langle\langle \varphi \rangle\rangle$

La transitivité de $\langle\langle \psi, \varphi \rangle\rangle$ implique que si on a $x \langle\langle \varphi \rangle\rangle y$, alors pour tout état z et toute formule ψ, on a d'une part $\langle\langle \psi, \varphi \rangle\rangle(z,x) \Leftrightarrow \langle\langle \psi, \varphi \rangle\rangle(z,y)$ et d'autre part $\langle\langle \varphi, \psi \rangle\rangle(x,z) \Leftrightarrow \langle\langle \varphi, \psi \rangle\rangle(y,z)$. De plus, si il n'existe pas de chemin menant de y à x, signifie que l'état x ne jouera aucun rôle dans les calculs de $\langle\langle \psi, \varphi \rangle\rangle(z,y)$ et de $\langle\langle \varphi, \psi \rangle\rangle(y,z)$. On peut donc remplacer l'état x par y car la suppression des transitions menant à x n'affectera donc pas ces valeurs.

8.0.14 Lemme 6 : $\sim^{\varphi}$ est une relation d'équivalence calculable

La preuve de la convergence est identique à celle de $\langle\langle \psi, \varphi \rangle\rangle$. La réflexivité est la symétrie de la relation se montrent par induction sur les règles de la définition et ne sont pas détaillées ici. Montrons qu'elle est transitive, c'est à dire que si on a $\sim^{\varphi} (x,y)$ et $\sim^{\varphi} (y,z)$, alors on a également $\sim^{\varphi} (x,z)$.

Cas B **:** On a $B@x \Leftrightarrow B@y$ et $B@y \Leftrightarrow B@z$, ce qui nous donne $B@x \Leftrightarrow B@z$.

Cas $\varphi_1 \wedge \varphi_2$ **:** La transitivité de $\sim^{\varphi_1}$ et de $\sim^{\varphi_2}$ obtenue grâce à l'hypothèse d'induction se propage à $\sim^{\varphi_1 \wedge \varphi_2}$.

Cas $\langle l \rangle \varphi$ **:** Supposons $\neg \mathsf{Fail}^{\langle l \rangle \varphi}(x)$, ce qui veux dire qu'on a donc $\bigwedge_{x \xrightarrow{l} x'} \bigvee_{y \xrightarrow{l} y'} \sim^{\varphi} (x', y')$. Cela implique également que $\mathsf{Fail}^{\langle l \rangle \varphi}(y)$ n'est pas vérifiée, et donc qu'on a $\bigwedge_{y \xrightarrow{l} y'} \bigvee_{z \xrightarrow{l} z'} \sim^{\varphi} (y', z')$.
On peut alors conclure $\bigwedge_{x \xrightarrow{l} x'} \bigvee_{z \xrightarrow{l} z'} \sim^{\varphi} (x', z')$.

Comme précédemment, pour toute formule φ il existe une formule φ' ne comportant pas de points fixes telle que la relation $\sim^{\varphi}$ soit égale a $\sim^{\varphi'}$. Cela simplifie les prochaines preuves car ils n'est pas nécessaire de considérer les cas correspondant à des points fixes.

8.0.15 Théorème 7 : compositionnalité de $\approx^{\psi}$

On considère deux modules S_1 et S_2, un état x_1 de S_1 et un autre x_2 de S_2, on remarque tout d'abord que les formules qui font partie de $\mathsf{Form}((x_1, x_2))$ font partie à la fois de $\mathsf{Form}(x_1)$ et de $\mathsf{Form}(x_2)$. Il reste à montrer que si on a $\sim^{\varphi} (x_1, y_1)$ et $\sim^{\varphi} (x_2, y_2)$ pour une formule φ, on a alors également $\sim^{\varphi} ((x_1, y_1), (x_2, y_2))$. Soient f_1, f_2 et f_3 des fonctions du même type que $\sim^{\varphi}$ sur les modules S_1,S_2 et $S_1 \times S_2$ (respectivement), on définie la propriété $\mathcal{P}(f_1, f_2, f_3)$ comme suit : quelques soient les états x_1, y_1 de S_1 et x_2, y_2 de S_2,

$$f_1(x_1, y_1) \Rightarrow f_2(x_2, y_2) \Rightarrow f_3((x_1, x_2), (y_1, y_2)).$$

On montre alors que les règles utilisées dans la construction de $\sim^{\varphi}$ préservent cette propriété $\mathcal{P}$.

Cas B **:** On a d'une part $B@x_1 \Leftrightarrow B@y_1$ donc aussi $B@x_1@x_2 \Leftrightarrow B@y_1@x_2$. On a d'autre part $B@x_2 \Leftrightarrow B@y_2$ donc notamment $B@x_2@y_1 \Leftrightarrow B@y_2@y_1$. Puisque $B@x_2@y_1$ est égal à $B@y_1@x_2$ on en déduit $B@(x_1, x_2) \Leftrightarrow B(y_1, y_2)$.

Cas $\wedge$ **:** Si les trois fonctions f_1, f_2, f_3 vérifient $\mathcal{P}(f_1, f_2, f_3)$, et les trois autres f_1', f_2', f_3' vérifient $\mathcal{P}(f_1', f_2', f_3')$, on a alors $\mathcal{P}(f_1 \wedge f_1', f_2 \wedge f_2', f_3 \wedge f_3')$.

Cas $\langle l \rangle \varphi$ **:** La preuve de la compositionnalité de Fail à déjà été effectué dans la preuve du lemme 2. Considérons sans perte de généralité le cas où l'action l est locale au premier module. On suppose que l'on a d'une part

$\bigwedge_{x_1 \xrightarrow{l} x_1'} \bigvee_{y_1 \xrightarrow{l} y_1'} \sim^{\varphi} (x_1', y_1')$, et d'autre part $\sim^{\varphi} (x_2, y_2)$. On a alors

$$\begin{aligned}
&\sim^{\langle l \rangle \varphi} ((x_1, x_2), (y_1, y_2)) \\
&= \bigwedge_{(x_1,x_2) \xrightarrow{l} (x_1,x_2)'} \bigvee_{(y_1,y_2) \xrightarrow{l} (y_1,y_2)'} \sim^{\varphi} ((x_1, x_2)', (y_1, y_2)') \\
&= \bigwedge_{x_1 \xrightarrow{l} x_1'} \bigvee_{y_1 \xrightarrow{l} y_1'} \sim^{\varphi} ((x_1', x_2), (y_1', y_2)).
\end{aligned}$$

Donc si la fonction $\sim^{\varphi}$ est compositionnelle, $\sim^{\langle l \rangle \varphi}$ l'est aussi.

8.0.16 Théorème 8 : $\approx^{\psi}$ préserve Pass

Soient ψ une formule et x, y deux états d'un module S. Montrons que si on a $x \approx^{\psi} y$ alors pour toute formule φ de $\mathsf{Form}(x)$, $\mathsf{Pass}^{\varphi}(x) \Leftrightarrow \mathsf{Pass}^{\varphi}(y)$.

Cas B **:** Supposons $\sim^{B} (x, y)$, c'est à dire $B@a \Leftrightarrow B@y$. La propriété est vraie car $\mathsf{Pass}^{B}(x)$ est égal à $B@x$ et $\mathsf{Pass}^{B}(y)$ à $B@y$.

Cas $\varphi_1 \wedge \varphi_2$ **:** On suppose $\sim^{\varphi_1 \wedge \varphi_2}$, c'est à dire $\sim^{\varphi_1} (x, y)$ et $\sim^{\varphi_2} (x, y)$, ainsi que $\mathsf{Pass}^{\varphi_1 \wedge \varphi_2}$ qui est égal à $\mathsf{Pass}^{\varphi_1}(x) \wedge \mathsf{Pass}^{\varphi_2}(x)$. On obtient alors $\mathsf{Pass}^{\varphi_1}(y)$ et $\mathsf{Pass}^{\varphi_2}(y)$, c'est à dire $\mathsf{Pass}^{\varphi_1 \wedge \varphi_2}(y)$ à l'aide de l'hypothèse d'induction.

Cas $\varphi_1 \vee \varphi_2$ **:** Comme précédemment, on suppose d'une part $\sim^{\varphi_1} (x, y) \wedge \sim^{\varphi_2} (x, y)$, et d'autre part $\mathsf{Pass}^{\varphi_1}(x) \vee \mathsf{Pass}^{\varphi_2}(x)$. Ce qui implique $\mathsf{Pass}^{\varphi_1}(y) \vee \mathsf{Pass}^{\varphi_2}(y)$ grâce à l'hypothèse d'induction.

Cas $\langle l \rangle \varphi$: On a les égalités :

$$\begin{aligned}
\sim^{\langle l \rangle \varphi} (x, y) = \mathsf{Fail}^{\langle l \rangle \varphi}(x) \vee \bigwedge_{x \xrightarrow{l} x'} \bigvee_{y \xrightarrow{l} y'} \sim^{\varphi} (x', y') \\
\bigwedge \mathsf{Fail}^{\langle l \rangle \varphi}(y) \vee \bigwedge_{y \xrightarrow{l} y'} \bigvee_{x \xrightarrow{l} x'} \sim^{\varphi} (y', x') \\
\text{et } \mathsf{Pass}^{\langle l \rangle \varphi}(x) = \bigvee_{x \xrightarrow{l} x'} \mathsf{Pass}^{\varphi}(x').
\end{aligned}$$

Si $\mathsf{Fail}^{\langle l \rangle \varphi}$ est vérifiée alors $\mathsf{Pass}^{\langle l \rangle \varphi}$ ne l'est pas et l'implication est vraie. Supposons donc $\bigwedge_{x \xrightarrow{l} x'} \bigvee_{y \xrightarrow{l} y'} \sim^{\varphi} (x', y')$ et $\bigvee_{x \xrightarrow{l} x'} \mathsf{Pass}^{\varphi}(x')$.
Ce qui nous permet d'obtenir $\bigvee_{y \xrightarrow{l} y'} \mathsf{Pass}^{\varphi}(y')$.

Cas $\langle s \rangle \varphi$ **et** $\langle e \rangle \varphi$ **:** Les formules $\mathsf{Pass}^{\langle s \rangle \varphi}(x)$ et $\mathsf{Pass}^{\langle e \rangle \varphi}(x)$ sont toutes deux égales à $\bot$ pour tout état x.

8.0.17 Théorème 9 : la réduction par quotient préserves l'équivalence $\sim^{\varphi}$

Soit x un état du LTS initial et $[x]$ l'état correspondant à la classe d'équivalence de x dans le graphe réduit par quotient. On a $\mathsf{Form}(x) = \mathsf{Form}([x])$. Montrons maintenant par induction que pour toute formule φ appartenant à $\mathsf{Form}(x)$, on a $x \sim^{\varphi} [x]$.

Cas B **:** les états x et $[x]$ on des labels identiques donc la formule $B@x$ et égale à $B@[x]$.

Cas $\varphi_1 \wedge \varphi_2$ **:** On a $\sim^{\varphi_1 \wedge \varphi_2} (x,[x]) = \sim^{\varphi_1} (x,[x]) \wedge \sim^{\varphi_2} (x,[x])$, où les deux membres de la conjonction sont égal a $\top$ d'après l'hypothèse d'induction.

Cas $\langle l \rangle \varphi$: On a l'égalité suivante :

$$\sim^{\langle l \rangle \varphi} (x,[x]) = \mathsf{Fail}^{\langle l \rangle \varphi}(x) \vee \bigwedge_{x \xrightarrow{l} x'} \bigvee_{[x] \xrightarrow{l} [x]'} \sim^{\varphi} (x',[x]') = \top$$
$$\wedge \mathsf{Fail}^{\langle l \rangle \varphi}([x]) \vee \bigwedge_{[x] \xrightarrow{l} [x]'} \bigvee_{x \xrightarrow{l} x'} \sim^{\varphi} ([x]',x') = \top.$$

La première partie de la conjonction est vraie car pour tout successeur x' de x, il existe un successeur $[x]'$ de $[x]$ tel que $[x]' = [x']$. Si x n'a pas de successeur par l'action l on a alors $\mathsf{Fail}^{\langle l \rangle \varphi}(x) = \top$. La seconde partie de la conjonction est également vraie et la justification est similaire.

Cas $\langle e \rangle \varphi$ **et** $\neg \varphi$ **:** Dans ces cas, on peut directement appliquer l'hypothèse d'induction car les formules $\sim^{\langle e \rangle \varphi}$ et $\sim^{\neg \varphi}$ sont toutes deux égales à $\sim^{\varphi}$.

Bibliographie

[1] *14th International Conference on Application of Concurrency to System Design, ACSD 2014, Tunis La Marsa, Tunisia, June 23-27, 2014*. IEEE Computer Society, 2014.

[2] Bowen Alpern and Fred B Schneider. Recognizing safety and liveness. *Distributed computing*, 2(3) :117–126, 1987.

[3] Rajeev Alur, Costas Courcoubetis, and David Dill. Model-checking for real-time systems. In *Logic in Computer Science, 1990. LICS'90, Proceedings., Fifth Annual IEEE Symposium on e*, pages 414–425. IEEE, 1990.

[4] Rajeev Alur and David L Dill. A theory of timed automata. *Theoretical computer science*, 126(2) :183–235, 1994.

[5] Henrik Reif Andersen. Partial model checking. In *Logic in Computer Science, 1995. LICS'95. Proceedings., Tenth Annual IEEE Symposium on*, pages 398–407. IEEE, 1995.

[6] Henrik Reif Andersen and Jorn Lind-Nielsen. Partial model checking of modal equations : A survey. *International Journal on Software Tools for Technology Transfer*, 2(3) :242–259, 1999.

[7] Nathaniel Ayewah, David Hovemeyer, J David Morgenthaler, John Penix, and William Pugh. Using static analysis to find bugs. *Software, IEEE*, 25(5) :22–29, 2008.

[8] Roberto Barbuti, Nicoletta De Francesco, Antonella Santone, and Gigliola Vaglini. Selective mu-calculus and formula-based equivalence of transition systems. *Journal of Computer and System Sciences*, 59(3) :537–556, 1999.

[9] Samik Basu and CR Ramakrishnan. Compositional analysis for verification of parameterized systems. *Theoretical computer science*, 354(2) :211–229, 2006.

[10] Béatrice Bérard, Michel Bidoit, Alain Finkel, François Laroussinie, Antoine Petit, Laure Petrucci, and Philippe Schnoebelen. *Systems and software verification : model-checking techniques and tools*. Springer Science & Business Media, 2013.

[11] Jan A Bergstra, Alban Ponse, and Scott A Smolka. *Handbook of process algebra*. Elsevier, 2001.

[12] Eike Best, Raymond Devillers, and Jon G Hall. *The box calculus : a new causal algebra with multi-label communication*. Springer, 1992.

[13] Eike Best, Raymond Devillers, and Maciej Koutny. *Petri net algebra*. Springer Science & Business Media, 2013.

[14] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. *Symbolic model checking without BDDs*. Springer, 1999.

[15] Robert V Binder. Testing object-oriented software : a survey. *Software Testing, Verification and Reliability*, 6(3-4) :125–252, 1996.

[16] Ahmed Bouajjani, Jean-Claude Fernandez, Susanne Graf, Carlos Rodriguez, and Joseph Sifakis. Safety for branching time semantics. In *Automata, Languages and Programming*, pages 76–92. Springer, 1991.

[17] Jonathan P Bowen and Michael G Hinchey. *Applications of formal methods*. Prentice Hall PTR, 1995.

[18] Stephen D Brookes, Charles AR Hoare, and Andrew W Roscoe. A theory of communicating sequential processes. *Journal of the ACM (JACM)*, 31(3) :560–599, 1984.

[19] Randal E Bryant. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on*, 100(8) :677–691, 1986.

[20] Jerry R Burch, Edmund M Clarke, Kenneth L McMillan, David L Dill, and Lain-Jinn Hwang. Symbolic model checking : 10< sup> 20</sup> states and beyond. *Information and computation*, 98(2) :142–170, 1992.

[21] Laurence Calzone, Laurent Tournier, Simon Fourquet, Denis Thieffry, Boris Zhivotovsky, Emmanuel Barillot, and Andrei Zinovyev. Mathematical modelling of cell-fate decision in response to death receptor engagement. *PLoS Computational Biology*, 6(3) :e1000702, 2010.

[22] Heino Carstensen and Rüdiger Valk. Infinite behaviour and fairness in petri nets. In *Advances in Petri nets 1984*, pages 83–100. Springer, 1985.

[23] Allan Cheng, Javier Esparza, and Jens Palsberg. Complexity results for 1-safe nets. In *Foundations of software technology and theoretical computer science*, pages 326–337. Springer, 1993.

[24] S. Christensen and L. Petrucci. Modular analysis of Petri nets. *The Computer Journal*, 43(3) :224–242, 2000.

[25] Søren Christensen and Laure Petrucci. *Towards a modular analysis of coloured Petri nets*. Springer, 1992.

[26] E Clarke, O Grumberg, K McMillan, and X Zhao. E cient generation of counterexamples and witnesses in symbolic model checking. Technical report, Citeseer, 1994.

[27] Edmund M Clarke and E Allen Emerson. *Design and synthesis of synchronization skeletons using branching time temporal logic*. Springer, 1982.

[28] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2) :244–263, 1986.

[29] Edmund M Clarke, Reinhard Enders, Thomas Filkorn, and Somesh Jha. Exploiting symmetry in temporal logic model checking. *Formal Methods in System Design*, 9(1-2) :77–104, 1996.

[30] Edmund M Clarke, Orna Grumberg, and David E Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(5) :1512–1542, 1994.

[31] Darren Cofer. Model checking : Cleared for take off. In *Model Checking Software*, pages 76–87. Springer, 2010.

[32] Yves-Stan Le Cornec and Franck Pommereau. Modular ¯-calculus model-checking with formula-dependent hierarchical abstractions. In *14th International Conference on Application of Concurrency to System Design, ACSD 2014, Tunis La Marsa, Tunisia, June 23-27, 2014* [1], pages 11–20.

[33] Gerardo Costa and Colin Stirling. Weak and strong fairness in ccs. *Information and Computation*, 73(3) :207–244, 1987.

[34] Patrick Cousot and Radhia Cousot. Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN*

symposium on Principles of programming languages, pages 238–252. ACM, 1977.

[35] Pepijn Crouzen and Holger Hermanns. Aggregation ordering for massively compositional models. In *Application of Concurrency to System Design (ACSD), 2010 10th International Conference on*, pages 171–180. IEEE, 2010.

[36] Pepijn Crouzen and Frédéric Lang. Smart reduction. In *Fundamental Approaches to Software Engineering*, pages 111–126. Springer, 2011.

[37] R. De Nicola and F. Vaandrager. Three logics for branching bisimulation. *Journal of the ACM (JACM)*, 42(2) :458–487, 1995.

[38] Edsger W. Dijkstra. Hierarchical ordering of sequential processes. *Acta informatica*, 1(2) :115–138, 1971.

[39] Edsger Wybe Dijkstra, Edsger Wybe Dijkstra, Edsger Wybe Dijkstra, and Edsger Wybe Dijkstra. *A discipline of programming*, volume 1. prentice-hall Englewood Cliffs, 1976.

[40] E Allen Emerson and A Prasad Sistla. Symmetry and model checking. *Formal methods in system design*, 9(1-2) :105–131, 1996.

[41] Adrien Fauré, Aurélien Naldi, Fabrice Lopez, Claudine Chaouiya, Andrea Ciliberto, and Denis Thieffry. Modular logical modelling of the budding yeast cell cycle. *Molecular BioSystems*, 5(12) :1787–1796, 2009.

[42] J. Fernandez and L. Mounier. "on the fly" verification of behavioural equivalences and preorders. In *Computer Aided Verification*, pages 181–191. Springer, 1992.

[43] Patrice Godefroid, J van Leeuwen, J Hartmanis, G Goos, and Pierre Wolper. *Partial-order methods for the verification of concurrent systems : an approach to the state-explosion problem*, volume 1032. Springer Heidelberg, 1996.

[44] Hans Hansson and Bengt Jonsson. A logic for reasoning about time and reliability. *Formal aspects of computing*, 6(5) :512–535, 1994.

[45] David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic logic*, volume 206. MIT press Cambridge, 2000.

[46] Matthew Hennessy and Robin Milner. *On observing nondeterminism and concurrency*. Springer, 1980.

[47] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10) :576–580, 1969.

[48] Charles Antony Richard Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8) :666–677, 1978.

[49] Charles Antony Richard Hoare and He Jifeng. *Unifying theories of programming*, volume 14. Prentice Hall Englewood Cliffs, 1998.

[50] Gerard Holzmann. On-the-fly model checking. *ACM Computing Surveys (CSUR)*, 28(4es) :120, 1996.

[51] Kurt Jensen. *Coloured petri nets*. Springer, 1987.

[52] Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Computing Surveys (CSUR)*, 41(4) :21, 2009.

[53] Cliff B Jones. Specification and design of (parallel) programs. 1983.

[54] Simon Peyton Jones. *Haskell 98 language and libraries : the revised report*. Cambridge University Press, 2003.

[55] Shmuel Katz and Doron Peled. Verification of distributed programs using representative interleaving sequences. *Distributed Computing*, 6(2) :107–120, 1992.

[56] W David Kelton and Averill M Law. *Simulation modeling and analysis*. McGraw Hill Boston, 2000.

[57] John G Kemeny and James Laurie Snell. *Finite markov chains*, volume 356. van Nostrand Princeton, NJ, 1960.

[58] F Kordon, A Linard, M Beccuti, D Buchs, L Fronc, F Hulin-Hubard, F Legond-Aubry, N Lohmann, A Marechal, E Paviot-Adet, et al. Model checking contest@ petri nets, report on the 2013 edition. *arXiv preprint arXiv :1209.2382*, 2012.

[59] Ron Koymans. Specifying real-time properties with metric temporal logic. *Real-time systems*, 2(4) :255–299, 1990.

[60] D. Kozen. Results on the propositional [mu]-calculus. *Theoretical Computer Science*, 27(3) :333–354, 1983.

[61] Saul A Kripke. Semantical analysis of modal logic i normal modal propositional calculi. *Mathematical Logic Quarterly*, 9(5-6) :67–96, 1963.

[62] Charles Lakos and Laure Petrucci. Modular analysis of systems composed of semiautonomous subsystems. In *proc. of ACSD'04*. IEEE Computer Society Press, 2004.

[63] Leslie Lamport. Proving the correctness of multiprocess programs. *Software Engineering, IEEE Transactions on*, (2) :125–143, 1977.

[64] Leslie Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 5(2) :190–222, 1983.

[65] Frédéric Lang. Exp. open 2.0 : A flexible tool integrating partial order, compositional, and on-the-fly verification methods. In *Integrated Formal Methods*, pages 70–88. Springer, 2005.

[66] Frédéric Lang and Radu Mateescu. Partial model checking using networks of labelled transition systems and boolean equation systems. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 141–156. Springer, 2012.

[67] Chang-Yeong Lee. Representation of switching circuits by binary-decision programs. *Bell System Technical Journal*, 38(4) :985–999, 1959.

[68] ISO Lotos. A formal description technique based on the temporal ordering of observational behaviour. *International Organisation for Standardization-Information Processing Systems-Open Systems Interconnection, Geneva*, 1988.

[69] Per Martin-Löf. Constructive mathematics and computer programming. *Studies in Logic and the Foundations of Mathematics*, 104 :153–175, 1982.

[70] Radu Mateescu and Mihaela Sighireanu. Efficient on-the-fly model-checking for regular alternation-free mu-calculus. *Science of Computer Programming*, 46(3) :255–281, 2003.

[71] Ernst Mayr. Persistence of vector replacement systems is decidable. *Acta Informatica*, 15(3) :309–318, 1981.

[72] Kenneth L McMillan. *Symbolic model checking*. Springer, 1993.

[73] Nuno D Mendes, Frédéric Lang, Yves-Stan Le Cornec, Radu Mateescu, Gregory Batt, and Claudine Chaouiya. Composition and abstraction of logical regulatory modules : application to multicellular systems. *Bioinformatics*, 29(6) :749–757, 2013.

[74] Robin Milner. *Communication and concurrency*. Prentice-Hall, Inc., 1989.

[75] Robin Milner. *Communicating and mobile systems : the pi calculus*. Cambridge university press, 1999.

[76] Robin Milner, Robin Milner, Robin Milner, and Robin Milner. *A calculus of communicating systems*, volume 92. springer-Verlag Berlin, 1980.

[77] John C Mitchell. *Foundations for programming languages*, volume 1. MIT press Cambridge, 1996.

[78] Tadao Murata. Petri nets : Properties, analysis and applications. *Proceedings of the IEEE*, 77(4) :541–580, 1989.

[79] Aurélien Naldi, Jorge Carneiro, Claudine Chaouiya, and Denis Thieffry. Diversity and plasticity of th cell types predicted from regulatory network modelling. *PLoS computational biology*, 6(9) :e1000912, 2010.

[80] D. Park. Concurrency and automata on infinite sequences. *Theoretical Computer Science, Lecture Notes in Computer Science*, 104 :167–183, 1981.

[81] Joachim Parrow. Fairness properties in process algebra with applications in communication protocol verification. 1985.

[82] Doron Peled. Combining partial order reductions with on-the-fly model-checking. In *Computer aided verification*, pages 377–390. Springer, 1994.

[83] James L Peterson. Petri nets. *ACM Computing Surveys (CSUR)*, 9(3) :223–252, 1977.

[84] Carl Adam Petri. Kommunikation mit automaten. 1962.

[85] Amir Pnueli. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 46–57. IEEE, 1977.

[86] Amir Pnueli. The temporal semantics of concurrent programs. *Theoretical computer science*, 13(1) :45–60, 1981.

[87] Amir Pnueli. *In transition from global to modular temporal reasoning about programs*. Springer, 1985.

[88] Frank Pommereau. *Algebras of coloured Petri nets, and their applications to modelling and verification*. LAP LAMBERT Academic Publishing, 2010.

[89] M Prasanna, SN Sivanandam, R Venkatesan, and R Sundarrajan. A survey on automatic test case generation. *Academic Open Internet Journal*, 15(part 6), 2005.

[90] Vaughan R Pratt. Semantical consideration on floyo-hoare logic. In *Foundations of Computer Science, 1976., 17th Annual Symposium on*, pages 109–121. IEEE, 1976.

[91] Corrado Priami. Stochastic π-calculus. *The Computer Journal*, 38(7) :578–589, 1995.

[92] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in cesar. In *International Symposium on Programming*, pages 337–351. Springer, 1982.

[93] Lucas Sánchez, Claudine Chaouiya, and Denis Thieffry. Segmenting the fly embryo : logical analysis of the role of the segment polarity cross-regulatory module. *International journal of developmental biology*, 52(8) :1059, 2008.

[94] Lucas Sanchez and Denis Thieffry. Segmenting the fly embryo : : a logical analysis of the pair-rule cross-regulatory module. *Journal of theoretical Biology*, 224(4) :517–537, 2003.

[95] Klaus Schneider. *Verification of reactive systems : formal methods and algorithms*. Springer Science & Business Media, 2013.

[96] Sharon Shoham and Orna Grumberg. Compositional verification and 3-valued abstractions join forces. In *Static Analysis*, pages 69–86. Springer, 2007.

[97] A Prasad Sistla. Deciding full branching time logic. 1984.

[98] A Prasad Sistla. On characterization of safety and liveness properties in temporal logic. In *Proceedings of the fourth annual ACM symposium on Principles of distributed computing*, pages 39–48. ACM, 1985.

[99] Colin Stirling. *Modal and temporal properties of processes*. Springer, 2001.

[100] Robert E Strom and Shaula Yemini. Typestate : A programming language concept for enhancing software reliability. *Software Engineering, IEEE Transactions on*, (1) :157–171, 1986.

[101] Kuo-Chung Tai and Pramod V Koppol. Hierarchy-based incremental analysis of communication protocols. In *Network Protocols, 1993. Proceedings., 1993 International Conference on*, pages 318–325. IEEE, 1993.

[102] René Thomas. Regulatory networks seen as asynchronous automata : a logical description. *Journal of theoretical biology*, 153(1) :1–23, 1991.

[103] René Thomas, Denis Thieffry, and Marcelle Kaufman. Dynamical behaviour of biological regulatory networks—i. biological role of feedback loops and practical use of the concept of the loop-characteristic state. *Bulletin of mathematical biology*, 57(2) :247–276, 1995.

[104] Alan Mathison Turing. On computable numbers, with an application to the entscheidungsproblem. *J. of Math*, 58(345-363) :5, 1936.

[105] Antti Valmari. A stubborn attack on state explosion. In *Computer-Aided Verification*, pages 156–165. Springer, 1991.

[106] R.J. Van Glabbeek and W.P. Weijland. Branching time and abstraction in bisimulation semantics. *Journal of the ACM (JACM)*, 43(3) :555–600, 1996.

[107] Rob J van Glabbeek. Bounded nondeterminism and the approximation induction principle in process algebra. In *STACS 87*, pages 336–347. Springer, 1987.

[108] Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engineering*, 10(2) :203–232, 2003.

[109] Jiacun Wang. *Timed Petri nets : Theory and application*, volume 9. Springer Science & Business Media, 2012.

[110] Glynn Winskel and Mogens Nielsen. Models for concurrency. *DAIMI Report Series*, 22(463), 1993.

[111] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 214–227. ACM, 1999.

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

Titre : Analyse de systèmes modulaires à l'aide de techniques d'abstractions hiérarchiques.

Mots clés : Model-checking, modularité, systèmes concurents, abstraction herarchique, mu-calcul.

Dans cette thèse, on s'intéresse au problème de l'explosion combinatoire du model-checking sur des systèmes modulaires. Les techniques d'abstractions hiérarchiques permettent de construire de manière incrémentale une abstraction d'un modèle en composant des abstractions de ses parties. Cette opération doit en outre préserver les propriétés d'intérêt du modèle. Dans un premier temps, on définit le formalisme des réseaux de régulation modulaires, dans le but de pouvoir étudier des systèmes biologiques en utilisant de l'abstraction hiérarchique. On utilise ensuite cette méthode pour détecter les états stables accessibles depuis un état donné sur un modèle multicellulaire intervenant lors du développement embryonnaire de la drosophile. L'opération d'abstraction SAFETY utilisée préserve l'accessibilité de tous les états stable. C'est une réduction classique et générique dans le sens ou elle préserve également toutes les propriétés de sûreté d'un système. Dans un second temps, on suppose que l'on a connaissance de la formule globale (exprimée en µ-calcul) que l'on veut vérifier, ainsi que de l'état initial du module. On définit alors deux opérations de réduction différentes. Ces réductions ont pour seule contrainte de préserver la valeur globale de la formule et il est donc possible d'obtenir des réductions plus importantes qu'avec les méthodes génériques (le but reste le même : réduire la taille d'un module ou d'un sous-système donné sans avoir connaissance du reste du système). Lors du calcul de cette réduction, on effectue de l'analyse partielle pour déterminer si le sous-système contient assez d'information pour connaître la valeur de vérité de la formule sur le système global. Si c'est le cas, on peut arrêter là l'analyse par abstraction hiérarchique ; sinon, cette étape est tout de même utile pour réduire le module. Enfin, on teste la première de ces réductions à l'aide d'un prototype sur quelques exemples simples. En plus de constater les coefficients de réduction obtenus, cela permet de fournir des données pour s'attaquer par la suite au problème du meilleur ordre d'analyse hiérarchique. C'est en effet un problème difficile et l'ordre d'analyse a une grande influence sur les performances de ces méthodes.

Title: Analysing modular systems with hierarchical abractions.

Keywords: Model checking, modularity, concurent systems, hierarchical abstraction, mu-calculus.

In this thesis, we are interested in limiting the combinatorial explosion that happens when model-checking modular systems. We use hierarchical abstraction techniques, which allow one to build an abstraction of a modular system by composing abstractions of his parts, while ensuring that this operation does not change the temporal properties we are interested in. At first, we define the modular regulation networks formalism in order to apply hierarchical abstraction techniques to the study of biological systems. We then use this approach to find the reachable stable states of a multi-cellular model involved in the development of the fruit fly embryo. For this, we use the abstraction called SAFETY which reduces a system while keeping all of its reachable stable states. This is a classical reduction which is quite general in the sens that it also preserves all the safety properties of the model. After this, we define two new reduction operations, which are dependent on the µ-calculus formula we seek to verify on the global system. We also assume that we know the initial state of the module or sub-system to be reduced. Since these reductions must only preserve the value of one formula over the global system, they should be able to return smaller systems than the general ones. While computing these reductions on one module or sub-system, we use partial analysis techniques to test if it contains enough information to conclude about the truth value of the formula on the global system. If it is the case, we can stop the incremental analysis right away; otherwise, this step is still useful for the computation of the reduced sub-system. Finally, we use a prototype to test the first one of our reduction operations on some simple examples. This enables us to observe how big the reductions are as well as getting data in order to tackle the problem of the order of analysis in the future. It is a difficult question and an important one since the hierarchical order in which we build the abstraction has a huge weight on the efficiency of these methods.

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