Département de formation doctorale en informatique **École doctorale IAEM Lorraine**
UFR STMIA

Logiques spatiales de ressources, modèles d'arbres et applications

THÈSE

présentée et soutenue publiquement le 9 décembre 2005

pour l'obtention du

Doctorat de l'université Henri Poincaré – Nancy 1
(spécialité informatique)

par

Nicolas Biri

Composition du jury

Président :	Abdel Belaïd	Professeur, Université de Nancy II, Nancy, France
Rapporteurs :	Philippa Gardner	Professeur, Imperial College, Londres, Angleterre
	Yassine Lakhnech	Professeur, Université Joseph Fourier, Grenoble, France
Examinateurs :	Abdel Belaïd	Professeur, Université de Nancy II, Nancy, France
	Patrick Blackburn	Directeur de recherche, INRIA Lorraine, Nancy, France
	Didier Galmiche	Professeur, Université Henri Poincaré, Nancy, France
	Daniel Hirschkoff	Maître de conférences, ENS Lyon, Lyon, France

Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503

Mis en page avec la classe thloria.

À mes parents, à ma famille

et à ceux qui m'ont soutenu.

Remerciements

Je tiens avant toute chose à remercier les membres du jury qui m'ont fait l'honneur et le plaisir de participer à la critique et à la soutenance de cette thèse :

- Mme. Philippa Gardner, Professeur à l'imperial college de Londres, qui a bravé la barrière de la langue pour être rapporteur de cette thèse et dont les nombreuses questions lors de la soutenance m'ont permis de mettre en valeur les résultats présentés ;
- M. Yassine Lakhnech, Professeur à l'université Université Joseph Fourier de Grenoble, qui a accepté le rôle de rapporteur pour cette thèse et dont les remarques pertinentes ont permis de clarifier certains résultats présents dans la version finale ;
- M. Daniel Hirschkoff, Maître de conférence à l'École Nationale Supérieure de Lyon, qui a accepté d'être membre du jury malgré ses obligations personnelles et dont les remarques m'ont confirmé l'intérêt des résultats présentés dans le document.
- M. Patrick Blackburn, Directeur de recherche à l'INRIA Lorraine, qui a accepté d'être membre du jury et dont les questions ont permis d'observer les résultats de la thèse sous un nouvel angle, ouvrant ainsi des perspectives nouvelles pour la suite de ces travaux ;
- M. Abdel Belaïd, Professeur à l'Université Nancy 2, qui m'a fait l'honneur de présider le jury et qui m'a permis de clarifier la voie à suivre concernant certain es perspectives de recherche à la suite de cette thèse ;
- M. Didier Galmiche, Professeur à l'université Henri Poincaré, Nancy 1, qui a non seulement été membre du jury mais qui est avant tout mon directeur de thèse. Je lui suis grandement reconnaissant pour la patience, l'énergie et le temps qu'il a consacré au bon déroulement de cette thèse, ainsi que le soutien et les conseils qu'il m'a donné tout au long de notre collaboration, même lors de choix délicats.

Je souhaite ensuite remercier ma famille et mes proches pour la patience dont ils ont fait preuve tout au long de ces années, ainsi que pour le soutien sans faille qu'ils m'ont accordé. Parmi eux, mes pensées vont plus particulièrement à mes parents et à ma compagne, Sandrine, pour le réconfort qu'ils m'ont apporté dans les moments les plus difficiles.

Parmi les collègues qui m'ont accompagné tout au long de cette thèse, et qui sont devenus des amis au fil des années, je tiens tout d'abord à saluer Daniel et Jean-Marc, avec qui j'ai partagé plus qu'un bureau durant la thèse, notamment pour les nombreux moments de convivialités que nous avons partagé, ainsi que Dominique, pour le soutien dont il a su faire preuve à mon égard. Armelle, pour son écoute et son soutien de tous les instants, Suzanne pour son énergie et sa joie de vivre et Franck avec qui j'ai partagé de nombreuses heures de plaisir en jouant de la musique dans les sous-sol du laboratoire.

Je tiens également à remercier Noëlle Carbonnell dont l'abscence justifiée le jour de la soutenance est totalement excusée à la vue de des nombreux encouragements qu'elle a su me faire lors de ma thèse. Enfin, Antoine Tabbone et Guillaume Hanrot, avec qui j'ai partagé l'organisation du pot, et que je félicite pour leurs habilitations à diriger des recherches, obtenues le jour de ma soutenance de thèse.

Enfin, merci à tous ceux qui sont venus assister à la soutenance et au pot de thèse, et qui m'ont permis d'appécier cette journée à sa juste valeur.

« Il faut aller à l'idéal en passant par le réel. »
- Jean Jaurès (1859 - 1914).

Résumé

La modélisation et la spécification de systèmes distribués nécessitent une adaptation des modèles logiques utilisés pour leur représentation. Les notions d'emplacements et de ressources jouent notamment un rôle centrale dans la représentation de ces systèmes. Dans ce cadre, on propose tout d'abord une première logique, la logique linéaire distribuée et mobile (DMLL) qui intégre les notions de distribution et de mobilité. On propose également une sémantique à la Kripke et un calcul des séquents supportant l'élimination des coupures pour cette logique.

Cette première étude a mis en avant le rôle centrale de la sémantique pour la modélisation de systèmes distribués. On propose alors la structure des arbres de ressources, des arbres dont les nœuds possèdent des labels et contiennent des ressources appartenant à monoïde partiel de ressources et BI-Loc, une logique pour raisonner sur ces arbres, un langage permettant de modifier les arbres et son axiomatisation correcte et complète sous forme de triplets de Hoare. Concernant BI-Loc, on détermine des conditions suffisantes pour décider de la satisfaction et de la validité par model-checking et on développe une méthode de preuves fondée sur les tableaux sémantiques correcte et complète.

On montre comment on peut raisonner sur les tas de pointeurs grâce aux arbres de ressources. Enfin, on determine comment le modèle des arbres partiel peut être utilisé pour représenter et spécifier les données semi-structurées et raisonner sur la transformation de ce type de données.

Mots-clés: Logiques de ressources ; Logiques spatiales ; Sémantiques ; Composition partielle ; Arbres de ressources ; Vérification ; Preuves.

Abstract

Modelling and specifying distributed systems require an adaptation of logical model habitually used to represent those systems. The notions of location and resource are one of the key points for representing such systems. We begin with a first proposition of logic, the Distributed and Concurrent Linear Logic (DMLL), which integrates notion of distribution and of mobility. We also propose a Kripke's semantic and a sequent calculus that supports cut-elimination.

This first study emphasizes the central role of semantics in the modelling of distributed systems. We propose then a new structure, the resource trees, which are labelled trees containing resources from a partial monoid inside their nodes and a new logic, BI-Loc, to reason about these trees. We also propose a language to transform these trees and its correct and complete logical axiomatisation using Hoare's triples. Concerning BI-Loc, we determine sufficient condition to decide satisfaction and validity and we develop a correct and complete proof-search method using semantic labelled tableaux. This method is inspired of the one developed for BI.

Then we show some application of the partial tree model. Firstly, we show how the resource trees can be used to reason about the heap of pointers and a refinement of this model, the permission model. Then, we focus on semi-structured data, and show how we can represent and specify such data using resource trees and how it can be used to reason about transformation of these data.

Keywords: Resource logics ; Spatial logics ; Semantic ; Partial Composition ; Resource Trees ; Model-Checking ; Proof.

Table des matières

Table des figures

Introduction

La notion de *ressource* est une notion centrale dans le domaine des systèmes informatiques. Le terme de ressource est un terme général qui désigne ici tout élément avec lequel un système doit interagir. Sa signification dépend donc du niveau d'observation où l'on se place et du domaine d'application. Il peut s'agir par exemple d'éléments matériels, d'emplacements mémoires, de fichiers ou de variables d'un programme, cette liste n'étant pas exhaustive. Selon les systèmes étudiés et le type de ressources que l'on souhaite manipuler, on s'intéresse également à divers types d'actions sur ces ressources, comme la possibilité de les composer entre elles, d'en contrôler l'accès, de les consommer pour produire de nouvelles ressources ou de les distribuer et de les déplacer entre divers emplacements. Par exemple, si l'on considère le domaine des calculs distribués [10], les ressources manipulées sont des processus et des messages ; les interactions à considérer sont les notions de consommation et de production de ces ressources, ce qui correspond à l'envoi et la réception de messages entre les processus. De même, les notions de *distribution* et de *mobilité* des ressources sont importantes dans ce domaine car elles permettent d'exprimer clairement quelles ressources sont détenues par chaque processus et comment se déroule la communication entre ces processus. Ces notions sont également au cœur d'applications comme les web-services, où il est nécessaire de pouvoir spécifier le cheminement des informations et les traitements réalisés à divers emplacements sur le réseau [54], mais aussi d'applications manipulant des données semi-structurées comme les documents XML, où les informations à l'intérieur d'un document sont distribuées entre divers emplacements formant la structure du document.

On s'intéresse dans cette thèse aux modèles logiques qui permettent de spécifier et de vérifier ces divers types d'actions sur les ressources, en se focalisant toutefois sur les aspects de distribution et de mobilité des ressources. Il s'agit de définir des modèles permettant de représenter des systèmes où les ressources sont réparties en divers emplacements (*distribuées*) et où la communication entre les ressources est importante (*mobiles*).

Parmi les travaux s'intéressant à la distribution et à la mobilité des ressources, on peut notamment citer des structures comme les ambients [28] ou les pointeurs [62]. Les ambients sont des structures arborescentes avec labels dont les nœuds contiennent des programmes permettant de faire évoluer cette structure en déplaçant ou en consommant des nœuds. Cette structure permet de représenter des systèmes distribués où des processus doivent se mouvoir entre différents emplacements, comme par exemple le fonctionnement d'un firewall [28]. Une logique a également été proposée pour spécifier les propriétés des ambients [29]. Elle inclut notamment des opérateurs de la logique classiques pour exprimer la conjonction ($\wedge$), la disjonction ($\vee$) ou l'implication ($\rightarrow$) de propriétés et des opérateurs de séparation pour raisonner sur la (dé)composition d'ambients ($|$) et ses conséquences ($\triangleleft$). Des modalités spatiales et temporelles ont enfin été introduites pour raisonner sur la structure des ambients et sur l'évolution d'une configuration donnée au cours du temps. Des versions statiques de ce modèle ont également été proposées [23, 24]. Elles permettent notamment de représenter partiellement des données semi-structurées (SSD) telles que les documents XML, la structure de l'arbre reprenant la structure des données et des labels représentant partiellement les informations des SSD. La nécessité d'utiliser ces labels pour représenter les informations vient principalement de l'absence de la notion de ressource dans le calcul des ambients. Seuls les labels permettent alors de différencier le contenu des nœuds, en conséquence, on doit adapter le modèle en fonction des données que l'on veut représenter dans ces labels. Cette représentation a notamment utilisées pour vérifier les pointeurs de données semi-structurées [26] ou des programmes mettant à jour ces données [24].

Les tas de pointeurs quant à eux servent à représenter les données manipulées par un programme. Un pointeur associe une valeur à un nom (qui peut être vu comme un *emplacement*). On peut représenter des structures comme des arbres, les DAG ou des graphes [16] à l'aide des pointeurs. Les logiques permettant de raisonner sur ces structures font partie d'une famille logique que l'on nomme les *logiques de séparation* [62, 76, 81]. Ces logiques mélangent des opérateurs de la logique classique [62, 81] ou de la logique classique intuitionniste [76] ($\wedge, \vee, \rightarrow$) et des opérateurs permettant également de raisonner sur la composition des ressources ($*$) et dans certaines variantes [62, 76] sur les conséquences de de certaines compositions de ressources ($-\!\!*$). Des commandes de base pour manipuler les pointeurs [62]. Cela a conduit à une axiomatisation de ces commandes [62] sous forme de triplets de Hoare [60], ce qui permet de raisonner sur les propriétés des algorithmes manipulant ces structures [62], permettant par exemple de prouver des algorithmes de copie de DAG ou de création de *fringes*, une liste chaînée reliant les feuilles d'un arbre [16]. Une des particularités de cette structure est d'être partielle, c'est à dire que la composition de deux tas de pointeurs n'est pas toujours possible. En effet, un nom doit permettre de désigner un unique pointeur et il est impossible de composer deux tas qui auraient des noms de pointeurs en commun. Dans un cadre plus général, cette définition partielle de la composition nous semble importante car elle permet de contrôler de manière fine les ressources manipulables dans le modèle en excluant certaines compositions de ressources. Récemment, des travaux étendant le modèle des pointeurs ont été proposé pour vérifier l'accès aux variables par des programmes concurrent [17] ou pour vérifier la notion d'abstraction dans des programmes Java [77].

Ces différents modèles démontrent l'importance, dans le cadre de la spécification et de la vérification de systèmes et de programmes informatiques distribués, des notions de partage des ressources et de séparation. Selon le modèle manipulé, le terme de ressource peut désigner ici un ambient, un arbre, un pointeur ou des informations contenues dans un SSD. On a également souligné l'intérêt que pouvait avoir la prise en compte de la composition partielle dans ces modèles afin de gérer plus finement la notion de ressources. On remarque cependant que dans ces systèmes, la modification des types de données à prendre en compte implique la nécessité d'étendre ou de modifier le modèle initial, comme cela a été le cas pour représenter les pointeurs dans des données semi-structurées [26] à partir d'un modèle d'arbres [23] ou pour représenter les permissions [17] à partir du modèles des pointeurs [62].

On peut alors se demander s'il n'est pas possible de proposer un modèle général traitant des interactions entre les ressources et de leur distribution dans l'espace, modèle dont les instances permettraient de représenter les différents types de ressources évoqués ci-dessus, comme par exemple les informations contenues dans des données semi-structurées ou des pointeurs, avec ou sans permissions. Il s'agit donc de proposer des modèles de ressources généraux et d'y ajouter la prise en compte de la répartition spatiale de ces ressources. On peut pour cela considérer comme point de départ des modèles traitant de la production et de consommation de ressources comme ceux de la logique linéaire [55] ou des modèles traitant de la séparation et du partage des ressources comme ceux de la logique BI *« the Logic of Bunched Implications »* [76, 79]. Une approche possible pour cela est donc de considérer tout d'abord une structure de ressources très générale, pouvant être instanciée de différentes manières, et de l'étendre avec une modalité permettant de représenter la distribution des ressources dans l'espace.

C'est cette approche qui est développée dans cette thèse. Elle aboutit principalement à la proposition d'une nouvelle structure, les *arbres de ressources* [13], dont l'originalité est de mélanger une représentation générale et partielle des ressources et une représentation explicite de l'espace. On propose également une logique pour spécifier le comportement des arbres de ressources. À la manière des travaux de [62, 81], on définit ensuite un langage permettant de modifier les arbres de ressources et l'axiomatisation de ce langage sous forme de triplets de Hoare. On montre enfin que le modèle ainsi créé peut s'instancier facilement pour représenter les modèles reposant sur le modèle des pointeurs [17, 62] ou pour représenter les données semi-structurées et raisonner sur leur transformation.

Ces travaux sont détaillés dans les cinq chapitres qui composent cette thèse. Le premier chapitre est un préambule à notre étude et analyse la logique DCLL (*Distributed Concurrent Linear Logic*) [65] qui est une extension distribuée et mobile de la logique linéaire avec une modalité d'emplacement et de déplacement. On montre que l'utilisation d'une modalité unique pour gérer la distribution des ressources et leur mobilité n'est pas correct sémantiquement et on propose donc une nouvelle logique nommé DMLL *«Distributed and Mobile Linear Logic»* dotée d'un nouveau système constitué de modalités permettant de la séparation de l'espace et la mobilité des ressources. On introduit pour cela trois modalités, la première $[l]$ pour indiquer qu'une proposition est vraie en l, les deux autres, $\{l\}$ et $\langle l \rangle$ pour indiquer respectivement que l'on entre et sort d'un emplacement l. On établit ensuite un calcul des séquents internalisant les règles de mobilité, alors que celle-ci est gérée par congruence dans [65], ce qui permet d'internaliser dans la recherche de preuve le déplacement des ressources. On démontre alors que le calcul des séquents à la propriété de l'élimination des coupures et on présente un nouveau modèle de ressources *à la Kripke* pour la logique. Afin d'illustrer l'intérêt de la distinction entre distribution et mobilité dans la logique, on étudie la représentation et à la vérification de protocoles d'authentification. Ce thème étant déjà abordé dans de nombreux travaux basé sur l'utilisation de la logique linéaire [20, 31], on dispose de points de comparaison suffisants pour expliquer que les modalités fournissent plus d'informations dans les preuves d'attaques de protocoles et aident à la spécification de ces protocoles.

Ces travaux préliminaires ont permis de clarifier la façon dont les modalités devaient être utilisées pour représenter la distribution spatiale et la mobilité. Il apparaît ainsi clairement que ces deux aspects ne doivent pas être traiter simultanément dans les modèles proposés. On peut alors proposer un modèle de ressources mêlant gestion fine des ressources et l'expression de leur distribution dans l'espace. Les travaux réalisés autour de BI [76, 79], notamment sur les pointeurs [62, 75] montrent l'intérêt d'un modèle prenant en compte les aspects de *partage* et de la *composition partielle* des ressources. Le partage permet entres autres d'établir qu'une partie du modèle vérifie plusieurs propriétés et la composition partielle permet de contrôler les compositions *acceptables* de ressources. On définit donc dans le chapitre 2 la structure des *arbres de ressources*, un arbre de ressources étant un arbre dont les nœuds contiennent des ressources appartenant à un monoïde de ressources partiel [49, 72]. Cette structure permet à la fois de représenter explicitement la séparation de l'espace et un modèle de ressources complexe. Pour raisonner sur ces arbres, une nouvelle logique nommée BI-Loc, qui peut être considérée comme une extension de BI avec une modalité d'emplacement $[\cdot]$, est également proposée. L'ensemble arbres de ressources et logique associée forment un modèle logique que l'on nomme *le modèle d'arbres partiel*. Notre but étant de définir des modèles pour raisonner sur des programmes distribués et mobiles, on souhaite disposer de commandes pour modifier les arbres de ressources et également d'outils de spécification pour raisonner sur ces commandes. En suivant la démarche proposée dans [62, 75], on définit un langage de base pour le modèle à manipuler et on établit pour ces commandes des pré/post-conditions sous formes d'expressions logiques. On détermine ainsi pour un programme donné la pré-condition qui doit être vérifiée avant l'exécution pour aboutir à la post-condition désirée. On propose des nouvelles primitives pour ajouter, lire et supprimer des nœuds et des ressources aux arbres de ressources. On définit ensuite pour ces commandes une sémantique opérationnelle puis une sémantique sous forme de triplets de Hoare, les pré/post-conditions de ces triplets étant des formules de BI-Loc. Un résultat important est d'établir que ces pré-conditions sont les plus faibles possibles (*weakest precondition*) et on discute du problème de localité (*frame property*) des programmes de manipulation d'arbres créés à partir de ces commandes.

Dans le chapitre 3, on étudie les méthodes poure vérifier si une instance donnée du modèle vérifie une certaine formule logique (*model-checking*) et pour raisonner directement sur les propriétés de la logique (*recherche de preuves*). Ces deux aspects sont très importants dans le cadre de la vérification de programme sous forme de triplets de Hoare car ils permettent respectivement de vérifier si une configuration donnée respecte la pré-condition d'un programme et de s'assurer que la formule logique correspondant à la spécification des données initiales d'un programme permet bien de déduire la pré-condition nécessaire à l'exécution de ce dit programme.

Concernant le model-checking, on étudie tout d'abord le problème dans le modèle des ressources partiels pour la logique BI, on détermine des conditions suffisantes sur le monoïde pour décider de la satisfaction et de la validité dans BI propositionnelle. Ce résultat sert de base pour établir que la satisfaction et la validité pour la version propositionnelle de BI-Loc pour le modèle d'arbres partiel. Concernant la version avec quantificateurs de BI-Loc, on montre tout d'abord que la satisfaction est décidable si on ne considère pas l'opérateur $-\!\!*$ mais que la décidabilité est indécidable pour ce fragment. Ce résultat nous permet de prouver que la satisfaction et la décidabilité sont indécidables pour BI-Loc avec quantificateurs.

Pour la recherche de preuves, on définit une méthode de recherche de preuves pour la version propositionnelle de BI-Loc. La méthode proposée se fonde sur les travaux sur la prouvabilité par tableaux sémantiques de la version propositionnelle de BI proposée dans [72] qui est une méthode de preuve par réfutation avec des tableaux sémantiques avec labels [45, 49]. Il s'agit de définir un ensemble de labels qui accompagnent les formules des tableaux et qui correspondent à des contraintes qui doivent être respecter pour construire à un contre modèle. On cherche alors à aboutir à des contradictions sur ces contraintes pour obtenir une preuve. L'originalité de cette approche réside dans le fait qu'elle consiste à tenter de construire simultanément une preuve et un contre-modèle de la formule. On définit donc une telle méthode et on la présente en détail dans ce chapitre en insistant sur les spécificités due à la présence de la modalité d'emplacement dans BI-Loc. On démontre ensuite que la méthode ainsi obtenue est correcte et complète pour la sémantique des modèles d'arbres partiels.

L'un de nos objectifs est de présenter un modèle suffisamment général pour pouvoir être instancié pour modéliser des problèmes de natures différentes. Les chapitres 4 et 5 montrent sur divers exemples comment on peut utiliser le modèle et le langage de transformation présenté au chapitre 2. On a expliqué que les notions de composition partielle et de partage mise en évidence dans les arbres de ressources sont inspirés des notions présentes dans le modèle des pointeurs. C'est pour cela qu'il nous semble naturel d'étudier comment les arbres de ressources peuvent représenter ce modèle. Cette démarche est détaillée dans le chapitre 4. On détermine un monoïde d'arbres partiel particulier pour construire des arbres de ressources représentant un tas de pointeurs. On poursuit en montrant que l'on peut exprimer les propriétés initialement décrites en logique des pointeurs sous forme de formules logiques de BI-Loc. On termine en montrant que l'on peut alors utiliser les commandes de manipulation des arbres de ressources pour simuler les programmes sur les pointeurs et on montre sur un exemple comment traduire tas de pointeurs, programmes et pré/post-conditions de la logique des pointeurs pour raisonner directement sur les arbres de ressources en utilisant BI-Loc.

On a expliqué précédemment que le modèle des permissions de [17] est un raffinement du modèle des pointeurs qui inclut pour chaque cellule un poids indiquant si la cellule est disponible en lecture et/ou en écriture. Ce modèle pose des problèmes de spécification nouveaux si on le compare au modèle des pointeurs classique [75] car la composition de deux tas de pointeurs ne nécessite pas nécessairement que ces tas soient disjoints. Cette particularité est déjà prise en compte dans le cas général du modèle d'arbres partiel. De plus, on montre que ce modèle s'adapte facilement à celui des permissions en modifiant légèrement le monoïde de ressources utilisé pour le modèle des pointeurs. La connaissance des problèmes de composition dans le modèle d'arbres partiel permet de résoudre le problème de la spécification des arbres dans le modèle des permissions, problème laissé ouvert dans [17].

Dans le chapitre 5, on propose de représenter des documents XML sous forme d'arbres de ressources. Le but de cette représentation est d'utiliser BI-Loc pour vérifier la transformation de documents XML, ce qui représente un enjeu crucial pour assurer l'intéropérabilité entre divers programmes utilisant ce format de document [41]. Pour cela on détermine l'ensemble des arbres de ressources que l'on fait correspondre à des documents XML. Contrairement à de nombreux autres travaux, notre représentation de ces documents comprend l'intégralité des attributs des nœuds, alors qu'on se limite habituellement aux identifiants et aux pointeurs [24, 39]. On utilise BI-Loc pour spécifier le format des documents que l'on manipule. Pour cela, on propose des formules permettant de représenter les principales règles d'une DTD *«Document Type Definition»*, le format de description de document XML le plus courant. Afin de manipuler les documents XML, on propose un langage de transformation d'arbres spécifiques inspiré du langage présenté au chapitre 2. On définit pour ce langage les triplets de Hoare correspondant afin de pouvoir vérifier la transformation de documents et on donne un exemple d'utilisation de ce langage. On montre enfin que les axiomes proposés définissent les plus faibles pré-conditions pour chaque commande et on propose une règle de fenêtrage *(frame property)* pour ce langage.

Dans les conclusions et perspectives, on revient sur le rôle central de la recherche de preuves pour assurer qu'une assertion est la conséquence d'une autre. Dans ce cadre, il est crucial d'implanter les algorithmes décrits dans le chapitre 3. Pour cela, on peut envisager d'étendre le prouveur BILL [48] qui met en œuvre la méthode des tableaux pour BI afin d'obtenir un prouveur pour BI-Loc. On cherchera aussi à étendre les résultats obtenus pour la version de BI-Loc avec quantificateurs et pour raisonner sur les modèles spécifiques des chapitres 5 et 6. L'application aux documents XML offre elle aussi des perspectives intéressantes. On pourra par exemple étendre le langage proposé dans cette thèse pour, par exemple, proposer de traiter systématiquement tous les nœuds vérifiant certains critères. Il nous semble également indispensable d'établir le lien entre le langage proposé et les langages de manipulation usuels du domaine, comme DOM ou XSL.

Chapitre 1

Logique Linéaire Distribuée et Mobile

Le développement des applications distribuées nécessite une adaptation des outils logiques. L'incorporation des notions de séparation, de distribution spatiale et de mobilité dans un cadre logique a fait l'objet de nombreux travaux. De tels aspects ont cependant rarement été exprimés dans des logiques de ressources. Dans ce cadre, la logique DCLL[1] [65] présente une approche intéressante, visant à introduire une modalité d'emplacement explicite dans la logique linéaire. Ces travaux permettent donc d'ajouter à une logique de ressources une notion de distribution spatiale.

DCLL est une logique basée sur la logique linéaire [55] et ayant pour but premier d'être utilisée dans le cadre du paradigme *proof-search-as-computation*. Il s'agit de représenter sous forme de formules logiques les processus et d'établir le lien entre recherche de preuves et exécution des processus correspondants. Dans ce cadre, puisque la preuve représente une trace de l'exécution, plus les informations qu'elle présente sont importantes, plus l'analyse de l'exécution est facilitée. On comprend donc bien ici l'intérêt de représenter explicitement les emplacements des ressources et leur déplacements. Puisque l'on doit modéliser ici des programmes distribués devant interagir entre eux, la mobilité des ressources échangées entre les différents processus joue un rôle central. Le problème est que ces aspects de mobilité sont peu traités dans DCLL. On propose donc ici DMLL[2], une logique fondée sur DCLL qui incorpore le concept de mobilité des ressources.

Dans ce chapitre, on commence par un rappel du paradigme *proof-search-as-computation* et de DCLL. On présente ensuite DMLL et on donne sa formalisation sous forme de calcul des séquents. On établit les propriétés inhérentes à ce système logique, notamment l'élimination des coupures et on définit une sémantique de ressources *à la Kripke* basée sur celles pour la logique linéaire [6] pour DMLL. Enfin, pour illustrer les applications qui peuvent être faites de DMLL, on présente comment la logique peut être utilisée pour simuler l'exécution de protocoles et rechercher des attaques possibles.

1.1 Logique Linéaire, processus et emplacements

1.1.1 Logique linéaire

La logique linéaire (LL), proposée par Girard [55], est un raffinement de la logique classique. La syntaxe de LL peut être présentée comme une modification du calcul des séquents de la logique classique (CL). Tout d'abord, les règles structurelles de CL :

[1] Distributed Concurrent Linear Logic
[2] Distributed Mobile Linear Logic

$$\frac{\Gamma, \phi, \phi \vdash \Delta}{\Gamma, \phi \vdash \delta}\ \text{Contraction} \qquad \frac{\Gamma \vdash \Delta}{\Gamma, \phi \vdash \delta}\ \text{Affaiblissement}$$

ne peuvent plus être utilisées. Les formules des séquents ne peuvent donc plus être dupliquées ou supprimées lors de la recherche de preuve. Ainsi, alors que les séquents en CL sont composés d'un ensemble de formules où chaque élément peut être dupliqué ou supprimé, les séquents de LL sont composés d'un multi-ensemble de formules, le nombre d'occurrences d'une formule est important et toutes les formules doivent être consommées. C'est pour cela que l'on peut voir les formules de LL comme des ressources.

La logique linéaire se distingue aussi par la présence de deux types d'opérateurs, les *additifs* ($\&$, $\oplus$) et les *multiplicatifs* ($\otimes$, $\wp$). La différence entre ces deux catégories d'opérateurs découle du comptage des ressources. Les séquents étant des multi-ensembles, les règles d'inférence peuvent soit dissocier les ressources (ceci correspond aux multiplicatifs)

$$\frac{\Gamma_1, \phi \vdash \Delta_1 \quad \Gamma_2, \psi \vdash \Delta_2}{\Gamma_1, \Gamma_2, \phi \wp \psi \vdash \Delta_1, \Delta_2}\ (\wp)$$

soit les partager (pour les additifs)

$$\frac{\Gamma, \phi \vdash \Delta \quad \Gamma, \psi \vdash \Delta}{\Gamma, \phi \oplus \psi \vdash \Delta}\ (\oplus)$$

On a donc vu que les propositions de la logique linéaire peuvent être vu comme des ressources. Il est parfois nécessaire d'indiquer que l'on possède une infinité de ressources d'un certain type ou qu'une ressource peut ne pas être présente. Ces deux possibilités s'expriment respectivement en logique linéaire à l'aide des opérateurs unaires ! et ?.

Du fait de la gestion fine des ressources, la logique linéaire et ses fragments ont permis d'aborder sous un angle nouveau des applications informatiques dans différents domaines et suivant différents paradigmes de programmation [5]. Dans le cadre de la programmation fonctionnelle, elle a conduit à des nouvelles études concernant l'isomorphisme de Curry-Howard et le paradigme *proofs-as-programs* [2] ainsi que de nouvelles techniques d'implantation.

Dans le cadre de la programmation logique et du paradigme *proof-search-as-computation*, la logique linéaire, vue comme une logique de ressources, a permis de raffiner un certain nombre de concepts et a conduit à des modélisations précises et naturelles de certains mécanismes et systèmes. Cela a conduit en parallèle à l'étude de la recherche de preuves en logique linéaire et à sa mise en œuvre (méthodes, implantations) suivant différentes approches : preuves canoniques [7, 52], preuves uniformes [61], réseaux de preuve [51]. Ainsi la programmation logique linéaire illustre de nouveaux concepts et mécanismes nécessaires à l'utilisation de la logique linéaire pour modéliser des systèmes ou des propriétés ainsi que pour développer des preuves pour le calcul ou la vérification.

1.1.2 Logique et calculs de processus

Pour illustrer le paradigme *proof-search-as-computation*, on détaille ici comment la logique linéaire peut être utilisée pour représenter des processus concurrents et la communication (envoi/réception de message) entre ces processus. Chaque élément impliqué dans un calcul concurrent peut alors être vu sous la forme d'une formule de logique linéaire et l'exécution peut-être vue comme la déduction (recherche de preuves) dans cette logique. La correspondance ci-dessous est celle qui est utilisée notamment dans [18, 66, 67].

Considérons deux processus P et Q. La formule $P \otimes Q$ indique que l'on a simultanément P et Q, elle est donc interprétée comme la mise en parallèle de P et Q.

De la même manière, puisqu'un message doit être consommé par un processus, il peut être représenté en logique linéaire par la ressource $m(v)$. Dans cette traduction, m est vu comme le nom de canal de communication et v comme le message transmis le long de m. Le récepteur, lui, utilise la ressource $m(v)$ pour produire autre chose ; il est donc traduit par une formule du type $m(v) \multimap P$ où le processus P correspond au processus à exécuter après la communication (cette formule se lit en logique linéaire *« la ressource $m(v)$ est nécessaire pour fournir P »*). La communication ainsi définie se traduit donc en logique linéaire de la manière suivante :

$$(m(v) \otimes (m(v) \multimap P)) \multimap P$$

Ce qui se lit, d'un point de vue opérationnel : *"s'il existe un message $m(v)$ et un processus $m(v) \multimap P$ alors, ce dernier reçoit le message et se comporte comme P"*. On peut aussi, à l'aide de la formule $\forall x.(m(x) \multimap P)$ définir un processus qui peut reçoit tout message de la forme $m(v)$. On a donc :

$$(m(v) \otimes \forall x.(m(x) \multimap P)) \multimap P[v/x]$$

On peut ainsi aussi transmettre des noms de canaux par exemple :

$$(m(n) \otimes \forall x.(m(x) \multimap x(1))) \multimap n(1)$$

Ici, le message n (qui est un canal), envoyé le long du canal m, est utilisé ensuite pour envoyer le message 1.

De la même manière, $\exists x.P$ permet de *cacher* x de l'extérieur. En effet, x doit être remplacé par un nom n'apparaissant pas déjà dans la formule lors de la recherche de preuves, il permet donc de représenter un canal *original* x et d'exécuter alors P. La formule $!P$ qui représente un nombre infini de copies de la ressource P. Cette utilisation est notamment utile pour définir une définition de processus : la formule $!\forall x.f(x) \multimap P$ qui indique que chaque ressource du type $f(v)$ peut être utilisée pour obtenir $P[x/v]$. Il s'agit donc de la définition de processus $f(x) = P$.

Les notions d'emplacements et de mobilité ne sont pas présentes explicitement en logique linéaire. Ces notions sont toutefois centrales lorsqu'il s'agit de différencier par exemples les ressources disponibles à divers emplacement d'un système distribué. Les travaux de Kobayashi et al. [65] qui sont présentés ci-dessous enrichissent la logique linéaire avec des modalités permettant de rendre compte de ces emplacements. La méthodologie suivie pour inclure cette modalité est d'ailleurs proche de celle utilisée par Kanovitch et Ito pour inclure des modalités spatiales à la logique linéaire [64].

1.1.3 Distributed Concurrent Linear Logic (DCLL)

DCLL est une extension de la logique linéaire définie pour être utilisée selon le paradigme *proof-search-as-computation* [65]. Aux opérateurs classiques de la logique linéaire, on ajoute une modalité d'emplacement $[\cdot]$ dont le but est d'indiquer à quel emplacement la formule est vérifiée. Ainsi, la formule $[l]A$ indique que la formule A est vérifiée à l'emplacement l. Malgré la présence de modalités, cette logique ne peut être appelée comme une logique modale au sens de [14], le comportement et la signification des modalités d'emplacements et de déplacements définies ici étant différents du comportement usuel des modalités des logiques modales.

Le type d'adressage pour les emplacements est un système d'adresses *absolues*, c'est-à-dire qu'un nom d'emplacement réfère toujours au même emplacement, quelque soit le contexte dans lequel il est utilisé. Ce choix est forcé par le fait qu'une seule et même modalité gère les emplacements et les déplacements. La mise en place d'un système à *adresses relatives* comme celui utilisé dans les *ambients* [28] nécessiterait entres autres une modalité permettant de sortir d'un emplacement. En effet, l'utilisation d'adresses absolues correspond en fait à la définition d'un nom unique pour chaque emplacement. Le déplacement d'une ressource d'un emplacement à un autre se fait alors directement en indiquant à quelle adresse devra se trouver la ressource. Dans un système à adresses relatives, l'adresse d'une ressource correspond au chemin menant de la racine à cette ressource. Le système d'adresses utilisé a pour conséquence que la modalité d'emplacement $[\cdot]$ permet également de modéliser les déplacements. Ceux-ci sont en effet gérés par la règle de congruence $[l_1][l_2]A \cong [l_2]A$. Les règles de la logique sont données en figure 1.1.

On a expliqué que cette logique avait été établie pour être utilisée selon le paradigme *proof-search-as-computation*. Cette utilisation se base sur la correspondance entre formules et processus détaillée dans la section précédente. L'ajout des emplacements permet alors de *localiser* les processus et les ressources que l'on manipule. Ainsi un processus ne peut consommer que les ressources présentes à son emplacement. La figure 1.2 montre deux séquents illustrant ce fait. Le séquent de gauche est prouvable puisque le processus $[l](A \multimap B)$ peut consommer A pour produire B puisque A est en l. Dans le séquent de droite, on ne peut conclure puisque A est en l_2 alors qu'il doit être consommé en l_1.

1.1.4 Le problème des emplacements de DCLL

La gestion des déplacements dans DCLL pose différents problèmes. L'utilisation d'une règle de congruence notamment, ajoutée au fait qu'il n'y ait pas de modalité particulière pour la sortie d'un emplacement, font que la présentation manque de clarté. Ce choix est étroitement lié à l'utilisation d'adresses absolues. Cette solution a été choisie car la notion de mobilité n'était pas centrale dans leur travaux or, avec un tel système d'adressage, le déplacement des ressources s'exprime de manière plus concise. En contrepartie, le déplacement des ressources n'apparaît pas explicitement dans la logique comme on l'explique ci-dessous.

L'utilisation de la congruence $[l_1][l_2]A \cong [l_2]A$ est en effet la seule façon d'indiquer qu'un déplacement a lieu de l_1 vers l_2. Le choix d'une règle de congruence pour exprimer la mobilité n'est toutefois pas adaptée. Tout d'abord, l'utilisation de la règle dans le sens gauche-droite, si elle est compréhensible, est toutefois peu adéquate d'un point de vue sémantique, puisqu'elle donne une impression de déplacement instantané. Cette impression est d'ailleurs due en grande partie à l'utilisation d'adresses absolues, puisque ce type d'adressage ne permet pas de suivre le cheminement de l'information. L'utilisation dans le sens droite-gauche, elle, n'a aucun sens puisqu'elle revient à rajouter des informations inutiles en tête de formule. Nous proposons donc de définir un système gérant les déplacements de manière explicite (avec une modalité de déplacement) et d'orienter les règles gérant ce déplacement.

1.2 Présentation du système logique DMLL

L'idée première est de disposer de modalités pour sortir d'un emplacement et pour entrer dans un emplacement. Nous noterons la modalité de sortie $\langle\text{-}\rangle$ et celle d'entrée $\{\text{-}\}$. Nous devons aussi disposer d'une modalité indiquant que l'on est dans un emplacement. Nous noterons cette

Règles de congruence :

$[l](A \otimes B) \cong [l]A \otimes [l]B$ $\quad$ $[l](A\&B) \cong [l]A\&[l]B$

$[l](A \multimap B) \cong [l]A \multimap [l]B$ $\quad$ $[l]!A \cong ![l]A$

$[l]\exists x.A \cong \exists x.[l]A$ $\quad$ $[l]\forall x.A \cong \forall x.[l]A$

$[l_1][l_2]A \cong [l_2]A$ $\quad$ $[l]1 \cong 1$

Axiome :

$B \vdash B$ (DCLL-Id)

Règles structurelles :

$$\frac{\Gamma_1, A, B, \Gamma_2 \vdash D}{\Gamma_1, B, A, \Gamma_2 \vdash D}\ (\text{DCLL-Exch}) \qquad \frac{\Gamma \vdash B \quad B, \Gamma' \vdash D}{\Gamma, \Gamma' \vdash D}\ (\text{DCLL-Cut})$$

$$\frac{A', \Gamma \vdash D \quad A \cong A'}{A, \Gamma \vdash D}\ (\text{DCLL-} \cong : \text{L}) \qquad \frac{\Gamma \vdash A' \quad A \cong A'}{\Gamma \vdash A}\ (\text{DCLL-} \cong : \text{R})$$

$$\frac{\Gamma \vdash D}{!A, \Gamma \vdash D}\ (\text{DCLL-Weak}) \qquad \frac{!A, A, \Gamma \vdash D}{!A, \Gamma \vdash D}\ (\text{DCLL-Con})$$

Connecteurs logiques :

$$\frac{\Gamma \vdash D}{\Gamma, 1 \vdash D}\ (\text{DCLL-}1 : \text{L}) \qquad \vdash 1 (\text{DCLL-}1 : \text{R})$$

$$\frac{A, B, \Gamma \vdash D}{A \otimes B, \Gamma \vdash D}\ (\text{DCLL-} \otimes : \text{L}) \qquad \frac{\Gamma_1 \vdash A \quad \Gamma_2 \vdash B}{\Gamma_1, \Gamma_2 \vdash A \otimes B}\ (\text{DCLL-} \otimes : \text{R})$$

$$\frac{A_i, \Gamma \vdash D \quad i = 1 \text{ ou } 2}{A_1\&A_2, \Gamma \vdash D}\ (\text{DCLL-}\& : \text{L}) \qquad \frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A\&B}\ (\text{DCLL-}\& : \text{R})$$

$$\frac{B, \Gamma_1 \vdash D \quad \Gamma_2 \vdash A}{A \multimap B, \Gamma_1, \Gamma_2 \vdash D}\ (\text{DCLL-} \multimap : \text{L}) \qquad \frac{A, \Gamma \vdash B}{\Gamma \vdash A \multimap B}\ (\text{DCLL-} \multimap : \text{R})$$

$$\frac{A, \Gamma \vdash D}{!A, \Gamma \vdash D}\ (\text{DCLL-}! : \text{L}) \qquad \frac{!\Gamma \vdash A}{!\Gamma \vdash !A}\ (\text{DCLL-}! : \text{R})$$

$$\frac{A[t/x], \Gamma \vdash D}{\forall x.A, \Gamma \vdash D}\ (\text{DCLL-}\forall : \text{L}) \qquad \frac{\Gamma \vdash A[y/x]}{\Gamma \vdash \forall x.A}\ (\text{DCLL-}\forall : \text{R})^*$$

$$\frac{A[y/x], \Gamma \vdash D}{\exists x.A, \Gamma \vdash D}\ (\text{DCLL-}\exists : \text{L})^* \qquad \frac{\Gamma \vdash A[t/x]}{\Gamma \vdash \exists x.A}\ (\text{DCLL-}\exists : \text{R})$$

* : y n'est libre ni dans Γ, ni dans A, ni dans D.

FIG. 1.1 – Calcul des séquents pour DCLL

$$\dfrac{\dfrac{}{[l]B \vdash [l]B}\,(\text{Id}) \quad \dfrac{}{[l]A \vdash [l]A}\,(\text{Id})}{[l](A \multimap B) \otimes [l]A \vdash [l]B}\,(\multimap : \text{L}) \qquad \dfrac{\dfrac{}{[l_1]B \vdash [l_1]B}\,(\text{Id}) \quad [l_2]A \vdash [l_1]A}{[l_1](A \multimap B) \otimes [l_2]A \vdash [l_1]B}\,(\multimap : \text{L})$$

FIG. 1.2 – Exemple de distribution dans DCLL

modalité [-]. Ainsi l'information $[l_1][l_2]A$ ne signifie pas, comme dans DCLL [65] que *le processus A s'est déplacé de l'emplacement* l_1 *à l'emplacement* l_2 mais que *le processus A est à l'emplacement* l_2 *qui lui-même est sous l'emplacement* l_1. L'espace des emplacements est donc tel que chaque emplacement peut contenir d'autres sous-emplacements. Il est donc *hiérarchisé* comme c'est le cas dans les logiques spatiales dérivés des ambients [29] par exemple. Nos modalités [-], $\langle$-$\rangle$ et {-} ont ici une sémantique claire : $[\cdot]$ est une modalité désignant un emplacement, $\langle$-$\rangle$ indique la sortie d'un emplacement et {-} l'entrée dans un emplacement. Dans [65] ce n'était pas le cas, puisque la modalité [-] correspondait selon les cas à un emplacement ou à un déplacement.

L'ajout de ces modalités permet de modéliser plus finement la mobilité des processus et des ressources. Par exemple, la configuration $[l_1]\{l_2\}a$ signifie que l'information qui est en $[l_1]$ se déplacera en $[l_1][l_2]$, alors que $[l_1][l_2]a$ indique que la ressource a est *déjà* en $[l_1][l_2]$. Cette différentiation permet de représenter explicitement les différents emplacements par lesquels vont transiter les ressources et d'intégrer la notion de mobilité dans les formules logiques.

Le type de règle permettant de gérer la mobilité est également différent de celui de [65]. Les règles de congruence seront identiques à celles de DCLL pour la plupart des opérateurs. Cependant, la règle gérant la mobilité sera maintenant orientée. Ce choix se fonde sur la sémantique de ces règles. Pour les règles permettant de déplacer les modalités d'emplacement à l'intérieur des formules, la congruence se justifie par l'équivalence entre les deux interprétations. Prenons l'exemple de la règle $[l_1](A \otimes B) \cong [l_1]A \otimes [l_1]B$. La partie gauche se lit : *l'emplacement* $[l_1]$ *contient le processus A et le processus B*. La partie droite elle correspond à *le processus A est en* l_1 *et le processus B est en* l_1 *(aussi)*. On voit bien que les deux interprétations peuvent être utilisées indifféremment et que le passage de l'une à l'autre ne pose aucun problème. Pour le déplacement, la signification même de la règle conduit à devoir l'orienter. En effet comme cela a été expliqué précédemment, la réalisation d'un déplacement est fortement orientée. Il serait effectivement tout à fait absurde de permettre de rajouter des déplacements lors de la recherche de preuves. Cela irait de plus à l'encontre de ce qui a été dit précédemment à propos de l'apport des adresses relatives en matière d'analyse de la sécurité d'un système puisqu'on ferait passer les ressources dans des emplacements qu'elles n'auraient pas dû visiter.

DMLL se fonde sur l'utilisation de trois modalités : {-}, $\langle$-$\rangle$ et [-] qui permettent respectivement d'indiquer la sortie, l'entrée, et le fait d'être dans un emplacement. Les règles de congruence pour ces opérateurs sont uniquement celles permettant de diffuser l'information relative à l'emplacement d'un processus. Les règles originales de DMLL (celles gérants les emplacements) sont indiquées dans la figure 1.3. Les autres règles sont celles de DCLL et sont donc présentées en figure 1.1.

Les règles de congruence reprises de DCLL sont celles qui ne correspondent pas à un déplacement mais qui permettent de diffuser ou de rassembler les informations relatives à l'emplacement des processus. Les règles (DMLL-In) et (DMLL-Out)

$$\frac{[l_1]\dots[l_{n+1}]A, \Gamma \to D}{[l_1]\dots[l_n]\{l_{n+1}\}A, \Gamma \to D}\ (\mathsf{DMLL\text{-}In : L}) \qquad \frac{\Gamma \to [l_1]\dots[l_{n+1}]A}{\Gamma \to [l_1]\dots[l_n]\{l_{n+1}\}A}\ (\mathsf{DMLL\text{-}In : R})$$

$$\frac{[l_1]\dots[l_{n-1}]A, \Gamma \to D}{[l_1]\dots[l_n]\langle l_n\rangle A, \Gamma \to D}\ (\mathsf{DMLL\text{-}Out : L})\ ^* \qquad \frac{\Gamma \to [l_1]\dots[l_{n-1}]A}{\Gamma \to [l_1]\dots[l_n]\langle l_n\rangle A}\ (\mathsf{DMLL\text{-}Out : R})\ ^*$$

Règles de congruence :

$$[l](A \otimes B) \cong [l]A \otimes [l]B \qquad [l](A \& B) \cong [l]A \& [l]B$$

$$[l](A \multimap B) \cong [l]A \multimap [l]B \qquad [l]!A \cong ![l]A$$

$$[l]\exists x.A \cong \exists x.[l]A \qquad [l]\forall x.A \cong \forall x.[l]A$$

Règles orientées pour la mobilité :

$$\frac{[l_1]\dots[l_{n-1}]A, \Gamma \to D}{[l_1]\dots[l_n]\langle l_n\rangle A, \Gamma \to D}\ (\mathsf{DMLL}\text{-Out : L})\ ^* \qquad \frac{\Gamma \to [l_1]\dots[l_{n-1}]A}{\Gamma \to [l_1]\dots[l_n]\langle l_n\rangle A}\ (\mathsf{DMLL}\text{-Out : R})\ ^*$$

$$\frac{[l_1]\dots[l_{n+1}]A, \Gamma \to D}{[l_1]\dots[l_n]\{l_{n+1}\}A, \Gamma \to D}\ (\mathsf{DMLL}\text{-In : L}) \qquad \frac{\Gamma \to [l_1]\dots[l_{n+1}]A}{\Gamma \to [l_1]\dots[l_n]\{l_{n+1}\}A}\ (\mathsf{DMLL}\text{-In : R})$$

$$\frac{1, \Gamma \to D}{[l_1]\dots[l_n]1, \Gamma \to D}\ (\mathsf{DMLL}\text{-Loc1} : L) \qquad \frac{\Gamma \to 1}{\Gamma \to [l_1]\dots[l_n]1}\ (\mathsf{DMLL}\text{-Loc1} : R)$$

* : $n \geq 2$

FIG. 1.3 – Règles de gestion des emplacements avec DMLL

permettent *d'effectuer* le déplacement. Les règles (DMLL-Loc1) quant à elles

$$\frac{1, \Gamma \to D}{[l_1]\dots[l_n]1, \Gamma \to D}\ (\mathsf{DMLL}\text{-Loc1} : L) \qquad \frac{\Gamma \to 1}{\Gamma \to [l_1]\dots[l_n]1}\ (\mathsf{DMLL}\text{-Loc1} : R)$$

permettent explicitement de supprimer les informations sur les emplacements liées à des processus inactifs.

1.3 Propriétés de DMLL

Il est nécessaire, pour juger de la pertinence du système proposé, de prouver quelques propriétés de la logique. L'élimination des coupures est très importante pour pouvoir faire le lien entre l'exécution et la preuve. En effet, l'utilisation d'une coupure correspond en logique à la démonstration d'un lemme. Dans le cadre de l'exécution, l'utilisation d'une coupure indique qu'un processus peut être ajouté sans modifier le comportement global du système. L'élimination des coupures permet donc de montrer qu'un tel processus peut être supprimé sans conséquence pour le comportement global du système.

Il est ensuite important de pouvoir faire le lien entre la logique et un modèle sémantique formel, afin de fixer la sémantique de notre logique. Il s'agit en fait de définir le rôle de chaque opérateur et leur signification. Il faut donc définir le modèle correspondant à cette logique et vérifier qu'il est adapté.

1.3.1 Élimination des coupures

On peut facilement démontrer l'élimination des coupures, pour un noyau de DMLL limité au règles de congruence et aux règles d'inférence ne gérant pas la mobilité (donc sans (DMLL-Out), (DMLL-In), (DMLL-Loc1)), comme dans DCLL [65]. Il reste alors à traiter les cas des règles gérant la mobilité.

Il s'agit tout d'abord de montrer que les règles (DCLL-$\cong$: L) et (DCLL-$\cong$: R) peuvent être éliminées en modifiant légèrement les autres règles. Il s'agit de fusionner les règles (DCLL-$\cong$: L) et (DCLL - $\forall$: L) et les règles (DCLL-$\cong$: R), (DCLL - $\forall$: R), dans les règles suivantes :

$$\frac{B, \Gamma \to D \quad A[t/x] \cong B}{\forall x.A, \Gamma \to D} \ (\mathsf{DCLL}\ \text{-}\forall' : \mathrm{L}) \qquad \frac{\Gamma \to B \quad A[t/x] \cong B}{\Gamma \to \exists x.A,} \ (\mathsf{DCLL}\ \text{-}\exists' : \mathrm{R})$$

Il convient pour la suite de la preuve de définir ce qu'est une formule en $\beta[l]$-forme normale.

Définition 1.1 ($\beta[l]$-forme normale). *Une formule est en* $\beta[l]$-forme normale *s'il ne contient pas de* β*-redex de la forme* $[l](A\&B)$*,* $[l](A \otimes B)$*,* $[l](A \multimap B)$*,* $[l](\exists x.A)$*,* $[l](\forall x.A)$ *et* $[l]!A$*.*

On suppose que chaque terme à une $\beta[l]$-forme normale unique. Un séquent est en $\beta[l]$-forme normale si ses formules sont en $\beta[l]$-forme normale. On donne maintenant la règle :

$$\frac{\text{A est en } \beta[l]\text{-forme normale}}{A \to A} \ (\mathsf{DCLL}\text{-Id'})$$

Un séquent DMLL' est un séquent obtenu à partir de DMLL en remplaçant les règles (DCLL-Id), (DCLL-$\cong$: L), (DCLL-$\cong$: R), (DCLL-$\forall$: L), (DCLL-$\exists$: R) par les règles (DCLL-Id'), (DCLL-$\forall'$: L) et (DCLL-$\exists'$: R). Nous admettrons que tous les séquents de DMLL sont en $\beta[l]$-forme normale.

Lemme 1.1.
(1) Si un séquent $\Gamma \to A$ est prouvable en DMLL', il est aussi prouvable avec DMLL.
(2) Si un séquent $\Gamma \to A$ est prouvable en DMLL, alors il existe un séquent $\Gamma' \to A'$ tel que :
- $\Gamma' \to A'$ est prouvable en DMLL', et
- $\Gamma \to A$ est dérivable de $\Gamma' \to A'$ en utilisant les règles (DCLL-$\cong$: L) et (DCLL-$\cong$: R).

Preuve.
(1) Immédiat : chaque règle de DMLL' est la composée d'une ou plusieurs règles de DMLL.
(2) Si $\Gamma \to A$ est un séquent prouvable dans DMLL, il est aussi prouvable dans DMLL' en y ajoutant les règles (DCLL-$\cong$: L) et (DCLL-$\cong$: R). Le résultat découle alors du fait que l'utilisation de (DCLL-$\cong$: L) avant (DCLL-$\forall$: L) et celle de (DCLL-$\cong$: R) avant (DCLL-$\exists$: R) peuvent être remplacées respectivement par (DCLL-$\forall'$: L) et (DCLL-$\exists'$: R). Dans les autres cas, on peut descendre les règles (DCLL-$\cong$: L) et (DCLL-$\cong$: R) en dessous de la règle précédente. Nous présentons ici uniquement la preuve de la règle (DCLL-Cut). Les autres cas sont similaires.

Dans le cas de (DCLL-Cut), on a :

$$\frac{\dfrac{\Gamma \to B' \quad B' \cong B}{\Gamma \to B} \qquad \dfrac{B'', \Gamma' \to D \quad B'' \cong B}{B, \Gamma' \to D}}{\Gamma, \Gamma' \to D}$$

On peut supposer, sans perte de généralité, que $\Gamma \to B'$ et $B'', \Gamma' \to D$ sont dérivés uniquement en utilisant des règles de DMLL'. Puisque B' et B'' sont en forme normale, ils sont identiques. On peut donc remplacer ce qui précède par :

$$\frac{\Gamma \to B' \quad B', \Gamma' \to D}{\Gamma, \Gamma' \to D}$$

$\square$

Théorème 1.1 (Élimination des coupures). *Si $\Gamma \to A$ est dérivable dans* DMLL *alors il est aussi dérivable dans* DMLL *sans utiliser la règle (*DCLL*-Cut).*

Preuve. D'après le lemme 1.1, il suffit de démontrer le théorème pour DMLL'. Le résultat découle du fait que l'on peut soit éliminer, soit faire remonter la règle (DCLL-Cut) quand elle est appliquée sous d'autres règles. Nous ne traiterons pas des cas similaires à ceux de la logique linéaire *classique*. La permutation de la règle (DMLL-Cut) et de la règle suivante doit donc être démontrée quand (DCLL-Cut) est utilisée sous une règle du type (DCLL'-$\forall$: L), (DCLL'-$\exists$: R), (DMLL-In : x) ou (DMLL-Out : x).

- Cas (DCLL-$\forall'$: L) : On a tout d'abord la dérivation suivante :

$$\dfrac{\dfrac{\dfrac{\cdots}{\Gamma \to A[y/x]}}{\dfrac{\cdots}{\Gamma \to \forall x.A}} \qquad \dfrac{\dfrac{\cdots}{B, \Gamma' \to D \quad A[t/x] \cong B}}{\forall x.A, \Gamma' \to D}}{\Gamma, \Gamma' \to D}$$

qui peut être remplacée par :

$$\dfrac{\dfrac{\cdots}{\Gamma \to B} \qquad \dfrac{\cdots}{B, \Gamma' \to \forall x.A}}{\Gamma, \Gamma' \to D}$$

où la dérivation $\Gamma \to B$ est obtenue à partir de la dérivation de $\Gamma \to A[y/x]$.

- Cas (DCLL-$\exists'$: R) : Cette preuve se fait de la même façon que la précédente.

- Cas (DMLL-In : L) : On a tout d'abord la dérivation suivante :

$$\dfrac{\dfrac{\dfrac{\cdots}{\Gamma \to [l_1][l_2]A}}{\dfrac{\cdots}{\Gamma \to [l_1]\{l_2\}A}} \qquad \dfrac{\dfrac{\cdots}{[l_1][l_2]A, \Gamma \to D}}{[l_1]\{l_2\}A, \Gamma' \to D}}{\Gamma, \Gamma' \to D}$$

qui peut être remplacée par :

$$\dfrac{\dfrac{\cdots}{\Gamma \to [l_1][l_2]A} \qquad \dfrac{\cdots}{[l_1][l_2]A, \Gamma' \to D}}{\Gamma, \Gamma' \to D}$$

- Cas (DMLL-In : R) : Identique à (DMLL-In : L)

- Cas (DMLL-Out : L) : Identique à (DMLL-In : L)

- Cas (DMLL-Out : R) : Identique à (DMLL-In : L)

$\square$

1.3.2 Un modèle sémantique

Le modèle sémantique qui est présenté ici correspond à la partie propositionnelle du système. Le modèle utilisé est une adaptation de celui qui a été utilisé pour DCLL.

Le modèle est basé sur l'algèbre des ressources présentée ci-dessous. Notons R l'ensemble des ressources. Un élément r de R définit quelle ressource est disponible et à quel emplacement elle se trouve. R doit nécessairement contenir la ressource vide 1. Les opérations associées à cet ensemble sont les suivantes : $*$ pour la composition, $\sqcup$ pour l'alternative et ! pour la copie. $r1*r2$ indique que l'on doit consommer les deux ressources en même temps. $r1 \sqcup r2$ définit une ressource où l'on peut choisir entre $r1$ et $r2$. $!r1$ indique que l'on possède un nombre illimité de copies de la ressource $r1$. Le fait qu'une ressource $r1$ puisse en remplacer une autre $r2$ (exemple : on peut utiliser $r1 \sqcup r2$ à la place de $r1$) se note $r1 \succcurlyeq r2$. On peut donc définir cet opérateur comme étant un opérateur de type $R \times R \to bool$.

Définition 1.2 (Algèbre des ressources). *Une structure* $\mathcal{R} = \langle R, *, \sqcup, !, 1, \succcurlyeq \rangle$ *est une* algèbre des ressources *si les opérateurs* $*$, $\sqcup$ ($\in R \times R \to R$), $!(\in R \to R)$, $\succcurlyeq(\in R \times R \to bool)$ *et la constante 1 satisfont les propriétés suivantes :*

1. $\langle R, *, 1\rangle$ *est un monoïde commutatif ;*
2. $\sqcup$ *est commutatif et associatif ;*
3. $\succcurlyeq$ *est un pré-ordre ;*
4. $*$, $\sqcup$ *et* ! *sont monotones par rapport à* $\succcurlyeq$ *;*
5. $\forall r, r' \in R.(r \sqcup r' \succcurlyeq r)$ *;*
6. $\forall r \in R.((r \succcurlyeq 1) \wedge (!r \succcurlyeq r*!r))$ *;*
7. $\forall r \in R.(!r \succcurlyeq !!r)$ *;*
8. $\forall r \in R.(!r1*!r2 \succcurlyeq !(r1 * r2))$.

Nous devons maintenant faire le lien entre le fragment propositionnel de DMLL et cette algèbre. Nous noterons cette restriction pDMLL.

Définition 1.3. *L'ensemble des formules de* pDMLL *est donné par la grammaire suivante :*
$A ::= 1 \mid a \mid A_1 \& A_2 \mid A_1 \multimap A_2 \mid A_1 \otimes A_2 \mid !A \mid [l]A \mid \{l\}A \mid \langle l\rangle A$
où a *est une constante de l'ensemble des propositions Prop.*

Il s'agit maintenant de définir le lien entre les ressources et les constantes propositionnelles. DCLL [65] lie une constante propositionnelle et un emplacement à une ressource. Nous gardons le même lien en adaptant la définition d'emplacement à celle qui nous intéresse maintenant. C'est à dire que l'emplacement d'une ressource est défini par une liste de noms d'emplacements qui définit son adresse relative.

Définition 1.4 (Structure des Ressources). *Une paire* $\langle \mathcal{R}, I\rangle$ *est une structure de ressources si* $\mathcal{R}$ *est une algèbre de ressources et si* $I \in Prop \times Loc \to R$, *où Loc est l'ensemble des adresses relatives.*

La relation de satisfaction $I, r, (l_1, \ldots, l_n) \models A$ définie ci-dessous correspond à *"A est obtenue à l'adresse* $(l_1, \ldots, l_n)$ *en utilisant la ressource r".*

Définition 1.5. $\models$ *est une relation définie par induction sur la structure de la formule par les règles suivantes :*
- $I, r, (l_1, \ldots, l_n) \models 1$ *ssi* $r \succcurlyeq 1$.
- $I, r, (l_1, \ldots, l_n) \models a$ *ssi* $r \succcurlyeq I(a, (l_1, \ldots, l_n))$.
- $I, r, (l_1, \ldots, l_n) \models A_1 \& A_2$ *ssi* $I, r, (l_1, \ldots, l_n) \models A_1$ *et* $I, r, (l_1, \ldots, l_n) \models A_2$.
- $I, r, (l_1, \ldots, l_n) \models A_1 \multimap A_2$ *ssi* $I, r', (l_1, \ldots, l_n) \models A_1$ *implique que* $I, r*r', (l_1, \ldots, l_n) \models A_2$ *pour tout* $r' \in R$
- $I, r, (l_1, \ldots, l_n) \models A_1 \otimes A_2$ *ssi il existe* $r1, r2 \in R$ *tel que :* $r \succcurlyeq r1*r2$, $I, r_1, (l_1, \ldots, l_n) \models A_1$ *et* $I, r_2, (l_1, \ldots, l_n) \models A_2$.

- $I, r, (l_1, \ldots, l_n) \models !A$ *ssi il existe* $r' \in R$ *tel que si* $r \succcurlyeq !r'$ *et* $I, r', (l_1, \ldots, l_n) \models A$.
- $I, r, (l_1, \ldots, l_n) \models \{l_{n+1}\}A$ *ssi* $I, r, (l_1, \ldots, l_n) \models [l_{n+1}]A$.
- $I, r, (l_1, \ldots, l_n) \models [l_{n+1}]A$ *ssi* $I, r, (l_1, \ldots, l_n, l_{n+1}) \models A$.
- $I, r, (l_1, \ldots, l_{n-1}, l_n) \models [l_1][\ldots][l_n]\langle l_n\rangle A$ *ssi* $I, r, (l_1, \ldots, l_{n-1}) \models A$.

Définition 1.6 (Validité - Tautologie). *Une formule* A *est* valide *dans une structure de ressource* $\mathcal{M} = \langle \mathcal{R}, I\rangle$*, si* $I, 1, l \models A$ *pour tout* $l \in Loc$*. Nous noterons cette validité* $\mathcal{M} \models A$.

Une formule A *est une* pDMLL*-tautologie, notée* $\models A$ *si* $\mathcal{M} \models A$ *pour toute structure de ressource* $\mathcal{M}$.

La définition de la validité est étendue aux séquents par les équivalences suivantes : $\mathcal{M} \models \Gamma \to A \Leftrightarrow \mathcal{M} \models (\otimes\Gamma) \multimap A$ et $\models \Gamma \to A \Leftrightarrow \models (\otimes\Gamma) \multimap A$.

1.3.3 Correction du modèle

Nous montrons que les règles d'inférence de pDMLL sont correctes par rapport au modèle sémantique, c'est à dire que chaque formule prouvable avec pDMLL est une pDMLL-tautologie.

Pour prouver ce résultat, nous commençons par prouver que la congruence $\cong$ préserve la validité. Cette preuve est similaire à celle développée pour DCLL [65] pour un système sémantique du même type. Comme notre système a une relation de congruence plus restreinte et que l'ajout des adresses relatives ne change rien de fondamental à la preuve, on peut donc reprendre le schéma de la preuve pour DCLL.

Lemme 1.2. Supposons que $A \cong B$ Alors $I, r, (l_1, \ldots, l_n) \models A$ ssi $I, r, (l_1, \ldots, l_n) \models B$ pour tout I, r et pour toute séquence d'emplacements $(l_1, \ldots, l_n)$.

Preuve. Par induction sur $A \cong B$ avec une analyse par cas en fonction de la dernière règle utilisée. □

Nous avons aussi besoin de quelques résultats supplémentaires notamment sur la commutativité et l'associativité de $\otimes$.

Lemme 1.3.
1. $I, r, (l_1, \ldots, l_n) \models A \otimes 1$ ssi $I, r, (l_1, \ldots, l_n) \models A$;
2. $I, r, (l_1, \ldots, l_n) \models A \otimes B$ ssi $I, r, (l_1, \ldots, l_n) \models B \otimes A$;
3. $I, r, (l_1, \ldots, l_n) \models A \otimes (B \otimes C)$ ssi $I, r, (l_1, \ldots, l_n) \models (A \otimes B) \otimes C$.

Preuve. Il s'agit prendre la définition de $\models$ et d'utiliser le fait que $\langle R, *, 1\rangle$ soit un monoïde commutatif. On a donc les résultats suivants :

1. $I, r, (l_1, \ldots, l_n) \models A \otimes 1 \Leftrightarrow I, r1, (l_1, \ldots, l_n) \models A$ et $I, r2, (l_1, \ldots, l_n) \models 1$ avec $r \succcurlyeq r1 * r2$. Ici on peut prendre $r1 = r$ et $r2 = 1$ ce qui donne le résultat souhaité.
2. $I, r, (l_1, \ldots, l_n) \models A \otimes B \Leftrightarrow I, r1, (l_1, \ldots, l_n) \models A$ et $I, r2, (l_1, \ldots, l_n) \models B$ avec $r \succcurlyeq r1 * r2$. Grâce à la commutativité de $\langle R, *, 1\rangle$, on peut conclure.
3. $I, r, (l_1, \ldots, l_n) \models A \otimes (B \otimes C) \Leftrightarrow I, r1, (l_1, \ldots, l_n) \models A$, $I, r2, (l_1, \ldots, l_n) \models B$ et $I, r3, (l_1, \ldots, l_n) \models C$ avec $r \succcurlyeq r1 * r'$ et $r' \succcurlyeq r2 * r3$ et on conclut avec l'associativité $\langle R, *, 1\rangle$ et le fait que $\succcurlyeq$ soit un pré-ordre.

□

Il est également important de montrer que la relation d'ordre entre les ressources préserve la validité de I :

Lemme 1.4. Si $I, r', (l_1, \ldots, l_n) \models A$ et $r \succcurlyeq r'$ alors $I, r, (l_1, \ldots, l_n) \models A$

Preuve. Par induction sur la structure de A. □

On peut finalement prouver le théorème suivant :

Théorème 1.2 (Correction). *Si* $\Gamma \vdash_{DMLL} D$*, alors* $\models \Gamma \rightarrow D$.

Preuve. Il faut montrer que pour chaque règle, si les prémisses sont valides, la conclusion est elle aussi valide. Les règles (DMLL-$\otimes$:L), (DMLL-1 :R) (DMLL-In) vérifient cette condition directement d'après définition de $\models$. Pour les règles (DMLL-$\cong$:L) et (DMLL-$\cong$:R) on peut le prouver directement à partir du lemme 1.2. De même pour les règles (DMLL-Exch) et (DMLL-1 : L) qui sont des conséquences directes du lemme 1.3. Il reste à démontrer les autres cas :

- Cas (DMLL-Id) : Trivial, par définition
  $I, 1, (l_1, \ldots, l_n) \models A \multimap A \Leftrightarrow \forall r.(I, 1, (l_1, \ldots, l_n) \models A \Rightarrow I, r, (l_1, \ldots, l_n) \models A)$.
- Cas (DMLL-Cut) : On suppose que (1) $I, 1, (l_1, \ldots, l_n) \models (\otimes\Gamma) \multimap B$,
  (2) $I, 1, (l_1, \ldots, l_n) \models B \otimes (\otimes\Gamma') \multimap D$ et (3) $I, r, (l_1, \ldots, l_n) \models \otimes(\Gamma, \Gamma')$.
  On doit alors prouver que $I, r, (l_1, \ldots, l_n) \models D$. D'après le lemme 1.3 et (3), il existe r_1, r_2 tel que (4) $I, r_1, (l_1, \ldots, l_n) \models \otimes\Gamma$, (5) $I, r_2, (l_1, \ldots, l_n) \models \otimes\Gamma'$ et (6) $r \succcurlyeq r_1 * r_2$.
  A l'aide de (1) et de (4), on obtient : $I, r_1, (l_1, \ldots, l_n) \models B$. Grâce à ce résultat et en utilisant les résultats, (2), (5) et (6) et le lemme 1.4, on a enfin : $I, r, (l_1, \ldots, l_n) \models D$.
- Cas (DMLL-$\otimes$: R) : On suppose (1) $I, 1, (l_1, \ldots, l_n) \models (\otimes\Gamma_1) \multimap A$,
  (2) $I, 1, (l_1, \ldots, l_n) \models (\otimes\Gamma_2) \multimap B$ et (3) $I, r, (l_1, \ldots, l_n) \models \otimes(\Gamma_1, \Gamma_2)$. On doit alors prouver que $I, r, (l_1, \ldots, l_n) \models A \otimes B$. Le raisonnement est le même que précédemment. D'après le lemme 1.3 et (3), il existe r_1, r_2 tel que (4) $I, r_1, (l_1, \ldots, l_n) \models \otimes\Gamma_1$, (5) $I, r_2, (l_1, \ldots, l_n) \models \otimes\Gamma_2$ et (6) $r \succcurlyeq r_1 * r_2$.
  A l'aide de (1), (2), (4)et (5) on a alors $I, r_1, (l_1, \ldots, l_n) \models A$ et $I, r_2, (l_1, \ldots, l_n) \models B$. Avec les résultats précédents et en utilisant (2), (4), (5) et le lemme 1.4, on obtient $I, r, (l_1, \ldots, l_n) \models A \otimes B$.
- Cas (DMLL-& : L) : On suppose que (1) $I, 1, (l_1, \ldots, l_n) \models (A_i \otimes (\otimes\Gamma)) \multimap D$ pour $i = 1$ ou 2 et (2) $I, r, (l_1, \ldots, l_n) \models (A_1 \& A_2) \otimes (\otimes\Gamma)$.
  On doit alors prouver que $I, r, (l_1, \ldots, l_n) \models D$. Par (2), il existe r_1, r_2 tel que (3) $I, r_1, (l_1, \ldots, l_n) \models (A_1 \& A_2)$, (4) $I, r_2, (l_1, \ldots, l_n) \models \otimes\Gamma$ et (5) $r \succcurlyeq r_1 * r_2$.
  Par (3), on a $I, r_1, (l_1, \ldots, l_n) \models A_j$ pour $j = 1$ ou 2. En utilisant ceci, (4) et (5) on obtient $I, r, (l_1, \ldots, l_n) \models A_j \otimes (\otimes\Gamma)$. Par (1) on a $I, r, (l_1, \ldots, l_n) \models D$.
- Cas (DMLL-& : R) : Soit (1) $I, 1, (l_1, \ldots, l_n) \models \otimes\Gamma \multimap A$, (2) $I, 1, (l_1, \ldots, l_n) \models \otimes\Gamma \multimap B$ et (3) $I, r, (l_1, \ldots, l_n) \models \otimes\Gamma$. On a alors $I, r, (l_1, \ldots, l_n) \models A$ et $I, r, (l_1, \ldots, l_n) \models B$ d'où l'on obtient par définition $I, r, (l_1, \ldots, l_n) \models A \& B$.
- Cas (DMLL-$\multimap$: L) : Soit (1) $I, 1, (l_1, \ldots, l_n) \models (B \otimes (\otimes\Gamma_1)) \multimap D$,
  (2) $I, 1, (l_1, \ldots, l_n) \models (\otimes\Gamma_2) \multimap A$ et (3) $I, r, (l_1, \ldots, l_n) \models (A \multimap B) \otimes (\Gamma_1, \Gamma_2)$.
  On doit montrer que $I, r, (l_1, \ldots, l_n) \models D$. D'après le lemme 1.3 et (3), il existe r_1, r_2 et r_3 tel que (4) $I, r_1, (l_1, \ldots, l_n) \models A \multimap B$, (5) $I, r_2, (l_1, \ldots, l_n) \models \otimes\Gamma_1$, (6) $I, r_3, (l_1, \ldots, l_n) \models \otimes\Gamma_2$ et (7) $r \succcurlyeq r_1 * r_2 * r_3$. Par (2) et (6), on a $I, r_3, (l_1, \ldots, l_n) \models A$. A partir de ce résultat et de (6), on a $I, r_1 * r_3, (l_1, \ldots, l_n) \models A$. En utilisant (5) et (7), on obtient alors $I, r, (l_1, \ldots, l_n) \models B \otimes (\otimes\Gamma_1)$; on obtient alors le résultat voulu par (1).

- Cas (DMLL-$\multimap$: R) : Soit (1) $I, 1, (l_1, \ldots, l_n) \models (A \otimes (\otimes\Gamma)) \multimap D$ et (2) $I, r, (l_1, \ldots, l_n) \models \otimes\Gamma$. On doit montrer que $I, r, (l_1, \ldots, l_n) \models A \multimap B$. Soit r' tel que $I, r, (l_1, \ldots, l_n) \models \otimes\Gamma$. On a alors $I, r * r', (l_1, \ldots, l_n) \models (A \otimes (\otimes\Gamma))$. De (1) on obtient alors $I, r' * r, (l_1, \ldots, l_n) \models B$. Par la définition de $\models$ on a bien $I, r' * r, (l_1, \ldots, l_n) \models A \multimap B$.
- Cas (DMLL-Weak) : Soit (1) $I, 1, (l_1, \ldots, l_n) \models (\otimes\Gamma) \multimap D$ et (2) $I, r, (l_1, \ldots, l_n) \models !A \otimes (\otimes\Gamma)$. On doit alors montrer que $I, r, (l_1, \ldots, l_n) \models D$. Par la définition de $\models$, il existe r_1 et r_2 tel que (3) $I, r_1, (l_1, \ldots, l_n) \models !A$, (4) $I, r_2, (l_1, \ldots, l_n) \models (\otimes\Gamma)$ et (5) $r \succcurlyeq r_1 * r_2$.
A l'aide de (1) et de (4), on a alors $I, r_2, (l_1, \ldots, l_n) \models D$. De plus par (3), il existe r_3 tel que $r_1 \succcurlyeq !r_3$. On a donc : $r \succcurlyeq r_1 * r_2 \succcurlyeq !r_3 * r_2 \succcurlyeq 1 * r_2 = r_2 *$. Finalement, par le lemme 1.4, on a bien $I, r, (l_1, \ldots, l_n) \models D$.
- Cas (DMLL-Con) : Soit (1) $I, 1, (l_1, \ldots, l_n) \models !A \otimes (!A \otimes (\otimes\Gamma)) \multimap D$ et
(2) $I, r, (l_1, \ldots, l_n) \models !A \otimes (\otimes\Gamma)$. On doit alors montrer que $I, r, (l_1, \ldots, l_n) \models D$. Par (2) et la définition de $\models$, il existe r_1 et r_2 tel que : (3) $I, r_1, (l_1, \ldots, l_n) \models A$, (4) $I, r_2, (l_1, \ldots, l_n) \models (\otimes\Gamma)$, (5) $r \succcurlyeq r_1 * r_2$.
D'après les règles sur !, on a aussi $!r \succcurlyeq !!r \succcurlyeq !r*!!r \succcurlyeq !r*!r \succcurlyeq !r*!r*!!r \succcurlyeq !r*!r$. On a aussi (6) $r \succcurlyeq !r_1*r_2 \succcurlyeq (!r_1*!r_1)*r_2 = !r_1*(!r_1*r_2)$. Enfin, de (3), (4) et (6) on obtient $I, r, (l_1, \ldots, l_n) \models !A \otimes !A \otimes (\otimes\Gamma)$ et la conclusion se déduit directement de (1).
- Cas (DMLL-! :L) : Soit (1) $I, 1, (l_1, \ldots, l_n) \models A \otimes (\otimes\Gamma) \multimap D$ et (2) $I, r, (l_1, \ldots, l_n) \models !A \otimes (\otimes\Gamma)$. On doit alors montrer que $I, r, (l_1, \ldots, l_n) \models D$. Par (2), il existe r_1 et r_2 tel que
(3) $I, r_1, (l_1, \ldots, l_n) \models !A$, (4) $I, r_2, (l_1, \ldots, l_n) \models \otimes\Gamma$ et (5) $r \succcurlyeq r_1 * r_2$.
Par (3), il existe aussi $r_1' \in R$ tel que (6) $I, r_1', (l_1, \ldots, l_n) \models A$ et (7) $r_1 \succcurlyeq r_1'$.
Donc on a (8) $r \succcurlyeq r_1' * r_2$. Par (4), (6) et (8) on obtient $I, 1, (l_1, \ldots, l_n) \models A \otimes (\otimes\Gamma)$ et la conclusion se déduit directement de (1).
- Cas (DMLL-! :R) : On suppose (1) $I, 1, (l_1, \ldots, l_n) \models \otimes(!\Gamma) \multimap A$ et
(2) $I, r, (l_1, \ldots, l_n) \models \otimes(!\Gamma)$. On doit alors montrer que $I, r, (l_1, \ldots, l_n) \models !A$. On pose (sans perte de généralité) $\Gamma = B_1, \ldots, B_n$. Par (2), il existe $r_1, \ldots, r_n$ tel que (3) $I, r_i, (l_1, \ldots, l_n) \models !B_i$ pour tout $i \in 1, \ldots, n$ et (4) $r \succcurlyeq r_1 * \ldots * r_n$.
Par (3), il existe $r_i' \in R$ tel que $r_i \succcurlyeq !r_i'$ et $I, r_i', (l_1, \ldots, l_n) \models B_i$ pour chaque i. On a alors $I, !r_1' * \ldots *!r_n', (l_1, \ldots, l_n) \models B_1 \otimes \ldots \otimes B_n$. En utilisant ceci et (1), on a $I, !r_1' * \ldots *!r_n', (l_1, \ldots, l_n) \models A$. La preuve de $I, r, (l_1, \ldots, l_n) \models !A$ découle alors de $r \succcurlyeq r_1 * \ldots * r_n \succcurlyeq !r_1' * \ldots *!r_n' \succcurlyeq !!r_1' * \ldots *!!r_n' \succcurlyeq !(!r_1' * \ldots *!r_n')$.
- Cas (DMLL-Out :L) : On suppose (1) $I, 1, (l_1, \ldots, l_n) \models [l']A \otimes (\otimes\Gamma) \multimap D$ et
(2) $I, r, (l_1, \ldots, l_n) \models [l'][l'']\langle l'' \rangle A \otimes (\otimes\Gamma)$. On doit alors montrer que $I, r, (l_1, \ldots, l_n) \models D$. Par (2) on sait qu'il existe r_1 et r_2 tel que
(3) $I, r_1, (l_1, \ldots, l_n) \models [l'][l'']\langle l'' \rangle A$, (4) $I, r_2, (l_1, \ldots, l_n) \models (\otimes\Gamma)$, (5) $r \succcurlyeq r_1 * r_2$.
Par définition, de (3), on obtient (6) $I, r_1, (l_1, \ldots, l_n) \models [l']A$.
De (1), (2) et (6) en utilisant aussi la relation (3), on obtient le résultat.
- Cas (DMLL-Out :R) : Soit (1) $I, 1, (l_1, \ldots, l_n) \models \otimes\Gamma \multimap [l']A$ et
(2) $I, r, (l_1, \ldots, l_n) \models \otimes\Gamma$. On doit alors montrer que $I, r, (l_1, \ldots, l_n) \models [l'][l'']\langle l'' \rangle A$. De (1) et (2), on a : $I, r, (l_1, \ldots, l_n) \models \otimes[l']A$ qui est le résultat cherché.
- Cas (DMLL-Loc1 :L) : On suppose (1) $I, 1, (l_1, \ldots, l_n) \models 1 \otimes (\otimes\Gamma) \multimap D$ et
(2) $I, r, (l_1, \ldots, l_n) \models [l_1'] \ldots [l_m']1 \otimes (\otimes\Gamma)$. On doit alors montrer que $I, r, (l_1, \ldots, l_n) \models D$. De (2), on a $I, 1 * r, (l_1, \ldots, l_n) \models [l_1'] \ldots [l_m']1 \otimes (\otimes\Gamma)$. Par définition de $\otimes$ et de 1, on peut alors montrer que $I, 1, (l_1, \ldots, l_n) \models [l_1'] \ldots [l_m']1$ et (3) $I, r, (l_1, \ldots, l_n) \models (\otimes\Gamma)$.
De (3), on peut prouver facilement : $I, r * 1, (l_1, \ldots, l_n) \models 1 \otimes (\otimes\Gamma)$, soit $I, r, (l_1, \ldots, l_n) \models 1 \otimes (\otimes\Gamma)$. On finit alors avec (1).

- Cas (DMLL-Loc1 :R) : On suppose (1) $I, 1, (l_1, \ldots, l_n) \models \otimes\Gamma \multimap 1$ et (2) $I, r, (l_1, \ldots, l_n) \models \otimes\Gamma$. On doit alors montrer que $I, r, (l_1, \ldots, l_n) \models [l'_1] \ldots [l'_m]1$. Or, comme $r \succcurlyeq 1$ on a $I, r, (l_1, \ldots, l_n, l'_1, \ldots, l'_m) \models 1$. D'où on obtient immédiatement la conclusion.

$\square$

1.3.4 Complétude du modèle

Nous devons montrer que chaque séquent dérivable avec pDMLL est une pDMLL tautologie. Pour ce faire, nous construirons un modèle canonique tel qu'un séquent prouvable avec pDMLL soit valide avec le modèle canonique. Une ressource dans le modèle canonique est représentée par une formule de pDMLL.

Lemme 1.5 (Algèbre de ressources canoniques). Soit $\mathcal{F}$ un ensemble de formules de pDMLL et $\backsim$ la plus petite relation de congruence close par les règles de commutativité et d'associativité de $\otimes$, & et la règle $1 \otimes A \backsim A$. alors $\mathcal{R}_c = \langle \mathcal{F}/\backsim, \otimes_\backsim, \&_\backsim, !_\backsim, [1]_\backsim, \vdash_\backsim \rangle$ est une algèbre de ressources où $[A]_\backsim$ est la classe d'équivalence de A induite par $\backsim$. La définition de $\otimes$, $\&_\backsim$, $!_\backsim$, $[\mathbf{1}]_\backsim$, $\vdash_\backsim$ est la suivante :

$$\begin{array}{ll} [A]_\backsim \otimes_\backsim [B]_\backsim = [A \otimes B]_\backsim & [A]_\backsim \&_\backsim [B]_\backsim = [A \& B]_\backsim \\ !_\backsim [A]_\backsim = [!A]_\backsim & [A]_\backsim \vdash_\backsim [B]_\backsim = A \vdash_{DMLL} B \end{array}$$

Preuve. Comme $\backsim$ est une congruence, on a $(A \backsim A') \wedge (B \backsim B') \Rightarrow (A \otimes B \backsim A' \otimes B') \wedge (A \& B \backsim A' \& B') \wedge (!A \backsim !A')$

De plus, les opérations $\otimes_\backsim$, $!_\backsim$ sont bien définies. On peut vérifier chaque condition d'une pDMLL-algèbre.

- Cas 1,2,4,5 : Évident.
- Cas 3 : La réflexivité de $\vdash_{DMLL}$ découle de la règle (DMLL-Id) et la transitivité de (DMLL-Cut).
- Cas 6 : Par (DMLL-1 : R) et (DMLL-Weak), $!A \vdash_{DMLL} 1$ pour tout A. Les séquents $!A \rightarrow A$ et $!A \rightarrow !A$ sont dérivables de (DMLL-Id) et (DMLL-! :L). De plus on a $!A \otimes !A \rightarrow A \otimes !A$ par (DMLL-$\otimes$: R), d'où on déduit $!A \otimes A \otimes !A$ par (DMLL-Con).
- Cas 7 : Immédiat par (DMLL-Id) et (DMLL-! : L).
- Cas 8 : $!A_1 \otimes !A_2 \vdash_{DMLL} !(A_1 \otimes A_2)$ découle de la relation suivante :

$$\dfrac{\dfrac{\dfrac{\dfrac{A_1 \rightarrow A_1}{!A_1 \rightarrow A_1}\ (\text{DMLL-! : L}) \quad \dfrac{A_2 \rightarrow A_2}{!A_2 \rightarrow A_2}\ (\text{DMLL-! : L})}{!A_1, !A_2 \rightarrow A_1 \otimes A_2}\ (\text{DMLL-}\otimes\text{ : R})}{!A_1, !A_2 \rightarrow !(A_1 \otimes A_2)}\ (\text{DMLL-! : R})}{!A_1 \otimes !A_2 \rightarrow !(A_1 \otimes A_2)}\ (\text{DMLL-}\otimes\text{ : L})$$

$\square$

À partir de maintenant on écrit $\langle \mathcal{F}, \otimes, \&, !, \mathbf{1}, \vdash_{DMLL} \rangle$ pour $\langle \mathcal{F}/\backsim, \otimes_\backsim, \&_\backsim, !_\backsim, [\mathbf{1}]_\backsim, \vdash_\backsim \rangle$ et on écrit A pour $[A]_\backsim$. On suppose toutefois que les formules sont toujours données en fonction de $\backsim$.

Définition 1.7 (Modèle canonique). *Un modèle canonique $\mathcal{M}_c$ est une paire $\langle \mathcal{R}_c, I_c \rangle$ où $I_c \in Prop \times Loc \rightarrow \mathcal{F}$ est défini par $I_c(a, l) = [l]a$.*

Le lemme suivant indique que la validité d'une formule dans le modèle canonique correspond à sa prouvabilité dans le calcul des séquents de DMLL.

Lemme 1.6. $I_c, A, (l_1, \ldots, l_n) \models B$ si et seulement si $A \vdash_{DMLL} [l_1] \ldots [l_n]B$.

Preuve. La preuve se fait par induction structurelle sur B.

- Cas 1 : Par définition de $\models$ et (DMLL-*Loc*1 : R), $I_c, A, (l_1, \ldots, l_n) \models 1 \Leftrightarrow A \vdash_{DMLL} 1 \Leftrightarrow A \vdash_{DMLL} [l_1] \ldots [l_n]1$.

- Cas $a \in Prop$: Par définition de $\models$, $I_c, A, (l_1, \ldots, l_n) \models B$ ce qui équivaut à $A \vdash_{DMLL} [l_1] \ldots [l_n]a$.

- Cas $B_1 \& B_2$: Par définition de $\models$, on a $I_c, A, (l_1, \ldots, l_n) \models B$ ce qui équivaut à $(I_c, A, (l_1, \ldots, l_n) \models B_1) \wedge (I_c, A, (l_1, \ldots, l_n) \models B_2)$

  Ce qui donne par hypothèse d'induction :

$$\begin{array}{ll} & I_c, A, (l_1, \ldots, l_n) \models B \\ \text{équivalent à} & (I_c, A, (l_1, \ldots, l_n) \models B_1) \wedge (I_c, A, (l_1, \ldots, l_n) \models B_2) \\ \text{équivalent à} & (A \vdash_{DMLL} [l_1] \ldots [l_n]B_1) \wedge (A \vdash_{DMLL} [l_1] \ldots [l_n]B_2) \\ \text{équivalent à} & A \vdash_{DMLL} [l_1] \ldots [l_n](B_1 \& B_2). \end{array}$$

- Cas $B_1 \multimap B_2$: Par définition, on a :

$$\begin{array}{ll} & I_c, A, (l_1, \ldots, l_n) \models B \\ \text{équivalent à} & \forall C.(I_c, C, (l_1, \ldots, l_n) \models B_1 \Rightarrow I_c, A \otimes C, (l_1, \ldots, l_n) \models B_2) \\ \text{équivalent à} & \forall C.(C \vdash_{DMLL} [l_1] \ldots [l_n]B_1 \Rightarrow A \otimes C \vdash_{DMLL} [l_1] \ldots [l_n]B_2) \end{array}$$

  Supposons que l'on ait $\forall C.(C \vdash_{DMLL} [l_1] \ldots [l_n]B_1 \Rightarrow A \otimes C \vdash_{DMLL} [l_1] \ldots [l_n]B_2)$.
  Alors, $A \otimes [l_1] \ldots [l_n]B_1 \vdash_{DMLL} [l_1] \ldots [l_n]B_2$, d'où on obtient (par (DMLL-$\multimap$: R)) : $A \vdash_{DMLL} [l_1] \ldots [l_n]B_1 \multimap [l_1] \ldots [l_n]B_2$.
  Par (DMLL-$\cong$: R), on a $A \vdash_{DMLL} [l_1] \ldots [l_n](B_1 \multimap B_2)$.
  Supposons maintenant que l'on ait $A \vdash_{DMLL} [l_1] \ldots [l_n](B_1 \multimap B_2)$ et $C \vdash_{DMLL} [l_1] \ldots [l_n]B_1$.
  Par (DMLL-$\otimes$: L), on a $A \otimes C \vdash_{DMLL} ([l_1] \ldots [l_n](B_1 \multimap B_2)) \otimes [l_1] \ldots [l_n]B_1$.
  Par (DMLL-*Id*), (DMLL-$\multimap$: L) et (DMLL-$\otimes$: R), on a aussi $([l_1] \ldots [l_n](B_1 \multimap B_2)) \otimes [l_1] \ldots [l_n]B_1 \vdash_{DMLL} [l_1] \ldots [l_n]B_2$.
  En utilisant (DMLL-Cut), on obtient $A \otimes C \vdash_{DMLL} [l_1] \ldots [l_n]B_2$.
  De plus, on a $\forall C.(C \vdash_{DMLL} [l_1] \ldots [l_n]B_1 \Rightarrow A \otimes C \vdash_{DMLL} [l_1] \ldots [l_n]B_2)$ ce qui équivaut à $A \vdash_{DMLL} [l_1] \ldots [l_n]B$, ce qui permet de conclure.

- Cas $B_1 \otimes B_2$: Par définition de $\models$, on a $I_c, A, (l_1, \ldots, l_n) \models B$, ce qui est équivalent à $\exists C_1, C_2.(A \vdash_{DMLL} C_1 \otimes C_2 \wedge I_c, C_1, (l_1, \ldots, l_n) \models B_1 \wedge I_c, C_2, (l_1, \ldots, l_n) \models B_2)$.

  Et les hypothèses d'induction, ceci équivaut à :
  $\exists C_1, C_2.(A \vdash_{DMLL} C_1 \otimes C_2 \wedge C_1 \vdash_{DMLL} [l_1] \ldots [l_n]B_1$
  $\wedge C_2 \vdash_{DMLL} [l_1] \ldots [l_n]B_2)$

  Supposons que l'on ait $\exists C_1, C_2.(A \vdash_{DMLL} C_1 \otimes C_2 \wedge C_1 \vdash_{DMLL} [l_1] \ldots [l_n]B_1 \wedge C_2 \vdash_{DMLL} [l_1] \ldots [l_n]B_2)$. Alors par (DMLL-Cut), (DMLL-$\otimes$: R) et (DMLL-$\cong$: R), on a : $A \vdash_{DMLL} [l_1] \ldots [l_n](B_1 \otimes B_2)$. Supposons que l'on ait $A \vdash_{DMLL} [l_1] \ldots [l_n](B_1 \otimes B_2)$. Il est alors évident que $\exists C_1, C_2.(A \vdash_{DMLL} C_1 \otimes C_2 \wedge C_1 \vdash_{DMLL} [l_1] \ldots [l_n]B_1 \wedge C_2 \vdash_{DMLL} [l_1] \ldots [l_n]B_2)$ soit vérifié (il suffit de prendre $C_1 \equiv [l_1] \ldots [l_n]B_1$ et $C_2 \equiv [l_1] \ldots [l_n]B_2$).

- Cas $!B_1$: Par définition de $\models$, on a $I_c, A, (l_1, \ldots, l_n) \models B$ ce qui équivaut à $\exists C.(A \vdash_{DMLL} !C \wedge (I_c, C, (l_1, \ldots, l_n) \models B_1))$

  Et les hypothèses d'induction donne alors : $\exists C.(A \vdash_{DMLL} !C \wedge (C \vdash_{DMLL} [l_1] \ldots [l_n]B_1)$ ce qui est équivalent à $A \vdash_{DMLL} [l_1] \ldots [l_n]!B_1$.

- Cas $[l']B_1$: Par définition de $\models$, on a $I_c, A, (l_1, \ldots, l_n) \models B$, soit $I_c, A, (l_1, \ldots, l_n, l') \models B_1$

  Ce qui donne, par hypothèse d'induction, $A \vdash_{DMLL} [l_1] \ldots [l_n][l']B_1$ ce qui est équivalent à $A \vdash_{DMLL} [l_1] \ldots [l_n]B$.

- Cas $[l']\{l'\}B_1$: Par définition de $\models$ et par hypothèse d'induction, $I_c, A, (l_1, \ldots, l_n) \models B$ ce qui équivaut à $I_c, A, (l_1, \ldots, l_n) \models [l']B_1$. Le résultat s'obtient alors par le cas précédent.
- Cas $[l'][l'']\langle l''\rangle B_1$: Par définition de $\models$, $I_c, A, (l_1, \ldots, l_n) \models B$ équivaut à $I_c, A, (l_1, \ldots, l_n) \models [l']B_1$ et donc à $I_c, A, (l_1, \ldots, l_n, l') \models B_1$, ce qui est équivalent à $A \vdash_{DMLL} [l_1] \ldots [l_n][l']B_1$. Par application de (DMLL-Out : R) on a aussi $A \vdash_{DMLL} [l_1] \ldots [l_n][l'][l'']\langle l''\rangle B_1$. Ce qui correspond à $A \vdash_{DMLL} [l_1] \ldots [l_n]B$.

$\square$

Lemme 1.7. Si $[l_1] \ldots [l_n]\Gamma \vdash_{DMLL} [l_1] \ldots [l_n]A$ pour toute liste $(l_1, \ldots, l_n)$, alors $\Gamma \vdash_{DMLL} A$.

Preuve. Soit $(l_1, \ldots, l_n)$ une liste d'emplacements qui ne soient pas dans Γ, A. Ainsi la dérivation du séquent $\Gamma \rightarrow A$ est obtenue à partir de $[l_1] \ldots [l_n]\Gamma \rightarrow [l_1] \ldots [l_n]A$ en retirant toutes les occurrences de $[l_1]$, ..., $[l_n]$. $\square$

La complétude découle directement des lemmes 1.6 et 1.7.

Théorème 1.3 (Complétude). *Si* $\models \Gamma \rightarrow A$ *alors* $\Gamma \vdash_{DMLL} A$.

Preuve. Supposons que $\models \Gamma \rightarrow A$. Par définition, on a $I_c, 1, (l_1, \ldots, l_n) \models (\otimes\Gamma) \multimap A$ pour tout $(l_1, \ldots, l_n) \in Loc$. Par le lemme 1.6, on a $1 \vdash_{DMLL} [l_1] \ldots [l_n]((\otimes\Gamma) \multimap A)$ pour tout $(l_1, \ldots, l_n)$. Par le lemme 1.7, on obtient $1 \vdash_{DMLL} (\otimes\Gamma) \multimap A$, i.e., $\Gamma \vdash_{DMLL} A$. $\square$

1.3.5 Relations avec DCLL

On peut se demander si l'utilisation des adresses relatives et l'ajout des modalités de déplacement change l'expressivité de la logique. On montre ici que ce n'est pas le cas en proposant une équivalence entre DCLL et DMLL. Pour cela, puisque la sémantique des opérateurs est la même dans les deux logiques, il suffit de montrer que l'on peut remplacer les modalités d'emplacement et de déplacement d'un système logique par celle de l'autre système et garder les mêmes déplacements.

- De DCLL vers DMLL

Puisque DMLL rajoute des informations relatives aux déplacements aux formules, il semble possible de pouvoir transformer une formule de DCLL en une formule de DMLL.

Pour passer d'une logique à l'autre, il suffit de faire précéder tous les changements d'emplacements par la modalité permettant de sortir de l'emplacement actuel. L'emplacement initial indiquant le positionnement initial du processus ne change donc pas. Par contre, tous ceux situés dans le processus sont modifiés et remplacés par une sortie de l'emplacement dans lequel ils se trouvent et par une entrée dans l'emplacement destination. Par exemple, la configuration initiale (en DCLL) :

$$[l_1]n(m) \otimes \forall x.(nx \multimap [l_2]P) \vdash [l_2]P$$

devient

$$[l_1]n(m) \otimes \forall x.(nx \multimap \langle l_1\rangle\{l_2\}P) \vdash [l_2]P \text{ (en DMLL).}$$

On voit bien dans cet exemple que le déplacement du processus P de l_1 vers l_2 est rendu explicite dans la formule de DCLL. La différence entre les deux formules illustre bien la différence entre les logiques : dans DMLL, tous les déplacements sont explicites.

Proposition 1.1.
Pour tout séquent $\Gamma \vdash_{DCLL} \Delta$ de DCLL*, il existe un séquent $\Gamma' \vdash_{DMLL} \Delta'$ tel que $\Gamma' \vdash_{DMLL} \Delta'$ est prouvable si et seulement si $\Gamma \vdash_{DCLL} \Delta$ est prouvable.*

Preuve. Il suffit de remplacer toute modalité correspondant à un déplacement dans Γ et Δ par les modalités de sortie et d'entrée dans un emplacement correspondantes pour obtenir la transformation dans DMLL. $\square$

- De DMLL vers DCLL

Le codage de DMLL vers DCLL est également possible si l'on considère des formules où les quantificateurs ne portent pas sur les emplacements. Pour passer de notre nouveau système logique à DCLL, on utilise le fait que l'emplacement où sera libérée chaque ressource (sa destination) est connu à l'avance. En effet, tous les noms d'emplacements utilisés lors des déplacements sont définis dès la configuration initiale, on peut donc déterminer sans ambiguïté la destination des ressources. Si l'on détermine maintenant une adresse absolue pour chaque chemin de notre configuration, on peut donc remplacer toute suite de déplacements directement par l'adresse absolue équivalente. Prenons un exemple pour illustrer la transformation :

$$[l][l_1](\langle l_1\rangle\{l_2\}c \otimes \langle l_1\rangle\{l_2\}c(m)), [l][l_2]\forall x.(x \multimap \forall y.(x(y) \multimap 1))$$

Ici, il n'y a pas de renommage à faire puisque les noms d'emplacement (l_1, l_2 et l) sont distincts. Il reste donc à remplacer les indications relatives aux emplacements et de les remplacer par l'emplacement destination correspondant pour obtenir la formule équivalent en DCLL :

$$[l_1]([l_2]c \otimes [l_2]c(m)), [l_2]\forall x.(x \multimap \forall y.(x(y) \multimap 1)).$$

On voit malgré tout que toutes les informations relatives à l'emplacement l ont été supprimés lors de la transformation. Ceci illustre une autre différence cruciale entre les deux systèmes : l'absence de hiérarchie dans DCLL ne permet pas d'indiquer le cheminement que doit suivre les messages.

Proposition 1.2.
Pour tout séquent $\Gamma \vdash_{DMLL} \Delta$ de DMLL*, il existe un séquent $\Gamma' \vdash_{DCLL} \Delta'$ tel que $\Gamma' \vdash_{DCLL} \Delta'$ est prouvable si et seulement si $\Gamma \vdash_{DMLL} \Delta$ est prouvable.*

Preuve. Il suffit de remplacer toute modalité d'entrée et de sortie d'emplacement dans Γ et Δ par les modalités d'emplacement correspondantes pour obtenir la transformation dans DCLL. $\square$

Les deux résultats de cette section montre que les différents systèmes de modalité ne modifient en rien la prouvabilité des séquents. On a cependant montré sur les exemples de transformations que les formules DMLL contenaient des informations supplémentaires sur la configuration de l'espace et les déplacements. De plus, le passage de DCLL à DMLL montre la difficulté que l'on peut avoir à différencier les modalités correspondant à des emplacements et celles correspondant à des déplacements. Les modalités spécifiques (aux déplacements) de DMLL permettent donc d'obtenir plus d'informations lors de la recherche de preuves et de différencier plus clairement mobilité et distribution d'un point de vue sémantique. On va montrer comment exploiter les informations supplémentaires relatives à la mobilité en se fondant sur l'utilisation de DMLL pour la spécification et la preuve de protocoles.

1.4 Applications aux protocoles

Pour illustrer l'intérêt des modalités de déplacement de DMLL et la façon dont elles peuvent être utilisées pour extraire des informations de la recherche de preuves, on va considérer DMLL pour la modélisation des protocoles de sécurité, notamment les protocoles d'authentification [37].

Il existe des travaux qui utilisent des fragments de la logique linéaire pour représenter les protocoles [20, 30, 32] et pour analyser les attaques possibles. Ces problèmes sont des exemples typiques où les informations sont distribuées et mobiles. Il est donc intéressant d'étudier l'impact que peut avoir l'introduction explicite de ces concepts dans la logique pour faciliter la modélisation ou la lecture des attaques.

1.4.1 Représentation des protocoles

Pour coder les protocoles d'authentification sous forme de formules de DMLL, on doit formuler le comportement des agents (qui reçoivent et envoient les messages) mais on doit également représenter les messages et leur éventuelle méthode d'encodage. On détaille ici cette formulation.

- Les agents

Un agent est une entité impliquée dans les transactions d'un protocole. Dans notre représentation, chaque agent possède un emplacement propre. L'emplacement contient les formules nécessaires pour que l'agent puisse tenir son rôle, c'est à dire qu'il contient l'état actuel de la (ou des) protocoles d'authentification qu'il doit gérer et la connaissance de cet agent (les clés qu'il connaît par exemple). Un intrus est un type particulier d'agent puisqu'il peut chercher à composer des messages autres que ceux dictés par le protocole. Son comportement sera détaillé plus loin dans ce chapitre.

- Les messages

Un message basique x est défini par le terme $msg(x)$. Un message composé est de la forme $msg(x,y)$ où x et y sont des messages. Notre définition des messages ne contient aucune information relative à l'emplacement où se trouve le message et où il doit être envoyé. Ces indications sont gérées uniquement par les modalités d'emplacement. Les termes définis ci-dessus sont relatifs à des messages non codés. Un message codé a quant à lui une forme qui dépend du type de codage utilisé, comme expliqué ci-dessous :

$msg(pubK(a,X))$:	le message X est codé avec la clé publique de l'agent a ;
$msg(privK(a,X))$:	le message X est codé avec la clé privée de l'agent a ;
$msg(symk(k,X))$:	le message X est codé avec la clé symétrique k.
$msg(xor(X,Y))$:	les messages X et Y sont codés grâce à la méthode de codage xor.

- Les nonces

Les nonces sont des nombres uniques qui peuvent être créés par le protocole. Ils correspondent donc à de nouvelles constantes créées par l'opérateur $\exists$.

- Envoyer et recevoir des messages

L'envoi de message est indiqué explicitement par les modalités de déplacement. Par exemple, un message X qu'un agent a situé en l_1 souhaite envoyer à un agent situé en l_2 est représenté par la formule $\langle l_1 \rangle send(l_2, l_1, X)$ que l'on situe en l_1. La modalité de cette formule indique uniquement que l'agent a envoie un message X et que ce message est envoyé depuis l'emplacement l_1. Le relais est alors assuré à l'emplacement situé au-dessus de a pour transmettre l'information à la bonne cible : $\forall x, y, t.(send(x, y, t) \multimap \{x\} from(y, t))$.

Le prédicat $from(y, t)$ indique alors que le message t provient de l'emplacement y. L'indication de la provenance est utile pour savoir *où* l'on doit répondre si une réponse est attendue.

Il aurait été possible de procéder directement à l'envoi du message grâce à la formule : $\langle l_1 \rangle \{l_2\} from(l_1, X)$. Cette solution nous semble moins intéressante. La solution choisie est en effet plus proche de ce qui se passe au niveau d'un réseau : le message est envoyé depuis l_1 et il est alors *routé* vers l_2. Avec une telle solution, on peut facilement exprimer le fait qu'un message soit intercepté, comme nous le verrons par la suite.

La réception des messages est alors gérée par l'opérateur $\multimap$. La formule $from(l, X) \multimap B$ indique qu'un agent attend un message X de l'emplacement l avant de passer à l'étape B. Un exemple de l'envoi et de la réception de messages est donné dans la section 1.4.5

1.4.2 L'intrus

Un intrus est un agent malicieux qui ne respecte pas les règles habituelles du protocole et qui cherche à usurper l'identité d'un agent ou à récupérer une information qui devrait être transmise de manière sécurisée entre deux agents. L'intrus que l'on modélise ici est celui proposé par le modèle de Dolev-Yao [69]. On donne le même nom z à cet agent et à l'emplacement où il se trouve. Ainsi, les actions de l'intrus seront celles qui impliquent des processus situés à l'emplacement z ou qui redirigent un message vers cet emplacement.

- Redirection

On exprime ici la possibilité, pour un intrus, de récupérer un message qu'il n'est normalement pas autorisé à récupérer (donc de détourner ce message). Pour ce faire, le message doit être redirigé vers l'intrus après avoir quitté son émetteur et avant d'arriver à son destinataire. La redirection doit donc se faire à l'emplacement situé au dessus des agents. On a alors :

$$\forall x, y, t.(send(x, y, t) \multimap \{z\} from(y, t))$$

Un point important à souligner est que l'intrus a besoin, pour rediriger les messages, d'ajouter des formules à un emplacement correspondant au routeur. On retrouve dans la spécification la même contrainte que celle qui se pose réellement à un intrus cherchant à dérouter des messages.

- Décomposition et recomposition de messages

Une fois un message à l'emplacement z, il peut être décomposé et recomposé pour former d'autres messages. La décomposition est gérée par les formules suivantes :

$$\begin{array}{l} \forall x, y.decomp(from(x, y)) \multimap decomp(x) \\ \forall x, y.decomp(msg(x, y)) \multimap (decomp(x) \otimes decomp(y)) \\ \forall x.decomp(msg(x)) \multimap !msg(x) \end{array}$$

À la troisième ligne de la formule ci-dessus, l'utilisation de $!msg(x)$ permet d'indiquer que tout message que possède l'intrus peut être utilisé en de multiples copies par l'intrus. L'intrus peut également recomposer les messages :

$$\forall x, y.(msg(x) \otimes msg(y)) \multimap msg(msg(x), msg(y))$$

Un intrus peut également usurper l'identité des autres agents. Il y a deux façons de gérer cette usurpation. La première est d'autoriser l'intrus à usurper uniquement l'identité d'agents dont il a connaissance pour avoir déjà récupéré leurs noms. La deuxième consiste à penser que l'intrus connaît le nom de tous les agents. Cette dernière approche nécessite que l'on ajoute à un intrus la connaissance du nom d'un agent dès que cet agent joue un rôle dans le protocole. Ainsi, si un agent a situé en l_1 doit exécuter le processus P, on indique à l'intrus sa présence en même temps que l'on définit son rôle de la façon suivante :

$$\forall a.(\underline{[z](msg(a) \otimes ag(a))} \otimes [l_1]P)$$

- Codage et décodage des messages

S'il possède la clé adéquate, un intrus peut coder et décoder les messages qu'il reçoit ou qu'il veut envoyer. Ces actions correspondent aux formules ci-dessous :

$$\text{– } Codage : \begin{array}{l} [z]!\forall x, y.(ag(x) \otimes msg(y)) \multimap msg(pubK(x, msg(y))) \\ [z]!\forall x, y.(msg(x) \otimes msg(y)) \multimap msg(xor(msg(x), msg(y))) \\ [z]!\forall k, y.(msg(k) \otimes msg(y)) \multimap msg(symk(k, msg(y))) \\ [z]!\forall x.msg(x) \multimap msg(privK(z, msg(x))) \end{array}$$

$$\text{– } Décodage : \begin{array}{l} [z]!\forall x.msg(pubK(z, x)) \multimap decomp(x) \\ [z]!\forall x, y.(msg(xor(x, y)) \otimes y) \multimap (decomp(x) \otimes decomp(y)) \\ [z]!\forall x, y.(msg(xor(x, y)) \otimes x) \multimap (decomp(x) \otimes decomp(y)) \\ [z]!\forall x, y.(msg(symk(k, y)) \otimes k) \multimap decomp(y) \\ [z]!\forall x, y.(msg(privK(x, y)) \otimes ag(x)) \multimap decomp(y) \end{array}$$

On retrouve les contraintes sur le codage que l'on avait expliqué précédemment. En effet, tout le monde peut coder un message avec la clé publique d'un agent, mais la seule clé publique que possède un agent (même l'intrus) est la sienne. Le xor est une méthode de cryptage où deux messages sont cryptés en s'utilisant mutuellement comme clé l'un-l'autre, le résultat du cryptage étant l'application de l'opérateur booléen «XOR» sur la représentation binaire des deux messages. Il est nécessaire de connaître au moins un message pour pouvoir retrouver l'autre. Enfin, les clés symétriques permettent bien à la fois de coder et de décoder un message qu'elles auraient codé. Ces méthodes de cryptage sont celles principalement utilisées dans les protocoles.

- Envoi des messages

Une fois que l'intrus a créé un message, il doit choisir un destinataire et modifier l'emplacement de provenance du message. Il peut choisir n'importe quel agent intervenant dans le protocole, on a donc : $[z]\forall x, y, t.(msg(y) \multimap \langle z\rangle send(x, t, msg(y)))$.

1.4.3 Détecter une attaque

Le principe de la détection d'attaque avec DMLL est de trouver une preuve d'une formule attestant que l'intrus a bien réussi à corrompre le protocole. Son expression dépend du but du protocole. On détermine ici ce but et la configuration de départ du protocole. On s'intéresse ensuite à la recherche de preuves et à ce que peuvent apporter les modalités d'emplacement dans cette recherche.

- Déterminer le but

Comme indiqué, le but d'une intrusion dépend du rôle du protocole. En fait, la formule de but doit correspondre à ce qu'on ne veut pas voir arriver dans un protocole. Considérons un protocole d'identification. On utilise dans ce cadre le prédicat $contacted(y)$ à un emplacement x pour indiquer que l'agent x a réussi à contacter l'agent y. De même, le prédicat $contacted\_by(y)$ situé à l'emplacement x indique que l'agent x a été contacté par l'agent y. Le but du protocole est d'assurer que l'agent x et un agent y se reconnaissent mutuellement, donc que le prédicat $contacted(y)$ satisfait en x et que le prédicat $contacted\_by(x)$ est satisfait en y. Pour l'intrus, le but est de faire croire à un des deux agents qu'il est l'autre. On doit donc obtenir à l'emplacement y le prédicat $contacted\_by(x)$ sans que x ait réellement tenté de contacter y. Avec les notations données précédemment, ceci correspond simplement à la formule suivante :

$$\exists l.[l]contacted\_by(a)$$

Le contexte du but peut toutefois être plus important si d'autres sessions d'identification doivent se dérouler pour que l'attaque puisse avoir lieu.

Si le but est l'échange d'un message X qui doit être secret, le but de l'intrus est alors d'avoir accès à ce message, ce que l'on note :

$$[z]msg(X) \otimes \top$$

L'utilisation de $\top$ indique que l'on ne se soucie pas du reste de la configuration : si z a récupéré le message qui devait être secret ou a réussi à usurper une identité, l'état des autres agents n'est pas important.

1.4.4 Application au protocole de Needham-Schroeder

On a expliqué globalement comment pouvait se coder un intrus et un protocole à l'aide de DMLL. On met ceci en œuvre sur un exemple d'attaque du protocole d'authentification de Needham-Schroeder avec clé publique [74], un exemple classique dans le monde des protocoles. Ce protocole a pour but une authentification mutuelle de deux agents, authentification suite à laquelle on crée une session entre les deux agents pour permettre la circulation de messages. Dans ce protocole, chaque agent x a une clé publique $pubK(x)$ qui peut être obtenue par tout autre agent mais également une clé secrète qui est l'inverse de $pubK(x)$ et noté $privK(x)$. Ainsi, un message m codé avec la clé publique de x ne peut être lu que par celui qui possède la clé privée de x. Un tel message est noté $\{m\}_k$. Le protocole utilise également des *nonces*, des nombres générés aléatoirement pour une exécution unique du protocole. N_x désigne généralement un nonce généré par un agent x. La présentation classique du protocole de Needham-Schroeder donnée en figure 1.4.

$$\begin{array}{lll}
1. & A \rightarrow B : & \{N_a \| A\}_{pubK(B)} \\
2. & B \rightarrow A : & \{N_a \| N_b\}_{pubK(A)} \\
3. & A \rightarrow B : & \{N_b\}_{pubK(B)}
\end{array}$$

FIG. 1.4 – Le protocole de Needham-Schroeder

Alice et Bob veulent communiquer en étant tous deux sûrs de l'identité de leur interlocuteur. Alice démarre donc le protocole en envoyant à Bob un nombre aléatoire (N_a) à Bob ainsi qu'un message indiquant qu'elle est bien Alice, le tout codé avec la clé publique de Bob. Bob décrypte ce message et le renvoie accompagné d'un nombre aléatoire qu'il a généré (N_b), en codant le tout avec la clé de Alice. Puisque Bob a réussi à décrypter le message, il a donc assuré à Alice qu'il était bien celui qu'elle pensait. Enfin, Alice décrypte à son tour le nombre de Bob et le renvoie après l'avoir codé avec la clé publique de Bob. Bob est ainsi assuré de l'identité d'Alice puisqu'elle arrive à décoder les messages qui lui sont destinés. Les deux agents s'étant assuré de l'identité de l'autre, la communication peut commencer. Contrairement aux apparences, cette authentification n'est pas sûre. Lowe a proposé en effet une attaque de ce protocole [69], attaque que l'on détaillera plus tard. Détaillons maintenant la transcription de ce protocole avec DMLL. Dans ce qui suit, les variables de type N_a, N_b, N_x correspondent respectivement à des nonces générées par l'agent a, b ou x. De même, afin de simplifier la lecture, le nom de variable l indique que cette variable correspondra à un emplacement.

Commençons par la partie relative à Alice, située en un emplacement arbitrairement nommé l_1. Dans le codage proposé, le prédicat $contact(x,l)$ indique que Alice veut contacter l'agent x situé à l'emplacement l. Quand elle reçoit cet ordre, Alice crée un nonce N_a ($\exists N_a$) et envoie immédiatement le message correspondant à la première étape du protocole à l'emplacement l, pour cela, elle envoie hors de son emplacement le prédicat : $send(l, l_1, msg(pubK(x, msg(msg(N_a), msg(a)))))$. Elle attend alors la réponse de l'agent x ($from(l, msg(pubK(a, msg(msg(N_a), msg(N_x))))))$ pour pouvoir lui renvoyer le message correspondant à la troisième étape du protocole (le prédicat $send(l, l_1, msg(pubK(x, msg(N_x)))))$. N'attendant alors plus de réponse, elle pourra indiquer qu'elle a contacté x ($contacted(x)$).

- *Alice (agent a à l'emplacement l_1) :*

$$\begin{aligned} Alice(a) \stackrel{def}{=} \; & [l_1]\forall x, l.(contact(x,l) \multimap \\ & \exists N_a.(\langle l_1\rangle send(l, l_1, msg(pubK(x, msg(msg(N_a), msg(a)))))\otimes \\ & \forall N_x.(from(y, msg(pubK(a, msg(msg(N_a), msg(N_x))))) \multimap \\ & (\langle l_1\rangle send(l, l_1, msg(pubK(x, msg(N_x)))) \otimes contacted(x))))) \end{aligned}$$

De manière analogue, la formule correspondant au rôle de Bob débute par l'attente du premier message du protocole d'authentification. Dès qu'il reçoit ce message, Bob crée un nonce (N_b) et envoie ce message à l'emplacement l d'où le premier message lui est parvenu. Il attend ensuite la réponse à son message pour pouvoir indiqué qu'il a bien été contacté par l'agent x ($contacted\_by(x)$).

- *Bob (agent b à l'emplacement l_2) :*

$$\begin{aligned} Bob(b) \stackrel{def}{=} \; & [l_2]\forall N_x, x, l.(from(l, msg(pubK(x, msg(msg(N_x), msg(x))))) \multimap \\ & \exists N_b.(\langle l_2\rangle send(l, l_2, msg(pubK(x, msg(msg(N_x), msg(N_b)))))\otimes \\ & (from(l, msg(pubK(b, msg(N_b)))) \multimap contacted\_by(x)))) \end{aligned}$$

- Configuration initiale

En plus du codage des deux agents, la configuration initiale doit comprendre le code correspondant à l'intrus. Ce codage reprend toutes les formules définissant le comportement de l'intrus qui ont été introduites précédemment pour décomposer, rediriger et recomposer des message et usurper des identités. Ce qui donne la configuration ci-dessous :

$$\Gamma_0 \equiv \begin{array}{l}
[z]!\forall x, y.decomp(from(x,y)) \multimap decomp(x))\otimes \\
[z]!\forall x, y.(decomp(msg(x,y)) \multimap (decomp(x) \otimes decomp(y)))\otimes \\
[z]!\forall x.(decomp(msg(x)) \multimap !msg(x))\otimes \\
[z]!\forall x, y.((msg(x) \otimes msg(y)) \multimap msg(x,y))\otimes \\
[z]!\forall x.(ag(x) \multimap msg(x))\otimes \\
[z]!\forall x, y.((msg(y) \otimes ag(x)) \multimap \langle z\rangle\{x\}msg(y))\otimes \\
[z]!\forall x, y.((ag(x) \otimes msg(y)) \multimap msg(pubK(x, msg(y))))\otimes \\
[z]!\forall x, y.((msg(x) \otimes msg(y)) \multimap msg(xor(msg(x), msg(y))))\otimes \\
[z]!\forall k, y.((msg(k) \otimes msg(y)) \multimap msg(symk(k, msg(y))))\otimes \\
[z]!\forall x.(msg(x) \multimap msg(privK(z, msg(x))))\otimes \\
[z]!\forall x, y.((ag(x) \otimes msg(y)) \multimap msg(pubK(x, msg(y))))\otimes \\
[z]!\forall x, y.((msg(x) \otimes msg(y)) \multimap msg(xor(msg(x), msg(y))))\otimes \\
[z]!\forall k, y.((msg(k) \otimes msg(y)) \multimap msg(symk(k, msg(y))))\otimes \\
[z]!\forall x.(msg(x) \multimap msg(privK(z, msg(x))))
\end{array}$$

On ajoute à ces formules fixant le routage des messages au niveau du réseau. En plus du routage classique, l'intrus peut dérouter les messages circulant sur le réseau pour se les approprier. Pour cela, on doit indiquer à la racine que tout message devant aller vers un des acteurs peut être redirigé vers l'intrus : $!\forall x, y.(send(y, msg(x)) \multimap \{y\}msg(x))$.

- L'attaque du protocole

L'attaque, proposée dans [69], consiste pour un intrus à utiliser les informations qu'un agent A lui fournirait lors d'une précédente transaction pour se faire passer pour usurper l'identité de A auprès d'un autre agent B. Le déroulement de l'attaque est expliqué en figure 1.5.

$$\begin{array}{lll}
1. & A \rightarrow Z : & \{N_A \| A\}_{pubK(Z)} \\
2. & Z(A) \rightarrow B : & \{N_A \| A\}_{pubK(B)} \\
3. & B \rightarrow Z : & \{N_A \| N_B\}_{pubK(A)} \\
4. & Z \rightarrow A : & \{N_A \| N_B\}_{pubK(A)} \\
5. & A \rightarrow Z : & \{N_B\}_{pubK(Z)} \\
6. & Z(A) \rightarrow B : & \{N_B\}_{pubK(B)}
\end{array}$$

Fig. 1.5 – Une attaque du protocole de Needham-Schroeder

Dans cette figure, $Z(A)$ indique que Z envoie un message en se faisant passer pour A. Z se sert donc de sa session avec A pour faire décoder des messages que B lui envoie. Des morceaux de preuves correspondant à cette attaque sont données dans la section suivante pour illustrer l'intérêt des emplacements et des déplacements dans la recherche de preuves. La configuration initiale doit donc contenir l'intrus, un agent jouant le rôle d'Alice et un autre jouant celui de Bob, Alice doit contacter l'intrus et Bob doit se faire contacter par l'intrus qui usurpe l'identité d'Alice. On a donc comme formule initiale : $\Gamma_0, [l_1]Alice(a), [l_2]Alice(b), [l_1]contact(z,z)$. La formule à prouver à partir de cette configuration initiale montre que le protocole s'est bien déroulé pour Alice et que Bob a l'impression de communiquer avec Alice alors qu'il communique avec l'intrus : $[l_1]contected(z) \otimes [l_2]contacted\_by(a)$. La preuve complète de cette attaque est donnée dans l'annexe A.

1.4.5 Preuves et analyse

En annexe A, on présente la preuve de l'attaque du protocole de Needham-Schroeder sous forme de séquent. La construction automatique de cette preuve n'est pas abordée dans cette thèse. Cependant, il semble que l'ajout de modalités d'emplacement et de déplacements puissent être intégrés à divers méthodes de preuves comme les preuves uniformes [53, 56, 78], les preuves avec focus [7]. De manière plus générale, des méthodes comme celles reposant sur des systèmes de ressources ou de labels [33, 57], des tableaux sémantiques [70] ou des connexions [44] pour la logique linéaires peuvent également être considérés. La forme des formules définies dans le cadre de la vérification de protocole suit celle proposée dans [34, 73] pour la programmation logique linéaire. Il est donc naturel de penser que les travaux réalisés dans ce cadre puissent également être adaptés à la recherche de preuves dans DMLL.

Dans ces méthodes, il semble en effet que la gestion des emplacements puisse se faire sans modifier le processus de recherche de preuves. Le calcul des séquents proposé dans ce chapitre en est l'illustration concrète : la suppression des modalités et de leurs règles de gestion permet de retrouver le calcul des séquents « classique » de la logique linéaire. On peut même penser que la gestion des emplacements puisse aider la recherche de preuves, en permettant par exemple de restreindre ponctuellement les formules considérées à celles situées à un emplacement donné.

Ici, notre étude ne se porte donc pas sur la construction d'une preuve mais sur l'analyse des informations contenues dans cette preuve. Si on la compare avec celles des travaux utilisant la logique linéaire [20, 30, 32], l'ajout des modalités d'emplacement et de déplacement permet de clarifier lors de la recherche de preuves les déplacements des messages et les connaissances de chacun des acteurs. Ainsi, à une étape donnée de la recherche de preuves, la connaissance de l'intrus est exactement l'ensemble de ressources présentes en z. De même, le rôle de l'intrus peut être isolé clairement en regardant les messages qui arrivent en z et ceux qui en sortent. Nous détaillons ces différents aspects sur quelques exemples de preuves donnés ci-dessous.

- Envoi de message

On présente en figure 1.6 le schéma de preuve correspondant à l'envoi du premier message du protocole de Needham-Schroeder. Cette preuve doit être lue de bas en haut, Γ représente l'ensemble des formules n'intervenant pas ici et P représente le but. On suppose donc qu'Alice (à l'emplacement l_1) veuille envoyer à Bob (en l_2) le message $\{Na\|A\}$. Cet envoi commence par la sortie de l'emplacement a du prédicat $send(l_2, l_1, msg(msg(Na), msg(a)))$. Le message arrive alors à la racine où il doit être routé par la formule $!\forall x, y, t.send(x, y, t) \multimap \{x\}from(y, t)$. On instancie donc x avec le prédicat correspondant au message, puis on utilise ensuite la règle ($\multimap$: L) et (In : L) pour assurer le déplacement du message en b.

Dans cette preuve, on remarque que le message doit nécessairement passer par la racine pour pouvoir atteindre son but. L'ajout des adresses relatives permet donc bien de vérifier le cheminement des ressources. Lors de son passage à la racine, le message aurait pu être intercepté par l'intrus (à l'emplacement z) si l'on avait utilisé la formule $\forall x, y.send(x, y) \multimap \{z\}y$ pour router le message. Il est donc important que le passage des messages à cet emplacement apparaisse dans la preuve.

- Action et connaissances de l'intrus

Dans le cadre de la vérification de protocole, il est important de pouvoir isoler clairement le rôle joué par l'intrus dans la preuve. L'utilisation des modalités est pour cela très utile car les

$$
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{\dfrac{\vdots}{\Gamma \otimes [l_2]from(l_1, NaA) \vdash P}}{\Gamma \otimes \{l_2\}from(l_1, NaA) \vdash P}\,(\text{In : L})
\quad
\dfrac{}{send(l_2, l_1, NaA) \vdash send(l_2, l_1, NaA)}\,(\text{Id})
}{\Gamma \otimes send(l_2, l_1, NaA) \otimes send(l_2, l_1, NaA) \multimap \{l_2\}from(l_1, NaA) \vdash P}\,(\multimap : \text{L})
}{\Gamma, send(l_2, l_1, NaA), \forall t.send(l_2, l_1, t) \multimap \{l_2\}from(l_1, t) \vdash P}\,(\forall : \text{L})
}{\Gamma, send(l_2, l_1, NaA), \forall y, t.send(l_2, y, t) \multimap \{l_2\}from(y, t) \vdash P}\,(\forall : \text{L})
}{\Gamma, send(l_2, l_1, NaA), \forall y, t.send(x, y, t) \multimap \{x\}from(y, t) \vdash P}\,(\forall : \text{L})
}{\Gamma, send(l_2, l_1, NaA), !\forall x, y, t.send(x, y, t) \multimap \{x\}from(y, t) \vdash P}\,(! : \text{L})
}{\Gamma, [l_1]\langle l_1\rangle send(l_2, l_1, NaA), !\forall x, y, t.send(x, y, t) \multimap \{x\}from(y, t) \vdash P}\,(\text{Out : L})
$$

avec :

$$NaA \overset{def}{=} msg(msg(Na), msg(a))$$

FIG. 1.6 – Preuve de l'envoi de message

$$
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{\vdots}{\Gamma, [z]\langle z\rangle send(l_2, z, m_b(Nb)) \vdash P}
\quad
\dfrac{}{[z]m_b(Nb) \vdash [z]m_b(Nb)}\,(\text{Id : L})
}{\Gamma, [z]m_b(Nb), [z](m_b(Nb) \multimap \langle z\rangle send(l_2, z, m_b(Nb))) \vdash P}\,(\multimap : \text{L})
}{\Gamma, [z]m_b(Nb), [z]\forall t.(msg(t) \multimap \langle z\rangle send(l_2, z, msg(t))) \vdash P}\,(\forall : \text{L})
}{\Gamma, [z]m_b(Nb), [z]\forall y, t.(msg(t) \multimap \langle z\rangle send(l_2, y, msg(t))) \vdash P}\,(\forall : \text{L})
}{\Gamma, [z]m_b(Nb), [z]\forall x, y, t.(msg(t) \multimap \langle z\rangle send(x, y, msg(t))) \vdash P}\,(\forall : \text{L})
\quad
\dfrac{}{[z]msg(Nb) \vdash [z]msg(Nb)}\,(\text{Id})
}{\begin{array}{l}\Gamma, [z]msg(Nb), [z](msg(Nb) \multimap m_b(Nb)),\\ [z]\forall x, y, t.(msg(t) \multimap \langle z\rangle send(x, y, msg(t)))\end{array} \vdash P}\,(\multimap : \text{L})
}{\begin{array}{l}\Gamma, [z]msg(Nb), [z]\forall x.(msg(x) \multimap msg(pubK(b, msg(x)))),\\ [z]\forall x, y, t.(msg(t) \multimap \langle z\rangle send(x, y, msg(t)))\end{array} \vdash P}\,(\forall : \text{L})
$$

avec :

$$m_b(Nb) \overset{def}{=} msg(pubK(b, msg(Nb)))$$

FIG. 1.7 – Preuve d'une composition de message

actions de l'intrus sont clairement identifiées comme celles ayant lieu à l'emplacement où il se trouve.

On a vu dans le cas de l'envoi de message que la redirection des messages par l'intrus menait à l'arrivée d'un message à son emplacement. Globalement, les actions dans lesquelles est impliqué l'intrus sont celles impliquant des formules situées ou entrant en z. Dans la figure 1.7, on montre comment l'intrus compose le message correspondant à l'étape 6 de l'attaque du protocole de Needham-Schroeder (celle de la figure 1.5). La preuve peut se décomposer en deux parties, la première (les deux premières règles d'inférence) correspond à la création du message à envoyer, la deuxième partie correspond au choix du destinataire et de l'identité de l'envoyeur. La création du message correspond au codage par la clé publique de b du message Nb, on utilise pour cela la ressource $msg(Nb)$ et une formule correspondant au codage par la clé publique de b d'un message : $[z]\forall x.(msg(x) \multimap msg(pubK(b, msg(x))))$.

On remarque ici que toutes les formules impliquées sont situées en z. Ainsi, les actions de l'intrus sont clairement identifiées comme étant celles impliquant des formules étant en z. De même les formules en z représentent les connaissances de l'intrus. Dans cette preuve, on remarque que toute les formules impliquées sont situées en z. On voit ici que l'utilisation des emplacements permet de déterminer sans ambiguïté l'acteur de l'action.

On a vu que les actions de l'intrus correspondent exactement aux morceaux de preuve impliquant l'emplacement z. Par conséquent, les connaissances de l'intrus à un instant donné du protocoles sont exactement les formules présentes en z. L'utilisation des emplacements est un moyen efficace pour différencier de manière efficace les acteurs.

Ainsi, on remarque que l'apport des emplacements et des déplacements permet de mieux différencier le rôle de chaque agent dans la preuve et de mieux suivre le cheminement des ressources. Ces informations permettent d'isoler facilement le comportement d'un acteur particulier et sont donc très intéressantes dans le cadre de la vérification de protocole dont il est question ici.

En l'état actuel des travaux, ces informations sont uniquement utilisées pour mieux comprendre une preuve donnée. Il est toutefois raisonnable de penser qu'elles peuvent être utilisées pour guider et améliorer la recherche de preuves. Ces possibilités dépassent les limites d'étude que l'on s'est fixé pour cette thèse et pourra faire l'objet de travaux futurs.

DMLL a été développée dans l'optique d'une utilisation dans le cadre du paradigme *proof-search-as-computation*. Le but était donc d'obtenir une logique dont la recherche de preuve est la plus proche de l'exécution et où la preuve contient un maximum d'informations sur la mobilité des ressources. Le rôle du modèle de ressources à la Kripke servant de sémantique à DMLL n'est donc pas central pour ce type d'utilisation. De plus, ce modèle n'a pas servi de point de départ à la construction de DMLL, il n'en est que la conséquence.

Dans la suite de ce document, on souhaite replacer le modèle au centre de notre étude. L'idée est de fournir un modèle assez général dont les instances peuvent servir à représenter des problèmes divers mettant en avant la distribution. Il semble donc naturel de commencer par proposer un modèle permettant de représenter les ressources et leur distribution de façon fine et d'y associer une logique.

Chapitre 2

Arbres de ressources et BI-Loc

On a vu dans le chapitre 1 avec la logique DMLL que l'on pouvait introduire la notion explicite d'emplacement et de déplacement au niveau d'une logique. L'utilisation qui est faite de DMLL vise à établir le lien entre l'exécution des systèmes et la recherche de preuves. On souhaite maintenant définir une logique prenant en compte distribution et mobilité pour un autre usage, celui de la spécification et de la vérification de propriétés des systèmes distribués. Pour ce type d'utilisation, le modèle associé à la logique joue un rôle central. Le but est ici de proposer un modèle général faisant cohabiter à la fois une gestion fine des ressources et une distribution spatiale. Ce choix se justifie par la nécessité de distinguer la structure de l'espace (les emplacements) des données qui s'y trouve (les ressources).

Dans ce chapitre, on commence par un tour d'horizon des modèles permettant l'expression de la distribution des ressources et de l'espace, et présentant les possibilités qui nous sont offertes pour gérer des modèles dynamiques, c'est à dire où les ressources peuvent être mobiles. On explique alors pourquoi le modèle de ressources partiel [72] de la logique BI [76, 79] nous semble centrale et constitue un point de départ important pour une extension où les emplacements sont explicites. On définit ensuite une nouvelle structure nommée *arbre de ressources* puis une logique associée, BI-Loc, où la distribution des ressources et de l'espace est explicite. On présente ensuite des commandes de transformation pour ce modèle et une sémantique pour ces commandes sous forme de triplets de Hoare.

2.1 Ressources, distribution et mobilité

La multiplication des web-services et des applications distribuées ont rendu centrale la notion d'emplacement dans la spécification et la vérification des systèmes informatiques. Les travaux de Cardelli et al. sur le *calcul des ambients* [28] et sur la logique associé à ce calcul [29] furent les premiers à proposer un modèle où cette notion apparaît explicitement et joue un rôle central. Le calcul des ambients est un calcul distribué et mobile permettant de modéliser des systèmes informatiques mobiles, comme par exemple la modélisation de pare-feu [28]. Un ambient est un emplacement clos et mobile désigné par un label (non nécessairement unique) et pouvant contenir d'autres ambients ou des processus. La possibilité pour un ambient de contenir d'autres ambients aboutit à une structure arborescente. L'exécution d'une configuration donnée consiste à effectuer des déplacements des ambients ou des interactions entre des processus contenues au sein d'un même ambient. Cardelli et Gordon ont également proposé la *logique des ambients* [29] dont le but est de spécifier des propriétés sur les ambients. Il s'agit de pouvoir décrire certaines propriétés des ambients (par exemple : tout ambient vérifiant A contient un ambient vérifiant B, tous les ambients contenues dans un ambient C vérifient D). Cette logique mêle des opérateurs de la logique classique ($\wedge, \vee, \rightarrow$) dont le but est d'exprimer la conjonction, la disjonction et la conséquence entre des propriétés, des opérateurs multiplicatifs ($|, \triangleright$) qui permettent d'exprimer la composition et la conséquence d'une composition, on a ensuite des modalités spatiales, dont le but est d'affirmer qu'il existe un emplacement vérifiant une propriété ou qu'un ambient donné existe ($n[\cdot]$) et enfin des modalités temporelles permettant de raisonner sur l'évolution d'une configuration donnée dans le temps. Les travaux de [58, 59] ont permis de définir les liens entre cette logique et la logique du premier ordre et de définir l'expressivité de cette logique. La logique des ambients permet d'exprimer de manière intuitive des propriétés telles que, par exemple, la possibilité pour un agent de passer un pare-feu nécessitant la connaissance d'une clé donnée.

La représentation arborescente des ambients présente des similitudes avec la structure des données semi-structurées (SSD) comme le format XML. Pour cette raison, de nombreux travaux se sont focalisés sur le fragment statique du modèle où les ambients ne contiennent pas de processus et où la logique n'utilise pas les opérateurs temporels. Les logiques ainsi obtenues sont regroupées sous le nom de *logiques d'arbres* [23, 24, 26]. Dans le modèle d'arbres ainsi obtenu, les seules données disponibles pour un nœud sont celles correspondant à son label. C'est ainsi que pour gérer les identificateurs et les pointeurs dans les arbres correspondants à des SSD, le modèle a du être étendu pour gérer des labels plus complexes [24, 26]. Des travaux récents [24] ont également montré que la gestion de la composition dans les ambients n'était pas suffisante pour exprimer la mise à jour dans les arbres. En effet, la composition d'ambients consiste à placer deux ambients côte-à-côte. On ne peut donc pas modifier un nœud interne d'un ambient par composition. Ces travaux ont amené à la proposition du modèle des contextes [24], un contexte étant un arbre dont un nœud donné attend des données. Ces travaux permettent alors de raisonner sur la modification de sous-arbres dans des SSD.

La solution proposée par [24] a pour caractéristique de différencier arbres et contexte. On se retrouve alors avec deux entités différentes pour pouvoir raisonner sur la modification d'arbres, cette particularité étant dû à la gestion de la composition dans les ambients. On peut donc se demander s'il n'y aurait pas d'autres types de composition plus adéquates pour modifier des arbres. De plus, l'utilisation des labels pour représenter les données des SSD a montré ses limites puisqu'elle nécessite une adaptation du modèle dès que la nature des données à représenter évolue. Il serait donc utile de proposer un modèle d'arbre où les données contenues dans les nœuds pourraient être gérées plus finement.

Il apparaît suite aux travaux sur les ambients que les notions de composition des ressources et de séparation des emplacement sont centrales. Ces notions sont également abordées dans le modèle des pointeurs [62, 81]. Un pointeur est un couple nom/valeur, la valeur pouvant désigner un autre pointeur et le nom pouvant être vu comme l'emplacement où se trouve cette valeur. On peut composer plusieurs pointeurs pour obtenir des *tas* de pointeurs. Toutefois, comme un nom ne peut correspondre qu'à un unique couple nom/valeur, certaines compositions ne sont pas définies. Pour cette raison, le modèle des pointeurs est dit *partiel*. La structure des pointeurs est centrale dans la vérification de programme, les pointeurs permettant de représenter naturellement la mémoire ou des variables d'un programme. Les travaux de [62, 81] visent à vérifier les propriétés de programmes manipulant de telles structures. Pour cela, ils proposent des commandes de base pour manipuler les pointeurs et une logique pour spécifier les propriétés des tas de pointeurs. Cette logique dispose d'opérateurs de la logique classique ($\wedge, \vee, \rightarrow$) ou selon les versions de ces mêmes opérateurs mais dans leur version intuitionniste et d'opérateurs linéaires multiplicatifs permettant d'exprimer la composition de tas de pointeurs ($*$) ou l'implication d'une telle composition ($-\!*$). Ces deux derniers opérateurs présentent des similitudes avec les opérateurs ($|$)

et ($\triangleright$) des ambients et de logiques spatiales comme celle de [21, 22]. Outre la spécification des propriétés d'un tas de pointeurs donné, cette logique permet de définir pour chaque commande de manipulation des pointeurs des pré et post-conditions sous forme de triplets de Hoare [60] et de raisonner sur les propriétés du tas avant et après exécution de la commande. Dans la pratique, cela a permis de démontrer la correction de certains algorithmes manipulant des arbres ou des graphes, comme l'algorithme de construction des *fringes* [16], une liste chaînée particulière construite à partir d'un arbre, ou de raisonner sur la vérification de code en Java [77].

Le modèle a récemment été étendu dans le modèle des permissions pour exprimer le contrôle d'accès à des variables [17]. Ce modèle ajoute au couple emplacement/valeur un poids indiquant si l'on a accès en lecture et/ou en écriture, avec pour but de raisonner sur l'accès aux variables de programmes concurrents. Une représentation possible de ce modèle est d'associer à chaque emplacement un couple poids/valeur. On voit là aussi, comme dans le cas des ambients, que la modification du type de données contenu dans les emplacements mène à une modification du modèle. Ce problème montre une fois de plus qu'il serait intéressant de proposer un modèle mêlant emplacements et ressources où le type de données serait très général et pourrait être instancié sans modifier le modèle sous-jacent.

Notre but ici est de mêler une représentation explicite de l'espace et la distribution des ressources afin d'obtenir un modèle général dont les instances permettent de raisonner sur des problèmes plus concrets. On doit pour cela fournir une notion d'emplacement explicite mais également un modèle de ressources assez souple pour que ces instances puissent s'adapter à divers problèmes. Outre les notions centrales de séparation, de partage et de distribution dont il a été question ci-dessus, il nous semble également crucial que le modèle proposé intègre la notion de composition partielle qui permet de déterminer finement les compositions de ressources qui sont possibles ou non à l'intérieur du modèle.

On a vu au chapitre précédent avec les travaux sur DCLL et DMLL qu'il était possible d'étendre les modèles existants pour y inclure des modalités d'emplacements. De plus, les travaux cités ci-dessus sont tous constitués d'un noyau logique mêlant opérateurs de la logique classique (éventuellement intuitionniste) et des opérateurs linéaires permettant d'exprimer la composition, la séparation et le partage des ressources. Or, ces opérateurs sont au cœur de la logique BI (*the Logic of Bunched Implications*) [76, 79]. De plus des travaux récents ont proposé un modèle de ressources partiel pour BI, modèle qui a été prouvé correct et complet pour BI [72] et qui intègre les notions de séparation, de partage et de composition partielle qui ont été introduites initialement pour [62].

Ce modèle de ressources nous semble un point de départ important pour intégrer la notion d'emplacement. On propose donc ici une nouvelle structure, nommée arbres de ressources, qui permet à la fois de prendre en compte explicitement la notion d'emplacement et la représentation fine des ressources dans ces emplacements. On définit ensuite une nouvelle logique, BI-Loc, permettant de raisonner sur cette structure, qui peut être vu comme une extension avec emplacements de la logique BI.

Arrivé à ce point, on étudie la modification des arbres de ressources et les méthodes disponibles pour raisonner sur ces manipulations. Deux axes principaux peuvent être considérés. Le premier, qui est celui choisi dans le chapitre 1 et qui est également celui choisi dans la logique des ambients [29], consiste à intégrer dans la sémantique de la logique une notion de transformation du modèle et de mobilité des ressources. Le second, introduit initialement par Hoare [60] consiste à définir un langage de commandes permettant de manipuler le modèle et d'établir les pré-conditions et post-conditions logiques pour chaque commande. Ainsi, on peut définir la formule que doit initialement vérifier une configuration pour pouvoir vérifier certaines propriétés après exécution du programme. C'est cette approche qui est utilisée pour la vérification de programmes avec pointeurs [62, 75] ou la manipulation d'arbres [24]. Elle a l'avantage de ne pas surcharger la logique en vue de gérer les aspects dynamiques du système et permet donc d'étudier plus facilement les propriétés de la logique. De plus, elle nous semble plus adaptée pour un modèle générique où la nature des commandes de manipulation peut être amenée à varier en fonction du domaine d'étude.

2.2 Les arbres de ressources

L'idée de faire cohabiter au sein d'un même modèle les ressources et la notion de représentation de l'espace n'est pas nouvelle. Les travaux de Kobayashi et al. sur l'ajout de modalité spatiale à la logique linéaire et la proposition de DCLL (Distributed and Concurrent Linear Logic) [65] par exemple montre comment on peut introduire dans un modèle de ressources donné une notion de distribution des ressources en divers emplacements. À l'inverse, les modèles initiaux des logiques d'arbres contiennent peu d'informations [23] et les travaux de Cardelli et al. [26] ou de Calcagno et al. [24] sur les arbres avec pointeurs et sur les données semi-structurées ont montré qu'il était souvent nécessaire de rajouter des informations aux labels des arbres pour améliorer leur pouvoir expressif.

La démarche que l'on a choisie ici est plus proche de celle proposée avec DCLL au chapitre 1 puisqu'elle consiste à partir d'un modèle de ressources n'intégrant pas de notion d'emplacement et d'y rajouter cette notion. Cependant, le choix pour la représentation de l'espace est beaucoup plus proche de ce que l'on a proposé au chapitre 1 puisque l'on propose une structuration arborescente de l'espace alors que la vision présentée dans DCLL est plane. De ce point de vue, notre vision est plus proche de ce que l'on retrouve dans les modèles d'arbres issus des logiques spatiales. Cependant, contrairement à ce qui est proposé dans ces modèles, on souhaite ici conserver l'unicité des chemins dans nos arbres. De cette gestion particulière des chemins découle en grande partie l'originalité du modèle qu'on présente ici. C'est notamment ce choix qui nous permet de modifier le contenu du noeud d'un arbre par composition, alors qu'habituellement la composition d'arbres ne fait qu'une juxtaposition des arbres composés. Ainsi, ce choix permet d'éviter l'introduction d'un niveau d'abstraction supplémentaire pour effectuer ce type de raisonnement sur les arbres qu'on manipule, comme c'est le cas pour les arbres des logiques spatiales, avec l'introduction nécessaire de la notion de contexte [24] pour définir qu'un emplacement de l'arbre peut être modifié.

2.2.1 Représentation des arbres

Un arbre de ressources est un arbre dont les arêtes ont un label (nommé emplacement) appartenant à un ensemble de noms d'emplacement et où chaque noeud contient des ressources appartenant à un monoïde partiel de ressources. On rappelle tout d'abord cette structure avant de proposer une définition formelle des arbres de ressources.

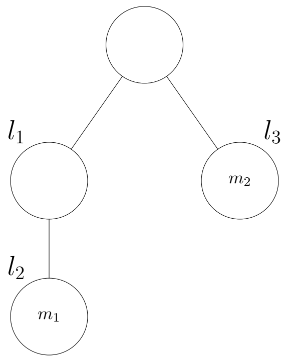
Définition 2.1 (Monoïde de ressources partiel). *Un* monoïde de ressources partiel $\mathcal{M} = (M, e, \times, \sqsubseteq)$ *est un monoïde commutatif pré-ordonné où la loi de composition* $\times$ *vérifie les conditions suivantes :*

1. $\forall m, n, k \in M.[(m \times n) \times k]\uparrow \Leftrightarrow [m \times (n \times k)]\uparrow$.
2. $\forall m, n, k \in M.[k \times n]\uparrow$ *et* $n \sqsubseteq m \Rightarrow [k \times m]\uparrow$ *et* $k \times n \sqsubseteq k \times m$.

où $[m]\uparrow$ indique que m est défini dans M.

Définition 2.2 (Ensemble d'arbres de ressources). *Soit un ensemble dénombrable de noms d'emplacements Loc et un ensemble de ressources M. L'*ensemble des arbres de ressources *construit à partir de M et Loc (noté $T_{M,Loc}$) est défini récursivement de la façon suivante :*
$T_{M,Loc} ::= M \times [Loc \rightharpoonup_{fin} T_{M,Loc}]$.

Par convention, les emplacements seront notés $l, l', l_1, l_2, \ldots$. On appelle un *chemin* une séquence finie d'emplacements. Les chemins sont notés $L, L', L_1, L_2, \ldots$. La concaténation de deux chemins L et L' est notée $L : L'$. Nous noterons les ressources $m, m', \ldots, m_1, m_2, \ldots$, les arbres de ressources $t, t', \ldots, t_1, t_2, \ldots$, et les fonctions déterminant les liens de filiations $f, f', \ldots$. Une fonction de filiation est une fonction finie qui associe chaque sous-arbre au nom d'emplacement qui y mène. On nomme *nil* la fonction de filiation totalement indéfinie. Ainsi, l'arbre vide correspond à (e, nil), un noeud contenant une ressource m correspond à (m, nil) et un arbre qui a juste un fils t le long d'une arrête l s'écrira : $(e, l \mapsto t)$. Un exemple d'arbre plus complexe et la représentation graphique correspondante est donné en figure 2.2.1.



$$(e, (l_1 \mapsto (e, l_2 \mapsto (m_1, nil)), l_3 \mapsto (m_2, nil)))$$

FIG. 2.1 – Exemple d'arbre de ressources et représentation

Soit un arbre de ressources $t = (m, f)$. Par convention, on autorisera la notation $t(l)$ à la place de $f(l)$. On étendra cette notation aux chemins de manière à ce que pour un chemin L, $t(L)$ renvoie le sous-arbre situé sous le chemin L. De même, pour un chemin $L = l_1, \ldots, l_n$, on notera $(e, L \mapsto t)$ l'arbre $(e, l_1 \mapsto (e, l_2 \mapsto (\ldots \mapsto (e, l_n \mapsto (t)) \ldots)))$.

À partir de cet ensemble, on peut créer des monoïdes partiels particuliers que nous appellerons monoïdes d'arbres partiels. Il s'agit en fait d'un sous ensemble des monoïdes partiels présentés dans la section précédente. Leur définition est la suivante :

Définition 2.3 (Monoïde d'arbres partiel). *Soit un ensemble dénombrable de noms d'emplacement Loc et un monoïde de ressources partiel $\mathcal{M} = (M, e, \times, \sqsubseteq)$. Un* monoïde d'arbres partiel $(T, (e, nil), |, \sqsubseteq_T)_{\mathcal{M},Loc}$ *vérifie les propriétés suivantes :*

1. *$T \subseteq T_{M,Loc}$.*
2. *$[(m, f)|(m'f')]\uparrow \Leftrightarrow [m \times m']\uparrow$ dans M et $\forall l.[f \cup f'(l)]\uparrow$ et $(m, f)|(m'f') = (m \times m', f \cup f')$ où $\forall l.f \cup f'(l) =$*
   – *$f(l)$ si $f'(l)$ n'est pas défini ;*
   – *$f'(l)$ si $f(l)$ n'est pas défini ;*

   – $f(l)|f'(l)$ *sinon.*
3. $\forall (m,f),(m',f').(m,f) \sqsubseteq_T (m',f')$ *ssi* $m \sqsubseteq m'$ *et* $\forall l.(([f(l)]\uparrow \Leftrightarrow [f'(l)]\uparrow)$ *et* $([f(l)]\uparrow \Rightarrow f(l) \sqsubseteq_T f'(l)))$.
4. $(T,(e,nil),|,\sqsubseteq_T)$ *est un monoïde de ressources partiel.*

où $[m]\uparrow$ *indique que* m *est défini dans* M *et* $[(m,f)]\uparrow$ *indique que* (m,f) *est défini dans* T.

Cette définition fixe le comportement de l'opérateur de composition d'arbres $|$. Il s'agit de composer les ressources présentes au même endroit et donc de fusionner les deux arbres à composer.

La relation d'ordre $\sqsubseteq_T$ utilisée pour les arbres est une extension simple de celle utilisée pour les ressources. Plus clairement, un arbre t subsume un arbre t' (ce qui est noté $t \sqsubseteq_T t'$) s'ils ont la même structure (les même chemins sont définis) et si pour chaque chemin de t, les ressources présentes au bout de ce chemin subsument celles présentes au bout du même chemin dans t.

La présentation des arbres de ressources sous la forme de relation ensembliste est proche de ce qui est proposé dans le modèle des tas de pointeurs proposé par O'Hearn et al. [62], pour la logique des pointeurs. Nous pouvons également donner une définition sous forme de grammaire de termes associée à une notion d'équivalence, que l'on appelle aussi notation *à la Cardelli*, telle qu'elle est par exemple proposée pour le modèle d'arbres présenté dans [23] :

Définition 2.4 (Arbres de ressources : grammaire). *Soit un ensemble dénombrable d'emplacement Loc et un ensemble de ressources M. Tout arbre de ressource de $T_{M,Loc}$ peut s'écrire avec la grammaire suivante :*

$$t ::= m \mid t|t \mid [l]t$$

$$\text{où } \forall m,m' \in M \;\; m|m' \stackrel{\text{def}}{=} m \times m'$$

Et nous avons pour cette représentation la relation d'équivalence structurelle suivante :

Définition 2.5 (Équivalence structurelle). *L'équivalence structurelle sur les arbres est la plus petite relation d'équivalence vérifiant les propriétés suivantes pour tout arbre t, t', r, r' :*

- $t \equiv t$	- $t \equiv t' \Rightarrow t' \equiv t$	- $t \equiv t'$ et $t' \equiv r \Rightarrow t \equiv r$
- $t|t' \equiv t'|t$	- $(t|t')|r \equiv t|(t'|r)$	- $t \equiv t' \Rightarrow t|r \equiv t'|r$
- $t|e \equiv t$	- $[l]t|[l]t' \equiv [l](t|t')$	- $t \equiv t' \Rightarrow [l]t \equiv [l]t'$

Avec cette définition, l'arbre vide s'écrit e et un arbre composé uniquement d'un sous-arbre t à l'emplacement l s'écrira $[l]t$. On peut donc facilement proposé une relation entre les deux présentations des arbres de ressources présentées ici :

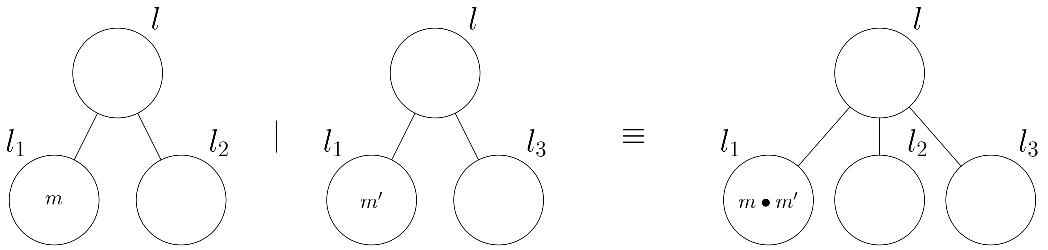
Définition 2.6. *La correspondance entre une représentation directe d'un arbre de ressources et une présentation* à la Cardelli *est donnée par la transformation* $[\![\cdot]\!]_{card}$ *définie récursivement ci-dessous :*
– $[\![m]\!]_{card} = (m, nil)$;
– $[\![t|t']\!]_{card} = [\![t]\!]_{card}|[\![t']\!]_{card}$;
– $[\![[l]t]\!]_{card} = (e, l \mapsto [\![t]\!]_{card})$.

Cette correspondance est telle que si on a deux arbres t et t' utilisant la notation *à la Cardelli* tels que $t \equiv t'$ alors $[\![t]\!]_{card} = [\![t']\!]_{card}$.

Dans la suite de ce chapitre, on utilise principalement la représentation ensembliste des arbres de ressources qui a l'avantage de définir l'arbre précisément puisque l'égalité entre deux arbres n'est pas sujette à une relation de congruence.

2.2.2 Chemins uniques et composition

Lorsque nous avons dû gérer l'espace, nous avons opté pour des chemins uniques, c'est à dire que nous considérons qu'une même séquence d'emplacements ne peut pas mener à des endroits différents dans l'arbre. C'est ce choix qui a permis d'obtenir une définition directe des arbres de ressources, tel que présentée dans la définition 2.2. Il est alors nécessaire de se demander alors ce que donne la composition de deux arbres de ressources ayant des chemins en commun. Le choix que nous avons retenu, qui est déjà apparent dans les définitions précédentes, est de fusionner les chemins communs et de composer les ressources présentes dans leurs emplacements. Ce comportement est fixé par la définition de la relation de composition $|$ et apparaît plus clairement dans la règle d'équivalence structurelle $[l](t|t') \equiv [l]t|[l]t'$. Le comportement de la composition est illustré par la figure 2.2.



$(e,(l_1 \mapsto (m,nil), l_2 \mapsto (m,nil)))$ | $(e,(l_1 \mapsto (m',nil), l_3 \mapsto (e,nil)))$ $\equiv$ $(e,(l_1 \mapsto (m \times m',nil), l_2 \mapsto (e,nil), l_3 \mapsto (e,il)))$

FIG. 2.2 – Composition d'arbres de ressources

Cette gestion de la composition mérite quelques remarques supplémentaires. Elle entraîne par exemple des compositions équivalentes surprenantes. Ainsi, on a $(e, l \mapsto (m, nil))|(e, nil) = (e, l \mapsto (m, nil))|(e, l \mapsto (e, nil))$. Ici les arbres (e, nil) et $(e, l \mapsto (e, nil))$ ne sont pas équivalents puisque le premier correspond à l'arbre vide alors que le second a un sous-arbre en l. Pourtant, leur composition avec l'arbre $(e, l \mapsto (m, nil))$ mène au même résultat. En effet, dans les deux cas, le sous-arbre situé en l contiendra au final uniquement la ressource m.
En effet, on a $(e, l \mapsto (m, nil))|(e, nil) = ((e \times e), (nil \cup l \mapsto (m, nil))) = (e, l \mapsto (m, nil))$,
et $(e, l \mapsto (m, nil))|(e, l \mapsto (e, nil)) = (e \times e, ((l \mapsto (m, nil) \cup (l \mapsto (e, nil))) = (e, (l \mapsto (m, nil))$

2.2.3 Composition partielle

Un autre point fort de ce modèle est d'étendre aux arbres la composition partielle du monoïde de ressources sur lequel il se fonde. Pour un monoïde d'arbres partiel donné, la composition partielle est au minimum fondée sur celle du monoïde de ressources : si la composition de deux arbres mène à la composition de deux ressources qui est indéfinie, la composition des arbres sera elle même indéfinie.

Cependant, la composition partielle peut également être définie au niveau des arbres. On peut, par exemple, définir que la composition de deux arbres contenant la même ressource, même à des emplacements différents, est indéfinie pour assurer que sa présence est unique dans tout l'arbre. De même, on peut empêcher le même nom d'emplacement d'apparaître à deux endroits différents dans l'arbre.

2.2.4 Un exemple

Pour illustrer l'avantage des spécificités des arbres de ressources, on considère l'utilisation des arbres de ressources pour représenter des données semi-structurées. Cette application des arbres de ressources est détaillée plus avant dans le chapitre 5 mais on en propose ici une première approche informelle. La figure 2.3 donne un exemple de données semi-structurées et propose une représentation sous forme d'arbre (l'arbre t) de ces données.

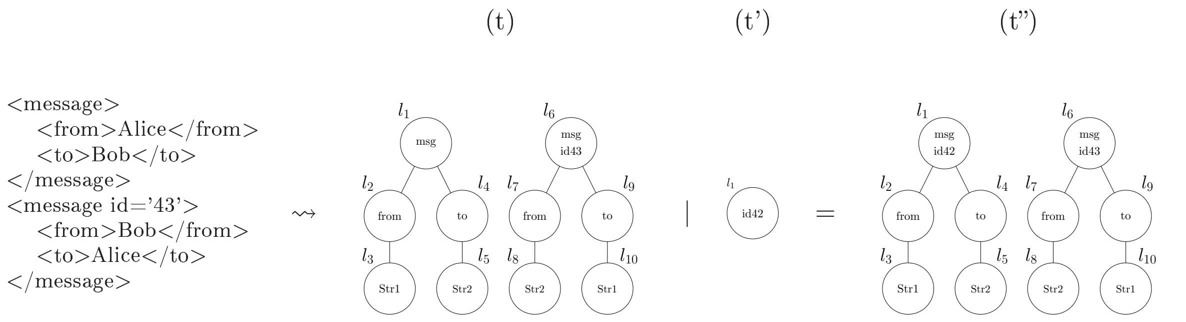


FIG. 2.3 – Représentation et modification de données semi-structurées

On propose alors d'ajouter à cet arbre l'attribut id = *'42'* dans le nœud correspondant au premier message. Cet ajout peut se faire en composant l'arbre t avec l'arbre t' et il est rendu possible par le fait que l'utilisation du chemin l_1 dans l'arbre t' permet de signaler que la ressource $id42$ doit être intégrée exactement à cet emplacement par l'arbre t. Le résultat de cette composition est l'arbre t'' qui inclut donc bien l'attribut voulu à l'emplacement souhaité.

Supposons maintenant que, comme c'est bien souvent le cas avec les données semi-structurées, on ne puisse pas déclarer deux fois le même attribut dans un même nœud. Supposons également qu'au lieu d'ajouter l'attribut id = *'42'* en l_1 on l'ajoute en l_6, c'est à dire dans un noeud possédant déjà un attribut id. On peut alors utiliser le fait que la composition soit partielle pour indiquer que la composition des deux arbres serait alors indéfinie. On voit que la composition partielle permet ici d'exclure de notre modèle des arbres syntaxiquement incorrects.

2.3 Présentation de BI-Loc

Comme il est décrit dans la définition 4, un monoïde d'arbres partiel est un type particulier de monoïde partiel. Par conséquent, on aurait pu utiliser la logique BI pour raisonner sur cette structure. Cependant, il est indispensable que la logique tienne compte intrinsèquement de la distribution spatiale des ressources. Dans cette optique, on considère une nouvelle modalité $[l]$, dite *modalité d'emplacement*, qui indique qu'une formule est vérifiée à l'emplacement l. Ainsi un arbre de ressources t satisfait une formule $[l]\phi$ s'il est constitué uniquement d'un fils t' situé en l et que t' vérifie ϕ.

2.3.1 La logique BI-Loc propositionnelle

On propose tout d'abord une version propositionnelle de la logique.

Définition 2.7 (Formules de BI-Loc). *Étant donné une signature* Σ *de variables propositionnelles, et un ensemble dénombrables d'emplacements Loc, la collection des* formules propositionnelles de BI-Loc *engendrées par* Σ *et Loc est donnée par la grammaire suivante :*

$$\begin{array}{llll} \phi & ::= & p\ (\in \Sigma) & \textit{Atomes} \\ & \mid & \phi * \phi \mid I \mid \phi -\!\!* \phi & \textit{Multiplicatifs} \\ & \mid & \phi \wedge \phi \mid \top \mid \phi \vee \phi \mid \bot \mid \phi \to \phi & \textit{Additifs} \\ & \mid & [l]\phi & \textit{Modalité spatiale} \end{array}$$

Nous adopterons les conventions d'écriture suivantes : pour une séquence d'emplacements $L = \{l_1, \ldots, l_n\}$, nous écrirons $[L]\phi$ à la place de $[l_1]\ldots[l_n]\phi$. Les liens entre BI-Loc et les monoïdes d'arbres partiels sont définis de la façon suivante :

Définition 2.8 (Modèle d'arbres de ressources partiel). *Un* modèle d'arbres de ressources partiel *(ou* modèle d'arbre partiel*) est un monoïde d'arbres de ressources* $(T, (e, nil), |, \sqsubseteq_T)_{\mathcal{M},Loc}$ *muni d'une relation de forcing* $\models_{\mathfrak{T}} \subseteq M \times \Sigma$ *satisfaisant la condition suivante :*

$$\forall p \in \Sigma. \forall t, t' \in T.(t \models_{\mathfrak{T}} p \textit{ et } t' \sqsubseteq_T t \Rightarrow t' \models_{\mathfrak{T}} p)$$

et s'étendant aux formules de BI-Loc *comme suit :*

- $t \models_{\mathfrak{T}} \phi * \psi$ *ssi* $\exists t', t''.[t|t']\uparrow$, $t \sqsubseteq_T t'|t''$, $t' \models_{\mathfrak{T}} \phi$ *et* $t'' \models_{\mathfrak{T}} \psi$ *;*
- $t \models_{\mathfrak{T}} I$ *ssi* $t \sqsubseteq_T (e, nil)$ *;*
- $t \models_{\mathfrak{T}} \phi -\!\!* \psi$ *ssi* $\forall t' \in M$, $t' \models_{\mathfrak{T}} \phi$ *et* $[t|t']\uparrow \Rightarrow t|t' \models_{\mathfrak{T}} \psi$ *;*
- $t \models_{\mathfrak{T}} \phi \wedge \psi$ *ssi* $t \models_{\mathfrak{T}} \phi$ *et* $t \models_{\mathfrak{T}} \psi$ *;*
- $t \models_{\mathfrak{T}} \top$ *toujours ;*
- $t \models_{\mathfrak{T}} \phi \vee \psi$ *ssi* $t \models_{\mathfrak{T}} \phi$ *ou* $t \models_{\mathfrak{T}} \psi$ *;*
- $t \models_{\mathfrak{T}} \bot$ *jamais ;*
- $t \models_{\mathfrak{T}} \phi \to \psi$ *ssi* $\forall t'.t' \sqsubseteq_T t$ *et* $t' \models_{\mathfrak{T}} \phi \Rightarrow t \models_{\mathfrak{T}} \psi$ *;*
- $t \models_{\mathfrak{T}} [l]\phi$ *ssi* $t \sqsubseteq (m, l \mapsto t')$ *avec* $t' \models_{\mathfrak{T}} \phi$.

Les règles des opérateurs $*$ et $\wedge$ de ce modèle illustrent l'expression de la séparation et du partage dans BI-Loc qui étend la logique BI [76, 79]. La règle de $*$ indique que l'on doit pouvoir séparer un arbre de ressources en deux sous-arbres distincts vérifiant chacune une sous formule, ces sous-formules *séparent* donc l'arbre. La règle de $\wedge$ indique que le même arbre vérifie les deux sous-formules, elle indique donc que ces sous-formules *partagent* le même arbre. Le modèle ainsi obtenu présente de nombreuses similitudes avec le modèle de ressources partiel pour BI [49, 72]. Cette similitude est naturelle car on cherche à représenter les mêmes notions (partage, séparation) pour les ressources présentes dans un emplacement d'un arbre que pour les ressources dans le modèle de ressources partiel.

Il est important de faire certaines remarques sur les spécificités introduites par l'ajout de la modalité $[l]$. En effet, sa cohabitation avec les autres opérateurs mérite quelques précisions.
Tout d'abord si l_1 et l_2 sont 2 emplacements distincts, quelques soient ϕ et ψ un arbre t satisfait $[l_1]\phi \wedge [l_2]\psi$ si et seulement s'il satisfait également $\bot$. En effet, un même arbre ne peut à la fois être constitué uniquement d'un fils à l'emplacement l_1 et uniquement d'un fils en l_2.
Les formules $\top$ et $[l]\top$ ne sont pas équivalentes. La première est vérifiée par tout arbre de ressource, la seconde uniquement par les arbres de ressources constitués uniquement d'un sous-arbre en l. Par contre, $\bot$ et $[l]\bot$ sont équivalentes : toutes deux ne sont jamais satisfaites.
Enfin, on peut noter que pour toute formule ϕ, $[l]\phi \to [l]\phi$ n'est pas équivalent à $[l](\phi \to \phi)$. La première est toujours vérifiée alors que la seconde est vérifiée uniquement par les arbres contenant

uniquement un sous-arbre en l.

Nous pouvons maintenant définir les notions classiques de satisfaction et de validité d'une formule.

Définition 2.9 (Satisfaction). *Soit un modèle d'arbre partiel $\mathcal{T} = (T, (e, nil), |, \sqsubseteq_T, \models_{\mathfrak{T}})_{\mathcal{M},Loc}$, un arbre de ressource $t \in T$ et une formule de* BI-Loc *ϕ, on dit que t* satisfait *ϕ ssi on a $t \models_{\mathfrak{T}} \phi$.*

Définition 2.10 (Validité (pour un modèle)). *Une formule ϕ est* valide *pour un modèle $\mathcal{T} = (T, (e, nil), |, \sqsubseteq_T, \models_{\mathfrak{T}})_{\mathcal{M},Loc}$ (noté $\models_{\mathfrak{T}} \phi$) ssi pour tout arbre de ressources $t \in \mathcal{T}$, on a $t \models_{\mathfrak{T}} \phi$.*

La relation entre la validité pour un modèle et la satisfaction peut être exprimée de la façon suivante :

Lemme 2.1. Soit une formule ϕ de BI-Loc, $\models_{\mathfrak{T}} \phi$ ssi $e \models_{\mathfrak{T}} \top \multimap \phi$.

Preuve. Ce lemme est une application directe des définitions de validité et de satisfaction. $e \models_{\mathfrak{T}} \top \multimap \phi$ si et seulement si pour tout t tel que $t \models_{\mathfrak{T}} \top$, $t \models_{\mathfrak{T}} \phi$. Or, par définition, on a $\forall t.t \models_{\mathfrak{T}} \top$. On peut donc conclure directement. □

On peut également définir les notions de *tautologie* et de *conséquence logique* qui sont indépendantes du modèle sous-jacents.

Définition 2.11 (Tautologie). *Une formule ϕ (noté $\models \phi$) est une* tautologie *si pour tout modèle $\mathcal{T} = (T, (e, nil), |, \sqsubseteq_T, \models_{\mathfrak{T}})_{\mathcal{M},Loc}$, on a $\models_{\mathfrak{T}} \phi$.*

Définition 2.12 (Conséquence logique). *Une formule ψ est la* conséquence logique *d'une formule ϕ (noté $\phi \models \psi$) si pour tout modèle $\mathcal{T} = (T, (e, nil), |, \sqsubseteq_T, \models_{\mathfrak{T}})_{\mathcal{M},Loc}$, et pour tout $t \in T$, on a $t \models_{\mathfrak{T}} \phi$ implique $t \models_{\mathfrak{T}} \psi$.*

Là aussi, on peut établir une relation entre tautologie et conséquence logique de la façon suivante :

Lemme 2.2. Soit deux formules ϕ et ψ de BI-Loc, $\phi \models \psi$ ssi $\models \phi \to \psi$.

Preuve. Ce lemme est une conséquence directe de la sémantique de l'opérateur $\to$. □

2.3.2 Extension avec quantificateurs

On a déjà évoqué plus haut l'utilisation possible des arbres de ressources pour représenter des données semi-structurées. Dans ce cadre, il est alors normal de vouloir utiliser BI-Loc pour spécifier ces documents et de vouloir exprimer des propriétés impliquant tous les emplacements ou un emplacement particulier mais dont on ne connaît pas le nom.

Prenons l'exemple de l'arbre de la figure 2.4. Cet arbre, qui est présenté plus en détail au chapitre 5, comporte deux pointeurs qui sont indiqués sur la figure par des pointillés. Pour vérifier qu'un pointeur fait référence à un nœud existant il faut s'assurer qu'il *existe* dans l'arbre un *chemin* menant vers la valeur indiqué par le pointeur. De plus, il faut bien évidemment vérifier que ceci est vrai pour *tous les emplacements* qui contiennent un pointeur.

Il apparaît donc essentiel d'étendre l'expressivité de BI-Loc avec des quantificateurs. En plus de la distinction habituelle entre quantificateurs existentiels (il existe un emplacement) et universels (pour tout emplacement), nous distinguons deux autres types de quantificateurs : ceux concernant les emplacements et ceux concernant les chemins (un chemin désignant une suite finie d'emplacements). Cette extension est appelée BI-Loc$_\forall$

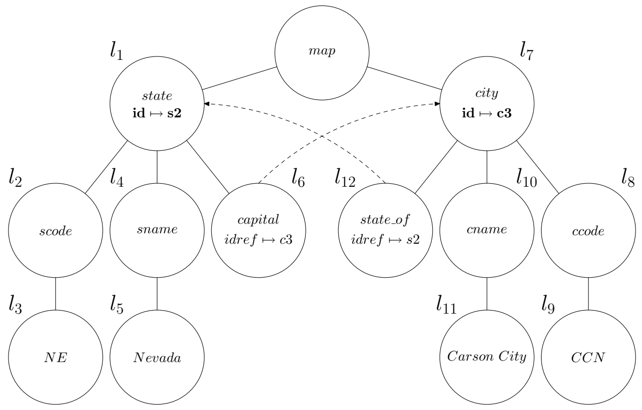


FIG. 2.4 – Un arbre représentant un document XML avec pointeurs

Définition 2.13 (Formule de BI-Loc$_\forall$). *Étant donné une signature Σ de variables propositionnelles, un ensemble de variables d'emplacements $\{x, x', y, X, X', Y\}$ et un ensemble dénombrable d'emplacements Loc, la* collection des formules de BI-Loc$_\forall$ *engendrées par Σ et Loc est donnée par la grammaire suivante :*

ϕ	$::=$	$p(\in \Sigma)$	*Atomes*
	$\mid$	$\phi * \phi \mid I \mid \phi \mathrel{-\!\!*} \phi$	*Multiplicatifs*
	$\mid$	$\phi \wedge \phi \mid \top \mid \phi \vee \phi \mid \bot$	*Additifs*
	$\mid$	$[l]\phi$	
	$\mid$	$[x]\phi$	*Modalité spatiale*
	$\mid$	$[X]\phi$	
	$\mid$	$\exists_{loc} x.\phi$	*Quantification existentielle sur les emplacements*
	$\mid$	$\forall_{loc} x.\phi$	*Quantification universelle sur les emplacements*
	$\mid$	$\exists_{path} X.\phi$	*Quantification existentielle sur les chemins*
	$\mid$	$\forall_{path} X.\phi$	*Quantification universelle sur les chemins*

Et la définition de modèle d'arbres partiel s'étend elle aussi. On étend également le modèle donné dans la définition 2.8 :

Définition 2.14 (Modèle d'arbres pour BI-Loc$_\forall$). *Un* modèle d'arbres de ressources partiel *est un monoïde d'arbres de ressources $(T, (e, nil), |, \sqsubseteq_T)_{\mathcal{M},Loc}$ muni d'une relation de forcing $\models_{\mathfrak{T}} \subseteq M \times \Sigma$ satisfaisant la condition suivante :*

$$\forall p \in \Sigma. \forall t, t' \in T.(t \models_{\mathfrak{T}} p \text{ et } t' \sqsubseteq_T t \Rightarrow t' \models_{\mathfrak{T}} p$$

et s'étendant aux formules de BI-Loc$_\forall$ *comme suit :*
- $t \models_{\mathfrak{T}} \phi * \psi$ *ssi* $\exists t', t''.[t|t']\uparrow,\ t \sqsubseteq_T t'|t'',\ t' \models_{\mathfrak{T}} \phi$ *et* $t'' \models_{\mathfrak{T}} \psi$ *;*
- $t \models_{\mathfrak{T}} I$ *ssi* $t \sqsubseteq_T (e, nil)$ *;*
- $t \models_{\mathfrak{T}} \phi \mathrel{-\!\!*} \psi$ *ssi* $\forall t' \in M,\ t' \models_{\mathfrak{T}} \phi$ *et* $[t|t']\uparrow \Rightarrow t|t' \models_{\mathfrak{T}} \psi$ *;*
- $t \models_{\mathfrak{T}} \phi \wedge \psi$ *ssi* $t \models_{\mathfrak{T}} \phi$ *et* $t \models_{\mathfrak{T}} \psi$ *;*
- $t \models_{\mathfrak{T}} \top$ *toujours ;*
- $t \models_{\mathfrak{T}} \phi \vee \psi$ *ssi* $t \models_{\mathfrak{T}} \phi$ *ou* $t \models_{\mathfrak{T}} \psi$ *;*
- $t \models_{\mathfrak{T}} \bot$ *jamais ;*
- $t \models_{\mathfrak{T}} \phi \rightarrow \psi$ *ssi* $\forall t'.t' \sqsubseteq_T t$ *et* $t' \models_{\mathfrak{T}} \phi \Rightarrow t \models_{\mathfrak{T}} \psi$ *;*
- $t \models_{\mathfrak{T}} [l]\phi$ *ssi* $t \sqsubseteq_T (m, l \mapsto t')$ *avec* $t' \models_{\mathfrak{T}} \phi$ *;*

– $t \models_{\mathfrak{T}} \exists_{loc} x.\phi$ *ssi il existe* $l \in Loc$ *tel que* $t \models_{\mathfrak{T}} \phi\{^{l}/_{x}\}$ *;*
– $t \models_{\mathfrak{T}} \forall_{loc} x.\phi$ *ssi pour tout* $l \in Loc$*, on a* $t \models_{\mathfrak{T}} \phi\{^{l}/_{x}\}$ *;*
– $t \models_{\mathfrak{T}} \exists_{path} X.\phi$ *ssi il existe* $L \in Loc^{*}$ *tel que* $t \models_{\mathfrak{T}} \phi\{^{L}/_{X}\}$ *;*
– $t \models_{\mathfrak{T}} \forall_{path} X.\phi$ *ssi pour tout* $L \in Loc^{*}$*, on a* $t \models_{\mathfrak{T}} \phi\{^{L}/_{X}\}$*.*

Il est nécessaire de faire quelques remarques concernant les quantifications universelles.

Lemme 2.3. Si une formule ϕ contient une variable d'emplacement $[x]$ (resp. la variable de chemin $[X]$) mais ne contient aucun opérateur $-\!\!*$ ou $\rightarrow$, alors il n'existe pas de t tels que $t \models_{\mathfrak{T}} \forall_{loc} x.\phi$ (resp. $t \models_{\mathfrak{T}} \forall_{path} X.\phi$).

Preuve. Pour que $t \models_{\mathfrak{T}} \forall_{path} X.\phi$, on montre que t doit contenir tous les emplacements de *Loc*. Or *Loc* est infini, donc t est un arbre infini, ce qui est contraire à la définition d'un arbre de ressources. Raisonnons maintenant par induction structurelle sur ϕ pour montrer que si $t \models_{\mathfrak{T}} \phi\{^{l}/_{x}\}$ (resp. $t \models_{\phi} \{^{L}/_{X}\}$) alors t contient l (resp. L). Comme ϕ doit contenir au moins une instance de $[x]$ (resp. $[X]$), le cas de base est le suivant :

- Cas $[l]\psi$: $t \models_{\mathfrak{T}} [l]\psi$, et par définition, t contient l.

A partir de là, on raisonne par cas sur $\phi\{^{l}/_{x}\}$

- Cas $\psi * \psi'$: $t \models_{\mathfrak{T}} \psi * \psi'$, par définition, il existe t', t'' tels que $t \sqsubseteq t'|t''$, $t' \models_{\mathfrak{T}} \psi$ et $t'' \models_{\mathfrak{T}} \psi'$. D'après les hypothèses, ψ ou ψ' contient $[l]$ et donc par hypothèse d'induction, t' ou t'' contient l. Donc t contient l.
- Cas $\psi \wedge \psi'$: $t \models_{\mathfrak{T}} \psi \wedge \psi'$, par définition, $t \models_{\mathfrak{T}} \psi$ et $t \models_{\mathfrak{T}} \psi'$. D'après les hypothèses, ψ ou ψ' contient $[l]$ et donc par hypothèse d'induction, t contient l.
- Cas $\psi \vee \psi'$: $t \models_{\mathfrak{T}} \psi \vee \psi'$, par définition, $t \models_{\mathfrak{T}} \psi$ ou $t \models_{\mathfrak{T}} \psi'$. D'après les hypothèses, ψ ou ψ' contient $[l]$ et donc par hypothèse d'induction, t contient l.
- Cas $[l']\psi$: $t \models_{\mathfrak{T}} [l']\psi$, par définition, $t \models_{\mathfrak{T}} \psi$ ou $t \models_{\mathfrak{T}} \psi'$. D'après les hypothèses, ψ contient $[l]$ et donc par hypothèse d'induction, t contient l.

$\square$

2.4 Un langage de transformations

La proposition du modèle d'arbres partiel vient de l'idée qu'une structure mêlant à la fois une structure spatiale et une représentation fine des ressources est adaptée pour représenter des données hiérarchisées et/ou réparties entre différents acteurs. La proposition d'un tel modèle n'est toutefois pertinente que si l'on possède les outils nécessaires pour modifier les structures et raisonner formellement sur ces modifications.

Dans cette optique, on souhaite donc proposer un langage pour manipuler les arbres de ressources. Un tel langage doit agir au niveau de la structure de l'arbre (ajouter ou enlever des nœuds) et des ressources présentes dans les nœuds. L'idée principale consiste, à partir du modèle d'arbres partiel, à définir des pré et des post-conditions pour chaque commandes sous forme de triplets de Hoare [60]. De tels triplets permettent de définir quelle forme doit avoir un arbre avant exécution d'un programme de manipulation donné pour vérifier certaines propriétés après vérification. Une méthodologie similaire a été suivie par O'Hearn et al. [62, 75] dans les travaux sur la manipulations des pointeurs et par Calcagno et al. pour la manipulation d'arbres [24]. Le modèle des arbres de ressources nécessite un positionnement intermédiaire entre ces deux langages. La logique BI-Loc étant une extension de BI, le langage d'assertions et les axiomes des commandes définis ici sont semblables dans leur énoncés à ceux des travaux de [62]. Les commandes de manipulation, quant à elles, sont globalement similaires à celles des travaux de [24] car dans les deux cas elles visent à manipuler un arbre.

2.4.1 Les commandes

Le langage introduit ici est fortement lié au langage pour la manipulation des pointeurs proposé dans [62, 75]. L'idée est de garder les commandes conditionnelles et les boucles introduite dans [81] et d'adapter les commandes d'affectation et d'accès au contenu pour gérer les arbres de ressources. Nous rappelons brièvement les commandes de contrôle qui sont standard avant de présenter celles spécifiques aux arbres de ressources :

Définition 2.15 (Commandes de contrôle).

$$\begin{array}{lll} C & ::= & \textit{if } (E) \textit{ then } C \textit{ else } C' & \textit{Si} \ldots \textit{alors} \ldots \textit{sinon} \ldots \\ & & \textit{while } (E) \textit{ do } C & \textit{Boucle} \end{array}$$

où E est une expression devant être interprétée comme une valeur booléenne.

Définition 2.16 (Commandes de manipulation).

$$\begin{array}{lll} C & ::= & x := new_{loc}E & \textit{Création d'un nouvel emplacement} \\ & & dispose_{loc}E & \textit{Suppression d'un emplacement} \\ & & x := E & \textit{Affectation d'une variable} \\ & & x := res(E) & \textit{Observer les ressources d'un emplacement} \\ & & x := tree(E) & \textit{Observer le sous-arbre d'un emplacement} \\ & & update_{res}(E_1, E_2) & \textit{Mettre à jour les ressources d'un emplacement} \\ & & update_{tree}(E_1, E_2) & \textit{Mettre à jour le sous-arbre d'un emplacement} \\ & & add(E_1, E_2) & \textit{Ajouter du contenu à un emplacement} \end{array}$$

E, E_i sont des expressions dont la forme sera précisée plus tard. Ce sont par contre des expressions simples, c'est à dire qu'elles ne contiennent pas d'autres commandes. Le domaine sémantique d'une expression dépend de la commande où elle est utilisée.

Les deux premières commandes modifient la structure de l'arbre en ajoutant ou en supprimant un nœud à un emplacement donné. L'expression E de ces deux commandes doit donc correspondre à un chemin, respectivement celui qui recevra le nœud à ajouter ou celui du nœud qu'on supprime. De plus, dans la commande new_{loc}, la variable x reçoit le chemin de l'emplacement nouvellement créé.

La commande suivante ($x := E$) effectue une affectation directe de variable. E peut donc correspondre à n'importe quel type d'expression autorisé par le langage. Nous avons ensuite deux commandes d'accès au contenu ($x := res(E)$ et $x := tree(E)$) qui permettent de récupérer les informations présentes dans un chemin donné. Par conséquent, dans ces deux commandes, l'expression E doit correspondre là aussi à un chemin.

Les trois dernières commandes permettent de modifier le contenu de l'arbre. L'expression ($update_{res}(E_1, E_2)$) remplace les ressources présentes sous un chemin donné. E_1 doit donc être interprétée comme un chemin, celui où a lieu la modification et E_2 doit être interprétée comme une ressource, celle qui est substituée à celles présentes en E_1. La deuxième ($update_{tree}(E_1, E_2)$) fait un remplacement de sous-arbres. E_1 doit là aussi correspondre à un chemin et E_2 doit correspondre à un arbre, celui qui remplace le sous-arbre présent en E_1. Enfin, la commande

$add(E_1, E_2)$ ajoute des informations sous un chemin donné. L'expression E_1 doit donc correspondre à un chemin et E_2 doit être un arbre, éventuellement réduit à une ressource, qui sera composé avec le contenu présent en E_1.

Une expression peut donc être une variable, une ressource, un chemin, un arbre de ressources ou toute autre expression que l'on peut interpréter comme appartenant à un des domaines sémantiques détaillés ci-dessous.

Définition 2.17 (Domaines sémantiques).

- $Val = M \cup T_M \cup Loc$;
- $S = Var \rightharpoonup_{fin} Val$;
- $T_M = M \times [Loc \rightharpoonup_{fin} T_M]$.

Comme dans les chapitres précédents, $Loc = \{l_1, l_2 \ldots, l_n, \ldots\}$ est un ensemble d'emplacements dénombrable, M est un ensemble de ressources appartenant à un monoïde de ressources partiel $\mathcal{M} = (M, e, \times, \sqsubseteq)$. $Var = \{x, y, z, x', y', 'z, \ldots\}$ est un ensemble de variables. La notation $\rightharpoonup_{fin}$ exprime une fonction partielle et finie. Un élément $s \in S$ est appelé une pile et l'ensemble T_m est l'ensemble des arbres de ressources construit à partir de M. On utilise également la notation $dom(s)$ pour désigner l'ensemble des variables définies dans une pile s.

La valeur d'une expression dépend d'une pile s. Une interprétation d'une expression E est donc la valeur $[\![E]\!]_s \in Val$ où la pile s est telle que $dom(s)$ inclut toues les variables intervenant dans E.

Les commandes effectuent donc des opérations sur un couple s, t, avec $s \in S$ et $t \in T_M$. Elles sont interprétées grâce à une relation $\rightsquigarrow$ entre configurations, la clôture transitive de cette relation étant notée $\rightsquigarrow^*$. Une configuration est soit un triplet C, s, t soit un couple s, t, avec C une commande, $s \in S$ et $t \in T_m$. Ce dernier type de configuration est dit *terminal*. La sémantique des commandes est donnée à travers la définition de la relation $\rightsquigarrow$.

On commence par rappeler la sémantique classique des commandes de contrôle :

$$\frac{i = 1 \text{ si } [\![E]\!]_s = vrai, i = 2 \text{ si } [\![E]\!]_s = faux}{\text{if } (E) \text{ then } C_1 \text{ else } C_2, s, t \rightsquigarrow C_i, s, t}$$

$$\frac{[\![E]\!]_s = vrai}{\text{while } (E) \text{ do } C \text{ end}, s, t \rightsquigarrow C; \text{while } (E) \text{ do } C \text{ end}, s, t} \qquad \frac{[\![E]\!]_s = false}{\text{while } (E) \text{ do } C \text{ end}, s, t \rightsquigarrow s, t}$$

On passe maintenant aux commandes spécifiques aux arbres de ressources :

$$\frac{[\![E]\!]_s = L \in Loc^*, l \in Loc \text{ tel que } [t(L)]\!\uparrow \text{ et } t(L : l) \text{ non défini}}{x := new_{loc}E, s, t \rightsquigarrow [s|x \mapsto L : l, t|(e, (L : l) \mapsto (e, nil))}$$

Comme indiqué lors de la présentation des commandes, la commande *new* crée un emplacement sous le chemin $[\![E]\!]_s$. Pour cela, on s'assure que cet emplacement n'existe pas ($t(L : l)$ non défini).

$$\frac{[\![E]\!]_s = L \in Loc^* \text{ tel que } t = t'|(e, L \mapsto t'') \text{ et } t'(L) \text{ non défini}}{dispose(E), s, t \rightsquigarrow s, t'}$$

Pour la commande *dispose*, le plus délicat est d'exprimer la séparation du sous-arbre situé en $[\![E]\!]_S$ du reste de l'arbre. On décompose donc t en deux sous-arbres, un qui ne contient rien ailleurs que sous le chemin $[\![E]\!]_S$ et qui sera supprimé, et un autre qui ne contient pas ce chemin et qui sert de résultat.

$$\frac{[\![E]\!]_s = v \in M \cup \mathcal{L}^*}{x := E, s, t \rightsquigarrow [s|x \mapsto v], t}$$

$$\frac{[\![E]\!]_s = L \in Loc^*, m \in M \text{ tel que } [t(L)]\!\uparrow \text{ et } t(L) = (m, f)}{x := res(E), s, t \rightsquigarrow [s|x \mapsto m], t}$$

Pour la commande $res(E)$, on doit s'assurer que l'on affecte bien à x le contenu de ce qui se trouve en $[\![E]\!]_s$. On s'assure donc que le chemin existe et on récupère son contenu.

$$\frac{[\![E]\!]_s = L \in Loc^*, m \in M \text{ tel que } [t(L)]\!\uparrow \text{ et } t(L) = (m, f)}{x := tree(E), s, t \rightsquigarrow [s|x \mapsto (m, f)], t}$$

$$\frac{[\![E_2]\!]_s = m \in M,\ [\![E_1]\!]_s = L \in Loc^* \text{ tel que } t = t'|(e, L \mapsto (m', nil)) \text{ et } t'(L) = (e, f)}{update_{res}(E_1, E_2), s, t \rightsquigarrow s, t'|(e, L \mapsto (m, nil))}$$

$$\frac{[\![E_2]\!]_s = t'',\ [\![E_1]\!]_s = L \in Loc^* \text{ tel que } t = t'|(e, L \mapsto t''') \text{ et } t'(L) \text{ non défini}}{update_{tree}(E_1, E_2), s, t \rightsquigarrow s, t'|(e, L \mapsto t'')}$$

$$\frac{[\![E_2]\!]_s = t',\ [\![E_1]\!]_s = L \in Loc^* \text{ tel que } [t(L)]\!\uparrow}{add(E_1, E_2), s, t \rightsquigarrow s, t|(e, L \mapsto t')}$$

Pour une configuration C, s, t, on dit que :
- C, s, t est *bloquée* s'il n'existe pas de configuration K telle que $C, s, t \rightsquigarrow K$;
- C, s, t est *sûre* si pour toute configuration K telle que $C, s, t \rightsquigarrow^* K$, K est une configuration terminale ou non bloquée.

Une configuration est bloquée si une expression n'appartient pas au bon domaine ou si elle essaye d'accéder à des parties inexistantes de l'arbre. Être bloqué correspond donc à une erreur d'exécution.

Il est aussi important de remarquer qu'être *sûre* n'implique pas la terminaison, on peut en effet retourner de manière cyclique à la configuration C, s, t sans terminer et être sûr.

2.4.2 Une logique d'assertions issue de BI-Loc

On propose ici d'exprimer des propriétés sur les configurations à l'aide d'une logique issue de BI-Loc. On modifie légèrement la logique pour y inclure des tests sur les expressions et des quantifications utiles pour raisonner sur le contenu des arbres de ressources.

Pour un ensemble d'emplacements *Loc*, les pré- et post-conditions des commandes sont des formules respectant la grammaire donnée ci-dessous :

$$\begin{array}{rcl} P, Q & ::= & \alpha \mid \bot \mid \top \mid P \vee Q \mid P \wedge Q \mid P \rightarrow Q \mid I \mid P * Q \mid P \mathrel{-\!\!*} Q \mid [l]P \mid \\ & & \mid \exists_{loc} x.P \mid \exists_{path} x.P \mid \exists_{res} x.P \mid \exists_{tree} x.P \mid \forall_{loc} x.P \mid \forall_{path} x.P \mid \forall_{res} x.P \mid \forall_{tree} x.P \end{array}$$

Si l'on compare les formules logiques présentées ici avec celles de la définition 2.7, on remarque que le principal ajout concerne les quantifications sur les ressources et les arbres de ressources, pour pouvoir s'intéresser syntaxiquement à l'arbre que l'on manipule. Une autre différence se situe au niveau des propositions. On étend aussi les propositions atomiques à l'ensemble α qui est défini comme suit : $\alpha ::= (E = E) \mid E \mid p$.

La sémantique des assertions est donnée par une relation de forcing $\models_{\mathfrak{A}}$ de la forme $s, t \models_{\mathfrak{A}} P$ qui signifie qu'une formule P est vérifiée pour une pile s et un arbre de ressources t. Les clauses sémantiques définies ici s'inspirent largement du modèle d'arbres partiel présenté en définition 2.8.

Définition 2.18 (Clauses sémantiques).

- $s, t \models_{\mathfrak{A}} [l]P$ *ssi il existe* t' *tel que* $(e, l \mapsto t') \preccurlyeq t$ *et* $s, t' \models_{\mathfrak{A}} P$.
- $s, t \models_{\mathfrak{A}} \exists_{loc} x.P$ *ssi il existe* l *tel que* $[s|x \mapsto l], t \models_{\mathfrak{A}} P$.
- $s, t \models_{\mathfrak{A}} \forall_{loc} x.P$ *ssi pour tout emplacement* l, $[s|x \mapsto l], t \models_{\mathfrak{A}} \phi$.
- $s, t \models_{\mathfrak{A}} \exists_{path} x.P$ *ssi il existe* $l_1 \ldots l_n$ $(n \in \mathbb{N})$, $[s|x \mapsto (l_1, \ldots l_n)], t \models_{\mathfrak{A}} \phi$.
- $s, t \models_{\mathfrak{A}} \forall_{path} x.P$ *ssi pour tout chemin* $l_1 \ldots l_n$ $(n \in \mathbb{N})$, $[s|x \mapsto (l_1, \ldots l_n)], t \models_{\mathfrak{A}} P$.
- $s, t \models_{\mathfrak{A}} \exists_{res} x.P$ *ssi il existe* $m \in M$, $[s|x \mapsto m], t \models_{\mathfrak{A}} P$.
- $s, t \models_{\mathfrak{A}} \forall_{res} x.\phi$ *ssi pour tout* $m \in M$, $[s|x \mapsto m], t \models_{\mathfrak{A}} P$.
- $s, t \models_{\mathfrak{A}} \exists_{tree} x.P$ *ssi il existe* $t' \in T_M$, $[s|x \mapsto t'], t \models_{\mathfrak{A}} P$.
- $s, t \models_{\mathfrak{A}} \forall_{tree} x.P$ *ssi pour tout* $t' \in T_M$, $[s|x \mapsto t'], t \models_{\mathfrak{A}} \phi$.
- $s, t \models_{\mathfrak{A}} \top$ *toujours.*
- $s, t \models_{\mathfrak{A}} \bot$ *jamais.*
- $s, t \models_{\mathfrak{A}} E = E'$ *ssi* $[\![E]\!]_s = [\![E']\!]_s$.
- $s, t \models_{\mathfrak{A}} E$ *ssi* $t = [\![E]\!]_s$ *si* $[\![E]\!]_s$ *est un arbre de ressources,* $[\![E]\!]_s \preccurlyeq t$ *sinon.*
- $s, t \models_{\mathfrak{A}} p$ *si* t *vérifie la proposition* p.
- $s, t \models_{\mathfrak{A}} P \vee Q$ *ssi* $s, t \models_{\mathfrak{A}} P$ *ou* $s, P \models_{\mathfrak{A}} Q$.
- $s, t \models_{\mathfrak{A}} P \wedge Q$ *ssi* $s, t \models_{\mathfrak{A}} P$ *et* $s, t \models_{\mathfrak{A}} Q$.
- $s, t \models_{\mathfrak{A}} P \rightarrow Q$ *ssi* $s, t \models_{\mathfrak{A}} P$ *implique* $s, t \models_{\mathfrak{A}} Q$.
- $s, t \models_{\mathfrak{A}} I$ *ssi* $e \preccurlyeq t$.
- $s, t \models_{\mathfrak{A}} P * Q$ *ssi il existe* t', t'' *tels que* $[t'|t'']\!\uparrow$, $t'|t'' \preccurlyeq t$, $s, t' \models_{\mathfrak{A}} P$ *et* $s, t'' \models_{\mathfrak{A}} Q$.
- $s, t \models_{\mathfrak{A}} P \mathrel{-\!\!*} Q$ *ssi pour tout* t' *tel que* $s, t' \models_{\mathfrak{A}} P$ *et* $[t|t']\!\uparrow$, $s, t|t' \models_{\mathfrak{A}} Q$.

Si l'on excepte que la structure d'arbre sur laquelle on raisonne ici est singulièrement différente que celle des pointeurs, le modèle détaillé ci-dessus présente de nombreuses similarités avec le langage d'assertions pour les pointeurs de [75]. De nombreux opérateurs sont communs aux deux modèles et ont un comportement identique si l'on excepte que l'on travaille sur des arbres de ressources et non sur des pointeurs. Nous n'avons pas d'opérateur *de pointage (points to)* permettant de définir qu'une cellule pointe vers une valeur donnée. À la place, nous avons la modalité d'emplacement qui permet de décrire le contenu d'un emplacement. Une autre différence se situe au niveau des quantificateurs. Pour les pointeurs, on utilisait un seul quantificateur alors qu'on propose ici un quantificateur par domaine sémantique couvert par les variables, cette différence n'est cependant pas cruciale, comme nous le verrons dans le chapitre 4.

- Propriétés de la logique d'assertions

L'introduction des expressions pour raisonner directement sur l'arbre introduit des nouvelles propriétés non triviales qui n'étaient pas présentes avec le modèle d'arbre partiel de la définition

2.8. Tout d'abord, il est important de signaler que pour tout $s \in S$ et $t \in T_M$, on a $s, t \models_{\mathfrak{A}} \exists_{tree} x.x$ puisque $[s|x \to t], t \models_{\mathfrak{A}} x$.

On présente ici un autre exemple moins trivial qui illustre le fait qu'une expression définit de manière unique un arbre : la formule $\forall_{tree} x, y.(((x * y) \wedge P) \to x * (x \mathrel{-\!\!*} P))$ est toujours vraie.

En effet, un couple s, t. Soit deux arbres t', t''. Si $[s|x \mapsto t'|y \mapsto t''], t \not\models_{\mathfrak{A}} (x * y) \wedge P)$ alors $[s|x \mapsto t'|y \mapsto t''], t \models_{\mathfrak{A}} (((x*y) \wedge P) \to x*(x \mathrel{-\!\!*} P))$. Sinon on a $[s|x \mapsto t'|y \mapsto t''], t \models_{\mathfrak{A}} (x*y) \wedge P)$, donc $t = t'|t''$ et $[s|(x \mapsto t')|(y \mapsto t'')], t \models_{\mathfrak{A}} P$. On peut alors affirmer que $[s|(x \mapsto t')|(y \mapsto t'')], t \models_{\mathfrak{A}} x * (x \mathrel{-\!\!*} P)$ et conclure. En effet, $t = t'|t''$ et on a bien $[s|(x \mapsto t')|(y \mapsto t'')], t' \models_{\mathfrak{A}} x$ et $[s|(x \mapsto t')|(y \mapsto t'')], t'' \models_{\mathfrak{A}} x \mathrel{-\!\!*} P$ puisque $[s|(x \mapsto t')|(y \mapsto t'')], t'|t'' \models_{\mathfrak{A}} P$.

Nous introduisons maintenant la relation de *conséquence sémantique* qui permet d'indiquer qu'une formule est la conséquence logique d'une autre. Une formule Q est une conséquence logique d'une formule P (ce que l'on note $P \models_{\mathfrak{A}} Q$) si et seulement si pour toute pile s et tout arbre t tels que $s, t \models_{\mathfrak{A}} P$ alors $s, t \models_{\mathfrak{A}} Q$. On suppose bien évidemment que $fv(P) \cup fv(Q) \subseteq dom(s)$ (où $fv(P)$ correspond à l'ensemble des variables libres de P).

La logique d'assertions est une adaptation de BI-Loc qui est elle-même une extension de la logique BI. Elle en reprend donc les règles usuelles concernant l'opérateur $\mathrel{-\!\!*}$. Nous avons en particulier :

$$\frac{P * Q \models_{\mathfrak{A}} Q'}{P \models_{\mathfrak{A}} Q \mathrel{-\!\!*} Q'} \quad \frac{P \wedge Q \models_{\mathfrak{A}} Q'}{P \models_{\mathfrak{A}} Q \to Q'} \quad \frac{P \models_{\mathfrak{A}} P' \quad Q \models_{\mathfrak{A}} Q'}{P * P' \models_{\mathfrak{A}} Q * Q'}$$

De plus on peut établir la règle suivante relative aux emplacements :

$$\frac{[L']P \models_{\mathfrak{A}} [L']P'}{[L]P \models_{\mathfrak{A}} [L]P'}$$

Elle définit un comportement important des emplacements : si une conséquence sémantique est valide à un emplacement (L) donné, on peut changer cet emplacement en (L') sans remettre en cause cette validité.

Enfin, nous introduisons la notion de *pureté*. Les assertions *pures* sont des assertions qui ne raisonnent pas sur l'arbre de ressources que l'on manipule. On peut les définir formellement de la façon suivante :

Définition 2.19 (Assertion pure). *Une assertion est dite* pure *si elle ne contient ni proposition, ni l'unité I, ni modalité d'emplacement.*

Lemme 2.4. Une assertion pure peut être vérifiée sans tenir compte de l'arbre de ressources. Si P est une assertion pure, on a : $s, t \models_{\mathfrak{A}} P$ ssi $\forall t' \in T_M.\ s, t' \models_{\mathfrak{A}} P$

Une assertion pure P est complètement additive. On a :
- $P * Q$ est équivalent à $P \wedge Q$; - $P \mathrel{-\!\!*} Q$ est équivalent à $P \to Q$.

2.5 Axiomes

On a défini le comportement des commandes et une logique d'assertion pour raisonner sur les arbres et les piles. Nous relions maintenant les deux en exprimant pour chaque commande l'axiome qui lui est associé, un axiome définissant la formule que doit vérifier une configuration avant l'exécution d'une commande (pour vérifier une propriété donnée après exécution de cette commande). Le problème majeur est que de nombreuses commandes nécessitent que l'on sépare complètement un sous-arbre situé sous un emplacement alors que l'opérateur $*$ n'assure pas une telle séparation dans la logique. Contrairement à ce que l'on a par exemple dans la logique des pointeurs [62, 75], $P * Q$ n'assure pas que l'on sépare un arbre de ressources donné en deux sous-arbres disjoints. Cette spécificité de $*$ dont les avantages ont été discutés dans le chapitre 2 mène toutefois à des formules plus complexes pour exprimer la séparation d'un sous-arbre donné du reste de l'arbre.

2.5.1 Sémantique des triplets

Les axiomes introduits dans cette partie des triplets de la forme $\{P\}C\{Q\}$ où P et Q sont des assertions définies comme ci-dessus et où C est une commande. L'interprétation que l'on choisit pour ces triplets permet d'assurer qu'une commande appliquée à un état vérifiant l'assertion P mène à un état vérifiant Q.

Définition 2.20 (Triplets corrects). *Un triplet $\{P\}C\{Q\}$ est dit* correct *si l'on a :*

Pour toute configuration C, s, t telle que $s, t \models_{\mathfrak{A}} P$ et $free(P) \cup free(Q) \subseteq dom(s)$ alors C, s, t est sûre et si $C, s, t \rightsquigarrow^* s', t'$ alors $s', t' \models_{\mathfrak{A}} Q$.

Cette interprétation ne garantit pas la terminaison. Cependant, on est sûr qu'il n'y a pas d'erreur dans le programme à l'exécution.

2.5.2 Axiomes de base

On commence par les règles classiques pour les commandes de contrôle. Ces règles sont les mêmes que celles présentés dans [81] :

$$\frac{\{P \wedge E\}C\{Q\} \quad \{P \wedge (E \rightarrow \bot)\}C'\{Q\}}{\{P\}\text{if } (E) \text{ then } C \text{ else } C'\{Q\}}$$

$$\frac{\{P \wedge E\}C\{Q\} \quad (P \wedge (E \rightarrow \bot)) \rightarrow Q}{\{P\}\text{while } (E) \text{ do } C \text{ end}\{Q\}}$$

On présente ensuite les règles de Hoare standard pour l'enchaînement de commandes, la conséquence et l'affectation.

- Enchaînement

$$\frac{\{P\}C\{Q\} \quad \{Q\}C\{R\}}{\{P\}C; C'\{Q\}}$$

- Conséquence

$$\frac{P \models_{\mathfrak{A}} P' \quad \{P'\}C\{Q'\} \quad Q' \models_{\mathfrak{A}} Q}{\{P\}C\{Q\}}$$

- Affectation simple

$$\{P\{^E/_x\}\}x := E\{P\}$$

Dans la règle pour la conséquence, la relation $\models_{\mathfrak{A}}$ utilisée est la relation de conséquence sémantique. Elle indique donc qu'on peut remplacer une pré-condition par une autre pré-condition plus générale et une post-condition par une autre post-condition moins générale.

On présente maintenant le triplet pour l'affectation à une variable des ressources présentes dans un nœud. Ce triplet est le premier de ceux présentés qui nécessite de *séparer* le contenu de l'arbre de manière précise.

- Observation des ressources contenues

$\{\exists_{res}x_1.(P\{^{x_1}/_x\} \wedge (\forall_{res}y.(y = e) \vee (([E]y * \top) \rightarrow \bot) * [E]x_1))\}x := res(E)\{P\}$

La pré-condition doit s'assurer que la ressource x_1 correspond bien à la ressource présente à l'emplacement E, il est donc nécessaire de pouvoir déterminer par notre assertion l'ensemble des ressources présentes en E. Pour cela, on décompose notre arbre de ressources en deux sous-arbres : l'un qui ne contient que l'emplacement E et ces ressources et l'autre qui ne contient aucune ressource à cet emplacement. La difficulté consiste à trouver une formulation logique qui corresponde à ce second sous-arbre. Il faut pour cela s'assurer que l'on ne peut trouver dans ce sous-arbre autre chose que la ressource e à l'emplacement E, cela se caractérise par la sous-formule : $\forall_{res}y.((y = e) \vee (([E]y * \top) \rightarrow \bot)$. Cette dernière sous-formule indique que soit la ressource à l'emplacement E est égale à e, l'élément neutre du monoïde, soit l'arbre est indéfini s'il contient une autre ressource. Notre arbre comprend donc ce sous-arbre et celui est constitué par la ressource x_1 en E.

- Observation de l'arbre contenu

$\{\exists_{tree}x_1.(P\{^{x_1}/_x\} \wedge (no(E) * [E]x_1))\}x := tree(E)\{P\}$ (avec $no(E) = ([E]\top * \top) \rightarrow \bot)$)

Là aussi, on doit séparer le sous-arbre présent en E du reste de l'arbre. Cette séparation est plus simple puisque le sous-arbre restant ne contient donc plus l'emplacement E. On décompose donc notre arbre en un sous-arbre sans l'emplacement E et un sous-arbre contenant exactement le sous-arbre sous cet emplacement.

- Création d'emplacement

$\{P \wedge exists(E) \wedge no(E : l)\}x := new_{loc}E\{P * [x]e \wedge x = (E : l)\}$ (avec $exists(E) = [E]\top * \top$)

La pré-condition assure que l'emplacement l que l'on va créer sous le chemin E n'existe pas déjà mais que E existe. La post-condition crée cet emplacement et place le chemin correspondant dans la variable x. Le problème de cet axiome est qu'il nécessite de donner explicitement le nom de l'emplacement qui va être créé alors que l'on souhaite pouvoir le choisir *arbitrairement*. On présente plus loin dans ce chapitre un *axiome arrière* pour la création d'emplacement qui évite ce problème.

- Suppression d'emplacement

$\{P \wedge no(E) * [E]\top\}dispose_{loc}E\{P\}$

On sépare dans la pré-condition l'emplacement à supprimer et son contenu du reste de l'arbre et seul ce qui concerne cette deuxième partie est vérifié dans la post-condition.

- Modifier et ajouter du contenu

$\{(\exists_{res}x.x = E_2) \wedge \exists_{res}y.((P \wedge \forall_{res}x.((x = e) \vee ((\top * [E_1]x) \rightarrow \bot))) * [E_1]y)\}$
$x := update_{res}(E_1, E_2)\{P * [E_1]E_2\}$

$\{(\exists_{tree}x.x = E_2) \wedge (P \wedge no(E_1) * [E_1]\top)\}x := update_{tree}(E_1, E_2)\{P * [E_1]E_2\}$

$\{(\exists_{tree} x.x = E_2) \wedge P \wedge exists(E_1)\} x := add(E_1, E_2)\{P * [E_1]E_2\}$

On rencontre pour la modification les mêmes difficultés que pour l'observation d'un emplacement : on doit isoler le contenu à modifier du reste de l'arbre. On décompose donc l'arbre en un sous-arbre contenant le contenu à remplacer et le reste de l'arbre. La post-condition correspond alors au reste de l'arbre auquel on ajoute les données correspondante à E_2.

La commande d'ajout est plus simple puisqu'il suffit d'ajouter l'arbre (éventuellement réduit à une ressource) à l'emplacement existant.

2.5.3 Axiomes arrières

Certains des axiomes présentés ci-dessus nécessitent une forme particulière à la fois pour la pré-condition et pour la post-condition. On souhaite trouver une écriture des axiomes qui est moins contraignante. On propose pour cela des *axiomes arrières* qui n'imposent pas une forme spéciale pour la post-condition.

Un *axiome arrière* décrit donc dans sa pré-condition comment la structure actuelle doit évoluer pour satisfaire la post-condition. On peut donc raisonner à partir de la post-condition que l'on cherche pour définir la pré-condition que doit satisfaire le programme.

- Création arrière d'un emplacement

$\{\forall_{loc} x'.exists(E) \wedge (no(E : x') \rightarrow ([E][x']e -\!\!* P\{^{E:x'}/_x\})\} x := new_{loc} E\{P\}$

Si l'on raisonne à partir de la post-condition pour la création d'emplacement, on doit démontrer que l'arbre actuel est tel que si on lui ajoute un nouvel emplacement au chemin donné, on obtiendra P. Dans la pré-condition, le quantificateur $\forall_{loc}$ indique que l'on peut choisir n'importe quel nom d'emplacement et la sous-formule $no(E : x')$ assure que l'emplacement choisi doit être un nouvel emplacement. Ainsi, si on ajoute cet emplacement à l'arbre actuel, on obtient P. L'utilisation du quantificateur $\forall_{loc}$ dans cet axiome permet donc d'éviter le problème du choix du nom d'emplacement que l'on avait précédemment.

- Ajout arrière de contenu

$\{(\exists_{tree} x.x = E_2) \wedge (exists(E_1) \wedge ([E_1]E_2 -\!\!* P))\} x := add(E_1, E_2)\{P\}$

Cet axiome arrière est le plus simple. Si l'emplacement où l'on doit ajouter du contenu existe, alors en composant l'arbre existant avec ce contenu, on obtient un arbre qui vérifie P.

- Modification arrière du contenu

$\{(\exists_{res} x.x = E_2) \wedge \exists_{res} y.[E_1]y * ((\forall_{res} x.((x = e) \vee (([E_1]x * \top) \rightarrow \bot))) \wedge ([E_1]E_2 -\!\!* P))\} update_{res}(E_1, E_2)\{P\}$

$\{(\exists_{tree} x.x = E_2) \wedge ([E_1]\top * (no(E_1) \wedge ([E_1]E_2 -\!\!* P)))\} update_{tree}(E_1, E_2)\{P\}$

Pour qu'une commande de modification mène à un arbre satisfaisant P, on doit séparer les données à modifier du reste de l'arbre et vérifier que ce reste vérifie P quand on lui ajoute le contenu de E_2.

2.6 Plus faibles pré-conditions

Commençons par définir ce qu'est une *plus faible pré-condition.* Il s'agit de définir, à partir d'une post-condition et d'une instruction donnée, l'ensemble des configurations qui vérifieront la post-condition si on leur applique cette commande.

Définition 2.21 (Plus faible pré-condition). *Soit une commande C et une formule Q la* plus faible pré-condition *$wp(C,Q)$ du couple C,Q est l'ensemble des configurations telles que :*
$s,h \in wp(C,Q)$ si C,s,h est sûre et si $C,s,h \rightsquigarrow s',h'$ implique $s,h' \models_{\mathfrak{A}} Q$.

Un axiome $\{P\}C\{Q\}$ est dit *correct* s'il est tel que toutes les configurations s,h vérifiant $s,h \models_{\mathfrak{A}} P$ vérifient $s,h \in wp(C,Q)$. Il est dit *complet* si toutes les configurations $s,h \in wp(C,Q)$ vérifient $s,h \models_{\mathfrak{A}} P$.

2.6.1 Correction

On démontre maintenant que les axiomes présentés précédemment sont tous corrects, qu'ils soient arrières ou non.

Lemme 2.5. Les axiomes de bases donnés en section 2.5 sont corrects d'après les clauses sémantiques.

Preuve. Par cas sur les différents axiomes. Soit un couple s,t :

- Cas $\{P\{^E/_x\}\}x := E\{P\}$:
  Supposons que $s,t \models_{\mathfrak{A}} P\{^E/_x\}$. On a donc $[s,x \mapsto E],t \models_{\mathfrak{A}} P$. Or, d'après la sémantique de $\rightsquigarrow$, $x := E, s,t \rightsquigarrow [s,x \mapsto E],t$. On peut donc conclure.
- Cas $\{\exists_{res}x_1.(P\{^{x_1}/_x\} \wedge (\forall_{res}y.(y=e) \vee (([E]y * \top) \rightarrow \bot) * [E]x_1))\}x := res(E)\{P\}$:
  Supposons que $s,t \models_{\mathfrak{A}} \exists_{res}x_1.(P\{^{x_1}/_x\} \wedge (\forall_{res}y.(y=e) \vee (([E]y * \top) \rightarrow \bot) * [E]x_1))$. D'après la sémantique de $\exists_{res}$, il existe donc m tel que $s,t \models_{\mathfrak{A}} P\{^m/_x\} \wedge (\forall_{res}y.(y = e) \vee (([E]y * \top) \rightarrow \bot) * [E]m)$. Par conséquent, l'emplacement E existe et la configuration C,s,t n'est pas bloquée.

  De plus, la formule ci-dessus assure que la ressource contenue par t à l'emplacement E est m. Par conséquent, $x := res(E), s,t \rightsquigarrow [s,x \mapsto m],t$ et on a bien : $[s,x \mapsto m],P \models_{\mathfrak{A}} P$.
- Cas $\{\exists_{tree}x_1.(P\{^{x_1}/_x\} \wedge (no(E) * [E]x_1))\}x := tree(E)\{P\}$:
  Supposons que $s,t \models_{\mathfrak{A}} \exists_{tree}x_1.(P\{^{x_1}/_x\} \wedge (no(E) * [E]x_1))$. Il existe donc t' et $L = [\![E]\!]_s$ tel que l'on décompose t en un sous-arbre $(e,(L \mapsto t'))$ et un sous-arbre t'' tel que $t''(L)$ n'est pas défini. Alors on a bien $[s,x_1 \mapsto t'],t \models_{\mathfrak{A}} P$.
- Cas $\{P \wedge exists(E) \wedge no(E:l)\}x := new_{loc}E\{P * [x]e \wedge x = (E:l)\}$:
  On considère la configuration $x = new_{loc}E, s,t$. Par définition, $new_{loc}E,s,t \rightsquigarrow [s|x \mapsto L : l],t|(e,L:l \mapsto (e,nil))$ avec $[\![E]\!]_s = L$ et où $L:l$ n'est pas un chemin de t. Supposons que $s,t \models_{\mathfrak{A}} P \wedge no(E:l)$ Par définition, $[\![E]\!]_s$ existe et donc la configuration n'est pas bloquée. De plus, $[s|x \mapsto L:l],t|(e,L:l \mapsto (e,nil) \models_{\mathfrak{A}} x = L:l$ et $[s|x \mapsto L:l],t|(e,L:l \mapsto (e,nil) \models_{\mathfrak{A}} P * [L][l]e$ donc $[s|x \mapsto L:l],t|(e,L:l \mapsto (e,nil) \models_{\mathfrak{A}} (x = L:l) \wedge (P * [L][l]e)$. On remarque au passage que pour ce cas, on a supposé que l'emplacement l était celui qui avait été choisi pour être créé, conformément à ce qu'on avait indiqué lors de la présentation de l'axiome.
- Cas $\{P \wedge no(E) * [E]\top\}dispose_{loc}E\{P\}$:
  Supposons que $s,t \models_{\mathfrak{A}} P \wedge no(E) * [E]\top$. $[\![E]\!]_s$ est donc un chemin (que nous noterons L)

et la configuration $dispose_{loc}E, s, t$ n'est pas bloquée. De plus, par définition des clauses sémantiques, il existe t', t'' tels que $s, t' \models_{\mathfrak{A}} P \wedge no(E)$ et $s, t'' \models_{\mathfrak{A}} [E]\top$. Donc par définition, $t'(L)$ n'est pas défini et t'' n'est défini qu'en l et on a : $dispose_{loc}E, s, t \rightsquigarrow s, t'$. Et par définition de $\wedge$, on a $s, t' \models_{\mathfrak{A}} P$.

- Cas $\begin{array}{l}\{(\exists_{res}x.x = E_2) \wedge \exists_{res}y.P \wedge \forall_{res}x.((x = e) \vee ((\top * [E_1]x) \to \bot)) * [E_1]y\} \\ x := update_{res}(E_1, E_2)\{P * [E_1]E_2\}\end{array}$:
  Supposons que $s, t \models_{\mathfrak{A}} (\exists_{res}x.x = E_2) \wedge \exists_{res}y.P \wedge \forall_{res}x.((x = e) \vee ((\top * [E_1]x) \to \bot)) * [E_1]y$. Par conséquent, d'après les clauses sémantiques, $[\![E_1]\!]_s = L$ est un chemin et $[\![E_2]\!]_s = m$ est une ressource. De plus, on peut séparer t en un sous-arbre t' vérifiant $s, t' \models_{\mathfrak{A}} P \wedge \forall_{res}x.((x = e) \vee ((\top * [E_1]x) \to \bot)$ et un sous-arbre t'' vérifiant, $s, t'' \models_{\mathfrak{A}} \exists_{res}y.[E_1]y$. Il existe donc une ressource m et un arbre $t^{(3)}$ tel que $t'' = (e, L \mapsto (m, t^{(3)}))$. Comme pour le cas de l'affectation de variable, la formule que vérifie t' assure qu'il ne contient pas de ressources à l'emplacement L. Donc on a $update(E_1, E_2), s, t \rightsquigarrow t'|[L]m$ et $t'|[L]m \models_{\mathfrak{A}} P * [E_1]E_2$.
- Cas $\{(\exists_{tree}x.x = E_2) \wedge (P \wedge no(E_1) * [E_1]\top)\}x := update_{tree}(E_1, E_2)\{P * [E_1]E_2\}$:
  Supposons que $s, t \models_{\mathfrak{A}} (\exists_{tree}x.x = E_2) \wedge (P \wedge no(E_1) * [E_1]m)$. Par conséquent, d'après les clauses sémantiques, $[\![E_1]\!]_s = L$ est un chemin et $[\![E_2]\!]_s = t'$ est un arbre. De plus, il existe deux arbres u et u' tels que $s, u \models_{\mathfrak{A}} P \wedge no(E_1)$ et $s, u' \models_{\mathfrak{A}} [E_1]\top$. Donc u ne contient rien en E_1 et u' contient uniquement un sous-arbre en E_1. On a donc $update_{tree}(E_1, E_2), s, t \rightsquigarrow s, u|(e, L \mapsto t')$ et par définition, $u|(e, L \mapsto t') \models_{\mathfrak{A}} P * [E_1]E_2$.
- Cas $\{(\exists_{tree}x.x = E_2) \wedge P \wedge exists(E_1)\}x := add(E_1, E_2)\{P * [E_1]E_2\}$:
  Supposons que $s, t \models_{\mathfrak{A}} (\exists_{tree}x.x = E_2) \wedge P \wedge exists(E_1)$. Donc $[\![E_1]\!]_s = L$ est un chemin et $[\![E_2]\!]_s = t'$ est un arbre. De plus, on a donc $add(E_1, E_2), s, t \rightsquigarrow s, t|(e, L \mapsto t')$. et par définition, $s, (e, L \mapsto t') \models_{\mathfrak{A}} [E_1]E_2$. Donc $s, t|(e, L \mapsto t') \models_{\mathfrak{A}} P * [E_1]E_2$.

□

On obtient les mêmes résultats pour les axiomes arrières que l'on a introduit par la suite.

Lemme 2.6. Les axiomes arrières de la section 2.5.3 sont corrects d'après les clauses sémantiques.

Preuve. Par cas sur les différents axiomes. Soit un couple s, t :

- Cas $\{\forall_{loc}x'.exists(E) \wedge (no(E : x') \to ([E][x']e \mathrel{-\!\!*} P\{^{E:x'}/_x\}))\}x := new_{loc}E\{P\}$:
  Supposons que $s, t \models_{\mathfrak{A}} \forall_{loc}x'.exists(E) \wedge (no(E : x') \to ([E][x']e \mathrel{-\!\!*} P\{^{E:x'}/_x\})$. On sait donc que $[\![E]\!]_s$ correspond à un chemin que l'on nomme L. On a donc $x := new_{loc}E, s, t \rightsquigarrow [s, x \mapsto L : l], t|(e, L : l \mapsto (e, nil))$ avec $t(L : l)$ non défini. Comme $t(L : l)$ est non défini on a donc $s, t \models_{\mathfrak{A}} [E][l]e \mathrel{-\!\!*} P\{^{E:l}/_x\})$ donc $[s, x \mapsto L : l], t|(e, L : l \mapsto (e, nil)) \models_{\mathfrak{A}} P$.
- Cas $\{(\exists_{tree}x.x = E_2) \wedge (exists(E_1) \wedge ([E_1]E_2 \mathrel{-\!\!*} P))\}x := add(E_1, E_2)\{P\}$:
  Supposons que $s, t \models_{\mathfrak{A}} (\exists_{tree}x.x = E_2) \wedge (exists(E_1) \wedge ([E_1]E_2 \mathrel{-\!\!*} P))$. La formule assure que $[\![E_2]\!]_s$ correspond à un arbre (noté t'), $[\![E_1]\!]_s$ à un chemin noté L, et que $t(L)$ est défini. Donc la configuration n'est pas bloquée et on a $add(E_1, E_2), s, t \rightsquigarrow s, t|(e, L \mapsto t')$. Comme $s, t \models_{\mathfrak{A}} [E_1]E_2 \mathrel{-\!\!*} P$, $s, t|(e, L \mapsto t') \models_{\mathfrak{A}} P$.
- Cas $\begin{array}{l}\{(\exists_{res}x.x = E_2) \wedge \exists_{res}y.[E_1]y * ((\forall_{res}x.((x{=}e) \vee (([E_1]x * \top) \to \bot))) \wedge ([E_1]E_2 \mathrel{-\!\!*} P))\} \\ update_{res}(E_1, E_2)\{P\}\end{array}$:

  Supposons que $s, t \models_{\mathfrak{A}} (\exists_{res}x.x = E_2) \wedge (\exists_{res}y.[E_1]y * \forall_{res}x.((x = e) \vee (([E_1]x * \top) \to \bot)) \wedge ([E_1]E_2 \mathrel{-\!\!*} P))$. La formule indique que $[\![E_2]\!]_s$ correspond à une ressource (noté m), $[\![E_1]\!]_s$ à un chemin noté L, et que $t(L)$ est défini. On assure aussi que t se décompose en deux sous-arbres. Un sous-arbre t' tel que $s, t' \models_{\mathfrak{A}} \exists_{res}y.[E_1]y$ et un sous-arbre t'' tel que $s, t'' \models_{\mathfrak{A}} \forall_{res}x.((x = e) \vee (([E_1]x * \top) \to \bot)) \wedge ([E_1]E_2 \mathrel{-\!\!*} P))$. On a donc $s, t'' \models_{\mathfrak{A}} \forall_{res} x.(x = e) \vee (([E_1]x * \top) \rightarrow \bot)$. Cette clause sémantique indique que le deuxième sous-arbre ne contient pas de ressource en L. On peut donc en déduire que $update_{res}(E_1, E_2), s, t \rightsquigarrow s, t''|(e, L \mapsto (m, nil))$. Et comme $s, t'' \models_{\mathfrak{A}} [E_1]E_2 -\!\!* P$, $s, t''|(e, L \mapsto (m, nil)) \models_{\mathfrak{A}} P$.

- Cas $\{(\exists_{tree} x.x = E_2) \wedge ([E_1]\top * (no(E_1) \wedge ([E_1]E_2 -\!\!* P)))\} update_{tree}(E_1, E_2)\{P\}$:
Supposons que $s, t \models_{\mathfrak{A}} (\exists_{tree} x.x = E_2) \wedge ([E_1]\top * (no(E_1) \wedge ([E_1]E_2 -\!\!* P)))$. Par définition des clauses sémantiques, $[\![E_2]\!]_s$ correspond à un arbre (noté u), $[\![E_1]\!]_s$ à un chemin noté L, et que $t(L)$ est défini. Elles assurent aussi que t se décompose en deux sous-arbres. Un sous-arbre t' tel que $s, t' \models_{\mathfrak{A}} [E_1]\top$ et un sous-arbre t'' tel que $s, t'' \models_{\mathfrak{A}} no(E_1) \wedge ([E_1]E_2 -\!\!* P)$. Donc t ne contient plus de sous-arbre en L et comme la configuration $update_{tree}(E_1, E_2), s, t$ n'est pas bloquée, on a : $update_{tree}(E_1, E_2), s, t \rightsquigarrow s, t''|(e, L \mapsto u)$. Et comme $s, t'' \models_{\mathfrak{A}} [E_1]E_2 -\!\!* P$ et $s, (e, L \mapsto u) \models_{\mathfrak{A}} [E_1]E_2$, on a $s, t''|(e, L \mapsto u) \models_{\mathfrak{A}} P$.

□

2.6.2 Complétude

On démontre maintenant que tous les axiomes arrières sont complets. Les axiomes arrières que nous traitons ici recouvrent l'ensemble des axiomes présentés dont la post-condition n'a pas de forme particulière.

Lemme 2.7. Les axiomes arrières sont complets d'après les clauses sémantiques.

Preuve. On montre par cas pour chaque axiome $\{P\}C\{Q\}$ que chaque configuration $s, t \in wp(C, Q)$ vérifie $s, t \models_{\mathfrak{A}} P$.

- Cas $s, t \in wp(x = E, P)$: Par définition, $[s, x \mapsto E], t \models_{\mathfrak{A}} P$, donc $s, t \models_{\mathfrak{A}} P\{^E/_x\}$.
- Cas $s, t \in wp(x := res(E), P)$: Par définition, on a $[s, x \mapsto m], t \models_{\mathfrak{A}} P$ (avec $t(L) = (m, t')$). On a également $s, t \models_{\mathfrak{A}} P\{^m/_x\}$. De plus, il existe t_1 tel que l'on puisse décomposer t de la façon suivante : $t = t_1|(e, L \mapsto (m, nil))$. Il est alors évident que l'arbre t_1 vérifie $t_1(L) = e$. Par conséquent, on a bien $s, t_1 \models_{\mathfrak{A}} \forall_{res} y.((y = e) \vee (([E]y * \top) \rightarrow \bot))$. On a également : $s, (e, L \mapsto (m, nil)) \models_{\mathfrak{A}} [E]m$ et par conséquent, on a bien $s, t \models_{\mathfrak{A}} \forall_{res} y.((y = e) \vee (([E]y * \top) \rightarrow \bot)) * [E]m$.
- Cas $s, t \in wp(x := tree(E), P)$: Par définition, $[s, x \mapsto t'], t \models_{\mathfrak{A}} P$ et il existe t, L tels que $t(L) = t'$ et $L = [\![E]\!]_s$. Donc il existe t'' tel que $t = t''|(e, L \mapsto t')$. Par conséquent, on a $s, (e, L \mapsto t') \models_{\mathfrak{A}} \exists_{tree} x_1.[E]x_1$ et $s, t'' \models_{\mathfrak{A}} no(E)$. On a donc $s, t \models_{\mathfrak{A}} \exists_{tree} x_1.(\phi\{^{x_1}/_x\} \wedge (no(E) * [E]x_1))$.
- Cas $s, t \in wp(x := new_{loc} E, P)$: Par définition, on a $[s|x \mapsto L : l], t|(e, L : l \mapsto (e, nil)) \models_{\mathfrak{A}} P$ où $L = [\![E]\!]_s$ et $t(L : l)$ n'est pas défini. Par définition de $-\!\!*$, on a : $[s|x \mapsto L : l], t \models_{\mathfrak{A}} [L][l]I \models_{\mathfrak{A}} P$. Comme $t(L : l)$ n'est pas défini, on a donc : $[s|x \mapsto L : l], t \models_{\mathfrak{A}} no(L : l) \rightarrow [L][l]I -\!\!* P$ et comme ces résultats sont vrais pour tout emplacement l, on obtient finalement $s, P \models_{\mathfrak{A}} \forall_{loc} x'.(no(E : x') \rightarrow ([E][x']I -\!\!* \phi\{^{E:x'}/_x\}))$.
- Cas $s, t \in wp(x := add(E_1, E_2), P)$: Par définition, $s, t|(e, L \mapsto (m, nil)) \models_{\mathfrak{A}} P$ avec $m = [\![E_2]\!]_s$, $L = [\![E_1]\!]_s$ et $[t(L)]\uparrow$. Par conséquent, on a $s, t \models_{\mathfrak{A}} [E_1]E_2 -\!\!* P$ et comme $t(L)$ est défini, on a également $s, t \models_{\mathfrak{A}} exists(E_2)$. On a donc bien $s, t \models_{\mathfrak{A}} exists(E_1) \wedge ([E_1]E_2 -\!\!* \phi)$.
- Cas $s, P \in wp(x := update_{res}(E_1, E_2), \phi)$: Par définition, on a donc $s, t'|(e, L \mapsto (m, nil)) \models_{\mathfrak{A}} P$ où $L = [\![E_1]\!]_s$, $m = [\![E_2]\!]_s$ et t' est tel que $t'(L) = (e, f)$ et il existe m' tel que $t = t'|(e, L \mapsto (m', nil))$. Comme $t'(L) = (e, f)$, on a $s, t' \models_{\mathfrak{A}} \forall_{res} x.((x = e) \vee (([E_1]x * \top) \rightarrow \bot))$ et comme $s, t'|(e, L \mapsto (m, nil)) \models_{\mathfrak{A}} P$, on a donc : $s, t' \models_{\mathfrak{A}} \forall_{res} x.(x = e) \vee (([E_1]x * \top) \rightarrow$

$\bot) \wedge ([E_1]E_2 \mathrel{-\!\!*} P)$. Enfin, comme $t = t'|(e, L \mapsto (m', nil))$, $s, t \models_{\mathfrak{A}} \exists_{res} y.[E_1]y * (\forall_{res} x.((x = e) \vee (([E_1]x * \top) \rightarrow \bot)) \wedge ([E_1]E_2 \mathrel{-\!\!*} P))$.

- Cas $s, P \in wp(x := update_{res}(E_1, E_2), P)$: Par définition, on a donc $s, t_1|(e, L \mapsto t')) \models_{\mathfrak{A}} P$ où $L = [\![E_1]\!]_s$, $t' = [\![E_2]\!]_s$ et t_1 est tel que $t_1(L)$ n'est pas défini et il existe t_2 tel que $t = t_1|(e, L \mapsto t_2)$. Comme $t_1(L)$ n'est pas défini, on a $s, t_1 \models_{\mathfrak{A}} no(L)$ et comme $s, t_1|(e, L \mapsto t')) \models_{\mathfrak{A}} P$, on a donc : $s, t_1 \models_{\mathfrak{A}} no(L) \wedge ([E_1]E_2 \mathrel{-\!\!*} P)$. Enfin, comme $t = t_1|(e, L \mapsto t_2)$, on obtient : $s, t \models_{\mathfrak{A}} \exists_{tree} y.[E_1]y * (no(E_1) \wedge ([E_1]E_2 \mathrel{-\!\!*} P))$.

$\square$

Les résultats de correction et de complétude établis pour les axiomes arrières permettent d'établir que les pré-conditions des axiomes sont minimales.

Théorème 2.1. *Soient une commande C et une post-condition Q, et un triplet $\{P\}C\{Q\}$ construit à partir des axiomes. Soit une pile s et un arbre de ressource t, on a : $s, t \in wp(C, Q)$ si et seulement si $s, t \models_{\mathfrak{A}} P$.*

Preuve. Le résultat est un conséquence directe des résultats de correction des lemmes 2.5 et 2.6 et du résultat de complétude du lemme 2.7. $\square$

2.7 La propriété de fenêtrage

On cherche maintenant à définir à quel point un programme donné est indépendant du contexte dans lequel il s'exécute. Pour une configuration donnée, on souhaite donc déterminer à quel point on peut ajouter des informations sans que ces dernières ne soient modifiées par l'exécution du programme ou ne le modifient. Cette possibilité d'étendre le contexte est appelée propriété de fenêtrage *(frame property)*. Concrètement, on cherche les conditions nécessaires sur un programme C, et les assertions P et P' pour que si on a $\{P\}C\{Q\}$ alors on ait $\{P*P'\}C\{Q*P'\}$.

2.7.1 Les restrictions imposées

On doit tout d'abord faire face à un problème de variables qui a déjà été signalé dans les travaux sur les tas de pointeurs [75]. On ne doit pas étendre le contexte en introduisant des variables dont la valeur sera modifiée par les commandes. Supposons qu'une commande C modifie la valeur d'une variable z et que l'on ait $\{P\}C\{Q\}$. On ne peut alors pas avoir $\{P*[z]p\}C\{Q*[z]p\}$ puisque la valeur de z va être modifiée et que l'on aura donc pas nécessairement $[z]p$ après exécution de C.

Un autre problème plus spécifique est dû au fait que la composition d'arbres réalise en fait une fusion des emplacements en commun. Par conséquent le contexte que l'on ajoute peut modifier une partie de l'arbre qui sera utilisée par le programme. Ce problème est d'autant plus gênant que l'on ne peut déterminer facilement les sous-arbres qui seront modifiés ou non par un programme. Nous devons donc nous assurer que le contexte ajouté ne modifie pas le contenu d'un emplacement que l'on doit lire ou modifier. Prenons par exemple le triplet $\{P\}dispose_{loc}E\{Q\}$. Le triplet $\{P * [E]p\}dispose_{loc}E\{Q * [E]p\}$ ne sera pas correct puisque l'emplacement E n'existe plus après l'exécution de la commande *dispose*.

Le problème peut être facilement résolu pour une commande atomique puisqu'on sait exactement à quelle partie de l'arbre on va accéder. Par contre, dès que l'on enchaîne plusieurs commandes, il devient impossible de déterminer clairement les sous-arbres qui seront concernés.

Enfin, le dernier problème provient de la nécessité de créer un nouveau nom quand on crée un nouvel emplacement dans l'arbre. Si la pré-condition assure que le nom créé n'est pas dans l'arbre initial, il se peut qu'il ne soit plus nouveau si on rajoute des informations, le triplet ne respecterait pas alors la sémantique des commandes.

2.7.2 Conditions suffisantes pour le fenêtrage

Les problèmes énumérés ci-dessus restreignent considérablement la propriété de fenêtrage pour le langage de manipulation des arbres proposé en section 2.4. Le plus gênant est le cas de la commande de création d'emplacement. On propose donc uniquement une règle de fenêtrage pour l'ensemble du langage sans la création d'emplacement.

Pour résoudre le problème de la modification des sous-arbres auxquels on accède, une solution suffisante est d'assurer que le contexte que l'on ajoute ne peut pas avoir d'emplacement en commun avec celui déjà en place. On est alors assuré qu'une commande qui modifie l'arbre initial ne modifie pas celui ajouté par le contexte. Soit P la pré-condition initiale de notre programme et P' la pré-condition du contexte que l'on ajoute. Pour s'assurer que des arbres satisfaisant P ne peuvent pas avoir de chemin commun avec des arbres vérifiant P', on doit s'assurer que tout arbre vérifiant $P * P'$ vérifie également $\exists_{loc} x.((P \wedge exists(x)) * (P' \wedge exists(x))) \to \perp$. Cette deuxième formule permet de vérifier exactement ce que l'on veut imposer : on ne peut avoir d'emplacement commun à un arbre vérifiant P et à un arbre vérifiant P'.

Pour le problème des variables, il suffit de vérifier que l'assertion définissant le contexte P' ne contient pas de variable libre modifiée par C. L'ensemble des variables modifiées par une commande C est noté $Modified(C)$.

Théorème 2.2 (Propriété de fenêtrage). *Si l'on considère les commandes de la section 2.4 à l'exception de la commande new_{loc}, la règle suivante est correcte si les arbres de ressources que l'on manipule ne contiennent pas de ressources à leur racine :*

$$\frac{\{(P * P') \wedge (\exists_{loc} x.((P \wedge exists(x)) * (P' \wedge exists(x))) \to \perp)\}C\{Q * P'\}}{\{P\}C\{Q\}} *$$
$$* : fv(P') \cap Modified(C) = \emptyset$$

Preuve. Soit un triplet $\{P\}C\{Q\}$, on montre par induction structurelle sur C que l'on a $\{(P * P') \wedge (\exists_{loc} x.((P \wedge exists(x)) * (P' \wedge exists(x))) \to \perp)\}C\{Q * P'\}$.

- Cas $dispose_{loc} E$: Soit une configuration s, t telle que $s, t \models_{\mathfrak{A}} (P * P') \wedge (\exists_{loc} x.((P \wedge exists(x)) * (P' \wedge exists(x))) \to \perp)$. Par définition de $*$, il existe donc t_1, t_2 tels que $s, t_1 \models_{\mathfrak{A}} P$ et $s, t_2 \models_{\mathfrak{A}} P'$. Par hypothèse, comme $s, t_1 \models_{\mathfrak{A}} P$, on a donc $C, s, t_1 \rightsquigarrow s', t_1'$ avec $s, t_1' \models_{\mathfrak{A}} Q$. De plus, par définition de new_{loc}, $t_1(\llbracket E \rrbracket_s)$ est défini. De plus, comme on a $s, t_1 | t_2 \models_{\mathfrak{A}} \exists_{loc} x.((P \wedge exists(x)) * (P' \wedge exists(x)) \to \perp$, t_1 et t_2 sont disjoints. $t_2(\llbracket E \rrbracket_s)$ n'est donc pas défini. Comme new_{loc} n'affecte que cet emplacement, t_2 n'est donc pas modifié par la commande et on a donc $s', t_1' | t_2 \models_{\mathfrak{A}} Q * P'$.
- Cas $x := E$: Cette commande ne modifie que la pile s et comme $fv(P') \cap Modified(C) = \emptyset$, on peut immédiatement conclure.

- Cas $x := res(E)$: Soit une configuration s, t telle que $s, t \models_{\mathfrak{A}} (P * P') \wedge (\exists_{loc} x.((P \wedge exists(x)) * (P' \wedge exists(x))) \rightarrow \bot)$. Par définition de $*$, il existe donc t_1, t_2 tels que $s, t_1 \models_{\mathfrak{A}} P$ et $s, t_2 \models_{\mathfrak{A}} P'$. Par hypothèse, comme $s, t_1 \models_{\mathfrak{A}} P$, on a donc $s, t_1 \models_{\mathfrak{A}} Q$, on a donc $C, s, t_1 \rightsquigarrow s', t'_1$ avec $s, t'_1 \models_{\mathfrak{A}} Q$. De plus, par définition de $x := res(E)$, $t_1(\llbracket E \rrbracket_s)$ est défini. De plus, comme on a $s, t_1 | t_2 \models_{\mathfrak{A}} \exists_{loc} x.((P \wedge exists(x)) * (P' \wedge exists(x)) \rightarrow \bot$, t_1 et t_2 sont disjoints. $t_2(\llbracket E \rrbracket_s)$ n'est donc pas défini. Le résultat de $x := res(E)$ n'est donc pas modifié par t_2 et on a donc $s', t'_1 | t_2 \models_{\mathfrak{A}} Q * P'$.
- Cas $x := tree(E)$: Similaire au cas $x := res(E)$.
- Cas $update_{res}(E_1, E_2)$: Similaire au cas $dispose_{loc} E$.
- Cas $update_{tree}(E_1, E_2)$: Similaire au cas $dispose_{loc} E$.
- Cas $add(E_1, E_2)$: Similaire au cas $dispose_{loc} E$.
- Cas $C'; C''$: Soit une configuration s, t telle que $s, t \models_{\mathfrak{A}} (P * P') \wedge (\exists_{loc} x.((P \wedge exists(x)) * (P' \wedge exists(x))) \rightarrow \bot)$. Par définition de $*$, il existe donc t_1, t_2 tels que $s, t_1 \models_{\mathfrak{A}} P$ et $s, t_2 \models_{\mathfrak{A}} P'$. Par hypothèse d'induction, t_2 ne modifie pas le résultat des commandes $CetC'$ et il n'est pas modifié par ses commandes. Donc si $C'; C'', s, t_1 \rightsquigarrow s', t'_1$ alors $s, t'_1 | t_2 \models_{\mathfrak{A}} P' * Q$.

□

On a introduit dans ce chapitre le modèle d'arbres partiel en expliquant les spécificités des arbres de ressources et de BI-Loc. On a également présenté un langage de manipulation pour ce modèle. La pertinence du modèle présenté repose maintenant sur deux aspects cruciaux.

Tout d'abord, le modèle n'est pertinent que si l'on dispose d'outils pour vérifier les propriétés que l'on énonce. Il s'agit donc de pouvoir décider si une configuration donnée d'une instance du modèle vérifie certaine propriété. On doit pour cela étudier la décidabilité du *model-checking* pour notre modèle. Ceci afin d'avoir une idée des fragments logiques pour lesquels la vérification de propriétés logiques est décidable. De même, le langage de manipulation que l'on propose définit des pré-conditions d'une forme particulière à l'aide d'axiomes arrières. Les pré-conditions ainsi exprimées peuvent alors avoir une forme différente des pré-conditions initialement établies pour une configuration donnée. Il est donc important de pouvoir établir une relation de conséquence entre les conditions, une *conséquence logique* indiquant qu'une formule en implique une autre. Pour cela, il est indispensable de développer des outils de preuve pour la logique d'assertions. Dans le chapitre suivant, on établit un premier pas dans cette direction en proposant une méthode de recherche de preuves pour BI-Loc propositionnelle. Ces aspects sont abordés dans le chapitre 3.

Ensuite, les ressources manipulées dans ce modèle étant très abstraites, il est important de montrer que le modèle peut être instancié pour répondre à des problèmes concrets. Ces aspects seront abordés aux chapitres 4 et 5, où l'on présentera les instances du modèle permettant de raisonner sur les pointeurs, les permissions et les données semi-structurées comme les documents XML.

Chapitre 3

Model-checking et recherche de preuves

On a présenté dans le chapitre précédent le modèle d'arbres partiel et le langage de manipulation des arbres de ressources issus de ce modèle. On propose maintenant d'étudier les outils utiles pour raisonner sur ce modèle. On se focalise dans ce chapitre sur les deux aspects centraux que sont le *model-checking* et la *recherche de preuves*.

Le model-checking [12] consiste à définir si une instance donnée du modèle vérifie une formule de BI-Loc. On propose dans ce chapitre d'étudier la décidabilité du model-checking dans BI-Loc. Il s'agit de déterminer des critères suffisants pour pouvoir affirmer de manière automatique si une formule est vérifiée par une configuration donnée. C'est un point important dans la perspective d'une utilisation de la logique pour la vérification de programmes car elle permet de s'assurer qu'une configuration vérifie bien une pré-condition nécessaire au bon déroulement d'un programme.

On s'intéresse ensuite à la recherche de preuves dont le but est dans le contexte de cette thèse de vérifier la conséquence logique dans BI-Loc. On propose pour cela une méthode des tableaux pour BI-Loc fondée sur celle proposée pour BI par [47]. L'étude de la recherche de preuve permet de proposer des méthodes efficaces pour établir le lien entre deux formules logiques, ce qui est utile pour vérifier qu'une pré-condition d'un programme est la conséquence logique de la formule correspondant à certaines propriétés d'une configuration donnée du modèle.

3.1 Satisfaction et validité par model-checking

La question posée par l'étude de la décidabilité du model-checking est la suivante : peut-on ou non déterminer automatiquement si un arbre de ressources t donné satisfait une formule ϕ, ou si une formule ϕ est satisfaite par tous les arbres d'un modèle (ce qu'on nomme *être valide pour ce modèle*) ?

De nombreux autres travaux sur les logiques spatiales et de séparation ont été proposé afin de décider de la décidabilité [23, 25, 35, 36] de cette logique ou de certains de ces fragments et des problèmes de complexité des fragments décidables [35, 36]. Dans ces logiques, il apparaît que les problèmes de décidabilité sont principalement liés à l'interaction entre l'implication multiplicative $-\!\!*$ et les autres opérateurs logiques. En effet, la sémantique de l'opérateur $-\!\!*$ introduit une quantification infinie puisqu'elle nécessite de vérifier la propriété pour tous les arbres vérifiant la sous-partie gauche de la formule. Il est important de souligner que le rôle clé de l'opérateur $-\!\!*$ dans cette étude avait déjà été identifié comme le cœur du problème dans la preuve de la décidabilité de la logique BI propositionnelle [49], même si l'approche était différente dans ce cadre.

BI-Loc et les modèles d'arbres partiels doivent faire face aux même problèmes que ces autres modèles. Les problèmes de complexité ne sont toutefois pas traités dans ce document et nous nous focalisons uniquement sur les problèmes de décidabilité. Tout d'abord, chaque modèle d'arbres repose sur un modèle de ressources partiel. Nous devons déjà nous assurer que pour ce modèle, la décidabilité est possible. Ensuite, nous devons gérer les problèmes spécifiques aux emplacements. Toute la difficulté des résultats présentés ici est de combiner les arguments de décidabilité présents dans les preuves pour les logiques de séparation et pour les logiques spatiales [23, 35].

Pour ce faire, nous commençons par nous intéresser aux modèles de monoïde partiels, ce qui permet de ne pas avoir à traiter le problème de la gestion des emplacements. Pour cette structure, nous proposons des conditions suffisantes pour assurer la décidabilité par *model-checking* de la satisfaction et de la validité. Puis nous intégrons les emplacements en montrant comment le nombre de chemins à considérer peut être maîtrisé.

3.1.1 Décidabilité de la validité dans BI

L'étude de la décidabilité par model-checking de la validité pour BI est un premier pas vers la décidabilité de la validité pour BI-Loc. Le problème de l'infinité de modèles à vérifier introduit par l'opérateur $-\!\!*$ n'est en effet pas directement lié aux arbres.

Nous commençons par définir les critères d'un monoïde de ressources partiel bornable. Par bornable, nous voulons exprimer le fait qu'il suffit de considérer un modèle fini pour recouvrir les différents cas nécessaires pour vérifier une propriété sur un modèle donné.

Définition 3.1. *Un modèle de ressources partiel* $(M, e, \times, \sqsubseteq)$ *est dit* bornable *pour un ensemble de proposition* Σ *si :*

1. $\forall m \in M.\exists n$ *tel que* $\exists m_1, \ldots, m_n \in M$ *telque* $m \sqsubseteq m_1 \times \ldots \times m_n$ *et il n'existe pas de* $i \in [1..n]$ *tel que* $\exists m', m'' \in M.m_i \sqsubseteq m' \times m'', m' \neq e$ *et* $m'' \neq e$ *;*
2. *Pour tout* $\sigma \subset_{fin} \Sigma$*, et tout entier* n *il existe une relation de congruence* $\cong_{\sigma,n}$ *tel que* $M_{/\cong_{\sigma,n}}$ *est fini et où* $\forall m, m'.m \cong_{\sigma,n} m'$ *ssi* $m = m'$ *ou :*
   - *(a)* $\forall p \in \Sigma.(m \models_{\mathfrak{P}} p \Leftrightarrow m' \models_{\mathfrak{P}} p)$ *et*
   - *(b)* $\forall r, r' \in M.(r \cong_{\sigma,n} r'$ *et* $[m \times r]\!\uparrow) \Rightarrow ([m' \times r']\!\uparrow$ *et* $m \times r \cong_{\sigma,n} m \times m')$ *et*
   - *(c)* $\forall m_1, \ldots, m_k.\ m \sqsubseteq m_1 \times \ldots m_k (k < n) \Rightarrow \exists m'_1, \ldots, m'_k.\ m' \sqsubseteq m'_1 \times \ldots m'_k$ *et* $\forall i \in [1..n] m_i \cong_{\sigma,0} m'_i$ *;*
   - *(d) si* $m \sqsubseteq e$*,* $m' \sqsubseteq e$*.*

Cette définition indique simplement que chaque ressource peut se décomposer en un ensemble fini de ressources *atomiques* (non décomposables) et que, pour chaque sous-ensemble de propositions, il existe une relation de congruence nous permettant de pouvoir nous ramener à un ensemble de ressources fini. Cela n'implique en aucun cas que l'ensemble de ressources du monoïde doit être fini.

Nous prouvons maintenant que pour toute formule de BI, il existe une classe d'équivalence finie qu'il suffit d'utiliser pour vérifier la validité de cette formule. Soit une formule ϕ on appelle σ_ϕ l'ensemble des formules atomiques présentes dans ϕ. De plus nous définissons la *taille* de la façon suivante :

Définition 3.2 (Taille d'une formule de la logique BI). *La* taille d'une formule de BI *est définie inductivement par :*

- $s(\phi' * \phi'') = s(\phi') + s(\phi'')$	- $s(\phi' \wedge \phi'') = max(s(\phi'), s(\phi''))$	- $s(p) = 1$
- $s(\phi' {-\!*} \phi'') = s(\phi'')$	- $s(\phi' \vee \phi'') = max(s(\phi'), s(\phi''))$	- $s(\top) = 0$
- $s(I) = 1$	- $s(\phi' \to \phi'') = max(s(\phi'), s(\phi''))$	- $s(\bot) = 0$

La taille d'une formule détermine le nombre maximal de décompositions d'une ressource qu'il est nécessaire de considérer pour vérifier si une formule ϕ est satisfaite ou non par une ressource donnée. Notre but est de démontrer que pour vérifier la satisfaction d'une formule du type $\phi {-\!*} \psi$ par une ressource ϕ, il suffit de vérifier que $m \times m'$ satisfait ψ pour un sous-ensemble fini de ressources m'.

Pour ce faire, nous commençons par démontrer les résultats suivants qui seront nécessaires pour établir la décidabilité de la satisfaction pour BI.

Lemme 3.1 (Monotonie). Soit un modèle partiel bornable $(M, e, \times, \sqsubseteq)$. Pour tout $m, m' \in M$, pour tout entier n et n' et pour tout $\sigma, \sigma' \subset \Sigma$, $(m \cong_{\sigma,n} m'$, $\sigma' \subseteq \sigma$ et $n' \leq n) \Rightarrow m \cong_{\sigma',n'} m'$

Preuve. On montre que les quatre conditions du point 2 de la définition 3.1 sont vérifiées :

Soit $m, m', n, n', \sigma, \sigma'$ tel que $m \cong_{\sigma,n} m'$, $\sigma' \subseteq \sigma$ et $n' \leq n$, on a bien :

(a) Comme on a $\sigma' \subseteq \sigma$, pour tout $p \in \sigma'$, $p \in \sigma$, et donc pour tout $p \in \sigma'.m \models_{\mathfrak{P}} p' \Rightarrow m' \models_{\mathfrak{P}} p'$ d'après le point 2a de la définition 3.1 appliquée à la relation de congruence $\cong_{\sigma,n}$.

(b) La propriété de composition (point 2b) est immédiate puisque cela ne dépend ni de n ni de σ.

(c) La preuve du point 2c se fait par induction sur n. Pour $n = 0$, $n' = 0$ et le point 2c est trivialement vérifié. S'il existe $m_1, \ldots, m_k$ tels que $m = m_1 \times \ldots \times m_k$ avec $k \leq n'$, par hypothèse d'induction, d'après le point 2c de la définition 3.1, comme $n' \leq n$, on a $k \leq n$ et on sait qu'il existe $m' = m'_1 \times \ldots \times m'_k$ tels que le point 2c est donc vérifié.

(d) Si $m \sqsubseteq e$, comme $m \cong_{\sigma,n} m'$, on a bien $m' \sqsubseteq e$ d'après le point 2d de la définition 3.1.

Donc d'après la définition 3.1, on a bien $m \cong_{\sigma',n'} m'$

□

Lemme 3.2. Soit un modèle partiel bornable $(M, e, \times, \sqsubseteq)$. Soit $m, m_1, m_2 \in M$ tels que $m_1 \times m_2 \cong_{\sigma',n_1+n_2} m$ alors il existe m'_1, m'_2, tels que $m \sqsubseteq m'_1 \times m'_2$, $m_1 \cong_{\sigma',n_1} m'_1$ et $m_2 \cong_{\sigma',n_2} m'_2$.

Preuve. Comme on considère un monoïde bornable, appelons $r_1 \times \ldots \times r_{k_1}$ la décomposition en ressources atomiques de m_1 et $r_{k_1+1} \times \ldots \times r_{k_1+k_2}$ la décomposition en ressources atomiques de m_2. On a donc $r_1 \times \ldots r_{k_1} \times r_{k_1+1} \times \ldots r_{k_1+k_2} \cong_{\sigma',n_1+n_2} m'_1 \times m'_2$. Et donc, par le point 2c de la définition 3.1, il existe $r'_1, \ldots r'_{k_1+k_2}$ tel que $m \sqsubseteq r'_1 \times \ldots \times r'_{k_1+k_2}$ et pour tout i, $r_i \cong_{\sigma',0} r'_i$.

Comme $r_i \cong_{\sigma',0} r'_i$ et comme chaque r_i est une ressource atomique, r_i n'est donc pas décomposable et on a donc $\forall n.r_i \cong_{\sigma',n} r'_i$.

Par conséquent d'après le point 2b de la définition 3.1, on a $r_1 \times \ldots \times r_{k_1} \cong_{\sigma',n_1} r'_1 \times \ldots \times r'_{k_1}$ et $r_1 \times \ldots \times r_{k_1} \cong_{\sigma',n_1} r'_1 \times \ldots \times r'_{k_1}$. Et on peut donc conclure. □

Lemme 3.3. $\forall m, m'.((m \cong_{\sigma_\phi,s(\phi)} m'$ et $m \models_{\mathfrak{P}} \phi) \Rightarrow m' \models_{\mathfrak{P}} \phi)$.

Preuve. Par induction sur le nombre d'opérateurs de la formule ϕ, avec une analyse par cas :

- Cas p : Par définition de $\cong_{\sigma_\phi,s(\phi)}$, $m \models_{\mathfrak{P}} p$ ssi $m \models_{\mathfrak{P}} p$.

- Cas $\psi * \psi'$: Supposons que $m \models_{\mathfrak{P}} \psi * \psi'$, il existe m_1, m_2 tel que $m_1 \models_{\mathfrak{P}} \psi$ et $m_2 \models_{\mathfrak{P}} \psi'$ et $m \sqsubseteq m_1 \times m_2$. On a donc également $m_1 \times m_2 \cong_{\sigma_\phi, s(\phi)} m'$. D'après le lemme 3.2, il existe $m'_1 \times m'_2 \sqsubseteq m'$ tels que $m'_1 \cong \sigma_\phi, s(\psi) m_1$ et $m'_2 \cong \sigma_\phi, s(\psi) m_2$. Par hypothèse d'induction, on peut conclure.
- Cas I : Par définition : $m \models_{\mathfrak{P}} I$ donc $m \sqsubseteq e$ et d'après **??**, $m' \sqsubseteq e$.
- Cas $\psi {-\!\!*} \psi'$: Soit m_1 tel que $m_1 \models_{\mathfrak{P}} \psi$ et $m \times m_1 \models_{\mathfrak{P}} \psi'$. Comme $m \cong_{\sigma_\phi, s(\phi)} m'$, d'après la définition de $\cong_{\sigma_\phi, s(\phi)}$, on a $m \times m_1 \cong_{\sigma_\phi, s(\phi)} m' \times m_1$. Par hypothèse d'induction, on peut conclure.
- Cas $\psi \wedge \psi'$: On a $m \models_{\mathfrak{P}} \psi \wedge \psi'$ donc, par définition, $m \models_{\mathfrak{P}} \psi$ et $m \models_{\mathfrak{P}} \psi'$. Comme $m \cong_{\sigma_\phi, s(\phi)} m'$, par hypothèse d'induction, on a bien $m' \models_{\mathfrak{P}} \psi$ et $m' \models_{\mathfrak{P}} \psi'$ et donc $m' \models_{\mathfrak{P}} \psi \wedge \psi'$.
- Cas $\top$: On a nécessairement $m \models_{\mathfrak{P}} \top$ et $m' \models_{\mathfrak{P}} \top$.
- Cas $\psi \vee \psi'$: On a $m \models_{\mathfrak{P}} \psi \vee \psi'$ donc, par définition, $m \models_{\mathfrak{P}} \psi$ ou $m \models_{\mathfrak{P}} \psi'$. Comme $m \cong_{\sigma_\phi, s(\phi)} m'$, par hypothèse d'induction, on a bien $m' \models_{\mathfrak{P}} \psi$ ou $m' \models_{\mathfrak{P}} \psi'$, donc $m' \models_{\mathfrak{P}} \psi \vee \psi'$.
- Cas $\perp$: $m \models_{\mathfrak{P}} \perp$ et $m' \models_{\mathfrak{P}} \perp$ ne sont jamais vérifiés.
- Cas $\psi \to \psi'$: On a $m \models_{\mathfrak{P}} \psi \to \psi'$ donc, par définition, si $m \models_{\mathfrak{P}} \psi$ alors $m \models_{\mathfrak{P}} \psi'$. Comme $m \cong_{\sigma_\phi, s(\phi)} m'$, par hypothèse d'induction, on a bien si $m' \models_{\mathfrak{P}} \psi$ alors $m' \models_{\mathfrak{P}} \psi'$, donc $m' \models_{\mathfrak{P}} \psi \to \psi'$.

$\square$

Corollaire 3.4. Pour toute ressource m et pour toute formule de la forme $\phi {-\!\!*} \psi$, $m \models_{\mathfrak{P}} \phi {-\!\!*} \psi$ ssi $\forall m' \in M_{/\cong_{\sigma_\phi, s(\phi)}}$ tel que $[m \times m']\uparrow$, $m \times m' \models_{\mathfrak{P}} \psi$.

Preuve. Le sens direct est immédiat puisque $M_{/\cong_{\sigma_\phi, s(\phi)}} \subseteq M$. Supposons maintenant que $\forall m' \in M_{/\cong_{\sigma_\phi, s(\phi)}}$ tel que $[m \times m']\uparrow$, on ait $m \times m' \models_{\mathfrak{P}} \psi$. Soit $r \in M$ tel que $r \models_{\mathfrak{P}} \psi$. Il existe $r' \in M_{/\cong_{\sigma_\phi, s(\phi)}}$ tel que $r' \cong_{\sigma_\phi, s(\phi)} r$. D'après le lemme 3.3, on a $r' \models_{\mathfrak{P}} \phi$ et donc $m \times r' \models_{\psi}$. De plus, par la définition 3.1, on a $m \times r' \cong_{\sigma_\phi, s(\phi)} m \times r$ et donc, par le lemme 3.3, on a $m \times r \models_{\mathfrak{P}} \psi$. $\square$

Ce dernier corollaire nous indique que pour vérifier une formule du type $\psi {-\!\!*} \psi'$, on peut travailler sur un ensemble fini de ressources. On contrôle ainsi l'explosion de cas qu'induit normalement l'opérateur ${-\!\!*}$. Ce cas crucial étant maîtrisé, nous pouvons donc maintenant établir le résultat de décidabilité suivant :

Théorème 3.1 (Satisfaction décidable dans BI). *Soit un modèle partiel de ressources bornable $\mathcal{M} = (M, \times, e, \sqsubseteq)$, pour toute formule de BI ϕ et toute ressource $m \in M$, $m \models_{\mathfrak{P}} \phi$ est décidable.*

Preuve. Par induction structurelle sur la formule ϕ :

- Cas p : Immédiat.
- Cas $\psi * \psi'$: Par définition de 3.1, il existe des ressources atomiques $m_1, \ldots, m_n$ tels que $m \sqsubseteq m_1 \times \ldots \times m_n$, on a donc un nombre limité de décompositions de m à tester.
- Cas I : Immédiat.
- Cas $\psi {-\!\!*} \psi'$: D'après le corollaire 3.4, il suffit de vérifier si pour tout $m' \in M_{/\cong_{\sigma_\phi, s(\psi)}}$, $m \times m' \models_{\mathfrak{P}} \psi'$. Comme $M_{/\cong_{\sigma_\phi, s(\psi)}}$ est fini, par hypothèse d'induction, on a un nombre fini de ressources à tester.

- Cas $\psi \wedge \psi'$: Il suffit de vérifier que l'on a $m \models_{\mathfrak{P}} \psi$ et $m \models_{\mathfrak{P}} \psi'$. Ce qui est décidable d'après hypothèse d'induction.
- Cas $\top$: Immédiat.
- Cas $\psi \vee \psi'$: Il suffit de vérifier que l'on a $m \models_{\mathfrak{P}} \psi$ ou $m \models_{\mathfrak{P}} \psi'$. Ce qui est décidable d'après hypothèse d'induction.
- Cas $\perp$: $m \models_{\mathfrak{P}} \perp$ n'est jamais vérifié.
- Cas $\psi \to \psi'$: Il suffit de vérifier que si $m \models_{\mathfrak{P}} \psi \to \psi$ alors $m \models_{\mathfrak{P}} \psi'$. Par hypothèse d'induction, ces deux problèmes sont décidables.

$\square$

La décidabilité de la validité est une conséquence immédiate de ce résultat :

Théorème 3.2 (Validité décidable dans BI). *Soit un monoïde de ressource bornable $\mathcal{M} = (M, \times, e, \sqsubseteq)$, pour toute formule de BI ϕ, $\models_{\mathfrak{P}}^{\mathcal{M}} \phi$ est décidable.*

Preuve. Ce résultat est une conséquence immédiate du théorème 3.1 et du lemme 2.1. $\square$

3.1.2 Décidabilité de la validité dans BI-Loc

Pour étendre les résultats présentés ci-dessus à BI-Loc, nous devons maîtriser également la taille des arbres que nous considérons. Nous devons donc trouver une solution pour restreindre la largeur et la hauteur maximale des arbres que l'on doit manipuler. Étant donné qu'un chemin mène à un emplacement unique, il suffit pour restreindre la largeur des arbres de restreindre l'ensemble des noms d'emplacement que l'on peut utiliser. Pour la restriction de la taille, nous utiliserons le même procédé que précédemment avec les ressources : nous définirons pour chaque formule une hauteur maximale des arbres qu'il est pertinent d'étudier.

Pour restreindre les noms d'emplacements, il est nécessaire que la partialité repose uniquement sur la décomposition des ressources. Dans cette optique, nous ne considérerons ici que les monoïdes d'arbres partiels maximalement défini dont voici la définition :

Définition 3.3 (Monoïde d'arbres partiel maximalement défini). *Un monoïde d'abres partiel $\mathcal{T} = (T, (e, nil), |, \sqsubseteq_T)_{\mathcal{M}, LocL}$ est* maximalement défini *si pour tout $t, t' \in T$, $t|t'$ est indéfini ssi il existe un chemin L tel que $t(L) = m$, $t'(L) = m'$ et $m \times m'$ n'est pas défini.*

Par la suite, nous utiliserons Loc_ϕ pour désigner l'ensemble des emplacements présents dans une formule de BI-Loc ϕ et Loc_t pour désigner l'ensemble des emplacements définis dans un arbre t.

Lemme 3.5. Soit une formule ϕ, un arbre de ressources t appartenant à un monoïde d'arbres partiel maximalement défini et deux emplacements $l, l' \notin Loc_\phi$, $t \models_{\mathfrak{T}} \phi$, $[t\{^l/_{l'}\}]\!\uparrow \Leftrightarrow t\{^l/_{l'}\} \models_{\mathfrak{T}} \phi$.

Preuve. Par induction selon la structure d'une formule ϕ :

- Cas p : $t \models_{\mathfrak{T}} p$ ssi $t = (m, nil)$ et $m \models_{\mathfrak{T}} p$ donc $t\{^l/_{l'}\} = t$ et donc $t\{^l/_{l'}\} \models_{\mathfrak{T}} p$ par le même raisonnement, si $t\{^l/_{l'}\} \models_{\mathfrak{T}} p$ alors $t \models_{\mathfrak{T}} p$.
- Cas $\psi * \psi'$: $t \models_{\mathfrak{T}} \psi * \psi'$ ssi il existe t', t'' tels que $t \sqsubseteq t'|t''$, $t' \models_{\mathfrak{T}} \psi$ et $t'' \models_{\mathfrak{T}} \psi'$. Par hypothèse d'induction, $t'\{^l/_{l'}\} \models_{\mathfrak{T}} \psi$ et $t''\{^l/_{l'}\} \models_{\mathfrak{T}} \psi'$. Et comme $t\{^l/_{l'}\}|t\{^l/_{l'}\}|t\{^l/_{l'}\}$ on a $t\{^l/_{l'}\} \models_{\mathfrak{T}} \psi * \psi'$. Supposons maintenant que l'on ait $t\{^l/_{l'}\} \models_{\mathfrak{T}} \psi * \psi'$, par un raisonnement similaire, on peut conclure que $t \models_{\mathfrak{T}} \psi * \psi'$.

- Cas I : $t \models_{\mathfrak{T}} I$, $t \sqsubseteq_T (e, nil)$ donc $t\{^l/_{l'}\} = t$ et donc $t\{^l/_{l'}\} \models_{\mathfrak{T}} I$. On obtient l'autre implication par le même raisonnement.
- Cas $\psi \mathrel{-\!\!*} \psi'$: $(\Rightarrow)$ $t \models_{\mathfrak{T}} \psi \mathrel{-\!\!*} \psi'$, donc $\forall t'.t' \models_{\mathfrak{T}} \psi$, $t' \times t \models_{\mathfrak{T}} \psi'$. Par hypothèse d'induction, on a donc $(t' \times t)\{^l/_{l'}\} \models_{\mathfrak{T}} \psi'$ et par cette même hypothèse, on obtient $t' \times t\{^l/_{l'}\} \models_{\mathfrak{T}} \psi'$ d'où $t\{^l/_{l'}\} \models_{\mathfrak{T}} \psi \mathrel{-\!\!*} \psi'$.
  $(\Leftarrow)$ $t\{^l/_{l'}\} \models_{\mathfrak{T}} \psi \mathrel{-\!\!*} \psi'$ donc pour tout $\forall t'.t' \models_{\mathfrak{T}} \psi$, $t' \times t\{^l/_{l'}\} \models_{\mathfrak{T}} \psi'$. Par hypothèse d'induction, $(t' \times t)\{^l/_{l'}\} \models_{\mathfrak{T}} \psi'$ et on obtient donc par cette même hypothèse : $(t' \times t) \models_{\mathfrak{T}} \psi'$ et au final $t \models_{\mathfrak{T}} \psi \mathrel{-\!\!*} \psi'$.
- Cas $\psi \wedge \psi'$: $t \models_{\mathfrak{T}} \psi \wedge \psi'$ donc $t \models_{\mathfrak{T}} \psi$ et $t \models_{\mathfrak{T}} \psi'$ par l'hypothèse d'induction, $t\{^l/_{l'}\} \models_{\mathfrak{T}} \psi$ et $t\{^l/_{l'}\} \models_{\mathfrak{T}} \psi'$ et donc $t\{^l/_{l'}\} \models_{\mathfrak{T}} \psi \wedge \psi'$. On obtient l'autre implication par un raisonnement similaire.
- Cas $\top$: On a $t \models_{\mathfrak{T}} \top$ et $t\{^l/_{l'}\} \models_{\mathfrak{T}} \top$.
- Cas $\psi \vee \psi'$: Le raisonnement est le même que pour $\psi \wedge \psi'$.
- Cas $\perp$: On a $t \not\models_{\mathfrak{T}} \perp$ et $t\{^l/_{l'}\} \not\models_{\mathfrak{T}} \perp$.
- Cas $\psi \rightarrow \psi'$: Le raisonnement est le même que pour $\psi \wedge \psi'$.
- Cas $[l'']\psi$: Par définition, $l'' \notin \{l, l'\}$. Or si $t \models_{\mathfrak{T}} [l'']\psi$ alors, $t \sqsubseteq (nil, l'' \mapsto t'')$ où $t'' \models_{\mathfrak{T}} \psi$. Donc $t' = (nil, l'' \mapsto t''\{^L/_{l'}\})$ et par hypothèse d'induction, $t''\{^L/_{l'}\} \models_{\mathfrak{T}} \psi$, donc $t' \models_{\mathfrak{T}} [l'']\psi$.

$\square$

Ce lemme indique que si deux emplacements ne sont pas dans une formule alors on peut remplacer un nom d'emplacement par l'autre sans altérer la satisfaction. Ainsi donc pour un arbre t et une formule ϕ, on peut remplacer tous les noms d'emplacements non présents dans Loc_ϕ par un seul nom d'emplacement.

Corollaire 3.6. Pour toute formule ϕ de BI-Loc et tout arbre de ressources t d'un monoïde d'arbres partiels maximalement défini, il existe un arbre de ressources t' tel que $Loc_{t'} \subseteq Loc_\phi \cup \{l_\pi\}$ (avec $l_\pi \notin Loc_\phi$) et $t \models_{\mathfrak{T}} \phi$ ssi $t' \models_{\mathfrak{T}} \phi$.

Preuve. Soit une formule ψ et un arbre de ressources t. On pose $Loc' = \{l_1, \ldots, l_n\} = Loc_t - Loc_\phi$. Alors, on remplace dans t chaque emplacement de Loc' par l_π. D'après le lemme 3.5, $t \models_{\mathfrak{T}} \phi$ ssi $t\{^{l_\pi}/_{l_1}\}\{\ldots\}\{^{l_\pi}/_{l_n}\} \models_{\mathfrak{T}} \phi$. Or si on pose $t' \equiv t\{^{l_\pi}/_{l_1}\}\{\ldots\}\{^{l_\pi}/_{l_n}\}$, on a bien en plus $Loc_{t'} = Loc_\phi \cup \{l_\pi\}$. $\square$

Ensuite nous définissons la taille d'un arbre t et d'une formule ϕ. La taille d'un arbre est donné par la définition usuelle et la taille d'une formule de BI-Loc est donnée par la définition suivante :

Définition 3.4 (Hauteur d'une formule). *La* hauteur $h(\phi)$ *d'une formule* ϕ *est définie récursivement de la façon suivante :*

- $h(p) = 1$;	- $h(\phi' * \phi'') = max(h(\phi'), h(\phi''))$;
- $h(I) = 1$;	- $h(\phi' \mathrel{-\!\!*} \phi'') = h(\phi'')$;
- $h(\top) = 0$;	- $h(\phi' \wedge \phi'') = max(h(\phi'), h(\phi''))$;
- $h(\perp) = 0$;	- $h(\phi' \vee \phi'') = max(h(\phi'), h(\phi''))$;
- $h([l]\phi') = 1 + h(\phi')$;	- $h(\phi' \rightarrow \phi'') = max(h(\phi'), h(\phi''))$;

La taille d'une formule correspond à la taille maximale des arbres qu'il est nécessaire de considérer pour vérifier si une formule donnée est valide ou non. L'idée est de considérer qu'une formule donnée ne vérifie pas le contenu des noeuds situés au delà d'une hauteur donnée. On peut donc restreindre la hauteur de l'arbre à considérer pour vérifier qu'il satisfait une formule donnée. On commence donc par définir la restriction d'un arbre à une hauteur donnée.

Définition 3.5. *La restriction d'un arbre t à une hauteur h, notée t_h, est telle que pour tout chemin $l_1, \ldots, l_n$, on a :*
- $t_h(l_1, \ldots, l_n) = t(l_1, \ldots, l_n)$ *si* $n < h$ *;*
- $t_h(l_1, \ldots, l_n) = (r, nil)$ *si* $\exists r \in M, t' \in T_{M,Loc}.t(l_1, \ldots, l_n) = (r, t')$ *et* $n = h$ *;*
- $t_h(l_1, \ldots, l_n)$ *n'est pas défini si* $n > h$.

On montre alors que pour une formule donnée, la restriction d'un arbre à la hauteur de cette formule se comporte comme cette formule.

Lemme 3.7. Pour toute formule ϕ et tout arbre de ressources t, $t \models_{\mathfrak{T}} \phi$ ssi $t_{h(\phi)} \models_{\mathfrak{T}} \phi$

Preuve. Le cas crucial est évidemment celui où $h(t) > h(\phi)$. Dans ce cas là, nous prouvons par induction structurelle sur ϕ que $t_{h(\phi)} \models_{\mathfrak{T}} \phi$ ssi $t_{h(\phi)} \models_{\mathfrak{T}} \phi$.

- Cas p : $t \models_{\mathfrak{T}} p$ uniquement si $t = (m, nil)$ avec $m \in M$ et $m \models_{\mathfrak{T}} p$ or, comme $h(p) = 1$ et $h(t) > h(\phi)$ on a $t \neq (m, nil)$ et également $t_{h(p)} \neq (m, nil)$ Donc $t \not\models_{\mathfrak{T}} p$ et $t_{h(p)} \not\models_{\mathfrak{T}} p$.
- Cas $\psi * \psi'$: Pour tout t' et t'' tels que $t \sqsubseteq t'|t''$. On pose $t_1 = t'_{h(\psi*\psi')}$ et $t_2 = t''_{h(\psi*\psi')}$. $t_1|t_2$ est défini puisque $t'|t''$ est défini. De même, il est clair que $t'_{h(\psi)} = t_{1_{h(\psi)}}$ et $t''_{h(\psi')} = t_{2_{h(\psi')}}$. Donc par l'hypothèse d'induction, on a $t' \models_{\mathfrak{T}} \psi$ ssi $t'_{h(\psi)} \models_{\mathfrak{T}} \psi$ ssi $t_1 \models_{\mathfrak{T}} \psi$ (idem pour t'' et t_2). Ainsi, on a pour tout t', t'', $t|t'' \models_{\mathfrak{T}} \psi * \psi'$ ssi $t_1|t_2 \models_{\mathfrak{T}} \psi * \psi'$ et on peut conclure.
- Cas I : $t \models_{\mathfrak{T}} p$ uniquement si $t = (m, nil)$ avec $m \in M$ et $e \sqsubseteq m$. Or comme $h(I) = 1$, $t_{h(I)} = (m, nil)$ ssi $t = (m, nil)$. Donc $t \models_{\mathfrak{T}} p$ ssi $t_{h(p)} \models_{\mathfrak{T}} p$.
- Cas $\psi \mathrel{-\!\!*} \psi'$: Pour tout t', par hypothèse d'induction, $t'|t \models_{\mathfrak{T}} \psi'$ ssi $(t'|t)_h(\psi') \models_{\mathfrak{T}} \psi'$. Et par l'hypothèse d'induction également, $(t'|t)_{h(\psi')} \models_{\mathfrak{T}} \psi'$ ssi $t'|t_{h(\psi')} \models_{\mathfrak{T}} \psi'$. $t \models_{\mathfrak{T}} \psi \mathrel{-\!\!*} \psi'$ ssi $t_{h(\psi')} \models_{\mathfrak{T}} \psi \mathrel{-\!\!*} \psi'$ et comme $h(\psi') = h(\psi \mathrel{-\!\!*} \psi')$, on peut conclure.
- Cas $\psi \wedge \psi'$: $t \models_{\mathfrak{T}} \psi \wedge \psi'$ ssi $t \models_{\mathfrak{T}} \psi$ et $t \models_{\mathfrak{T}} \psi'$. Par hypothèse d'induction, $t \models_{\mathfrak{T}} \psi$ (resp. $t \models_{\mathfrak{T}} \psi'$) ssi $t_{h(\psi)} \models_{\mathfrak{T}} \psi$ et $t_{h(\psi')} \models_{\mathfrak{T}} \psi'$. Par la même hypothèse d'induction $t_{max(h(\psi),h(\psi'))} \models_{\mathfrak{T}} \psi$ (resp. $t_{max(h(\psi),h(\psi'))} \models_{\mathfrak{T}} \psi'$) ssi $t_{h(\psi)} \models_{\mathfrak{T}} \psi$ (resp. $t_{h(\psi')} \models_{\mathfrak{T}} \psi'$).
- Cas $\top$: On a $t \models_{\mathfrak{T}} \top$ et $t_{h(\top)} \models_{\mathfrak{T}} \top$.
- Cas $\psi \vee \psi'$: Idem au cas $\psi * \psi'$.
- Cas $\bot$: On a $t \not\models_{\mathfrak{T}} \bot$ et $t_{h(\bot)} \not\models_{\mathfrak{T}} \bot$.
- Cas $\psi \to \psi'$: Idem au cas $\psi \wedge \psi'$.

$\square$

Nous avons vu précédemment comment borner les ressources à considérer sans prendre en compte les emplacements et comment nous pouvions borner l'ensemble des noms et la hauteur des arbres à prendre en compte. En limitant l'ensemble des noms, nous avons de ce fait limiter la largeur de nos arbres, puisque deux frères ne peuvent correspondre au même nom d'emplacement. Tous ces résultats nous permettent d'établir la décidabilité de la satisfaction pour BI-Loc :

Théorème 3.3 (Satisfaction décidable dans BI-Loc). *Soit un modèle d'arbres partiel* $\mathcal{T} = (T, (e, nil), |, \sqsubseteq_T, \models_{\mathfrak{T}})_{\mathcal{M},Loc}$ *où* $(T, (e, nil), |, \sqsubseteq_T)$ *est un monoïde d'arbres partiel maximalement défini et où* $\mathcal{M}$ *est un monoïde partiel bornable. Soit* $t \in T$ *et* ϕ *une formule de* BI-Loc*,* $t \models_{\mathfrak{T}} \phi$ *est décidable.*

Preuve. On s'assure pour chaque règle qu'il suffit d'un nombre fini de vérifications pour s'assurer de la satisfaction de la formule. De ce point de vue, les deux règles posant le plus de problèmes sont dans l'ordre celles de $*$ et $-\!\!*$.

- Cas $*$: Supposons que l'on doive vérifier si $t \models_{\mathfrak{T}} \phi * \phi'$ est satisfait. Nous devons donc vérifier que parmi toutes les décompositions de t en deux sous-arbres t' et t'' il y en a au moins une telle que $t' \models_{\mathfrak{T}} \phi$ and $t'' \models_{\mathfrak{T}} \phi'$. Or comme les arbres contiennent des ressources appartenant à un monoïde de ressources bornable, il existe un nombre fini de décompositions de t.
- Cas $-\!\!*$: Supposons que l'on doive vérifier si $t \models_{\mathfrak{T}} \phi * \phi'$ est satisfait. Nous devons donc vérifier que pour tout t' tels que $t' \models_{\mathfrak{T}} \phi$, $t|t' \models \phi'$. Le problème ici est qu'il peut y avoir une infinité d'arbres vérifiant ϕ. Cependant, d'après le lemme 3.7, on sait que tout arbre tel que $h(t) > h(\phi -\!\!* \phi')$ se comporte comme sa restriction à la hauteur $h(\phi -\!\!* \phi')$. Par conséquent il suffit pour vérifier si pour tout t' tel que $h(t') \leq h(\phi -\!\!* \phi')$ et tel que $t' \models_{\mathfrak{T}} \phi$ alors $t|t' \models \phi'$. On limite ensuite les noms d'emplacement des arbres à considérer grâce au lemme 3.5. Enfin, on limite le nombre de ressources dans chaque nœud grâce au théorème 3.1.

□

Par conséquent, nous pouvons donc affirmer que la validité de BI-Loc est décidable :

Théorème 3.4 (Validité décidable dans BI-Loc). *Soit un modèle d'arbres partiel* $\mathcal{T} = (T, (e, nil), |, \sqsubseteq_T, \models_{\mathfrak{T}})_{\mathcal{M},Loc}$ *où* $(T, (e, nil), |, \sqsubseteq_T)$ *est un monoïde d'arbres partiel maximalement défini et où* $\mathcal{M}$ *est un monoïde partiel bornable. Soit* ϕ *une formule de* BI-Loc*,* $t \models_{\mathfrak{T}} \phi$ *est décidable.*

Preuve. C'est une conséquence directe du théorème 3.3 et du lemme 2.1. □

3.1.3 Étude de la décidabilité pour BI-Loc$_\forall$

L'introduction des quantificateurs a nécessairement une conséquence sur les résultats de décidabilité de BI-Loc. La situation est alors similaire à celle étudiée par Charatonik et al. dans la logique des ambients [35] ou encore à celle de la logique des pointeurs étudiée par Calcagno et al. dans [25]. Dans leur travaux, ils soulignent les problèmes posés par la cohabitation des quantificateurs avec l'implication multiplicative $-\!\!*$. Nous montrons ici que cette même cause provoque les mêmes difficultés dans BI-Loc.

Commençons par mettre à part ce problème en considérant l'ensemble des formules qui ne contiennent pas l'opérateur $-\!\!*$. Nous proposons pour cela une procédure de model-checking permettant de vérifier la satisfaction. Nous montrons ensuite que cette procédure termine.

Définition 3.6 (Procédure de décision pour BI-Loc$_\forall$ sans $-\!\!*$). *Soit un arbre* t *et une formule* ϕ *ne comportant pas l'opérateur* $-\!\!*$*,* $Check(t, \phi)$ *est alors défini récursivement comme suit :*

- $Check(t, p) = vrai$ *si* $t = (m, nil)$ *et* $m \models_{\mathfrak{T}} p$*,* $faux$ *sinon ;*
- $Check(t, \psi * \psi') = \bigvee_{t_1|t_2 \sqsubseteq t}(Check(t_1, \psi) \wedge Check(t_2, \psi'))$ *;*
- $Check(t, I) = vrai$ *si* $t = (m, nil)$ *et* $e \sqsubseteq m$*,* $faux$ *sinon ;*
- $Check(t, \psi \wedge \psi') = Check(t, \psi) \wedge Check(t, \psi')$

- $Check(t, \top) = vrai$;
- $Check(t, \phi \vee \psi) = Check(t, \phi) \vee Check(t, \psi)$;
- $Check(t, \bot) = faux$;
- $Check(t, \psi \rightarrow \psi') = \neg Check(t, \psi) \vee Check(t, \psi')$;
- $Check(t, [l]\psi) = vrai$ *si* $t = (m, l \mapsto t')$ *et* $Check(t', \psi)$, $faux$ *sinon ;*
- $Check(t, \exists_{loc} x.\psi) = \bigvee_{l \in \{l_0\} \cup Loc_\psi \cup Loc_t} Check(t, \psi\{^l/_x\})$ *(avec* $l_0 \in Loc - (Loc_\psi \cup Loc_t)$*) ;*
- $Check(t, \forall_{loc} x.\psi) = \bigwedge_{l \in \{l_0\} \cup Loc_\psi \cup Loc_t} Check(t, \psi\{^l/_x\})$ *(avec* $l_0 \in Loc - (Loc_\psi \cup Loc_t)$*) ;*
- $Check(t, \exists_{path} X.\psi) = \bigvee_{L \in Reachable(t,\psi)} Check(t, \psi\{^L/_X\})^*$;
- $Check(t, \forall_{path} X.\psi) = \bigwedge_{L \in Reachable(t,\psi)} Check(t, \psi\{^L/_X\})^*$.

* : *Reachable*(t, ψ) *définit l'ensemble des chemins* $l_1, \ldots l_n$ *tels que* $n < h(\psi)$ *et* $l_i \in l_0 \cup Loc_\phi \cup Loc_t$ *où* $l_0 \in \mathcal{L} - (fn(\phi) \cup fn(P))$

Dans cette définition, $\bigvee_{t_1|t_2 \sqsubseteq t} P(t_1, t_2)$ *correspond à la disjonction des propositions* $P(t_1, t_2)$ *pour tous les couples* t_1, t_2 *tels que* $t_1|t_2 \sqsubseteq t$.

De même, les notations $\bigvee_{x \in E} P(x)$ *et* $\bigwedge_{x \in E} P(x)$ *représentent respectivement la disjonction et la conjonction des propositions* $P(x)$ *où* x *prend toutes les valeurs présentes en* E.

Lemme 3.8. Soit un modèle d'arbres partiel $\mathcal{T} = (T, (e, nil), |, \sqsubseteq_T)_{\mathcal{M}, Loc}$ où $(T, (e, nil), |, \sqsubseteq_T)$ est un monoïde d'arbres partiel maximalement défini et où $\mathcal{M}$ est un monoïde partiel bornable. Soit $t \in T$ et ϕ une formule de BI-Loc$_\forall$. Si ϕ ne contient pas l'opérateur $-\!\!*$, $Check(t, \phi)$ termine.

Preuve. Par induction structurelle sur ϕ :

- $Check(t, p) = vrai$ est immédiat, donc $Check(t, p)$ termine.
- $Check(t, \psi * \psi')$ comme M est un monoïde partiel bornable, il existe un nombre de décompositions fini de t. Donc il existe un nombre fini de t_1 et t_2 tels que $t_1|t_2 \sqsubseteq t$. Donc par hypothèse d'induction, l'évaluation de $\bigvee_{t_1|t_2 \sqsubseteq t}(Check(t_1, \psi) \wedge Check(t_2, \psi'))$ termine et donc $Check(t, \psi * \psi')$ termine.
- $Check(t, I)$ est immédiat, donc $Check(t, I)$ termine.
- $Check(t, \psi \wedge \psi') = Check(t, \psi) \wedge Check(t, \psi')$. Par hypothèse d'induction, $Check(t, \psi)$ et $Check(t, \psi')$ terminent, donc $Check(t, \psi \wedge \psi')$ termine.
- $Check(t, \top)$ est immédiat, donc $Check(t, \top)$ termine.
- $Check(t, \phi \vee \psi) = Check(t, \phi) \vee Check(t, \psi)$. Par hypothèse d'induction, $Check(t, \psi)$ et $Check(t, \psi')$ terminent, donc $Check(t, \psi \vee \psi')$ termine.
- $Check(t, \bot)$ est immédiat, donc $Check(t, \bot)$ termine.
- $Check(t, \psi \rightarrow \psi') = \neg Check(t, \psi) \vee Check(t, \psi')$. Par hypothèse d'induction, $Check(t, \psi)$ et $Check(t, \psi')$ terminent, donc $Check(t, \psi \rightarrow \psi')$ termine.
- $Check(t, [l]\psi) = vrai$ si $t = (m, l \mapsto t')$ et $Check(t', \psi)$, $faux$ sinon. Par hypothèse d'induction, $Check(t', \psi)$ termine donc $Check(t, [l]\psi)$ termine.
- $Check(t, \exists_{loc} x.\psi) = \bigvee_{l \in \{l_0\} \cup Loc_\psi \cup Loc_t} Check(t, \psi\{^l/_x\})$ (avec $l_0 \in Loc - (Loc_\psi \cup Loc_t)$). Comme $Loc_\psi \cup Loc_t$ est fini, on doit évaluer $Check(t, \psi\{^l/_x\}$ pour un ensemble fini de valeurs de l. Comme par hypothèse d'induction, $Check(t, \psi\{^l/_x\})$ termine, $Check(t, \exists_{loc} x.\psi)$ termine.
- $Check(t, \forall_{loc} x.\psi) = \bigwedge_{l \in \{l_0\} \cup Loc_\psi \cup Loc_t} Check(t, \psi\{^l/_x\})$ (avec $l_0 \in Loc - (Loc_\psi \cup Loc_t)$). Comme $Loc_\psi \cup Loc_t$ est fini, on doit évaluer $Check(t, \psi\{^l/_x\})$ pour un ensemble fini de valeurs de l. Comme par hypothèse d'induction, $Check(t, \psi\{^l/_x\})$ termine, $Check(t, \forall_{loc} x.\psi)$ termine.

– $Check(t, \exists_{path}X.\psi) = \bigvee_{L \in Reachable(t,\psi)} Check(t, \psi\{^L/_X\})^*$. Comme $Loc_\psi \cup Loc_t$ est fini, et que $h(\psi)$ est fini, l'ensemble des chemins de taille n tels constitués à partir de l'ensemble $\{l_0\} \cup Loc_\psi \cup Loc_t$ est lui aussi fini. Comme par hypothèse d'induction, $Check(t, \psi\{^L/_X\})$ termine, $Check(t, \exists_{path}X.\psi)$ termine également.
– $Check(t, \forall_{path}X.\psi) = \bigwedge_{L \in Reachable(t,\psi)} Check(t, \psi\{^L/_X\})^*$. Comme $Loc_\psi \cup Loc_t$ est fini, et que $h(\psi)$ est fini, l'ensemble des chemins de taille n tels constitués à partir de l'ensemble $\{l_0\} \cup Loc_\psi \cup Loc_t$ est lui aussi fini. Comme par hypothèse d'induction, $Check(t, \psi\{^L/_X\})$ termine, $Check(t, \forall_{path}X.\psi)$ termine.

□

Pour démontrer la décidabilité de ce fragment, nous avons également besoin de démontrer les lemmes de substitution suivants :

Lemme 3.9 (Substitution de noms d'emplacement). Soit une formule ϕ de BI-Loc$_\forall$ ne comportant pas d'opérateur $-\!\!*$, un arbre de ressources t et deux emplacements $l, l' \notin Loc_t$. Alors $t \models_{\mathfrak{T}} \phi$ ssi $t \models_{\mathfrak{T}} \phi\{^l/_{l'}\}$.

Preuve. Par induction sur la structure de ϕ. Le cas le plus difficile est celui où $\phi = [l']\psi$. Dans ce cas, comme $l, l' \notin Loc_t$ on a ni $t \models_{\mathfrak{T}} [l']\psi$, ni $t \models_{\mathfrak{T}} [l]\psi$. □

Lemme 3.10 (Substitution de chemins). Soit une formule ϕ de BI-Loc$_\forall$ ne comportant pas d'opérateur $-\!\!*$, un arbre de ressources t et un chemin L tels que $h(L) = h(t) + 1$. Alors $t \models_{\mathfrak{T}} \phi$ ssi pour tout chemin L', $t \models_{\mathfrak{T}} \phi\{^L/_{L:L'}\}$

Preuve. Le cas où $L : L'$ n'appartient pas à ϕ est trivial puisque $\phi = \phi\{^L/_{L:L'}\}$.

Dans le cas contraire, on fait la preuve par induction sur ϕ. Le cas de base est le plus complexe, il s'agit de celui où $t \models_{\mathfrak{T}} [L : L']\psi$. Quelque soit la formule ψ, on ne peut pas avoir $t \models_{\mathfrak{T}} [L, L']\psi$ ni $t \models_{\mathfrak{T}} [L]\psi$. En effet, comme $h(t) < h(L)$ et $h(t) < h(L : L')$, $t(L)$ et $t(L : L')$ n'existent pas donc il n'existe pas de t' tel que $t = [L : L']t'$ (respectivement $t = [L]t'$) et $t' \models_{\mathfrak{T}} \psi$.

Les autres cas se traitent trivialement en décomposant la formule et en utilisant les hypothèses d'induction. □

On peut enfin démontrer la décidabilité de ce fragment de BI-Loc$_\forall$.

Théorème 3.5 (Décidabilité de la satisfaction dans BI-Loc$_\forall$ sans $-\!\!*$). *Soit un modèle d'arbres partiel $\mathcal{T} = (T, (e, nil), |, \sqsubseteq_T, \models_{\mathfrak{T}})_{\mathcal{M},Loc}$ où $(T, (e, nil), |, \sqsubseteq_T)$ est un monoïde d'arbres partiel maximalement défini et où $\mathcal{M}$ est un monoïde partiel bornable. Soit $t \in T$ et ϕ une formule de* BI-Loc$_\forall$. *Si ϕ ne contient pas l'opérateur $-\!\!*$, $t \models_{\mathfrak{T}} \phi$ est décidable et on a $t \models_{\mathfrak{T}} \phi$ ssi $Check(t, \phi)$.*

Preuve. Nous avons montré dans le lemme 3.8 que la procédure $Check$ termine. Donc par conséquent il suffit de montrer que $t \models_{\mathfrak{T}} \phi$ ssi $Check(t, \phi)$.
La preuve se fait par induction structurelle sur ϕ :

- Cas p : $Check(t, p) = vrai$ ssi $t = (m, nil)$ et $m \models_{\mathfrak{T}} p$ donc par définition, ssi $t \models_{\mathfrak{T}} p$.
- Cas $\psi * \psi'$: $Check(t, \psi * \psi') = \bigvee_{t_1|t_2 \sqsubseteq t}(Check(t_1, \psi) \wedge Check(t_2, \psi'))$. Donc par définition, $Check(t, \psi * \psi') = vrai$ ssi il existe t_1 et t_2 tels que $t_1|t_2 \sqsubseteq t$, $Check(t_1, \psi)$ et $Check(t_2, \psi')$. Par hypothèse d'induction, ceci est vrai ssi $t_1 \models_{\mathfrak{T}} \psi$ et $t_2 \models_{\mathfrak{T}} \psi'$ donc ssi $t \models_{\mathfrak{T}} \psi \times \psi'$.
- Cas I : $Check(t, p) = vrai$ ssi $t = (m, nil)$ et $e \sqsubseteq m$ donc par définition, ssi $t \models_{\mathfrak{T}} I$.
- Cas $\top$: $Check(t, \top) = vrai$ toujours, donc par définition, ssi $t \models_{\mathfrak{T}} \top$.

- Cas $\psi \wedge \psi'$: $Check(t, \psi \wedge \psi') = vrai$ ssi $Check(t, \psi)$ et $Check(t, \psi')$ donc par hypothèse d'induction, ssi $t \models_{\mathfrak{T}} \psi$ et $t \models_{\mathfrak{T}} \psi'$, donc ssi $t \models_{\mathfrak{T}} \psi \wedge \psi'$.
- Cas $\perp$: $Check(t, \perp) = vrai$ jamais, donc par définition, ssi $t \models_{\mathfrak{T}} \perp$.
- Cas $\psi \to \psi'$: $Check(t, \psi \to \psi') = vrai$ ssi non $Check(t, \psi)$ ou $Check(t, \psi')$ donc par hypothèse d'induction, ssi non $t \models_{\mathfrak{T}} \psi$ ou $t \models_{\mathfrak{T}} \psi'$, donc ssi $t \models_{\mathfrak{T}} \psi \to \psi'$.
- Cas $[l]\psi$: $Check(t, [l]\psi) = vrai$ ssi $t = (e, l \mapsto t')$ et $Check(t', \psi)$, donc par hypothèse d'induction, ssi $t = (e, l \mapsto t')$ et $t' \models_{\mathfrak{T}} \psi$, donc ssi $t \models_{\mathfrak{T}} [l]\psi$.
- Cas $\exists_{loc}x.\psi$: $Check(t, \exists_{loc}x.\psi) = vrai$ ssi il existe $l \in \{l_0\} \cup Loc_\psi \cup Loc_t$ tel que $Check(t, \psi\{^l/_x\})$ (avec $l_0 \in Loc - (Loc_\psi \cup Loc_t)$), donc par hypothèse d'induction, ssi il existe $l \in \{l_0\} \cup Loc_\psi \cup Loc_t$ tel que $t \models_{\mathfrak{T}} \psi\{^l/_x\}$. Il reste à montrer qu'il existe un tel l ssi il existe un $l' \in Loc$ tels que $t \models_{\mathfrak{T}} \psi\{^l/_x\}$. Soit un tel l' prenons le cas où $l' \notin \{l_0\} \cup Loc_\psi \cup Loc_t$, dans ce cas et par le lemme 3.9, on a également $t \models_{\mathfrak{T}} \psi\{^{l_0}/_x\}$.
- Cas $\forall_{loc}x.\psi$: $Check(t, \exists_{loc}x.\psi) = vrai$ ssi $\forall l \in \{l_0\} \cup Loc_\psi \cup Loc_t$ tel que $Check(t, \psi\{^l/_x\})$ (avec $l_0 \in Loc - (Loc_\psi \cup Loc_t)$), donc par hypothèse d'induction, ssi il existe $l \in \{l_0\} \cup Loc_\psi \cup Loc_t$ tel que $t \models_{\mathfrak{T}} \psi\{^l/_x\}$. Il reste à montrer que $t \models_{\mathfrak{T}} \psi\{^l/_x\}$ pour tout $l \in \{l_0\} \cup Loc_\psi \cup Loc_t$ ssi $t \models_{\mathfrak{T}} \psi\{^l/_x\}$ pour tout $l \in Loc$. Si $l \in \{l_0\} \cup Loc_\psi \cup Loc_t$ sinon, comme $l \notin Loc_t$, par le lemme 3.9, on a également $t \models_{\mathfrak{T}} \psi\{^{l_0}/_x\}\{^l/_{l_0}\}$ donc $t \models_{\mathfrak{T}} \psi\{^l/_x\}$.
- Cas $\exists_{path}X.\psi$: $Check(t, \exists_{loc}x.\psi) = vrai$ ssi il existe un chemin $L \in Reachable(t, \psi)$ tel que $Check(t, \psi\{^L/_X\})$, donc par hypothèse d'induction, ssi il existe $L \in Reachable(t, \psi)$ tel que $t \models_{\mathfrak{T}} \psi\{^L/_X\}$. Il reste à montrer qu'il existe un tel L ssi il existe un $L \in Loc^*$ tels que $t \models_{\mathfrak{T}} \psi\{^L/_X\}$. Soit un tel L, prenons le cas où $L \notin Reachable(t, \psi)$. dans ce cas, on pose L_t le sous-chemin de taille $h(t)$ de L. D'après le lemme 3.10, $t \models_{\mathfrak{T}} \psi\{^L/_X\}$ ssi $t \models_{\mathfrak{T}} \psi\{^L/_X\}\{^{L_t}/_L\}$. Ensuite, on nomme $l_1, \ldots, l_k$ l'ensemble des emplacements de L qui ne sont pas dans $\{l_0\} \cup Loc_\psi \cup Loc_t$. D'après le lemme 3.9 $t \models_{\mathfrak{T}} \psi\{^{L_t}/_X\}\{^{l_0}/_{l_1}\} \ldots \{^{l_0}/_{l_k}\}$. Par définition $L_t\{^{l_0}/_{l_1}\} \ldots \{^{l_0}/_{l_k}\} \in Reachable(t, \psi)$ et nous pouvons conclure.
- Cas $\forall_{path}X.\psi$: $Check(t, \forall_{loc}x.\psi) = vrai$ ssi pour tout $L \in Reachable(t, \psi)$, $Check(t, \psi\{^L/_X\})$, donc par hypothèse d'induction, ssi pour tout $L \in Reachable(t, \psi)$, $t \models_{\mathfrak{T}} \psi\{^L/_X\}$. Il reste que ceci est vrai ssi pour tout $L \in Loc^*$ tels que $t \models_{\mathfrak{T}} \psi\{^L/_X\}$. Soit un tel L, détaillons le cas où $L \notin Reachable(t, \psi)$. Dans ce cas, on pose L_t le sous-chemin de taille $h(t)$ de L. D'après le lemme 3.10, $t \models_{\mathfrak{T}} \psi\{^L/_X\}$ ssi $t \models_{\mathfrak{T}} \psi\{^L/_X\}\{^{L_t}/_L\}$. On nomme $l_1, \ldots, l_k$ l'ensemble des emplacements de L qui ne sont pas dans $\{l_0\} \cup Loc_\psi \cup Loc_t$. D'après le lemme 3.9 on a $t \models_{\mathfrak{T}} \psi\{^{L_t}/_X\}\{^{l_0}/_{l_1}\} \ldots \{^{l_0}/_{l_k}\}$. Par définition, $L_t\{^{l_0}/_{l_1}\} \ldots \{^{l_0}/_{l_k}\} \in Reachable(t, \psi)$ et nous pouvons conclure.

□

3.1.4 Indécidabilité de BI-Loc$_\forall$

Nous montrons ici que même sans prendre en compte $-\!\!*$, la validité pour un modèle donné de BI-Loc$_\forall$ est indécidable. De plus, par la relation entre satisfaction et validité établie dans le lemme 2.1, nous indique que si la validité est indécidable pour ce fragment, la satisfaction est indécidable pour ce fragment et $-\!\!*$. La preuve de l'indécidabilité de la validité repose sur le résultat d'indécidabilité suivant :

Théorème 3.6 (Trakhtenbrot [83]). *Même si une signature consiste uniquement en une relation binaire, on ne peut pas décider si une formule close du première ordre ϕ construite sur cette signature admet un modèle fini.*

La preuve de l'indécidabilité de la validité dans $\mathsf{BI\text{-}Loc}_\forall$ consiste à réduire le problème ci-dessus à celui du test de validité. Pour établir le lien entre les deux problèmes, nous commençons par associer à chaque formule logique ϕ du premier ordre, reposant sur une unique relation binaire R, une formule $[\![\phi]\!]_{FO}$ de $\mathsf{BI\text{-}Loc}_\forall$. Selon le schéma suivant :

- $[\![\phi \vee \phi']\!]_{FO} = [\![\phi]\!]_{FO} \vee [\![\phi']\!]_{FO}$;
- $[\![\phi \wedge \phi']\!]_{FO} = [\![\phi]\!]_{FO} \wedge [\![\phi']\!]_{FO}$;
- $[\![\phi \rightarrow \phi']\!]_{FO} = [\![\phi]\!]_{FO} \rightarrow [\![\phi']\!]_{FO}$;
- $[\![\neg\phi]\!]_{FO} = [\![\phi]\!]_{FO} \rightarrow \bot$;
- $[\![\exists x.\phi]\!]_{FO} = \exists_{loc} x.(([d][x]I * \top) \wedge [\![\phi]\!]_{FO})$;
- $[\![R(x_1, x_2)]\!]_{FO} = [r][x_1][x_2]I * \top$.

L'emplacement d sert à regrouper tous les éléments du domaine fini $\mathcal{D}$. Tout élément de $\mathcal{D}$ est représenté par un emplacement sous l'emplacement d et de représenter la relation $R(x_1, x_2)$ par le chemin r, l_1, l_2. Le point crucial de notre démonstration est donné par le lemme suivant :

Lemme 3.11. Soit une formule du premier ordre ϕ, $\not\models_{\mathfrak{T}} [\![\phi]\!] \rightarrow \bot$ est vérifié ssi il existe un modèle fini de ϕ.

Preuve. Pour prouver ce lemme, nous définissons une relation $\rightsquigarrow_{tree}$ entre $\mathcal{S}$, une structure non vide et finie pour la signature $\{R\}$ (où R est une relation binaire sur l'ensemble fini $\mathcal{D}$) et un arbre de ressources t.
Nous avons $\mathcal{S} \rightsquigarrow_{tree} t$ ssi :

1. $a \in \mathcal{D}$ ssi il existe t' tel que $t = t'|(e, d \mapsto (e, a \mapsto (e, nil)))$;
2. Si $a_1, a_2 \in \mathcal{D}$, $R(a_1, a_2) \in \mathcal{S}$, ssi il existe t' tel que $t = t'|(e, r \mapsto (e, a_1 \mapsto (e, a_2 \mapsto (e, nil))))$.

Soit une formule close du premier ordre ϕ et une structure $\mathcal{S}$, on montre par induction structurelle sur ϕ que $\mathcal{S} \models \phi$ ssi pour t tel que $\mathcal{S} \rightsquigarrow_{tree} t$, $t \models_{\mathfrak{T}} [\![\phi]\!]_{FO}$. Seuls les cas les plus significatifs sont donnés ici :

- Cas $R(a_1, a_2)$:

1. $\mathcal{S} \models R(a_1, a_2)$ donc par définition, $R(a_1, a_2) \in \mathcal{S}$ donc il existe t' tel que $t = t'|(e, r \mapsto (e, a_1 \mapsto (e, a_2 \mapsto (e, nil))))$ donc $t \models_{\mathfrak{T}} [r][x_1][x_2]I * \top$, donc $t \models_{\mathfrak{T}} [\![R(a_1, a_2)]\!]_{FO}$;
2. $t \models_{\mathfrak{T}} [\![R(a_1, a_2)]\!]_{FO}$ donc $t \models_{\mathfrak{T}} [r][x_1][x_2]I * \top$. Par définition, on a $t \sqsubseteq t'|t''$ tels que $t \models_{\mathfrak{T}} [r][x_1][x_2]I$ et $t'' \models_{\mathfrak{T}} \top$. Donc par définition, $R(a_1, a_2) \in \mathcal{S}$ et $\mathcal{S} \models R(a_1, a_2)$.

- Cas $\exists x.\psi$:

1. $\mathcal{S} \models \exists x.\psi$ donc par définition il existe $a \in \mathcal{D}$ tel que $\mathcal{S} \models \psi\{^a/_x\}$. Donc par définition, il existe t' tel que $t = t'|(e, d \mapsto (e, a \mapsto (e, nil)))$ et donc $t \models_{\mathfrak{T}} [d][a]I * \top$. Par hypothèse d'induction, on a également $t \models_{\mathfrak{T}} [\![\psi\{^a/_x\}]\!]_{FO}$. Donc $t \models_{\mathfrak{T}} ([d][a]I * \top) \wedge [\![\psi\{^a/_x\}]\!]_{FO}$ et par conséquent $t \models_{\mathfrak{T}} \exists x.(([d][x]I * \top) \wedge [\![\psi]\!]_{FO})$;
2. $t \models_{\mathfrak{T}} [\![\exists x.\psi]\!]_{FO}$, donc par définition, $t \models_{\mathfrak{T}} \exists x.(([d][x]I * \top) \wedge [\![\psi]\!]_{FO})$, donc, il existe a tel que $t \models_{\mathfrak{T}} [d][a]I * \top$ et $t \models_{\mathfrak{T}} [\![\psi\{^a/_x\}]\!]_{FO})$. Par définition, comme $t \models_{\mathfrak{T}} [d][a]I * \top$, il existe t' tel que $t = t'|(e, d \mapsto (e, a \mapsto (e, nil)))$ et donc, $a \in \mathcal{D}$. De même, par hypothèse d'induction, comme $t \models_{\mathfrak{T}} [\![\psi\{^a/_x\}]\!]_{FO})$, on a : $\mathcal{S} \models \psi\{^a/_x\})$. Donc, $\mathcal{S} \models \exists x.\psi$.

- Cas $\psi \vee \psi'$:

$\mathcal{S} \models \psi \vee \psi'$ ssi $\mathcal{S} \models \psi$ ou $\mathcal{S} \models \psi'$. D'après l'hypothèse d'induction, $\mathcal{S} \models \psi$ ssi $t \models_{\mathfrak{T}} [\![\psi]\!]_{FO}$. De même, $\mathcal{S} \models \psi'$ ssi $t \models_{\mathfrak{T}} [\![\psi']\!]_{FO}$. Donc, on a bien $\mathcal{S} \models \psi \vee \psi'$ ssi $t \models_{\mathfrak{T}} [\![\psi \vee \psi']\!]_{FO}$.

□

Nous pouvons alors établir le résultat d'indécidabilité suivant :

Théorème 3.7 (Indécidabilité de la validité dans BI-Loc$\forall$, sans $-\!\!*$). *Soit un modèle d'arbre partiel $\mathcal{T} = (T, (e, nil), |, \sqsubseteq_T, \models_{\mathfrak{T}})_{\mathcal{M},Loc}$ où $(T, (e, nil), |, \sqsubseteq_T)$ est un monoïde d'arbres partiel maximalement défini et où $\mathcal{M}$ est un monoïde partiel bornable. On ne peut décider de la validité d'une formule ϕ de* BI-Loc$_\forall$ *sans $-\!\!*$ pour ce modèle.*

Preuve. Conséquence directe du lemme 3.11. □

De plus, par le lien entre satisfaction et validité, l'indécidabilité de la satisfaction pour BI-Loc$_\forall$ est un corollaire du théorème précédent :

Théorème 3.8 (Indécidabilité de la satisfaction dans BI-Loc$_\forall$). *Soit un modèle d'arbre partiel $\mathcal{T} = (T, (e, nil), |, \sqsubseteq_T, \models_{\mathfrak{T}})_{\mathcal{M},Loc}$ où $(T, (e, nil), |, \sqsubseteq_T)$ est un monoïde d'arbres partiel maximalement défini et où $\mathcal{M}$ est un monoïde partiel bornable. . On ne peut décider de la satisfaction d'une formule ϕ de* BI-Loc$_\forall$ *par un arbre $t \in T$.*

Preuve. Conséquence directe du théorème 3.7 et du lemme 2.1. □

3.2 Preuves et tableaux sémantiques dans BI-Loc

Dans ce chapitre, on propose l'élaboration d'une méthode de recherche de preuves basée sur la méthode des tableaux [9, 42]. D'autres méthodes, comme celle des connexions [46, 68] ou des réseaux de preuves [8, 11] auraient pu être considérées mais elles ne sont pas abordées dans cette thèse. BI-Loc étant une logique dont le noyau est la logique BI [76, 79], il semble naturel de s'inspirer de la méthode de preuves par tableaux sémantiques réalisés pour cette logique (TBI) [45, 49, 72]. Le principe de cette méthode est de chercher à prouver une formule en montrant qu'il n'est pas possible de construire un contre-modèle respectant la sémantique. Par conséquent, on aboutit soit à une preuve d'une formule, soit à l'expression d'un contre-modèle indiquant pourquoi une formule donnée est non prouvable, ce qui permet d'analyser pourquoi une formule est fausse. Dans le cadre de la preuve de programmes, exposée au chapitre précédent, il s'agit de montrer par exemple qu'une assertion n'en implique pas une autre.

La principale difficulté de cette méthode est de capturer la sémantique lors de la décomposition en tableaux. Il faut en effet s'assurer qu'une formule est vraie ou fausse relativement à un arbre de ressources donnée. Pour cela, on définit un système de labels proche de celui utilisé pour BI [45, 49, 72], dans l'esprit des *«Labelled Deductive System»* [40, 43]. Les labels permettent de guider la recherche de preuves en représentant les contraintes sémantiques inhérentes à la décomposition des formules.

On commence donc par présenter les labels et les contraintes sémantiques sur ces labels. La forme de nos labels est différente de la forme usuelle des systèmes déductifs avec labels car ils doivent permettre de représenter la notion d'emplacement. On établit ensuite la relation entre contraintes et labels d'une part et arbres de ressources d'autre part, en introduisant notamment la structure des *graphes d'arbres de ressources*. Puis, on présente le système des tableaux TBI-Loc pour la logique BI-Loc. On introduit tout d'abord les règles de décomposition et les critères de clôture des tableaux déterminant la prouvabilité dans BI-Loc. Après quelques exemples illustrant le système de preuves et quelques spécificités de BI-Loc, on montre que ce système de preuves est correct et complet. On explique ensuite au travers d'un exemple comment construire un contre-modèle en cas de non-prouvabilité et on termine en donnant des arguments pour établir la terminaison de TBI-Loc et donc la décidabilité de BI-Loc (propositionnelle).

Contrairement à TBI [45, 49, 72], quand la construction complète d'un tableau ne permet pas d'affirmer que la formule est prouvable, on ne peut pas toujours trouver directement un contre-modèle à l'aide des contraintes générées. Celles-ci peuvent être trop fortes pour construire ce contre-modèle ou trop faibles pour le définir précisément. Dans le premier cas, la formule est prouvable même si le tableau n'est pas clos. Dans le second cas, on doit vérifier que l'on peut construire un contre-modèle vérifiant les contraintes déjà générées.

3.3 Un système déductif avec labels

Les récents travaux sur la recherche de preuves dans BI [72] ont permis de mettre en évidence comment l'utilisation de labels et de contraintes pouvait guider la recherche de preuves et la construction de contre-modèles à travers une méthode des tableaux pour BI (TBI) [49]. Les travaux réalisés proposent une *méthodologie générale* de la recherche de preuves dans les logiques de ressources. Cette méthodologie et sa mise en œuvre pour BI propositionnelle est un point de départ très intéressant pour étudier la recherche de preuve dans BI-Loc. En effet, comme nous l'avons vu précédemment pour le problème du *model-checking*, les raisonnements pour BI-Loc utilisent les bases posées pour BI.

Si les travaux proposés pour BI constituent donc une base indispensable, il reste toutefois à y intégrer tous les problèmes liés à la représentation de l'espace. Par conséquent, la solution la plus adaptée consiste à inclure à la méthodologie générale proposée avec TBI la notion d'emplacement afin d'obtenir une *méthodologie générale* non seulement pour les logiques de ressources mais également pour les logiques avec emplacements.

Dans TBI, les contraintes et les labels capturent la relation de conséquence sémantique induite par la décomposition des formules. Par conséquent, les labels sont fortement liés aux ressources. L'adaptation aux arbres de ressources a donc nécessité d'intégrer aux labels la notion d'emplacement.

Les contraintes et les labels permettent de guider la construction d'un monoïde d'arbres partiel. On explique ici quelles sont les conditions nécessaires pour pouvoir complètement déterminer ce monoïde et le construire.

3.3.1 Labels localisés

Les labels que nous utilisons pour BI-Loc incluent des emplacements pour prendre en compte la dimension spatiale de la recherche de preuves. Nous appelons ces labels des *labels localisés*. Le langage de labels localisés que nous utilisons contient donc les symboles suivants :

- l'unité 1 ;
- un ensemble dénombrable $\mathcal{C}$ de symboles de constantes $c, c_0, c_1, \ldots$;
- l'ensemble dénombrable Loc d'emplacements $l, l_0, l_1, \ldots$;
- la fonction unaire d'enracinement $[\cdot]$;
- la fonction binaire de composition $\times$;
- l'opérateur de marquage $\star$;
- la relation d'ordre binaire $\leq$.

La fonction d'enracinement permet d'indiquer qu'un label est à un emplacement donné.

Définition 3.7 (Contraintes et labels localisés). $\mathcal{L}_0 = \{1\} \cup \mathcal{C}$ *est l'ensemble des labels atomiques. L'ensemble des* labels localisés*, noté* $\mathcal{L}$ *est le plus petit ensemble qui est clos par composition et par enracinement, on a donc :*

1. $\mathcal{L}_0 \subseteq \mathcal{L}$;
2. $\forall x, y \in \mathcal{L}, x \times y \in \mathcal{L}$;
3. $\forall x \in Loc, \forall y \in \mathcal{L}, [x]y \in \mathcal{L}$.

Une contrainte *entre deux labels est une expression de la forme* $x \leq y$. *Un label est dit* pur *s'il ne contient pas de modalité d'emplacement. Un* label marqué *est une expression de la forme* $x^\star$ *où* x *est pur.*

Dans la suite de ce chapitre, nous établissons la relation entre les arbres de ressources. Lors de l'établissement de cette relation, certains labels doivent correspondre à des arbres de ressources alors que d'autres correspondent uniquement à des ressources. Le marquage d'un label permet d'indiquer que l'on est certain que le label en question corresponde à une ressource. Les règles de construction des tableaux montrent comment et pourquoi ce marquage est nécessaire.

L'ensemble des contraintes entre les labels $\{x \leq y \ / \ x, y \in \mathcal{L}\}$ est noté $\mathcal{K}$, l'ensemble des labels marqués $\{x^\star \ / \ x \in \mathcal{L} \textit{ et } x \textit{ est pur}\}$ est noté $\mathcal{M}$.

Si $x \leq y, y \leq x \in \mathcal{K}$, on le notera plus concisement : $x \cong y \in \mathcal{K}$. De même, pour alléger les notations ultérieures, on notera $x \sim y$ quand on a indifféremment $x \leq y$ ou $y \leq x$. L'ensemble des labels, des labels marqués et des contraintes qui leur sont associées $\mathcal{L} \cup \mathcal{M} \cup \mathcal{K}$ est noté $\mathcal{LMK}$.

Pour un sous-ensemble $X \subseteq \mathcal{LMK}$ (que l'on désignera par abus de langage et pour simplifier un ensemble de labels et de contraintes, alors qu'il contient également des labels marqués), on désignera par $\mathcal{L}(X)$ l'ensemble $\{x \ / \ x \in X \textit{ et } x \in \mathcal{L}\}$ des labels de X. Nous nommerons cet ensemble le *domaine* de X. De même, $\mathcal{C}(X)$ désigne l'ensemble des constantes de X, $\mathcal{K}(X)$ l'ensemble des contraintes de X et $\mathcal{M}(X)$ l'ensemble des labels marqués de X.

Pour ne pas surcharger les notations, si x et y sont des labels, on écrira simplement $x^\star \times y \in X$ pour indiquer que $x \times y \in X$ et que $x^\star \in X$. La fonction de composition $\times$ est soumise aux règles suivantes :

- Associativité : $x \times (y \times z) = (x \times y) \times z$;
- Commutativité : $x \times y = y \times x$;
- Neutralité : $1 \times x = x$;
- Fusion : $[l]x \times [l]y = [l](x \times y)$.

Conformément à la méthodologie présentée dans [72], le comportement de la fonction de composition des labels $\times$ est basé sur le comportement de l'opérateur de composition d'arbre $|$. Le but est de manipuler des labels qui reflètent de manière aussi proche que possible le comportement du modèle.

Ainsi, les labels $(1 \times [l](c_1 \times ([l_1]c_2 \times [l_2][l_3]c_3))$ et $[l]c_1 \times [l][l_1]c_2 \times [l][l_2]1 \times [l][l_2][l_3]c_3$ sont équivalents, ce que l'on note simplement $(1 \times [l](c_1 \times ([l_1]c_2 \times [l_2]c_3)) \equiv 1 \times [l]c_1 \times [l][l_1]c_2 \times [l][l_2]1 \times [l][l_2][l_3]c_3$. L'ensemble des labels comporte donc plusieurs labels équivalents. Par souci de clarté, nous continuerons malgré tout à parler de labels et de contraintes sur ces labels alors que nous manipulons en fait des classes d'équivalence de labels et des contraintes sur ces classes d'équivalence.

On propose pour chaque label une forme normale (modulo la commutativité) de ce label. Celle-ci est définie comme suit :

Définition 3.8 (Forme normale d'un label). *Soit x un label, x est en* forme normale *si $x = x_0 \times [L_1]x_1 \times \ldots \times [L_n]x_n$ avec :*
- *$n \geq 0$, $L_i \in Loc^*$, x_i pur ;*
- *$\forall L_i, L, L'$ tels que $L : L' = L_i$*[3]*, $L, L' \in \{L_1, \ldots, L_n\}$.*

Tout label possède donc une forme normale. On considère par la suite un label comme étant en forme normale sans perte de généralité. Dans l'exemple d'égalité de label présenté ci-dessus, le label de gauche est la forme normale du label de droite.

Pour alléger les notations, nous omettons lorsqu'il n'est pas nécessaire l'écriture de $\times$. Par exemple, on note $x[L_1]y[L_2]z$ le label $x \times [L_1]y \times [L_2]z$. Pour tout label en forme normale, l'écriture de $\times$ n'est pas nécessaire.

Définition 3.9 (Sous-labels). *La longueur d'un label x, notée $|x|$, est définie récursivement :*
- $|1| = 0$; - $|c_i| = 1$;
- $|[l]x| = |x|$; - $|x \times y| = |x| + |y|$;

Le label y est un sous-label du label x (noté $y \subseteq x$) s'il existe un label z tel que $z \times y = x$. Le sous-label y de x est dit strict (noté $y \subset x$) si $|y| < |x|$.

Étant donné un label x, nous notons $\mathcal{S}(x)$ l'ensemble $\{y \ / \ y \subseteq x\}$ des sous-labels de x. Il est important de signaler que nous travaillons ici sur les classes d'équivalence des labels plutôt que sur les labels eux-mêmes. En effet, si l'on se place au niveau des labels, l'ensemble $\mathcal{S}(x)$ est infini puisque $x1, x11, x111 \in \mathcal{S}(x)$. Par contre, si on se place au niveau des classes d'équivalence (ce qui est notre cas), on a alors un nombre fini de sous-labels et donc $\mathcal{S}(x)$ est fini.

3.3.2 Graphes de ressources

Nos labels sont fortement liés aux modèles d'arbres partiels. Nous montrons ici comment nous pouvons obtenir un monoïde de ressources à partir d'un tel ensemble de labels et de contraintes. Nous utilisons pour cela un opérateur de clôture. Nous allons ensuite extraire les informations de cette clôture et nous les représenterons sous forme de *graphes d'arbres de ressources*. Commençons par introduire la terminologie des ensembles de ressources et de contraintes qui est utilisé dans cette section.

Une ensemble X est dit *complètement défini* si pour tout $x \in \mathcal{L}(X)$ tel que $x \notin \mathcal{M}(X)$, il existe une contrainte $x \cong y_0[L_1]y_1 \ldots [L_n]y_n \in X$ avec $y_i \in \mathcal{M}(X)$.

On définit alors ce qu'est la valuation d'un ensemble de labels et de contraintes.

Définition 3.10 (Valuation). *Soit X un ensemble de labels et de contraintes. Une* valuation *X_v de X est obtenue en rajoutant à X pour chaque label non marqué $x \in X$ une contrainte du type*

$$x \cong [L_1]y_1^\star | \ldots [L_n]y_n^\star$$

avec $n \geq 1$, $c_1^\star, \ldots c_n^\star \in \mathcal{C}$ et $L_1, \ldots, L_n \in Loc^*$.

Un ensemble complètement défini fixe de manière précise la forme de l'arbre de ressources qui doit être associé à chaque label pour extraire un contre-modèle. En effet, on connaît exactement le

[3]pour rappel, $L : L'$ représente la concaténation des chemins L et L'

chemin qui le compose et on sait qu'au bout de chacun de ces chemins on trouvera des ressources. Il se peut cependant que les contraintes soient alors trop fortes pour qu'une telle construction soit possible.

Nous définissons donc le critère d'incohérence d'un ensemble de labels et de contraintes. Une incohérence indique que l'ensemble des contraintes générées est trop fort pour permettre de construire un modèle d'arbre respectant les contraintes.

Définition 3.11 (Incohérence structurelle). *Soit un ensemble X de labels et de contraintes. L'ensemble X est* incohérent *si :*

- *il existe des labels $x^\star, y, z \in X$ et un chemin L tel que $x^\star \sim y[L]z \in X$;*
- *il existe des emplacements $l, l_1, \dots, l_n, l'_1, \dots, l'_n$ (avec $n \geq 0$) et des labels $y, y', z_0, \dots, z_n, z'_1, \dots, z'_k$ tels que :*
  - $x \sim y^\star[l]z_0[l_1]z_1 \dots [l_n]z_n \in X$ *et*
  - $x \sim y'^\star[l'_1]z'_1 \dots [l'_k]z'_k \in X$ *et*
  - $l \notin \{l'_1, \dots l'_n\}$.

Un ensemble qui n'est pas incohérent est dit potentiellement cohérent*.*

Un ensemble potentiellement cohérent *et complètement défini est dit* cohérent*.*

Le premier cas d'incohérence indique que l'ensemble est incohérent quand un label marqué (qui doit donc correspondre à une ressource) est en relation avec un label correspondant à un arbre. Le second cas quant à lui indique qu'un label ne peut être similaire à (au moins) deux labels localisés n'ayant pas les mêmes emplacements.

On peut maintenant définir la $(\cdot)^\dagger$-clôture d'un ensemble de contraintes.

Définition 3.12 ($(\cdot)^\dagger$-clôture). *Soit X un ensemble de labels et de contraintes entre ces labels. Nous définissons $X^\dagger$ comme le plus petit ensemble de X tel que :*

- *Complétion :* $x \leq y \in X^\dagger \Rightarrow x, y \in X^\dagger$; $x^\star \sim y \in X^\dagger ety$ *est pur* $\Rightarrow y^\star \in X^\dagger$;
  $x^\star \in X^\dagger \bigwedge y^\star \in X^\dagger \bigwedge xy \in X^\dagger \Leftrightarrow (x \times y)^\star \in X^\dagger$;
- *Saturation :* $x \in X^\dagger ety \subseteq x \Rightarrow y \in X^\dagger$;
- *Réflexivité :* $x \in X^\dagger \Rightarrow x \leq x \in X^\dagger$;
- *Transitivité :* $x \leq y \in X^\dagger \bigwedge y \leq z \in X^\dagger \Rightarrow x \leq z \in X^\dagger$;
- *Compatibilité :* $y \times z \in X^\dagger \bigwedge y \leq x \in X^\dagger \Rightarrow y \times z \leq x \times z \in X^\dagger$;
  $x_1 \leq x_2 \in X^\dagger$, *si* $y[l]x_1 \in X^\dagger$ *ou* $y[l]x_2 \in X^\dagger$, $y[l]x_1 \leq y[l]x_2 \in X^\dagger$;
- *Similarité :* $y_0^\star \times [l_1]y_1 \times \dots \times [l_n]y_n \leq y_0'^\star \times [l_1]y'_1 \times \dots \times [l_n]y'_n \in X^\dagger \Rightarrow \forall i. y_i \leq y'_i$;
  $x \sim y_0^\star \times [l_1]y_1 \times \dots \times [l_n]y_n et \forall c_0, \dots, c_n . x \ncong c_0^\star \times [l_1]c_1 \times \dots \times [l_n]c_n \Rightarrow x \cong c_0'^\star \times [l_1]c'_1 \times \dots \times [l_n]c'_n$ *où les c'_i sont de nouvelles constantes.*

L'égalité sémantique entre les labels peut mener à ce que l'ensemble des contraintes contienne des informations redondantes. On définit pour cela une normalisation d'un ensemble de labels et de contraintes visant à limiter les informations redondantes présentes. L'idée de la normalisation est de supprimer les constantes qui sont équivalentes à un label plus développé.

Définition 3.13 (Normalisation). *Soit X un ensemble de labels et de contraintes. La* normalisation *$n(X)$de X est définie comme suit :*

$$\begin{array}{rcll} n(X) & = & X- & \{x[L]y \ / \ y \in \mathcal{C}(X) et \exists z \in X - \mathcal{C}(X). y \cong z\} \cup \\ & & & \{x \leq y'[L]y \ / \ y \in \mathcal{C}(X) et \exists z \in X - \mathcal{C}(X). y \cong z\} \cup \\ & & & \{y'[L]y \leq x \ / \ y \in \mathcal{C}(X) et \exists z \in X - \mathcal{C}X. y \cong z\} \end{array}$$

Définition 3.14 (Graphe de ressources). *Soit X un ensemble de labels et de contraintes. Le* graphe d'arbres de ressources *associé à X, noté $\mathcal{G}(X)$ est le graphe orienté $[N(X), E(X)]$ où l'ensemble des nœuds est donné par l'ensemble des labels purs de $X^\dagger$ et où l'ensemble des arêtes est déterminé par l'ensemble des contraintes de $X^\dagger$ entre ces labels.*

Considérons maintenant un exemple de construction de graphes. Prenons l'ensemble de labels et de contraintes : $X = \{c_1[l]c_2 \leq c_3c_4, c_4 \cong [l]c_5, c_1^\star, c_3^\star\}$. Les conditions de $(\cdot)^\dagger$-clôture nous permettent d'obtenir consécutivement :

- par complétion, $X_1 = X \cup \{1, c_1[l]c_2, c_3c_4, c_4, [l]c_5\}$;
- par compatibilité, $X_2 = X_1 \cup \{c_1[l]c_2 \leq c_3[l]c_5, c_3c_4 \cong c_3[l]c_5\}$;
- par complétion, $X_3 = X_2 \cup \{c_3[l]c_5\}$;
- par similarité, $X_4 = X_3 \cup \{c_1 \leq c_3, c_2 \leq c_5\}$;
- par complétion, $X_5 = X_4 \cup \{[l]1, c_1, c_3, c_2, c_5\}$;
- par compatibilité, $X_6 = X_5 \cup \{[l]c_2 \leq [l]c_5, c_3[l]c_2 \leq c_3[l]c_5, c_1[l]c_2 \leq c_3[l]c_2, c_1[l]c_2 \leq c_1[l]c_5, c_1[l]c_5 \leq c_3[l]c_5, c_1c_4 \cong c_1[l]c_5, c_1c_4 \leq c_3c_4\}$;
- par complétion, $X_7 = X_6 \cup \{c_1c_4, c_1[l]c_5\}$.

L'ensemble $X^\dagger$ s'obtient par clôture réflexive et transitive de X_7. Nous obtenons donc les ensembles de noeuds et d'arcs suivants :

- $N(X) = \{1, c, c_1, c_2, c_3, c_5, \}$
- $E(X) = \{c_5 \to c_2, c_3 \to c_1\}$

Ce qui correspond à la représentation suivante :

$$\begin{tikzcd} \boxed{c_3} & \boxed{c_1} \arrow[l] \\ \boxed{1} & \\ \boxed{c_5} & \boxed{c_2} \arrow[l] \end{tikzcd}$$

Le graphe de ressources, s'il correspond à un ensemble cohérent permet de déterminer complètement un monoïde de ressources partiel qui sert de base à la construction d'un monoïde d'arbres partiels. Ce dernier est appelé *monoïde atomique partiel associé.*

Définition 3.15 (Monoïde atomique partiel associé). *Soit $\mathcal{G}(X) = [N(X), E(X)]$ le graphe associé à un ensemble X. Le* monoïde atomique partiel *associé à X est la structure $\langle T, 1, \bullet, \sqsubseteq\rangle$ telle que :*

- $T = \{t \;/\; t | \forall L.t(L) = (m, t'), m \in N(X) et t' \in T\}$;
- $\bullet$ *est la loi de composition vérifiant :*
  - $t|1 = 1|t = t$;
  - $[t_1|t_2]\uparrow$ *ssi* $\forall L.\exists t_1', t_2, m_1, m_2.t_1(L) = (m_1, t_1'),\ t_1(L) = (m_2, t_2')$ *et* $m_1 \times m_2 \in N(X)$;
  - $t_1|t_2$ *n'est pas défini sinon ;*

- $\sqsubseteq$ *est la relation définie par :*
  - $\forall t_1, t_2 \in M.t_1 \sqsubseteq t_2$ *si et seulement si* $\forall L.\exists t_1', t_2, m_1, m_2.t_1(L) = (m_1, t_1'),\ t_1(L) = (m_2, t_2'),\ m_1 \to m_2 \in E(X)$ *et* $t_1' \sqsubseteq t_2'$.

Lemme 3.12. Le monoïde atomique partiel $\langle M, 1, \bullet, \sqsubseteq\rangle$ associé à un ensemble X de labels et de contraintes est un monoïde de ressources partiel.

Preuve. Par définition du monoïde associé, on sait déjà que la loi $\bullet$ est interne sur M et qu'elle admet 1 comme élément neutre. L'associativité de $\bullet$ est une conséquence directe de la condition de saturation de $(\cdot)^\dagger$ et de l'associativité de $\times$. Il est également évident de part la transitivité et la réflexivité de la $(\cdot)^\dagger$-clôture que $\sqsubseteq$ est un pré-ordre. Reste alors à montrer que ce pré-ordre est compatible avec $\bullet$. Pour cela, supposons que l'on ait $x, y \in M$ tels que $x \sqsubseteq y$. Soit $z \in M$ tel que $[z \bullet x]\uparrow$ and $[z \bullet y]\uparrow$ d'après la condition de compatibilité de la définition 3.12 on a $zx \sqsubseteq zy$. □

On remarque ensuite qu'un label sous sa forme normale peut être alors vu comme la représentation d'un arbre sous sa forme grammaticale (celle de la définition 2.4).

Définition 3.16 (Monoïde partiel associé). *Soit* $\mathcal{G}(X) = [N(X), E(X)]$ *le graphe associé à un ensemble* X. *Le* monoïde partiel *associé à* X *est la structure* $\langle T, (1, nil), |, \sqsubseteq\rangle$ *telle que :*

- $T = \{[\![x]\!]_{card} \ / \ x \in N(X)\}$;
- $|$ *est la loi de composition vérifiant :*
  - $[\![x]\!]_{card}|(1, nil) = (1, nil) \times [\![x]\!]_{card} = [\![x]\!]_{card}$;
  - $[[\![x]\!]_{card}|[\![y]\!]_{card}]\uparrow$ *si* $x \times y \in N(X)$;
  - $[\![x]\!]_{card}|[\![y]\!]_{card}$ *n'est pas défini sinon ;*
- $\sqsubseteq$ *est la relation définie par :*
  - $\forall x, y \in T.x \sqsubseteq y$ *si et seulement si* $x \to y \in E(X)$.

Lemme 3.13. Le monoïde partiel $\langle M, 1, \bullet, \sqsubseteq\rangle$ associé à un graphe $\mathcal{G}(X)$ est un monoïde d'arbres partiel construit sur l'ensemble d'emplacements *Loc* et le monoïde atomique partiel associé à $\mathcal{G}(X)$.

Preuve. Par définition du monoïde partiel associé à $\mathcal{G}(X)$, il est clair que $T \subseteq T_{M,Loc}$. De même, par définition, la loi $|$ est interne sur T et elle admet $(1, nil)$ comme élément neutre. L'associativité de $|$ est une conséquence directe de la condition de saturation de $(\cdot)^\dagger$ et de l'associativité de $\times$.

De part la transitivité et la réflexivité de la $(\cdot)^\dagger$-clôture, on montre que $\sqsubseteq$ est un pré-ordre et qu'il est compatible avec $|$ puisque la $(\cdot)^\dagger$-clôture est réflexive et transitive.

Il reste à démontrer que cet ordre partiel est bien une extension de celui du monoïde atomique, ce qui est une conséquence directe de la condition de similarité de la $(\cdot)^\dagger$-clôture. □

3.4 Tableaux pour BI-Loc

Nous avons proposé dans la section précédente une version des labels permettant de représenter au mieux les contraintes du modèle d'arbres partiel. On doit maintenant utiliser ces labels dans la construction de tableaux pour les formules de BI-Loc. Une démarche similaire a été proposée pour BI [45, 49, 72]. La construction des tableaux est toutefois ici légèrement plus complexe. Outre la gestion de la modalité d'emplacement $[\cdot]$, la principale difficulté est que l'algorithme de construction de tableaux proposé dans [72] ne permet pas toujours de déterminer complètement la forme d'un contre modèle lorsque la formule n'est pas prouvable. Nous détaillons plus loin dans cette section comment on solutionne ce problème.

3.4.1 Règles de décomposition

Nous commençons par introduire la notion de *formules signées*, une définition classique dans les preuves par tableaux sémantiques [42]. Dans le cas de BI-Loc la valeur de vérité d'une formule est relative à un arbre de ressources. Ainsi donc nous attribuons à nos formules signées des labels localisés qui codent les arbres de ressources qui sont associés aux formules.

Définition 3.17 (Formules signées). *Une* formule signée *est un triplet* (S, ϕ, x) *noté* $S\ A : x$*, où* $S \in \{T, F\}$ *est le signe de la formule* ϕ *et* $x \in \mathcal{L}$ *est son label.*

De plus, on est amené à devoir différencier les labels dont nous sommes sûrs qu'ils se rapportent à des ressources des autres labels. Nous introduisons pour cela un marquage des labels. Ainsi, on utilisera le label marqué $x^\star$ pour indiquer que x doit correspondre à une ressource. Enfin, nous qualifions de positives les formules signées dont le signe est T, et de négatives celles dont le signe est F.

Définition 3.18 (Tableaux). *Soit* ϕ *une formule de* BI-Loc*, un* tableau *pour* ϕ *est un arbre binaire obtenu par applications successives des règles de décomposition décrites à la figure 3.1 en respectant la structure de* ϕ*. Les noeuds de cet arbre sont étiquetés soit par des formules signées, soit par des contraintes entre labels ; la racine quant à elle est toujours étiquetées par* $F\ \phi : 1$.

La figure 3.1 décrit les règles de décomposition pour BI-Loc. Comme cette logique repose sur la même base que BI, les règles de décomposition pour les opérateurs communs aux deux logiques sont donc identiques. Ainsi, les règles de décomposition pour $\wedge$ et $\vee$ (première ligne) sont donc les règles standard de la logique intuitionniste pour la conjonction et la disjonction [84]. Ce sont des règles dites *propagatives* car les conclusions de la règle comportent les mêmes labels que l'hypothèse.

Les règles de la deuxième et de la troisième lignes sont des règles dite *assertives*. Ces règles sont nommées ainsi car elles introduisent de nouvelles constantes et de nouvelles contraintes entre labels appelées *assertions*. Ces contraintes sont identifiées par l'abréviation *ass* dans les règles de la figure 3.1. Une assertion est une contrainte factuelle que l'on impose à nos labels. Ainsi par exemple, la formule $T\ [l]\psi : x$ est décomposée en une formule signée $T\ \psi : c_i$ où c_i est une nouvelle constante sur laquelle on impose la contrainte suivante $x \cong c_i$. Les contraintes de réflexivité sont quant à elles implicites pour tout label qui vient d'être introduit. Les deux dernières règles (celle concernant les formules $T\ I : x$ et $T\ p : x$) introduisent le marquage des labels. En effet, seule une ressource peut vérifier I ou une proposition. Pour différencier ces labels de ceux dont on ne connaît pas la nature exacte, on le marque.

La quatrième ligne contient les règles *obligationnelles* qui décomposent une formule en réutilisant les labels déjà définis et introduisent des nouvelles contraintes qui sont à vérifier. Les contraintes de ce type se distinguent par l'utilisation de l'abréviation *req (requirement)* en début de règle. Par exemple, la formule $F\ [l]\phi : x$ est décomposée en $F\ \phi : y$ où y est un label déjà défini qui est censé satisfaire $x \leq [l]y$. Enfin, le système ne contient pas de règles pour $\top$ et $\bot$. En effet, il est plus simple de considérer ces unités en construisant des règles de clôture adaptées pour les tableaux.

Avant de continuer, nous présentons quelques notations relatives aux tableaux qui seront utilisées dans la suite de ce document. Soit $\mathcal{B}$ une branche d'un tableau $\mathcal{T}$, $Ass(\mathcal{B})$ et $Req(\mathcal{B})$ représentent respectivement l'ensemble des assertions et des obligations de $\mathcal{B}$. De la même manière, $Ass(\mathcal{T})$ et $Req(\mathcal{T})$ représentent l'ensemble des assertions (resp. des obligations) du tableau $\mathcal{T}$. Enfin, on établit une relation d'ordre partiel reposant sur l'ordre d'apparition des objets dans le

$$\begin{array}{c} F\ \phi \vee \psi : x \\ \hline F\ \phi : x \\ F\ \phi' : x \end{array} \qquad \begin{array}{c} T\ \phi \vee \psi : x \\ \hline T\ \phi : x \quad \mid \quad T\ \phi' : x \end{array} \qquad \begin{array}{c} T\ \phi \wedge \phi' : x \\ \hline T\ \phi : x \\ T\ \phi' : x \end{array} \qquad \begin{array}{c} F\ \phi \wedge \phi' : x \\ \hline F\ \phi : x \quad \mid \quad F\ \phi' : x \end{array}$$

$$\begin{array}{c} F\ \phi \to \phi' : x \\ \hline \boxed{ass : c_i \leq x} \\ \hline T\ \phi : c_i \\ F\ \phi' : c_i \end{array} \qquad \begin{array}{c} T\ \phi * \phi' : x \\ \hline \boxed{ass : x \leq c_i c_j} \\ \hline T\ \phi : c_i \\ T\ \phi' : c_j \end{array} \qquad \begin{array}{c} F\ \phi \mathrel{-\!\!*} \phi' : x \\ \hline T\ \phi : c_i \\ F\ \phi' : x c_i \end{array}$$

$$\begin{array}{c} T\ [l]\phi : x \\ \hline \boxed{ass : x \cong [l]c_i} \\ \hline T\ \phi : c_i \end{array} \qquad \begin{array}{c} T\ I : x \\ \hline \boxed{ass : x \leq 1 \wedge x^\star} \\ \hline \end{array} \qquad \begin{array}{c} T\ p : x \\ \hline \boxed{ass : x^\star} \\ \hline \end{array}$$

avec c_i, c_j de nouvelles constantes

$$\begin{array}{c} F\ [l]\phi : x \\ \hline \boxed{req : x \leq [l]y} \\ \hline F\ \phi : y \end{array} \qquad \begin{array}{c} T\ \phi \to \phi' : x \\ \hline \boxed{req : y \leq x} \\ \hline F\ \phi : y \quad \mid \quad T\ \phi' : y \end{array} \qquad \begin{array}{c} F\ \phi * \phi' : x \\ \hline \boxed{req : x \leq yz} \\ \hline F\ \phi : y \quad \mid \quad F\ \phi' : z \end{array} \qquad \begin{array}{c} T\ \phi \mathrel{-\!\!*} \phi' : x \\ \hline F\ \phi : y \quad \mid \quad T\ \phi' : xy \end{array}$$

avec y, z, $x \times y$ des labels déjà définis

FIG. 3.1 – Règles de décomposition de TBI-Loc

tableau. Soient deux objets x et y de $\mathcal{T}$, on note $x \ll y$ (ce qui se lit x précède y) dans $\mathcal{T}$, si x et y appartiennent à la même branche $\mathcal{B}$ de $\mathcal{T}$ et si x a été introduit avant y dans $\mathcal{B}$.

3.4.2 Conditions d'admissibilité

Les règles de décomposition définies ci-dessus introduisent deux types de contraintes. Les contraintes *assertives*, qui sont décrétées vraies et les contraintes *obligatives*, que l'on doit vérifier. On définit donc un critère d'admissibilité pour ces dernières. Il s'agit de déterminer les valuations qui vérifient une obligation donnée. Pour cela, on commence par introduire l'ensemble des assertions qui précèdent $x \leq y$ noté $Ass(\mathcal{B}, x \leq y)$.

Formellement, on a $Ass(\mathcal{B}, x \leq y) = \{x' \leq y \ / \ x' \leq y \in Ass(\mathcal{B}) \bigwedge x' \leq y' \ll x \leq y\}$.

Définition 3.19 (Tableau admissible). *Une obligation $x \leq y$ apparaissant dans une branche $\mathcal{B}$ d'un tableau $\mathcal{T}$ est* admissible *si elle appartient à la $(\cdot)^\dagger$-clôture des assertions qui la précèdent. Une branche est admissible si toutes ses obligations le sont, un tableau est admissible si toutes ses branches le sont. Formellement, le critère d'admissibilité pour un tableau est le suivant :*
$\forall \mathcal{B} \in \mathcal{T}. \forall x \leq y \in Req(\mathcal{B}). x \leq y \in Ass(\mathcal{B}, x \leq y)^\dagger$

Ainsi, on parle d'une *obligation satisfaite* quand une obligation satisfait la condition d'admissibilité. Comme les conditions de clôture n'introduisent pas de nouveaux labels, il est nécessaire pour satisfaire une obligation d'utiliser des labels déjà introduits dans le graphe de ressources associé à $\mathcal{B}$ au moment où la règle s'applique. Cela limite donc les choix possibles pour instancier les règles obligationnelles. L'admissibilité est un critère incontournable pour la bonne construction d'un tableau. Dans la suite de ce chapitre, nous ne considérerons donc que des tableaux admissibles.

Selon le type de règle qu'on peut lui appliquer, une formule peut servir une ou plusieurs fois. Ainsi, les formules nécessitant l'usage de règles *propagatives* (ligne 1 de la figure 3.1) ou *assertives* (lignes 2 et 3 de la figure 3.1) ne seront utilisés qu'une seule fois. Plus clairement, une fois ces formules décomposées, on ne peut les réutiliser. À l'opposé, les formules nécessitant une règle obligationnelle (ligne 4 de la figure 3.1) peuvent être instanciées plusieurs fois avec des labels différents puisque des labels différents peuvent satisfaire la même obligation. Ceci peut d'ailleurs mené, dans certaines configurations détaillées dans [72] à des branches infinies.

Il est important de souligner qu'à part pour la règle concernant les formules de type $F\ [l]\phi : x$, il existe toujours au moins une instanciation triviale des règles obligationnelles données en figure 3.2.

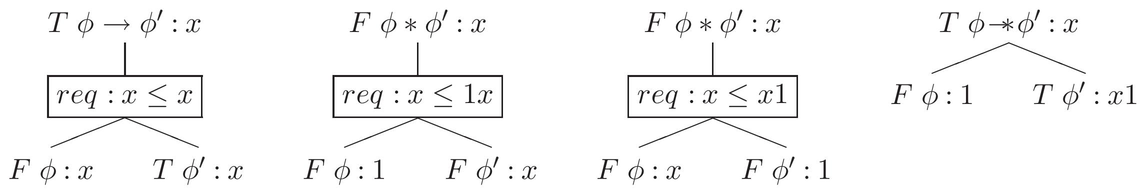


Fig. 3.2 – Règles de décomposition triviale de TBI-Loc

3.4.3 Clôture logique d'un tableau

Les méthodes de preuve par tableaux sémantiques sont des méthodes de preuves par réfutation. Nous devons donc définir maintenant les conditions qui nous permettent de clore une branche ou de déclarer qu'une branche ne peut être close. Pour clore une branche, nous devons être dans l'impossibilité de construire un contre-modèle de la formule initiale respectant les contraintes décrites à l'intérieur de la branche.

Nous nous intéressons ici à l'*impossibilité logique* de construire un contre-modèle, qui peut être due à une contradiction concernant les propositions qui doivent être vérifiées par une ressource ou à l'existence d'une ressource qui mènerait à l'incohérence.

Définition 3.20 (Complémentarité). *Deux formules signées $T\ \phi : x$ sont* complémentaires *dans une branche $\mathcal{B}$ d'un tableau $\mathcal{T}$ si $y \leq x \in Ass(\mathcal{B})^{\dagger}$.*

Définition 3.21 (Labels et contraintes incohérents). *Soit une branche $\mathcal{B}$ d'un tableau $\mathcal{T}$. Le label x est dit* incohérent *dans $\mathcal{B}$ s'il existe un label y tel que $x \leq y \in Ass(\mathcal{B})^{\dagger}$ et un sous-label z de y tel que $T\ \bot : z \in \mathcal{B}$. Dans le cas contraire le label est dit cohérent.*

Par extension, une contrainte $x \leq y$ est incohérente *dans une branche $\mathcal{B}$ si x ou y est incohérent dans $\mathcal{B}$.*

Définition 3.22 (Branche close logiquement). *Une branche $\mathcal{B}$ d'un tableau $\mathcal{T}$ est* close *(ou contradictoire), s'il vérifie l'une des conditions suivantes :*

1. *(CL1) $\mathcal{B}$ contient deux formules signées $T\ \phi : x$ et $F\ \phi : x$ complémentaires ;*
2. *(CL2) $\mathcal{B}$ contient une formule signée $F\ I : x$ et $x \leq 1 \in Ass(\mathcal{B})^\dagger$;*
3. *(CL3) $\mathcal{B}$ contient une formule signée $F\ \top : x$;*
4. *(CL4) $\mathcal{B}$ contient une formule signée $F\ \phi : x$ avec x incohérent dans $\mathcal{B}$.*

Une branche non close logiquement est dite potentiellement ouverte.

3.4.4 Complétion et clôture structurelle

Les formules obligationnelles peuvent être décomposées plusieurs fois dans le cadre de la construction d'un tableau. On introduit ici la notion de *formule potentiellement complète* pour indiquer que l'on a décomposé une formule de toutes les manières possibles relativement au graphe de ressources associé à la branche que l'on étudie.

Définition 3.23 (Formule analysée). *Une formule signée $S\ \phi : x$ est dite* analysée *dans une branche $\mathcal{B}$ d'un tableau $\mathcal{T}$ (ce qui est noté $\mathcal{B} \succ S\ \phi : x$) si et seulement si :*
- *$S \equiv F$ et il existe $F\ \phi : y \in \mathcal{B}$ tel que $y \leq x \in Ass(\mathcal{B})^\dagger$ ou*
- *$S \equiv T$ et il existe $T\ \phi : y \in \mathcal{B}$ tel que $x \leq y \in Ass(\mathcal{B})^\dagger$.*

Définition 3.24 (Formule complète). *Nous définissons en fonction des valeurs de S et de ϕ la relation $\mathcal{B} \Vdash S\ \phi : x$ qui indique qu'une formule signée $S\ \phi : x$ est* potentiellement complète *dans une branche $\mathcal{B}$ d'un tableau $\mathcal{T}$:*

- *$\mathcal{B} \Vdash F\ p : x$ ssi $\mathcal{B} \succ F\ p : x$;*
- *$\mathcal{B} \Vdash T\ p : x$ ssi $\mathcal{B} \succ T\ p : x$;*
- *$\mathcal{B} \Vdash F\ \top : x$ ssi $\mathcal{B} \succ F\ \top : x$;*
- *$\mathcal{B} \Vdash T\ \top : x$ ssi $\mathcal{B} \succ T\ \top : x$;*
- *$\mathcal{B} \Vdash F\ I : x$ ssi $\mathcal{B} \succ F\ I : x$ et $x \leq 1 \notin Ass(\mathcal{B})^\dagger$;*
- *$\mathcal{B} \Vdash T\ I : x$ ssi $\mathcal{B} \succ T\ I : x$ et $x \leq 1 \in Ass(\mathcal{B})^\dagger$;*
- *$\mathcal{B} \Vdash F\ \bot : x$ ssi $\mathcal{B} \succ F\ \bot : x$;*
- *$\mathcal{B} \Vdash T\ \bot : x$ ssi $\mathcal{B} \succ T\ \bot : x$;*
- *$\mathcal{B} \Vdash F\ [l]\psi : x$ ssi il existe y tel que $x \leq [l]y \in Ass(\mathcal{B})^\dagger$ et $\mathcal{B} \succ F\ \psi : y$;*
- *$\mathcal{B} \Vdash T\ [l]\psi : x$ ssi pour tout y tel que $x \leq [l]y \in Ass(\mathcal{B})^\dagger$ et $\mathcal{B} \succ F\ \psi : y$;*
- *$\mathcal{B} \Vdash F\ \psi \wedge \psi' : x$ ssi $\mathcal{B} \succ F\ \psi : x$ ou $\mathcal{B} \succ F\ \psi : x$;*
- *$\mathcal{B} \Vdash T\ \psi \wedge \psi' : x$ ssi $\mathcal{B} \succ T\ \psi : x$ et $\mathcal{B} \succ T\ \psi : x$;*
- *$\mathcal{B} \Vdash F\ \psi \vee \psi' : x$ ssi $\mathcal{B} \succ F\ \psi : x$ et $\mathcal{B} \succ F\ \psi : x$;*
- *$\mathcal{B} \Vdash T\ \psi \vee \psi' : x$ ssi $\mathcal{B} \succ T\ \psi : x$ ou $\mathcal{B} \succ T\ \psi : x$;*
- *$\mathcal{B} \Vdash F\ \psi \to \psi' : x$ ssi il existe $y \in Ass(\mathcal{B})^\dagger$ tel que $y \leq x \in Ass(\mathcal{B})^\dagger$ et $\mathcal{B} \succ T\ \psi : y$ et $\mathcal{B} \succ F\ \psi' : y$;*
- *$\mathcal{B} \Vdash T\ \psi \to \psi' : x$ ssi pour tout $y \in Ass(\mathcal{B})^\dagger$, $y \leq x \in Ass(\mathcal{B})^\dagger \Rightarrow \mathcal{B} \succ F\ \psi : y$ ou $\mathcal{B} \succ T\ \psi' : y$;*
- *$\mathcal{B} \Vdash F\ \psi * \psi' : x$ ssi pour tous $y, z \in Ass(\mathcal{B})^\dagger$, $x \leq yz \in Ass(\mathcal{B})^\dagger \Rightarrow (\mathcal{B} \succ F\ \psi : y$ ou $\mathcal{B} \succ F\ \psi' : z)$;*
- *$\mathcal{B} \Vdash T\ \psi * \psi' : x$ ssi il existe $y, z \in Ass(\mathcal{B})^\dagger$ tel que $x \leq yz \in Ass(\mathcal{B})^\dagger$ et $\mathcal{B} \succ T\ \psi : y$ et $\mathcal{B} \succ T\ \psi' : z$;*
- *$\mathcal{B} \Vdash F\ \psi \to \psi' : x$ ssi il existe $y \in Ass(\mathcal{B})^\dagger$ tel que $xy \in Ass(\mathcal{B})^\dagger$ et $\mathcal{B} \succ T\ \psi : y$ et $\mathcal{B} \succ F\ \psi' : xy$;*
- *$\mathcal{B} \Vdash T\ \psi \to \psi' : x$ ssi pour tout $y \in Ass(\mathcal{B})^\dagger$, $xy \in Ass(\mathcal{B})^\dagger \Rightarrow (\mathcal{B} \succ F\ \psi : y$ ou $\mathcal{B} \succ T\ \psi' : xy)$.*

Une formule potentiellement complète est dite complète *si* $Ass(\mathcal{B})^\dagger$ *est complètement défini.*

On montre alors que les relations $\succ$ et $\Vdash$ sont monotones par rapport aux formules signées d'une branche :

Lemme 3.14. Soit une branche $\mathcal{B}$ d'un tableau $\mathcal{T}$, alors :

1. $\mathcal{B} \succ F\ \phi : x$ et $x \leq y \in Ass(\mathcal{B})^\dagger \Rightarrow \mathcal{B} \succ F\ \phi : y$;
2. $\mathcal{B} \succ T\ \phi : x$ et $y \leq x \in Ass(\mathcal{B})^\dagger \Rightarrow \mathcal{B} \succ T\ \phi : y$;
3. $\mathcal{B} \Vdash F\ \phi : x$ et $x \leq y \in Ass(\mathcal{B})^\dagger \Rightarrow \mathcal{B} \Vdash F\ \phi : y$;
4. $\mathcal{B} \Vdash T\ \phi : x$ et $y \leq x \in Ass(\mathcal{B})^\dagger \Rightarrow \mathcal{B} \Vdash T\ \phi : y$.

Preuve. Commençons par démontrer la propriété 1. Si $\mathcal{B} \succ F\ \phi : x$, alors il existe $F\ \phi : z$ tel que $z \leq x \in Ass(\mathcal{B})^\dagger$. Comme par hypothèse $x \leq y \in Ass(\mathcal{B})^\dagger$, nous avons $z \leq y \in Ass(\mathcal{B})^\dagger$ car la $(\cdot)^\dagger$-clôture est transitive. Donc, $\mathcal{B} \succ F\ \phi : y$.

La propriété 2 se démontre de façon similaire à la propriété 1. Les propriétés 3 et 4 s'établissent simultanément en raisonnant par cas sur $S\ \phi : x$. On a donc les cas suivants :

- Cas $F\ p : x$; $F\ \top : x$; $F\ I : x$; $F\ \bot : x$: Conséquence immédiate de la propriété 1.
- Cas $T\ p : x$; $T\ \top : x$; $T\ I : x$; $T\ \bot : x$: Conséquence immédiate de la propriété 2.
- Cas $F\ \psi \wedge \psi' : x$: $\mathcal{B} \Vdash F\ \psi \wedge \psi' : x$ implique $\mathcal{B} \Vdash F\ \psi : x$ ou $\mathcal{B} \Vdash F\ \psi' : x$ donc $\mathcal{B} \Vdash F\ \psi : y$ ou $\mathcal{B} \Vdash F\ \psi' : y$ par hypothèse.
- Cas $T\ \psi \wedge \psi' : x$: Similaire au cas $F\ \psi \wedge \psi' : x$.
- Cas $F\ \psi \vee \psi' : x$: Similaire au cas $F\ \psi \wedge \psi' : x$.
- Cas $T\ \psi \vee \psi' : x$: Similaire au cas $F\ \psi \wedge \psi' : x$.
- Cas $F\ \psi * \psi' : x$: Soit $y \leq zz' \in Ass(\mathcal{B})^\dagger$, comme $x \leq y \in Ass(\mathcal{B})^\dagger$ par hypothèse, la transitivité de la $(\cdot)^\dagger$-clôture implique $x \leq zz' \in Ass(\mathcal{B})^\dagger$. Ainsi $\mathcal{B} \Vdash F\ \psi * \psi' : x$ implique alors $\mathcal{B} \succ F\ \psi : z$ ou $\mathcal{B} \succ F\ \psi' : z'$.
- Cas $T\ \psi * \psi' : x$: $\mathcal{B} \succ T\ \psi * \psi' : x$ implique qu'il existe deux labels $z, z' \in Ass(\mathcal{B})^\dagger$ tels que $x \leq zz' \in Ass(\mathcal{B})^\dagger$, $\mathcal{B} \succ T\ \psi : z$ et $\mathcal{B} \succ T\ \psi' : z'$. Comme $y \leq x \in Ass(\mathcal{B})^\dagger$ par hypothèse, on obtient $y \leq zz' \in Ass(\mathcal{B})^\dagger$ par transitivité de la $(\cdot)^\dagger$-clôture. d'où $\mathcal{B} \succ T\ \psi * \psi' : y$.
- Cas $F\ \psi \rightarrow \psi' : x$: Similaire au cas $T\ \psi * \psi' : x$.
- Cas $T\ \psi \rightarrow \psi' : x$: Similaire au cas $F\ \psi * \psi' : x$.
- Cas $F\ \psi \mathrel{-\!\!*} \psi' : x$: Soit $yz \in Ass(\mathcal{B})^\dagger$ comme $y \leq x \in Ass(\mathcal{B})^\dagger$, la condition de compatibilité de la $(\cdot)^\dagger$-clôture implique $yz \leq xz \in Ass(\mathcal{B})^\dagger$. $\mathcal{B} \Vdash F\ \psi \mathrel{-\!\!*} \psi' : x$ implique alors $\mathcal{B} \succ F\ \psi : z$ ou $\mathcal{B} \succ T\ \psi' : xz$. Par la propriété 2, on a alors : $\mathcal{B} \succ F\ \psi : z$ ou $\mathcal{B} \succ T\ \psi' : yz$.
- Cas $T\ \psi \mathrel{-\!\!*} \psi' : x$: $\mathcal{B} \Vdash T\ \psi \mathrel{-\!\!*} \psi' : x$ implique qu'il existe un label $z \in Ass(\mathcal{B})^\dagger$ tel que $xz \in Ass(\mathcal{B})^\dagger$, $\mathcal{B} \Vdash T\ \psi : z$ et $\mathcal{B} \Vdash F\ \psi' : xz$. Comme $x \leq y \in Ass(\mathcal{B})^\dagger$, grâce à la compatibilité de $(\cdot)^\dagger$-clôture, on a $x \leq y \in Ass(\mathcal{B})^\dagger$. Par la propriété 1, on a alors $\mathcal{B} \succ F\ \psi' : yz$.

$\square$

Définition 3.25 (Tableau complet). *Une branche est dite* (potentiellement) complète *si elle est potentiellement ouverte et si toutes ses formules signées sont (potentiellement) complètes. Un tableau est* (potentiellement) complet *s'il contient au moins une branche (potentiellement) complète.*

Au début de cette section, nous définissions informellement une branche potentiellement complète comme une branche dont toutes les formules sont analysées. Nous démontrons maintenant que cette correspondance est bien assurée par les définitions que l'on a données ci-dessus.

Lemme 3.15. Si $\mathcal{B}$ est une branche potentiellement complète d'un tableau $\mathcal{T}$, alors :

1. $\mathcal{B} \succ S\ \phi : x \Rightarrow \mathcal{B} \Vdash S\ \phi : x$;
2. $\mathcal{B} \Vdash T\ \phi : x \Rightarrow \mathcal{B} \nVdash F\ \phi : x$;
3. $\mathcal{B} \Vdash F\ \phi : x \Rightarrow \mathcal{B} \nVdash T\ \phi : x$.

Preuve. Pour la propriété 1. on considère le cas $S = F$, l'autre cas étant similaire. Si $\mathcal{B} \succ F\ A{:}x$, alors, par définition de $\succ$, il existe $F\ A : y \in \mathcal{B}$ tel que $y \leq x \in Ass(\mathcal{B})^{\dagger}$. Comme $\mathcal{B}$ est supposée complète, $F\ A : y$ implique $\mathcal{B} \Vdash F\ A : y$. D'après le lemme 3.14 $y \leq x \in Ass(\mathcal{B})^{\dagger}$ entraîne alors $F\ A : y$.

Pour les propriétés 2. et 3. on montre que l'on ne peut avoir à la fois $\mathcal{B} \Vdash T\ p{:}x$ et $\mathcal{B} \Vdash F\ p{:}x$. Supposons que l'on puisse l'avoir. Alors, il existe un label y tel que $T\ p : y$ et $x \leq y \in Ass(\mathcal{B})^{\dagger}$ et il existe un label z tel que $F\ p : z$ et $z \leq x \in Ass(\mathcal{B})^{\dagger}$. Par transitivité de $(\cdot)^{\dagger}$, on en déduit $z \leq y \in Ass(\mathcal{B})^{\dagger}$. Donc la branche $\mathcal{B}$ est close et on est en contradiction avec l'hypothèse d'induction. □

Une branche *potentiellement complète* n'est pas *complète* si et seulement si elle contient des labels dont on ne peut encore déterminer la forme et si par conséquent elle ne contient pas assez d'informations pour construire un contre-modèle. On doit alors déterminer une *valuation*[4] cohérente de notre ensemble de contraintes et de labels qui permet de compléter la branche.

Définition 3.26 (Extension d'une branche). *Soit une branche $\mathcal{B}$ potentiellement complète et une valuation $Ass(\mathcal{B})^{\dagger}_{v}$ de $Ass(\mathcal{B})^{\dagger}$. L'*extension de *$\mathcal{B}$* à *$Ass(\mathcal{B})^{\dagger}_{v}$ est obtenue en rajoutant à $Ass(\mathcal{B})$ les contraintes introduites dans $Ass(\mathcal{B})^{\dagger}_{v}$ et en complétant la branche à l'aide de ces nouvelles contraintes.*

Définition 3.27 (Branche close structurellement). *Soit une branche $\mathcal{B}$ potentiellement complète. $\mathcal{B}$ est* close structurellement *s'il n'existe pas de valuation cohérente $Ass(\mathcal{B})^{\dagger}_{v}$ de $Ass(\mathcal{B})^{\dagger}$ tel que l'extension de $\mathcal{B}$ à cette valuation soit complète.*

Une branche non close structurellement est dite ouverte.

Nous présentons maintenant un algorithme *équitable* de complétion des tableaux. Par équitable, on entend que la stratégie choisie ne privilégie pas une règle par rapport aux autres. Notamment, cela implique que l'on développe toutes les règles obligationnelles (qui peuvent être instanciées plusieurs fois) avec la même priorité. L'algorithme construit progressivement un tableau soit clos, soit complet. Pour définir quelles formules peuvent être développées par notre algorithme, on commence par définir la notion de formule active.

Définition 3.28 (Formule active). *Soit $\mathcal{B}$ une branche d'un tableau $\mathcal{T}$. Une formule signée est* active *dans $\mathcal{B}$ si elle appartient à $\mathcal{B}$, elle n'est pas complète dans $\mathcal{B}$ et que $\mathcal{B}$ est ouverte.*

Définition 3.29 (Procédure de complétion). *La* procédure de complétion *d'un tableau admissible $\mathcal{T}$ est la suivante :*

donnée : *un tableau $\mathcal{T}$ (admissible)*
tantque *$\mathcal{T}$ n'est ni clos, ni potentiellement complet* **faire**

[4] Voir définition 3.10.

si *toutes les formules actives de $\mathcal{T}$ sont marquées comme* visitées **alors**
marquer toutes les formules actives de $\mathcal{T}$ comme non-visitées
fsi
choisir dans une branche $\mathcal{B}$ une formule active non-visitée $S\ \phi : x$
si $S\ \phi : x$ *est de la forme* $F\ \phi * \psi : x$ **alors**
choisir deux labels $y, z \in Ass(\mathcal{B})^\dagger$
tels que $x \leq yz \in Ass(\mathcal{B})^\dagger$, $\mathcal{B} \not\vdash F\ \phi : y$ *et* $\mathcal{B} \not\vdash F\ \psi : z$
décomposer $F\ \phi * \psi : x$ *avec la règle* $F*$ *en utilisant les labels* y *et* z
sinon si $S\ \phi : x$ *est de la forme* $T\ \phi \mathrel{-\!\!*} \psi : x$ **alors**
choisir un label $y \in Ass(\mathcal{B})^\dagger$ *tel que* $xy \in Ass(\mathcal{B})^\dagger$ *et* $\mathcal{B} \not\vdash F\ \phi : y$ *et* $\mathcal{B} \not\vdash T\ \psi : xy$
décomposer $F\ \phi \mathrel{-\!\!*} \psi : x$ *avec la règle* $T \mathrel{-\!\!*}$ *en utilisant les labels* y *et* xy
sinon si $S\ \phi : x$ *est de la forme* $T\ \phi \rightarrow \psi : x$ **alors**
choisir un label $y \in Ass(\mathcal{B})^\dagger$ *tel que* $y \leq x \in Ass(\mathcal{B})^\dagger$ *et* $\mathcal{B} \not\vdash F\ \phi : y$ *et* $\mathcal{B} \not\vdash T\ \psi : xy$
décomposer $F\ \phi \mathrel{-\!\!*} \psi : x$ *avec la règle* $T \rightarrow$ *en utilisant le label* y
sinon si $S\ \phi : x$ *est de la forme* $T\ [l]\phi : x$ **alors**
choisir un label $y \in Ass(\mathcal{B})^\dagger$ *tel que* $x \leq [l]y \in Ass(\mathcal{B})^\dagger$ *et* $\mathcal{B} \not\vdash F\ \phi : y$
décomposer $F\ \phi \mathrel{-\!\!*} \psi : x$ *avec la règle* $T \rightarrow$ *en utilisant le label* y
sinon
décomposer $S\ \phi : x$ *avec la règle adaptée*
fsi
marquer la formule $S\ \phi : x$ *comme* visitée
ftant

Cette procédure de complétion est alors au cœur de l'algorithme de TBI-Loc.

Définition 3.30 (Algorithme de TBI-Loc). *L'algorithme général de recherche de preuves est le suivant :*

si *le tableau n'est pas clos* **alors**
tant que *aucune branche n'est ouverte*
et que toutes les valuations cohérentes n'ont pas été épuisée **faire**
- choisir une branche potentiellement complète $\mathcal{B}$;
- choisir une valuation cohérente de $Ass(\mathcal{B})^\dagger$;
- ajouter les contraintes introduites par la valuation à $Ass(\mathcal{B})$;
procédure de complétion
si *la branche n'est pas ouverte* **alors**
- supprimer la valuation ;
- remettre $\mathcal{B}$ dans l'état qu'elle avait avant l'ajout des nouvelles contraintes ;
fsi
ftant

Un tableau est donc clos si l'ensemble de ses branches est clos. Un tableau clos indique qu'aucune des alternatives considérables pour une formule donnée ne mène à un contre-modèle, donc un tableau clos est une preuve de la validité d'une formule.

La procédure proposée ici semble ne pas pouvoir terminer et ce pour deux raisons. Tout d'abord, comme dans TBI, l'algorithme de complétion peut générer une branche infinie. L'autre raison est spécifique à TBI-Loc et découle du fait que l'ensemble des valuations cohérentes à tester est infini si on ne lui impose aucune restriction. On aborde ces problèmes et y apporte quelques éléments de réponse dans la section 3.8.

Définition 3.31 (Preuve pour TBI-Loc). *Soit ϕ une formule de BI-Loc, un tableau $\mathcal{T}$ est une preuve de ϕ dans* TBI-Loc *(TBI-Loc-preuve de ϕ) s'il existe une séquence finie de tableaux $(\mathcal{T}_i)_{1\leq i\leq n}$ telle que :*

- *$\mathcal{T}_1$ a pour seul nœud la racine étiquetée par la formule signée $F\ \phi : c$;*
- *$\mathcal{T}_{i+1}$ est obtenu à partir de $\mathcal{T}_i$ par application d'une règle décrite à la figure 3.1 ;*
- *$\mathcal{T}_n = \mathcal{T}$ est un tableau clos et admissible.*

Une formule ϕ est prouvable dans TBI-Loc *(TBI-Loc-prouvable) s'il existe une* TBI-Loc*-preuve de ϕ.*

3.5 Exemples de tableaux

Afin d'illustrer les différents mécanismes de clôture des branches et de la création de graphe de ressources, on propose maintenant quelques exemples de tableaux avant de passer aux preuves de correction et de complétude de ce système de preuves. TBI-Loc repose sur les bases de la méthode TBI des tableaux pour BI [72]. Par souci de concision, on s'intéresse ici principalement à des exemples mettant en avant les spécificités de TBI-Loc et du modèle des arbres partiel. Nous développons ici trois exemples, mettant en avant les principes d'incohérence propres à la structure d'arbre, la clôture par similarité de l'ensemble de contraintes et le problème de la valuation. Le but de ces exemples est à la fois d'illustrer le fonctionnement de l'algorithme et de justifier par l'exemple certaines définitions données dans la section précédente.

Pour indiquer comment le tableau a été construit, on indique les formules signées du tableau qui ont été traitées à l'aide d'une marque $\surd$. Les numéros situés sous la marque indique l'ordre dans lequel le traitement des formules a eu lieu. La présence de plusieurs numéros indique que la même formule a été instanciée plusieurs fois avec des décompositions différentes. Chaque branche est nommée à son extrémité. De plus on indique les branches closes par une croix terminale $\times$.

3.5.1 Incohérence spécifique aux arbres

Le premier exemple que l'on développe est celui de la preuve de la formule $(p \wedge [l]p) \rightarrow \bot$, dont le tableau est donné en figure 3.3. Cet exemple met en avant le cas d'incohérence spécifique aux arbres.

Le tableau initial ne contient que la formule $F\ (p \wedge [l]p) \rightarrow \bot : c$. La première décomposition ($\surd_1$) concerne donc cette formule et introduit une nouvelle constante c_1 qui doit vérifier la partie gauche de l'implication et falsifier la partie droite, comme l'indique les formules $T\ p \wedge [l]p : c_1$ et $F\ \bot : c_1$. La deuxième étape décompose l'opérateur $\wedge$ en introduisant ni nouvelle constante, ni nouvelle contrainte, mais rajoutant au tableau les formules $T\ p : c_1$ et $T\ [l]p : c_1$. La troisième étape consiste à marquer le label c_1 qui doit correspondre à une ressource puisqu'il doit pouvoir vérifier la proposition p. L'étape 4 quant à elle indique que le label c_1 doit être égal à un label localisé en l. Les deux assertions introduites par les étapes 3 et 4 sont contradictoires puisque c_1 doit à la fois correspondre à un label localisé et à une ressource. La branche est donc incohérente. Elle est donc close et la formule est donc prouvable.

3.5.2 La similarité des labels

L'exemple développé en figure 3.4 montre comment l'argument de similarité de la condition de clôture peut être utilisé. Pour cela, on détaille donc la preuve de la formule suivante :

$$(q * [l]p) \wedge (q' * [l]p')) \rightarrow ((q \wedge q') * [l](p \wedge p')).$$

$$\surd_1 \; F\ (p \wedge [l]p) \to \bot : c$$
$$\boxed{ass1 : c_1 \leq c}$$
$$\surd_2 \; T\ p \wedge [l]p : c_1$$
$$F\ \bot : c_1$$
$$\surd_3 \; T\ p : c_1$$
$$\surd_4 \; T\ [l]p : c_1$$
$$\boxed{ass2 : c_1^\star}$$
$$\boxed{ass3 : c_1 \cong [l]c_2}$$
$$T\ p : c_2$$
$$\mathcal{B}_1$$
$$\times$$

FIG. 3.3 – Tableau de $(p \wedge [l]p) \to \bot$

La partie intéressante de ce tableau commence à l'étape 7 de la construction. Cette étape construit en effet l'assertion 6 qui marque le nœud c_2. Par transitivité, on a donc l'assertion suivante : $c_1 \leq c_2^\star[l]c_6$. Cette assertion vérifie la condition initiale du cas de clôture par similarité. Il existe donc des constantes d_1, d_2 tels que $c_1 \cong d_1[l]d_2$. De plus, toujours par similarité, on obtient les assertions suivantes : $d_1 \leq c_2$, $d_1 \leq c_4$, $d_2 \leq c_6$ et $d_2 \leq c_7$. Ces quatre assertions sont respectivement à la base de l'impossibilité de réaliser les branches 1,2,3 et 4. Cette formule est donc prouvable dans BI-Loc alors qu'elle semblerait non prouvable dans BI. En effet, la décomposition vérifiant $q * [l]p$ et celle vérifiant $q' * [l]p'$ est la même : q et q' sont toutes deux vérifiées par l'ensemble des ressources présentes à la racine et p et p' sont toutes deux vérifiées par l'ensemble des ressources en l.

3.5.3 Tableau ouvert et valuation

Le dernier exemple est présenté sur deux figures. La figure 3.5 présente le tableau ouvert initialement et qui nécessite de trouver une valuation de certains labels pour pouvoir s'assurer que le tableau est véritablement ouvert. La figure 3.6, montre le complément du tableau après valuation. Chaque label concerné par la valuation est remplacé par sa valuation et le tableau est alors complété en conséquence.

La formule étudiée dans la figure 3.5 est très proche de celle de la figure 3.4. Par conséquent, les décompositions sous forme de tableaux sont presque identiques. On remarque toutefois que l'on a ici dans la décomposition $T\ \top : c_2$ et $T\ \top : c_4$ au lieu de $T\ q : c_2$ et $T\ q' : c_4$. Du fait de cette différence, on ne peut appliquer les règles relatives aux propositions appliquées aux étapes 7 et 8 de l'exemple précédent. Et l'on ne connaît donc pas encore la forme de c_1. Ceci est dû au fait qu'un label forçant $\top$ n'est pas nécessairement une ressource. Par conséquent, on ne peut décomposer la formule $F\ \top * [l](p \wedge q) : c_1$ qu'avec les labels $c_2[l]c_6$ et $c_3[l]c_7$ et aucune de ces décompositions ne permet de clore complètement le tableau. Une fois arrivé à la décomposition de la figure 3.5, le tableau est potentiellement ouvert puisqu'aucune règle de décomposition n'est applicable et que les labels c, c_1, c_2 et c_4 sont indéfinis.

On doit alors choisir une valuation pour vérifier qu'il en existe *au moins une* qui laisse

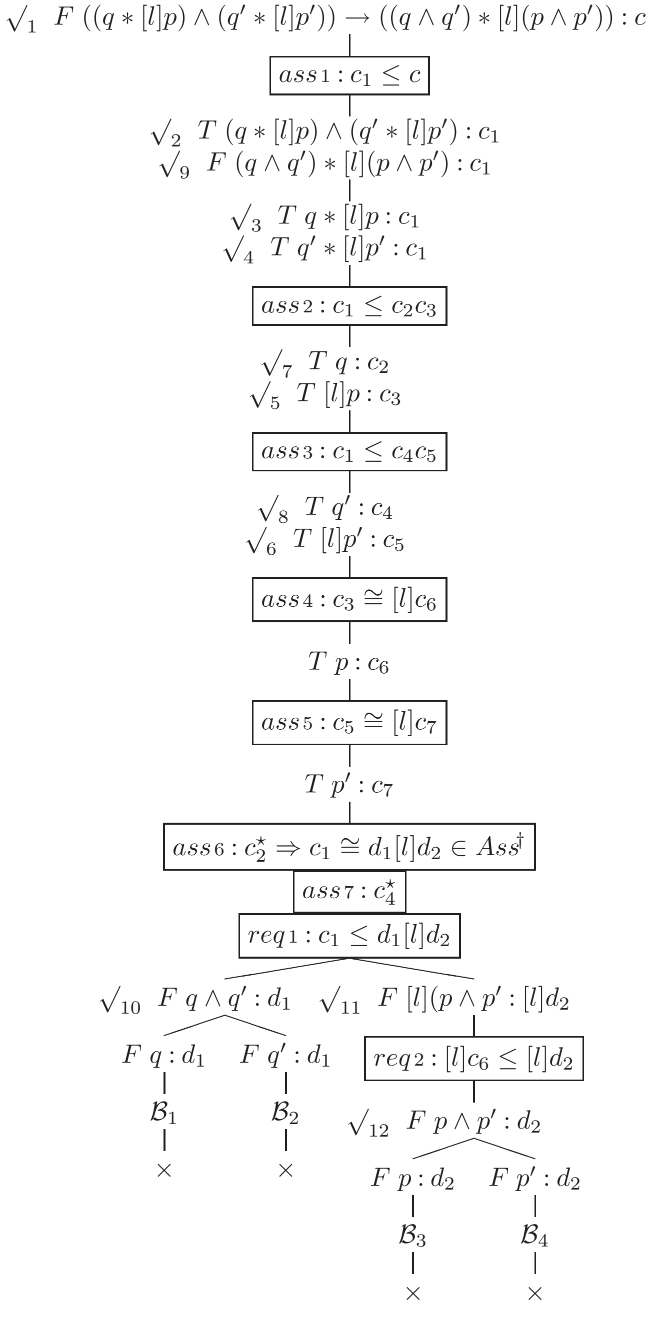


FIG. 3.4 – Tableau de ressources $((q * [l]p) \wedge (q' * [l]p')) \rightarrow ((q \wedge q') * [l](p \wedge p'))$

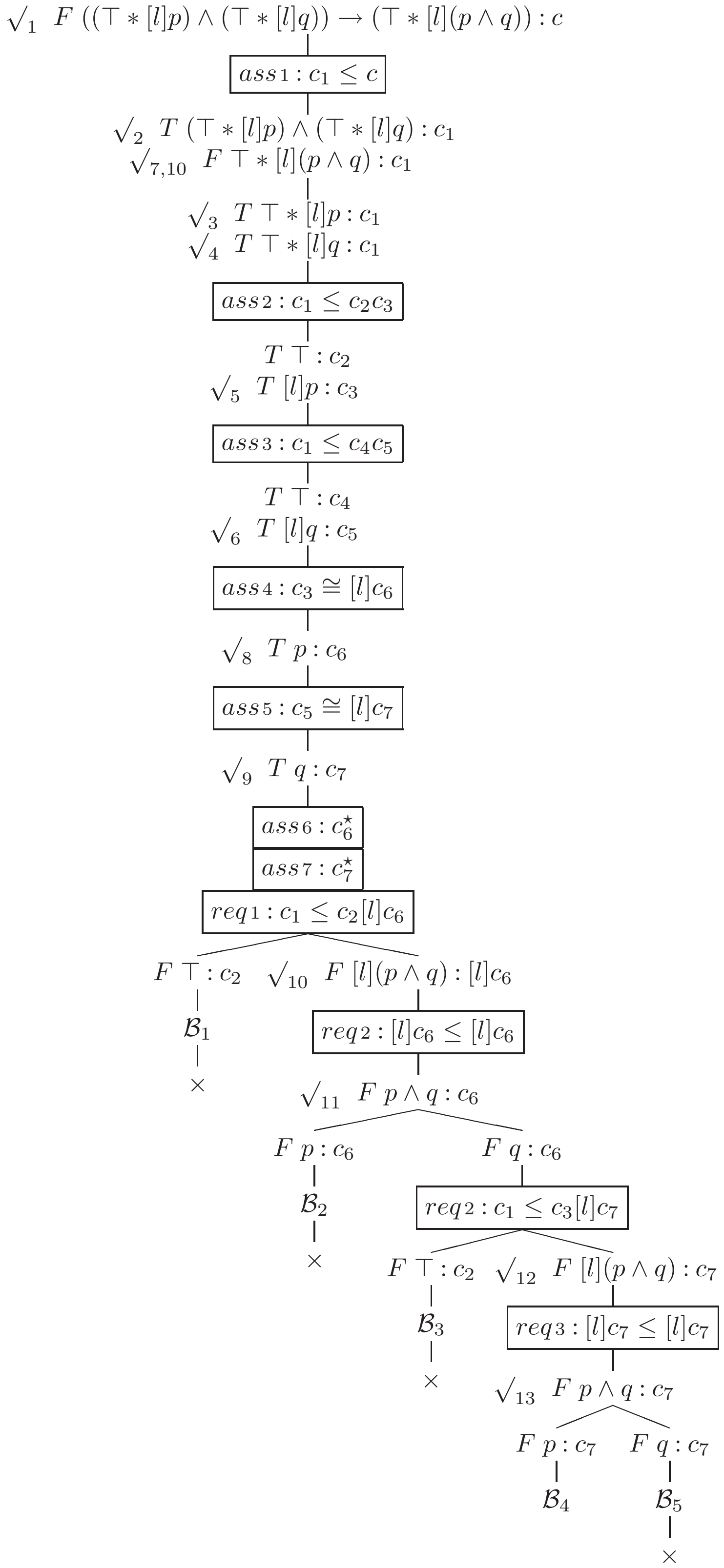


FIG. 3.5 – Tableau de $((\top * [l]p) \wedge (\top * [l]q)) \to (\top * [l](p \wedge q))$

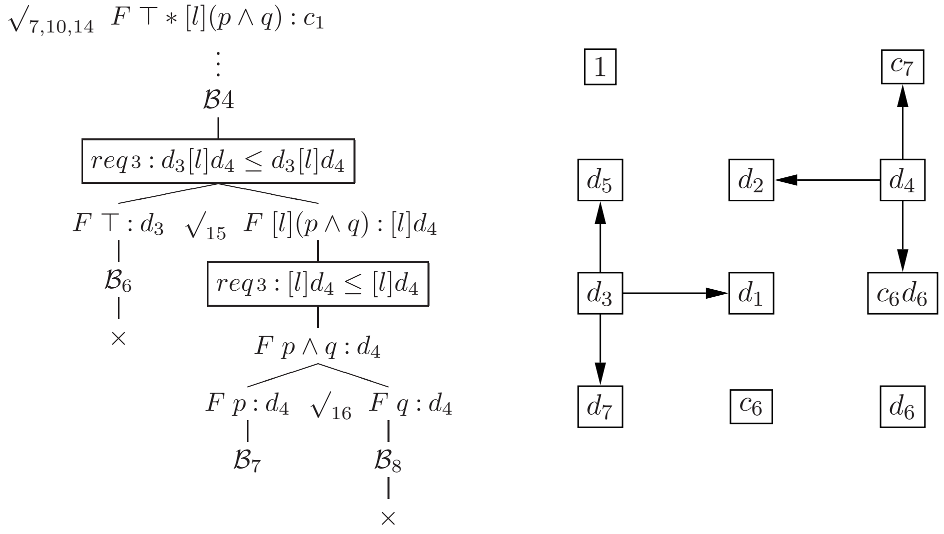


FIG. 3.6 – Tableau valué et graphe de ressources pour $((\top * [l]p) \wedge (\top * [l]q)) \to (\top * [l](p \wedge q))$

une branche ouverte. Par souci de concision, nous choisirons directement une valuation de ce type. Toutefois, pour automatiser l'algorithme, on devrait tester systématiquement toutes les valuations modulo certaines contraintes de taille, comme nous le verrons en section 3.8. On propose donc la valuation suivante :

$$c = d_1[l]d_2; c_1 = d_3[l]d_4; c_2 = d_4[l]d_5; c_4 = d_6$$

Avec cette valuation, les contraintes suivantes (entre autres) appartiennent à l'ensemble des assertions par clôture : $d_3 \leq d_5$, $d_3 \leq d_7$, $d_4 \leq d_6c_6$, $d_4 \leq c_7$ et donc, $d_3[l]d_4 \leq d_5[l]d_6c_6$ et $d_3[l]d_4 \leq d_7[l]c_7$. Partant de ce constat, la seule nouvelle décomposition pertinente apportée par cette valuation est celle marquée par l'étape 14 de la figure 3.6. Cependant, là aussi, cette valuation ne permet pas de fermer le tableau. On a alors une branche ouverte permettant de construire un contre modèle, comme nous le verrons dans la sous-section 3.7.2.

3.6 Correction de la méthode des tableaux

Dans cette section, nous démontrons la correction TBI-Loc par rapport à BI-Loc. Cela signifie que chaque formule de BI-Loc qui est prouvable dans TBI-Loc est valide dans le modèle des arbres partiels de BI-Loc. La preuve reprend les grands principes et la démarche mise en œuvre pour BI [72]. Il s'agit de montrer qu'une branche $\mathcal{B}$ close ne peut être *réalisée*. C'est à dire qu'on ne peut trouver de modèle d'arbres partiel vérifiant les conditions imposées par les labels de la branche $\mathcal{B}$.

Définition 3.32 (Réalisation). *Soit une branche $\mathcal{B}$ d'un tableau $\mathcal{T}$ et $Tree = (T, (e, nil), |, \sqsubseteq_T, \models_{\mathfrak{P}})_{\mathcal{M},Loc}$ un modèle d'arbres partiel reposant sur un ensemble d'emplacement Loc et un monoïde de ressources partiel $\mathcal{M} = (M, e, \times, \sqsubseteq)$. On dit que $\mathcal{B}$ est* réalisable *dans $Tree$ s'il existe une fonction $[\![\cdot]\!] : \mathcal{L}Ass(\mathcal{B})^\dagger \to T$ de l'ensemble des labels utilisés dans $\mathcal{B}$ dans l'ensemble des arbres de ressources telle que :*

1. $[\![1]\!] = e$;
2. $[\![[l]x]\!] = [l][\![x]\!]$;

3. $[\![x^\star]\!] = [\![x]\!]$ *et* $[\![x^\star]\!] \in M$;
4. $[\![x \times y]\!] = [\![x]\!] | [\![y]\!]$;
5. *pour toute formule signée* $T\ \phi : x$ *de* $\mathcal{B}$, $[\![x]\!] \models_{\mathfrak{P}} \phi$;
6. *pour toute formule signée* $F\ \phi : x$ *de* $\mathcal{B}$, $[\![x]\!] \not\models_{\mathfrak{P}} \phi$;
7. *pour toute assertion* $x \leq y$ *de* $Ass(\mathcal{B})$, *on a* $[\![x]\!] \sqsubseteq [\![y]\!]$.

Une telle fonction est une réalisation *de* $\mathcal{B}$ *dans* T.

Une réalisation fixe donc la valeur des labels. On peut donc établir un lien entre une réalisation et la valuation d'un ensemble d'assertions.

Lemme 3.16. Soit $[\![\cdot]\!]$ une réalisation d'une branche $\mathcal{B}$. Alors il existe une valuation cohérente $Ass(\mathcal{B})^\dagger_v$ de $Ass(\mathcal{B})^\dagger$.

Preuve. Directe par définition de $[\![\cdot]\!]$ puisqu'une réalisation fixe l'arbre correspondant à chaque label. □

Définition 3.33 (Tableau réalisable). *Une branche est* réalisable *s'il existe au moins un modèle d'arbres partiel dans lequel elle est réalisable. Un tableau est* réalisable *s'il contient au moins une branche réalisable.*

On montre maintenant que même si la définition 3.32 requiert uniquement la réalisation des assertions de $Ass(\mathcal{B})$, elle satisfait également celles obtenues par $(\cdot)^\dagger$-clôture de cet ensemble.

Lemme 3.17. Soit $\mathcal{T}$ un tableau, $\mathcal{B}$ une branche de $\mathcal{T}$, $(T, (e, nil), |, \sqsubseteq_T, \models_{\mathfrak{P}})_{\mathcal{M},Loc}$ un modèle d'arbres partiel et une réalisation $[\![\cdot]\!]$. Pour toute assertion $x \leq y \in Ass(\mathcal{B})^\dagger$, on a $[\![x]\!] \sqsubseteq [\![y]\!]$.

Preuve. La preuve est directe. Les assertions étant obtenues par la clôture réflexive, transitive et compatible, comme $[\![\cdot]\!]$ est un homomorphisme et comme par définition la condition est vérifiée sur $Ass(\mathcal{B})$, si on a $x \leq y \in Ass(\mathcal{B})^\dagger$, on a bien $[\![x]\!] \sqsubseteq [\![y]\!]$. □

Maintenant que nous disposons de la notion de réalisation, nous montrons qu'elle vérifie des propriétés cruciales pour établir la preuve de correction. Tout d'abord, elle est préservée par les règles de décomposition de la figure 3.1. Ensuite, on établit qu'une branche ne peut être à la fois réalisable et close.

Lemme 3.18. Les règles de décomposition de TBI-Loc préservent la réalisabilité.

Preuve. Soit $\mathcal{T}$ un tableau réalisable. Il existe donc dans $\mathcal{T}$ une branche $\mathcal{B}$ réalisable dans un modèle d'arbre partiel $(T, (e, nil), |, \sqsubseteq_T, \models_{\mathfrak{P}})_{\mathcal{M},Loc}$ pour une réalisation $[\![\cdot]\!]$. Si la règle que nous appliquons ne concerne pas la branche ouverte, cette branche reste ouverte et on peut conclure. Le cas qui nous intéresse est donc celui où l'on applique une règle de composition sur une formule appartenant à $\mathcal{B}$. On procède alors par cas sur le connecteur principal et la polarité S de la formule signée $S\ \phi : x$ décomposée :

- Cas $T\ \phi * \psi : x$: La branche $\mathcal{B}$ est étendue par l'ajout des deux formules signées $T\ \phi : c_i$ et $T\ \phi : c_j$ et de l'assertion $x \leq c_i c_j$ où c_i et c_j sont de nouvelles constantes. Par hypothèse, $[\![\cdot]\!]$ est une réalisation de $\mathcal{B}$, on a donc $[\![x]\!] \models_{\mathfrak{P}} \phi * \psi$. Donc par définition de $*$, il existe deux arbres t et t' tels que $[\![x]\!] \sqsubseteq t|t', t \models_{\mathfrak{P}} \phi$ et $t' \models_{\mathfrak{P}} \psi$. Il suffit alors d'étendre $[\![\cdot]\!]$ à c_i et c_j en posant $[\![c_i]\!] = t$ et $[\![c_j]\!] = t'$. On obtient $[\![c_i]\!] \models_{\mathfrak{P}} \phi$, $[\![c_j]\!] \models_{\mathfrak{P}} \psi$ et $[\![x]\!] \sqsubseteq [\![c_i c_j]\!]$. Donc l'extension de $\mathcal{B}$ est réalisable.

- Cas $F\ \phi * \psi : x$: La branche $\mathcal{B}$ s'étend en deux nouvelles branches $\mathcal{B}'$ et $\mathcal{B}''$. Ces deux branches partagent l'obligation $x \leq yz$, $\mathcal{B}'$ contient $F\ \phi : y$ et $\mathcal{B}''$ contient $F\ \psi : z$. Pour pouvoir appliquer la règle, l'obligation $x \leq yz$ doit appartenir à $Ass(\mathcal{B})^\dagger$ donc d'après le lemme 3.17, on doit avoir $[\![x]\!] \leq [\![yz]\!]$. Comme $[\![\cdot]\!]$ est une réalisation de $\mathcal{B}$, on a $[\![x]\!] \not\models_{\mathfrak{P}} \phi * \psi$ et donc par définition de $*$, pour tous les arbres t, t' tels que $[\![x]\!] \sqsubseteq t|t'$, soit $t \not\models_{\mathfrak{P}} \phi$, soit $t' \not\models_{\mathfrak{P}} \psi$. En particulier, on obtient soit $[\![y]\!] \not\models_{\mathfrak{P}} \phi$ ou $[\![z]\!] \not\models_{\mathfrak{P}} \psi$. Par conséquent, soit $\mathcal{B}'$ soit $\mathcal{B}''$ est réalisable.
- Cas $T\ [l]\phi : x$: La branche $\mathcal{B}$ s'étend avec la formule signé $T\ \phi : c_i$ et l'assertion $x \cong [l]c_i$. Par hypothèse, $[\![x]\!]$ est une réalisation de $\mathcal{B}$, donc on a $[\![x]\!] \models_{\mathfrak{P}} [l]\phi$. Donc par définition de $[\cdot]$, il existe t tel que $[\![x]\!]_=(e, l \mapsto t)$. Il suffit alors d'étendre $[\![\cdot]\!]$ en posant $[\![c_i]\!]_=t$. On obtient $[\![c_i]\!] \models_{\mathfrak{P}} \phi$ et $[\![x]\!] \sqsubseteq [\![[l]c_i]\!]$. Par conséquent, $\mathcal{B}$ est réalisable.

Les autres cas sont similaires à ceux détaillés ci-dessus.

□

Lemme 3.19. Un tableau clos n'est pas réalisable.

Preuve. Supposons que $\mathcal{T}$ soit réalisable. Donc, il existe une branche $\mathcal{B}$ de $\mathcal{T}$ qui est réalisable dans un modèle d'arbre partiel $(T, (e, nil), |, \sqsubseteq_T, \models_{\mathfrak{P}})_{\mathcal{M}, Loc}$ pour une réalisation $[\![\cdot]\!]$. Raisonnons par cas sur la condition de clôture de la branche $\mathcal{B}$:

- La branche $\mathcal{B}$ est close logiquement :
  1. La branche est close par la condition (CL1) de la définition 3.11. Donc il existe deux formules signées de $\mathcal{B}$, $T\ \phi : x$ et $F\ \phi : x$ telles que $y \leq x \in Ass(\mathcal{B})^\dagger$. Comme $[\![\cdot]\!]$ est une réalisation de $\mathcal{B}$, on a donc $x \models_{\mathfrak{P}} \phi$ et $y \not\models_{\mathfrak{P}} \phi$. Or, on déduit du lemme 3.17 que $[\![x]\!] \sqsubseteq [\![y]\!]$. De part la définition de $\sqsubseteq$, on a donc : $[\![y]\!] \models_{\mathfrak{P}} \phi$ et on obtient une contradiction.
  2. La branche est close par la condition (CL2) de la définition 3.11. Elle contient donc une formule signée $F\ I : x$ avec $x \leq 1 \in Ass(\mathcal{B})^\dagger$. Comme $[\![\cdot]\!]$ est une réalisation, on a donc $[\![x]\!] \models_{\mathfrak{P}} I$. De plus, on déduit du lemme 3.17 que $[\![x]\!] \sqsubseteq e$. On aboutit alors à une contradiction puisque cette dernière relation implique que $x \models_{\mathfrak{P}} I$.
  3. La branche est close par la condition (CL3) de la définition 3.11. Elle contient donc une formule signée $F \top : x$. Comme $[\![\cdot]\!]$ est une réalisation, on a donc $[\![x]\!] \not\models_{\mathfrak{P}} \top$, ce qui est une contradiction car par définition $\models_{\mathfrak{P}}$, $[\![x]\!] \models_{\mathfrak{P}} \top$.
  4. La branche est close par la condition (CL4) de la définition 3.11. Elle contient donc une formule signée $F\ \phi : x$ avec x incohérent dans $\mathcal{B}$. Comme x est incohérent, par définition, il existe y tel que $x \leq y \in Ass(\mathcal{B})^\dagger$ et un sous-label z de y tel que $T \perp : z \in \mathcal{B}$. Il existe donc $u \in Ass(\mathcal{B})^\dagger$ tel que $y = zu$ et donc on a $x \leq zu \in Ass(\mathcal{B})^\dagger$. Comme $[\![\cdot]\!]$ est une réalisation, on a donc également $[\![x]\!] \leq [\![zu]\!]$. Or, $[\![z]\!] \models_{\mathfrak{P}} \perp$ ssi z n'est pas défini. Donc $[\![x]\!]$ non plus n'est pas défini. Et donc, on ne peut avoir $[\![x]\!] \models_{\mathfrak{P}} \phi$ puisqu'un arbre non défini satisfait toutes les formules.
- La branche est close structurellement. Donc toute valuation de l'ensemble $Ass(\mathcal{B})^\dagger$ est incohérente. D'après le lemme 3.16 on ne peut donc pas obtenir de réalisation de $\mathcal{B}$.

□

Comme expliqué ci-dessus, ces deux lemmes nous permettent d'établir le résultat suivant de correction suivant :

Théorème 3.9. *Si une formule de* BI-Loc *est prouvable dans* TBI-Loc, *alors elle est valide dans la sémantique des arbres partiels.*

Preuve. Soit ϕ une formule prouvable dans TBI-Loc. Alors, il existe aucune séquence finie de tableaux, telle que le tableau initial $\mathcal{T}_1$ contient uniquement $F\ \phi : c$ satisfaisant les conditions de clôture de la définition 3.22 ou tel que l'ensemble des assertions est incohérent. Supposons que ϕ ne soit pas valide. Alors il existe un modèle d'arbres partiel $(T, (e, nil), |, \sqsubseteq, \models_{\mathfrak{P}})$ tel qu'il existe $t \in T$ vérifiant $t \not\models_{\mathfrak{P}} \phi$. Par conséquent, le tableau $\mathcal{T}_1$ est trivialement réalisable en posant $[\![c]\!] = t$. Mais alors, le lemme 3.18 implique que tous les tableaux de la séquence sont réalisables. Donc par le lemme 3.19, les tableaux ne sont pas clos et on aboutit à une contradiction. □

3.7 Complétude de la méthode des tableaux

Selon le principe habituellement mis en place pour la méthode des tableaux avec labels [72], la preuve de complétude est fondée une nouvelle fois sur la construction d'un contre-modèle dans la sémantique des arbres partiels. Pour cela, il faut donc réussir à fixer le monoïde partiel qui permet de construire un ensemble d'arbres dans lequel la formule n'est pas valide.

3.7.1 Construction de contre-modèles

Pour déterminer un contre-modèle dans la sémantique des arbres partiels, on doit déterminer un monoïde d'arbre partiel qui ne vérifie pas la formule. Celui que l'on utilise pour notre contre-modèle est extrait d'un graphe de ressources associé à une branche complète du tableau étudié.

Définition 3.34 ($\mathcal{H}$-modèle). *Soit $\mathcal{B}$ une branche ouverte d'un tableau $\mathcal{T}$ et $\mathcal{G}(Ass(\mathcal{B})^{\dagger}) = [N(Ass(\mathcal{B})^{\dagger}), E(Ass(\mathcal{B})^{\dagger})]$ le graphe de ressources associé. Le $\mathcal{H}$-modèle d'arbres partiel de $\mathcal{B}$ est la structure $(T, (e, nil), |, \sqsubseteq_T, \models_{\mathfrak{P}})_{Loc}$, telle que :*

- $T = \{t \ / \ t|\forall L.t(L) = (m, t') \wedge m \in N(Ass(\mathcal{B})^{\dagger})\}$ *;*
- $\bullet$ *est la loi de composition vérifiant :*
  - $t|1 = 1|t = t$ *;*
  - $[t_1|t_2]\uparrow$ *si* $\forall L.\exists t'_1, t'_2, m_1, m_2.t_1(L) = (m_1, t'_1),\ t_1(L) = (m_2, t'_2)$ *et* $m_1 \times m_2 \in N(Ass(\mathcal{B})^{\dagger})$ *;*
  - $t_1|t_2$ *n'est pas défini sinon ;*
- $\sqsubseteq$ *est la relation définie par :*
  - $\forall t_1, t_2 \in M.x \sqsubseteq y$ *ssi* $\forall L.t_1(L) = (m_1, t'_1),\ t_1(L) = (m_2, t'_2)$ *et* $m_1 \to m_2 \in E(Ass(\mathcal{B})^{\dagger})$.

La relation $\models_{\mathfrak{P}}$ quant à elle est engendrée à partir des relations suivantes :

$$x \models_{\mathfrak{P}} p \ ssi\ \mathcal{B} \Vdash T\ p : x$$

On montre maintenant qu'un $\mathcal{H}$-modèle est un modèle d'arbres partiel doté de propriétés remarquables établissant le lien entre l'analyse de la branche et la relation de forcing.

Lemme 3.20. Le $\mathcal{H}$-modèle d'une branche complète $\mathcal{B}$ est un modèle d'arbre partiel qui vérifie pour toute formule ϕ de BI-Loc :

1. $\mathcal{B} \succ T\ \phi : x$ et x cohérent dans $\mathcal{B} \Rightarrow x \models_{\mathfrak{P}} \phi$;
2. $\mathcal{B} \succ F\ \phi : x$ et x cohérent dans $\mathcal{B} \Rightarrow x \not\models_{\mathfrak{P}} \phi$.

Preuve. Tout d'abord, on doit démontrer que la relation de forcing définie dans la définition 3.34 satisfait les conditions nécessaires à l'obtention d'un modèle d'arbres partiel. Nous savons déjà que, d'après le lemme 3.15, la relation $\models_{\mathfrak{P}}$ est correctement définie. D'autre part, cette relation vérifie également la condition de monotonie requise par la définition 2.8. Supposons en effet que l'on ait $x \models_{\mathfrak{P}} p$ et $y \sqsubseteq x$. Par définition de $\models_{\mathfrak{P}}$ et de $\sqsubseteq$, $y \leq x \in Ass(\mathcal{B})^{\dagger}$, donc, d'après le lemme 3.14 on a $\mathcal{B} \Vdash T\ p : y$ et, par définition de $\models_{\mathfrak{P}}$, $y \models_{\mathfrak{P}} p$.

Il reste donc à démontrer les propriétés 1 et 2. On les démontre simultanément en raisonnant par cas sur $S\ \phi : x$. Dans ces raisonnements, on utilisera implicitement pour conclure le fait que $\mathcal{B} \succ S\ \phi : x$ implique $\mathcal{B} \Vdash S\ \phi : x$:

- Cas $T\ p : x$: $\mathcal{B} \succ T\ p : x$ implique $\mathcal{B} \Vdash T\ p : x$ d'où $x \models_{\mathfrak{P}} p$ par définition de $\models_{\mathfrak{P}}$.
- Cas $F\ p : x$: $\mathcal{B} \succ F\ p : x$ implique $\mathcal{B} \Vdash F\ p : x$ d'où $x \not\models_{\mathfrak{P}} p$ par définition de $\models_{\mathfrak{P}}$.
- Cas $T\ \top : x$: Par définition de $\models_{\mathfrak{P}}$, $x \models_{\mathfrak{P}} \top$.
- Cas $F\ \top : x$: L'antécédent $\mathcal{B} \succ F\ \top : x$ n'est jamais vérifié car sinon il existerait y tel que $y \leq x \in Ass(\mathcal{B})^{\dagger}$ et $F\ \top : y \in \mathcal{B}$ et $\mathcal{B}$ serait alors close d'après le cas (CL3) de la définition 3.22, ce qui contredit l'hypothèse selon la quelle $\mathcal{B}$ est une branche complète, donc ouverte.
- Cas $T\ I : x$: $\mathcal{B} \Vdash T\ I : x$ implique $x \leq 1 \in Ass(\mathcal{B})^{\dagger}$, d'où $x \sqsubseteq 1$ par définition de $\sqsubseteq$.
- Cas $F\ I : x$: $\mathcal{B} \Vdash F\ I : x$ implique $x \leq 1 \notin Ass(\mathcal{B})^{\dagger}$, d'où $x \not\sqsubseteq 1$ par définition de $\sqsubseteq$.
- Cas $T\ [l]\phi : x$: $\mathcal{B} \Vdash T\ [l]\phi : x$ implique l'existence d'un label y tel que $x \cong [l]y \in Ass(\mathcal{B})^{\dagger}$ et $\mathcal{B} \succ T\ \phi : y$. Par hypothèse d'induction, on a alors $y \models_{\mathfrak{P}} \phi$.
- Cas $F\ [l]\phi : x$: Soit $y \in \mathcal{L}(\mathcal{B})$ tel que $x \leq [l]y \in Ass(\mathcal{B})^{\dagger}$. $\mathcal{B} \succ F\ [l]\phi{:}x$ implique alors $\mathcal{B} \succ F\ \phi{:}y$ d'où $y \not\models_{\mathfrak{P}} \phi$ par hypothèse d'induction.
- Cas $T\ \phi \wedge \psi : x$: $\mathcal{B} \Vdash T\ \phi \wedge \psi : x$ implique $\mathcal{B} \succ T\ \phi : x$ et $\mathcal{B} \succ T\ \psi : x$. Donc par hypothèse d'induction, $x \models_{\mathfrak{P}} \phi$ et $x \models_{\mathfrak{P}} \psi$.
- Cas $F\ \phi \wedge \psi : x$: Similaire au cas $T\ \phi \wedge \psi : x$.
- Cas $T\ \phi \vee \psi : x$: Similaire au cas $T\ \phi \wedge \psi : x$.
- Cas $F\ \phi \vee \psi : x$: Similaire au cas $T\ \phi \wedge \psi : x$.
- Cas $T\ \phi \to \psi : x$: Soit $y \in \mathcal{L}(\mathcal{B})$ tel que $y \sqsubseteq x$ et $y \models_{\mathfrak{P}} \phi$. $y \sqsubseteq x$ implique $y \leq x \in Ass(\mathcal{B})^{\dagger}$ par définition de $\sqsubseteq$. De plus $\mathcal{B} \succ T\ \phi \to \psi : x$ implique $\mathcal{B} \succ F\ \phi : y$ ou $\mathcal{B} \succ T\ \psi : y$. D'où, par hypothèse d'induction, $y \not\models_{\mathfrak{P}} \phi$ ou $y \models_{\mathfrak{P}} \psi$. Et comme par hypothèse $y \models_{\mathfrak{P}} \phi$, on a $y \models_{\mathfrak{P}} \psi$.
- Cas $F\ \phi \to \psi : x$: $\mathcal{B} \Vdash F\ \phi \to \psi : x$ implique qu'il existe un label $y \in Ass(\mathcal{B})^{\dagger}$ tel que $y \leq xAss(\mathcal{B})^{\dagger}, \mathcal{B} \succ T\ \phi : x$ et $\mathcal{B} \succ F\ \psi : x$. De $y \leq xAss(\mathcal{B})^{\dagger}$, on obtient $y \sqsubseteq x$ par définition de $\sqsubseteq$ et on a $y \models_{\mathfrak{P}} \phi$ et $x \models_{\mathfrak{P}} \psi$ par hypothèse d'induction.
- Cas $T\ \phi * \psi : x$: $\mathcal{B} \Vdash T\ \phi * \psi : x$ implique qu'il existe deux labels $y, z \in Ass(\mathcal{B})^{\dagger}$ tels que $x \leq yz \in Ass(\mathcal{B})^{\dagger}, \mathcal{B} \succ T\ \phi : y$ et $\mathcal{B} \succ T\ \psi : z$.Comme $x \leq yz \in Ass(\mathcal{B})^{\dagger}$, on a $x \sqsubseteq yz$ par définition $\times$ et de $\sqsubseteq$. De plus, par hypothèse d'induction, on a $y \models_{\mathfrak{P}} \phi$ et $z \models_{\mathfrak{P}} \psi$.
- Cas $F\ \phi * \psi : x$: Soit $y, z \in \mathcal{L}(\mathcal{B})$ tels que $x \sqsubseteq yz$, par définition de $\times$, on a $x \leq yz$. Et comme $\mathcal{B} \Vdash F\ \phi * \psi : x$, on a $y \not\models_{\mathfrak{P}} \phi$ et $y \not\models_{\mathfrak{P}} \psi$ par hypothèse d'induction.
- Cas $T\ \phi {-\!\!*} \psi : x$: Soit $y \in \mathcal{L}(\mathcal{B})$ tel que $y \models_{\mathfrak{P}} \phi$. On a alors par définition de $\times$, $x \times y = xy \in Ass(\mathcal{B})^{\dagger}$. Et donc par définition de la $(\cdot)^{\dagger}$-clôture, on a $y \in Ass(\mathcal{B})^{\dagger}$. $\mathcal{B} \succ T\ \phi {-\!\!*} \psi{:}x$ implique alors $\mathcal{B} \succ F\ \phi : y$ ou $\mathcal{B} \succ T\ \psi : xy$. D'où on déduit $y \not\models_{\mathfrak{P}} \phi$ ou $xy \models_{\mathfrak{P}} \psi$. Comme par hypothèse on a $y \models_{\mathfrak{P}} \phi$, on a donc $xy \models_{\mathfrak{P}} \psi$.

- Cas $F\ \phi \multimap \psi : x$: $\mathcal{B} \Vdash F\ \phi \multimap \psi : x$ implique qu'il existe un label $y \in Ass(\mathcal{B})^{\dagger}$ tel que $xy \in Ass(\mathcal{B})^{\dagger}$.Comme $x \times y = xy$ par définition de $\times$, on a finalement $y \models_{\mathfrak{P}} \phi$ et $x \times y \models_{\mathfrak{P}} \psi$ par hypothèse d'induction.

□

On sait maintenant comment nos tableaux sont complétés et on sait aussi comment construire un contre-modèle quand une branche ouverte est complétée. On peut donc établir le théorème de complétude.

Théorème 3.10. *Si une formule de* BI-Loc *est valide alors elle est prouvable dans* TBI-Loc.

Preuve. Soit ϕ une formule de BI-Loc valide. Supposons que ϕ ne soit pas prouvable dans TBI-Loc, alors il existe une séquence finie de tableaux satisfaisant les conditions de la définition 3.31. Donc la procédure de complétion construit un tableau contenant une branche complète $\mathcal{B}$, éventuellement infinie. Comme $\mathcal{B}$ est complète, on peut construire son $\mathcal{H}$-modèle conformément à la définition 3.34. Puisque $\mathcal{B}$ contient la formule signée $F\ \phi : c$, le lemme 3.20 nous permet de conclure que $c \models_{\mathfrak{P}} \phi$ ce qui contredit l'hypothèse de départ. □

3.7.2 Un exemple d'extraction de contre-modèles

Dans cette section, on montre comment s'établit la construction d'un contre-modèle à partir d'un graphe de ressources. On utilise pour cela le graphe de ressources de la figure 3.6.

D'après le graphe, les ressources utilisées comme base pour construire le monoïde d'arbre partiel est le suivant :

$$\mathcal{L}(\mathcal{B}) = \{1, d_1, d_2, d_3, d_4, d_5, d_6, d_7, c_6, c_7, c_6d_6\}$$

Et la loi de composition sur ses ressources vérifie la loi suivante :
- $c_6 \times d_6 = c_6d_6$;
- $1 \times x = x (\forall x \in \mathcal{LB})$.

Et la relation de pré-ordre entre les ressources est donné par les arrêtes $E(Ass(\mathcal{B})^{\dagger})$. Dans le cas de l'exemple, on a donc $d_3 \sqsubseteq d_1, d_3 \sqsubseteq d_5, d_3 \sqsubseteq d_7, d_4 \sqsubseteq d_2, d_4 \sqsubseteq c_6d_6, d_4 \sqsubseteq c_7$.

L'ensemble des arbres constituant le contre modèle est donc l'ensemble des arbres dont les ressources appartiennent au monoïde décrit ci-dessus.

La relation de forcing est elle engendrée par les formules atomiques positives (de signe T) présentes dans la branche et propagée par monotonie suivant le pré-ordre. Le tableau ci-dessous indique pour chaque propositions les ressources qui les forcent :

	p	q
$\models_{\mathfrak{P}}$	c_6	c_7, d_4

Il reste alors à montrer que ce modèle est un contre modèle de la formule $((\top * [l]p) \wedge (\top * [l]q)) \rightarrow (\top * [l](p \wedge q))$. Tout d'abord, comme $c_6 \models_{\mathfrak{P}} p$, on a bien, $d_3[l]c_6d_6 \models_{\mathfrak{P}} \top * [l]p$ et par la relation $\sqsubseteq$, on a également : $d_3[l]d_4 \models_{\mathfrak{P}} \top * [l]p$. De même, comme $c_7 \models_{\mathfrak{P}} q$, on a : $d_3[l]d_4 \models_{\mathfrak{P}} \top * [l]q$. On a bien $d3[l]d_4 \models_{\mathfrak{P}} (\top * [l]p) \wedge (\top * [l]q)$ Cependant, il n'existe pas de décomposition de $d_2[l]d_4$ forçant $\top * [l](p \wedge q)$. Donc on a bien $\not\models_{\mathfrak{P}} ((\top * [l]p) \wedge (\top * [l]q)) \rightarrow (\top * [l](p \wedge q))$.

3.8 Mise en œuvre de la méthode et perspectives

La mise en œuvre pratique des solutions de recherche de preuves proposées ici est un point indispensable et est envisagée à court terme. On peut pour cela se fonder sur les travaux menés sur BILL, le prouveur développé pour BI [48]. Cette réalisation sort des limites d'études que l'on s'était fixé pour cette thèse mais on peut toutefois pointer quelques problème qui se poseront lors de la réalisation pratique et y apporter des premiers éléments de réponse.

Le problème principal que l'on doit gérer est celui de la terminaison de la méthode. On avons vu précédemment que la procédure de décision de TBI-Loc pouvait ne pas terminer pour deux raisons. La première, déjà présente dans TBI, [72] est la possibilité dans certaines configurations d'avoir des *branches redondantes*. Il s'agit de branches où l'utilisation d'une règle obligationnelle mène à la génération de nouveaux labels qui doivent alors être de nouveaux utilisés par cette règle obligationnelle. Le second problème, spécifique à TBI-Loc celui-ci, se pose quand on doit trouver une valuation pour clore une branche. Sans limitation sur les noms d'emplacements et sur la profondeur des arbres que l'on peut utiliser, on a en effet une infinité de possibilités pour chaque valuation.

Deux approches différentes sont alors possibles. La plus simple à mettre en œuvre est de se contenter de semi-décidabilié et de proposer un prouveur interactif. Cette solution est possible car on sait détecter les cas où la procédure peut ne pas terminer. En effet, dans [72], les branches redondantes sont clairement identifiés et on peut donc déterminer quelles formules peuvent y mener. Concernant les valuations, on devra passer en mode interactif dès qu'on sera en présence d'une branche potentiellement close pour laquelle une valuation sera trouvable.

L'autre possibilité consiste à rendre la méthode décidable, en contrôlant directement dans l'algorithme de recherche de preuves les cas de non-terminaison. Pour les branches redondantes, des arguments permettant de contrôler ces branches en élargissant la notion de classe d'équivalence entre les labels sont développés dans [72]. Concernant les valuations, les résultats exposés dans le cadre du chapitre 2 montrent comment on peut limiter la taille des arbres à considérer pour prouver qu'une formule n'est pas valide. On peut utiliser une démarche similaire pour contrôler la taille des labels intervenants dans une valuation.

Si l'on peut contrôler ces deux cas de non-terminaison, on sera alors capable de proposer un algorithme décidable. Néanmoins, notamment pour la gestion des valuations, une approche interactive peut être plus efficace pour un utilisateur expérimenté car il pourra certainement trouver plus rapidement la valuation permettant de construire un contre-modèle.

En marge de ces problèmes cruciaux se posent d'autres problèmes intéressant d'élaboration de stratégies efficaces de recherche de preuves et de complexité des algorithmes proposés. Ces problèmes n'ont pas encore fait l'objet de réflexion et peuvent être vus comme des perspectives de recherche pertinentes.

Dans la suite de ce document, on présente des instanciations du modèle d'arbres partiels et du langage de manipulation des arbres pour raisonner sur les tas de pointeurs ou les documents XML. Comme dans le cas général, il sera important pour ces applications de pouvoir établir si une assertion est la conséquence logique d'une autre. La méthode de recherche de preuves présentée ici constitue, dans ce cadre, un point de départ très important. Il sera toutefois utile de l'adapter pour pouvoir raisonner non pas sur tous les modèles mais sur un modèle particulier. De plus, il faudra inclure à la méthode actuelle la gestion des quantificateurs.

Chapitre 4

Arbres de ressources et modèles des pointeurs

Le but recherché avec la proposition du modèle d'arbres partiel (arbres de ressources) est de disposer d'un modèle suffisamment général pour pouvoir être instancié de manière à représenter différents modèles plus concrets. La vérification de programmes manipulant des pointeurs est une application potentielle importante. De nombreux modèles ont été proposés pour raisonner sur ces programmes [19, 63, 71, 80]. On s'intéresse dans ce chapitre aux modèles des pointeurs classiques de [62, 75] et aussi à celui avec permissions [17] utilisant des logiques de séparations.

Le modèle classique des pointeurs pouvant être vu comme une instanciation du modèle des monoïdes partiels pour BI [49], il semble normal qu'il puisse être vu également comme une instanciation du modèle d'arbres partiels. On montrera dans ce chapitre qu'une telle instanciation est effectivement possible mais également que la modalité d'emplacement de BI-Loc peut être rapprochée de l'opérateur logique de pointage (*points to*) de la logique des pointeurs [62]. Ainsi, la gestion explicite des emplacements permet d'établir plus clairement le lien entre BI-Loc et cette logique.

On s'intéresse également au modèle des pointeurs avec permissions. Ce modèle peut être vu comme une extension du modèle classique des pointeurs [17]. Une des particularités de ce modèle par rapport au modèle classique est d'introduire la possibilité de séparer un tas de pointeurs en des tas non disjoints. Cette spécificité est déjà traitée dans BI-Loc, puisque deux arbres ne doivent pas nécessairement être disjoints pour pouvoir être composés. Ceci illustre l'intérêt d'avoir un modèle général : les limitations logiques sont alors inhérentes à l'instanciation choisie et l'on peut raisonner de manière générale sur tous les modèles.

Dans ce chapitre, on commence donc par montrer comment on peut instancier le modèle des arbres de ressources pour établir une relation avec le modèle des pointeurs. On montre comment la manipulation des programmes avec pointeurs peut être considérée comme la manipulation d'arbres de ressources et comment on peut prouver un programme sur les pointeurs en raisonnant sur les arbres. On aborde ensuite le modèle avec permissions et la représentation de ce modèle avec BI-Loc qui permet d'apporter une réponse à la spécification des *arbres* dans ce modèle, répondant ainsi à un problème ouvert énoncé dans [17].

4.1 Modèles des pointeurs et arbres de ressources

Les travaux de O'Hearn et al. [62, 75] sur les pointeurs ont mené à la proposition d'un langage de manipulation des pointeurs et à une logique d'assertions pour raisonner sur les programmes utilisant ce langage. Nous avons vu, dans le chapitre 2, que le langage de transformation des arbres et les clauses sémantiques que l'on a proposé pour la manipulation des arbres de ressources sont très proches de ce qui existe pour les pointeurs. Après avoir rappelé le fonctionnement du modèle des pointeurs et ses applications possibles [16], on montre ici comment modéliser un tas de pointeurs sous forme d'arbre de ressources et plus généralement comment raisonner sur un modèle d'arbres partiel pour prouver des propriétés de programmes avec pointeurs.

4.1.1 Le modèle des pointeurs

Un *pointeur* est un couple associant une valeur à un emplacement dans la mémoire. Le modèle proposé dans [62, 75] représente les données d'un programme sous la forme d'un *tas* de pointeurs. L'idée est ensuite de proposer une logique pour définir des propriétés sur ces tas de pointeurs et des primitives de programmation réalisant des manipulations de base sur cette structure (suppression, création, lecture, écriture).

Les domaines sémantiques, utilisés dans [62], pour construire des *tas* de pointeurs sont les suivants :

$$
\begin{array}{l}
Ints = \{\ldots, -1, 0, 1, \ldots\} \\
Loc \subseteq Ints \\
Heap = Loc \rightharpoonup_{fin} Ints \\
Stack = Var \rightharpoonup_{fin} Ints \\
State = Stack \times Heap \\
\text{(généralement, on choisira } Loc = \{1, \ldots, n\}\text{)}
\end{array}
$$

$Ints$ est l'ensemble des entiers relatifs, cet ensemble a été choisi arbitrairement pour représenter de façon simple le nom des pointeurs et les données qu'ils contiennent. L'ensemble des pointeurs est donc un sous-ensemble des entiers relatifs, généralement celui des entiers strictement positifs. Un tas (ou *heap*) est un ensemble fini de pointeurs. Enfin, on propose aussi une pile de variables qui associe à des variables des entiers.

La logique et le langage des manipulations utilisés pour raisonner sur les pointeurs seront introduits plus loin dans le chapitre. Le modèle ainsi obtenu permet de raisonner la manipulation de structures de données diverses comme par exemple les listes [15] ou les graphes [16].

4.1.2 Pointeurs et arbres de ressources

L'idée de base de la représentation des pointeurs par des arbres de ressources est de faire correspondre à un emplacement des pointeurs et à sa valeur associée un emplacement de l'arbre ainsi que la ressource qu'il contient. Pour cela, rappelons tout d'abord comment sont présentés les pointeurs dans [75].

On compare tout d'abord les emplacements utilisés pour les pointeurs et pour les arbres de ressources. Pour les pointeurs, l'ensemble des emplacements correspond à un sous-ensemble des entiers relatifs. Les arbres de ressources utilisent pour leurs emplacements un ensemble dénombrable. On peut donc utiliser comme ensemble d'emplacements pour les arbres le même ensemble que celui utilisé pour les pointeurs.

On définit alors le monoïde de ressources partiel $\mathcal{M} = \{M, \bullet, e, \preccurlyeq\}$ tel que les ressources contenues dans les emplacements d'un arbre de ressources puissent correspondre à la valeur associée à un emplacement dans un tas de pointeurs. Intuitivement, on devrait donc poser $M = Ints$ pour l'ensemble de ressources et la relation $\bullet$ devrait être totalement indéfinie pour s'assurer qu'un emplacement ne peut contenir deux valeurs. Le problème avec un tel choix est que nous n'avons pas alors d'élément neutre pour le monoïde. Pour contourner ce problème, on ajoute artificiellement un élément neutre à l'ensemble des valeurs que l'on manipule. Le monoïde partiel que nous utilisons est alors le suivant :

$$\mathcal{M} = \{Ints \cup \{e\}, e, \bullet, \preccurlyeq\}$$

La relation $\bullet$ est définie comme suit :
- $\forall m, m' \in Ints, m \bullet m'$ n'est pas défini ;
- $\forall m \in Ints \cup \{e\}, m \bullet e = e \bullet m = m$.

et la relation d'ordre $\preccurlyeq$ est réflexive ($m \preccurlyeq m'$ si et seulement si $m = m'$) et totalement indéfinie ailleurs.

On a maintenant tous les éléments pour déterminer l'ensemble des arbres de ressources que nous allons manipuler et pour ensuite établir la correspondance entre un tas de pointeurs $h \in Heap$ et un arbre de ressource $\llbracket h \rrbracket_{RT} \in T_M$ est définie récursivement de la façon suivante :

- $\llbracket [h|l \mapsto v] \rrbracket_{RT} \equiv (e, l \mapsto v) | \llbracket h \rrbracket_{RT}$;
- $\llbracket nil \rrbracket_{RT} \equiv (e, nil)$.

Un arbre de ressources est dit *pur* s'il existe des emplacements $l_1, \dots l_n \in Loc$ et des ressources $m_1, \dots, m_n \in Ints$ tels que $t = (e, l_1 \mapsto m_1) | \dots | (e, l_n \mapsto m_n)$. Autrement dit, un arbre pur contient une valeur de $Ints$ dans chacun des emplacements qui le composent, dont la taille des chemins le composant n'est pas supérieur à 1 et qui ne contient pas de ressources à sa racine.

Lemme 4.1. Un arbre de ressources t est *pur* si et seulement si il existe un tas h de pointeurs tel que $\llbracket h \rrbracket_{RT} = t$

Preuve. Par la définition de $\llbracket \cdot \rrbracket_{RT}$, on sait que si $\llbracket h \rrbracket_{RT} = t$ alors t est pur.

Il reste alors à prouver l'autre implication. Supposons que t soit pur. Alors il existe $l_1, \dots l_n \in Loc$ et des ressources $m_1, \dots, m_n \in Ints$ tels que $t = (e, l_1 \mapsto m_1) | \dots | (e, l_n \mapsto m_n)$. Si on considère le tas de pointeurs $h = [l_1 \mapsto m_1 | l_2 \mapsto m_2 | \dots | l_n \mapsto m_n]$, on a bien $\llbracket h \rrbracket_{RT} = t$. $\square$

Définition 4.1 (Restriction à un arbre pur). *La* restriction d'un arbre $t = (m, t')$ à un arbre pur *est l'arbre $t_p = (e, t'')$ où t'' est tel que pour tout emplacement $l \in Loc$ s'il existe $m \in Ints$ tel que $t(l) = (m, nil)$ ssi $t''(l) = (m, nil)$, sinon $t''(l)$ n'est pas défini.*

4.1.3 Logiques d'assertions pour les pointeurs

On montre ici comment la logique de séparation utilisée pour exprimer les assertions sur les pointeurs dans [75] peut correspondre à la logique d'assertions proposée dans le chapitre 2. Afin de distinguer sans ambiguïté les assertions sur les pointeurs des assertions sur les arbres de ressources, on nomme P-assertion une assertion sur les pointeurs.

Commençons par rappeler la syntaxe et la sémantique de la logique d'assertions pour les pointeurs :

$$\begin{array}{rcl} P, Q & ::= & \alpha \mid E \mapsto F \\ & \mid & false \mid P \rightarrow Q \mid \exists x.P \\ & \mid & emp \mid P * Q \mid P \multimap\!\!* Q \end{array}$$

avec $\alpha ::= E = F \mid E < F$ et où E et F sont soit des variables, soit des entiers, soit des expressions de type $E+F$, $E-F$, $E \times F$.

La sémantique de ces formules est définie pour une pile $s \in Stack$ et un tas de pointeurs $h \in Heap$ par les clauses sémantiques suivantes :

- $s, h \models \alpha$ ssi $[\![\alpha]\!]_s = true$;
- $s, h \models \exists x.P$ ssi $\exists v \in Ints$ tel que $[s|x \mapsto v], h \models P$;
- $s, h \models E \mapsto F$ ssi $[\![E]\!]_s = dom(h)$ et $h([\![E]\!]_s) = [\![F]\!]_s$;
- $s, h \models false$ jamais ;
- $s, h \models P \to Q$ ssi $s, h \models P$; $s, h \models Q$;
- $s, h \models emp$ ssi $h = []$ est le tas vide ;
- $s, h \models P * Q$ if $\exists h_0, h_1$ (disjoint), $h_0 * h_1 = h$, $s, h_0 \models P$, $s, h_1 \models Q$;
- $s, h \models P \mathrel{-\!\!*} Q$ ssi $\forall h'$ tel que h' et h soient disjoints et $s, h' \models P$ alors $s, h' * h \models Q$.

On peut dès lors établir la correspondance entre une P-assertion propositionnelle P (sans quantificateurs) et sa correspondance $[\![P]\!]_{BI-Loc}$ dans le langage d'assertions des arbres de ressources. Cette relation se définit récursivement comme suit :

- $[\![E \mapsto F]\!]_{BI-Loc} = [E]F$;
- $[\![P \Box Q]\!]_{BI-Loc} = [\![P]\!]_{BI-Loc} \Box [\![Q]\!]_{BI-Loc}$ pour tous les opérateurs binaires $\Box$;
- $[\![emp]\!]_{BI-Loc} = I$;
- $[\![false]\!]_{BI-Loc} = \bot$;
- $[\![\alpha]\!]_{BI-Loc} = \alpha$;
- $[\![\exists x.P]\!]_{BI-Loc} = \exists_{res} x.P$.

On prouve maintenant une propriété cruciale pour la correspondance entre le modèle des pointeurs et l'instance des arbres de ressources que l'on considère ici. Il s'agit de montrer que si un arbre force une formule correspondant à une P-assertion pour les pointeurs, la restriction de cet arbre à un arbre pur la force également. Pour cela, nous analysons chacun des cas qui peut rendre un arbre *non-pur*.

Lemme 4.2. Soient deux arbres t et t', un emplacement l et une P-assertion P tels que $t = t'|(e, l \mapsto (e, nil))$ et $l \notin Loc'_t$ alors si $s, t \models_{\mathfrak{A}} [\![P]\!]_{BI-Loc}$ on a également $s, t' \models [\![P]\!]_{BI-Loc}$.

Preuve. Preuve par induction structurelle sur P. On ne détaille ici que le cas central, celui de $P := Q \mathrel{-\!\!*} Q'$:

On suppose que $s, t'|(e, l \mapsto (e, nil)) \models_{\mathfrak{A}} [\![Q \mathrel{-\!\!*} Q']\!]_{BI-Loc}$. Soit t'' tel que $s, t'' \models_{\mathfrak{A}} [\![q]\!]_{BI-Loc}$. Par définition $s, t''|t'|(e, l \mapsto (e, nil)) \models_{\mathfrak{A}} [\![Q']\!]_{BI-Loc}$. Par hypothèse d'induction on a donc $s, t''|t' \models_{\mathfrak{A}} [\![Q']\!]_{BI-Loc}$. Et donc $s, t' \models_{\mathfrak{A}} [\![Q \mathrel{-\!\!*} Q']\!]_{BI-Loc}$. $\Box$

Lemme 4.3. Soient trois arbres t et t' et (m, f), un emplacement l et une P-assertion P tels que $t = t'|(e, l \mapsto (m, f))$ et tels que $f \neq nil$ (f est défini quelque part) si $s, t \models_{\mathfrak{A}} [\![P]\!]_{BI-Loc}$ on a également $s, t' \models [\![P]\!]_{BI-Loc}$.

Preuve. Par induction structurelle sur P. $\Box$

Lemme 4.4. Soient deux arbres t et m, f et une P-assertion P tels que $t = (m, f)$ alors si $s, t \models_{\mathfrak{A}} [\![P]\!]_{BI-Loc}$ on a également $s, (e, f) \models [\![P]\!]_{BI-Loc}$.

Preuve. Par induction structurelle sur P. □

Corollaire 4.5. Soit un arbre t, t_p la restriction pure de cet arbre et une assertion sur les pointeurs P, si $s, t \models_{\mathfrak{A}} [\![P]\!]_{BI-Loc}$ on a également $s, t_p \models [\![P]\!]_{BI-Loc}$.

Preuve. Ce résultat est la conséquence directe des lemmes 4.2, 4.3 et 4.4. □

Ce corollaire est indispensable pour établir le lien entre la prouvabilité pour les arbres et celle pour les pointeurs.

Lemme 4.6. Soit un tas de pointeurs $h \in Heap$, une pile s et une assertion sur les pointeurs P sans $-\!\!*$, on a : $s, [\![h]\!]_{RT} \models_{\mathfrak{A}} [\![P]\!]_{BI-Loc}$ ssi $s, h \models P$.

Preuve. La preuve se fait par induction sur P :

- Cas emp : Supposons que $s, [\![h]\!]_{RT} \models_{\mathfrak{A}} [\![emp]\!]_{BI-Loc}$ donc $s, [\![h]\!]_{RT} \models_{\mathfrak{A}} I$. Et par définition de $\models_{\mathfrak{A}}$, on a donc $[\![h]\!]_{RT} = e$, donc h est le tas vide et $s, h \models P$. L'autre implication suit le même schéma.
- Cas $false$: Triviale : dans les deux cas, la formule n'est jamais satisfaite.
- Cas $\exists x.P$: Triviale : la relation de forcing est la même dans les 2 cas.
- Cas α : $[\![\alpha]\!]_{BI-loc} = \alpha$. La satisfaction de cette formule dépend donc uniquement de s et on peut conclure.
- Cas $E \mapsto F$: On pose $[\![E]\!]_s = l$ et $[\![F]\!]_s = m$. Supposons que $s, [\![h]\!]_{RT} \models_{\mathfrak{A}} [\![E \mapsto F]\!]_{BI-Loc}$, on a alors $[\![H]\!]_{RT} = (e, l \mapsto (m, nil))$ et par définition, le seul tas de pointeurs qui correspond à cet arbre est $[l \mapsto m]$. Or on a bien $s, [l \mapsto m] \models E \mapsto F$. L'autre implication suit le même schéma.
- Cas $Q \to Q'$: Immédiat par hypothèse d'induction.
- Cas $Q * Q'$: On suppose $s, [\![h]\!]_{RT} \models_{\mathfrak{A}} [\![P]\!]_{BI-Loc}$, donc $s, [\![h]\!]_{RT} \models_{\mathfrak{A}} [\![Q]\!]_{BI-Loc} * [\![Q']\!]_{BI-Loc}$. Il existe donc deux arbres partiels t_1, t_2 tels que $[\![h]\!]_{RT} = t|t_2$, $s, t_1 \models_{\mathfrak{A}} [\![Q]\!]_{BI-Loc}$ et $s, t_2 \models_{\mathfrak{A}} [\![Q']\!]_{BI-Loc}$. Il est alors évident que $[\![h]\!]_{RT} = t_{1p}|t_{2p}$ et d'après le corollaire 4.5, on a $s, t_{1p} \models_{\mathfrak{A}} [\![Q]\!]_{BI-Loc}$ et $s, t_{2p} \models_{\mathfrak{A}} [\![Q']\!]_{BI-Loc}$. Par le lemme 4.1, on sait qu'il existe deux tas h_1 et h_2 tels que $[\![h_1]\!]_{RT} = t_1$ et $[\![h_2]\!]_{RT} = t_2$ et on a $h = h_1 * h_2$. D'où $s, h \models P$. L'autre implication suit le même schéma.

□

Corollaire 4.7. Soit une assertion P telle qu'il n'existe pas de sous formule de P de la forme $Q \to Q'$ où Q contient l'opérateur $-\!\!*$. Soit un tas de pointeurs $h \in Heap$, une pile s si : $s, [\![h]\!]_{RT} \models_{\mathfrak{A}} [\![P]\!]_{BI-Loc}$ alors $s, h \models P$.

Preuve. Le corollaire découle du lemme précédent. Il reste simplement à montrer que l'implication est vrai pour les formules du type $Q -\!\!* Q'$:

Supposons que $s, [\![h]\!]_{RT} \models_{\mathfrak{A}} [\![Q -\!\!* Q']\!]_{BI-Loc}$. Soit un tas h' tel que $s, h' \models Q$. Par hypothèse d'induction, $s, [\![h']\!]_{RT} \models_{\mathfrak{A}} [\![Q]\!]_{BI-Loc}$. On a $s, [\![h']\!]_{RT}|[\![h]\!]_{RT} \models_{\mathfrak{A}} [\![Q']\!]_{BI-Loc}$ et on en déduit $s, h * h' \models Q'$. □

Il n'est pas possible de démontrer la réciproque de ce corollaire car $P -\!\!* Q$ nécessite de vérifier la propriété P pour tous les arbres de ressources, pas uniquement sur les arbres purs.

4.1.4 Manipulation des pointeurs

Dans la section 2.4, on a proposé un langage de générique de manipulation des arbres. Nous l'utilisons ici pour établir un lien entre la manipulation d'un tas de pointeurs la manipulation de l'arbre de ressources correspondant.

Rappelons les commandes de bases proposées pour la manipulation des tas de pointeurs :

Affectation d'une cellule :

$$\frac{[\![E]\!]_s = l \in Loc, [\![E]\!]_s = m \in M}{[E] := F, s, h \rightsquigarrow s, [h|l \mapsto m]}$$

Libération :

$$\frac{[\![E]\!]_s = l \in Loc \cap dom(h)}{dispose(E), s, h \rightsquigarrow s, (h-l)^*}$$

Où $(h - l)$ représente le tas h privé de la cellule l.

Affectation de variable :

$$\frac{[\![E]\!]_s = m \in M}{x := E, s, h \rightsquigarrow [s|x \mapsto m], h}$$

Observation du contenu :

$$\frac{[\![E]\!]_s = l \in Loc, h(l) = m \in M}{x := [E], s, h \rightsquigarrow [s|x \mapsto m], h}$$

Création d'une cellule :

$$\frac{[\![E_i]\!]_s = m_i \in M, \forall i \in [\![0..n-1]\!]\, (l+i) \in, Loc \cap dom(h)}{cons(E_1, \ldots, E_n), s, h \rightsquigarrow s, [h|l \mapsto m_1| \ldots |l+n-1 \mapsto m_n]}$$

Les axiomes arrières *(backward axioms)* correspondants à ces commandes exprimées dans le langage d'assertions des pointeurs détaillé ci-dessus sont les suivants :

$$\begin{array}{lll}
\{\exists x.(E \mapsto x) * ((E \mapsto F) -\!\!* P)\} & [E] := F & \{P\} \\
\{\exists x.(E \mapsto x) * P\} & dispose(E) & \{P\} \\
\{\forall x'.((x' \mapsto E_1) * \ldots * ((x'+n-1) \mapsto E_n)) -\!\!* P\{^{x'}/_x\}\} & x := cons(E_1, \ldots E_n) & \{P\} \\
\{P\{^E/_x\}\} & x := E & \{P\} \\
\{\exists x'.P\{^{x'}/_x\} \wedge ((E \mapsto x') * \top)\} & x := [E] & \{P\}
\end{array}$$

Ces commandes sont très proches de certaines commandes pour la manipulation des arbres. La relation entre la manipulation d'arbres et de pointeurs est donc naturelle pour la plupart des commandes. Si l'on excepte la commande *cons*, la commande C sur les pointeurs a comme équivalent la commande $[\![C]\!]_{point}$ dans le langage de manipulation des arbres, définie par :

- $[\![[E] := F]\!]_{point} \equiv update_{res}(E, F)$;
- $[\![dispose(E)]\!]_{point} \equiv dispose_{loc}(E)$;
- $[\![x := E]\!]_{point} \equiv x := E$;
- $[\![x := [E]]\!]_{point} \equiv x := res(E)$.

Gérer la création de nouvelles cellules (la commande *cons*) n'est pas possible avec le langage de manipulation d'arbres présenté en section 2.4. En effet, la commande de création d'emplacements dans les arbres new_{loc} permet de créer un seul emplacement à la fois et choisit dynamiquement le nom d'emplacement à utiliser. On ne peut donc pas assurer la création de n emplacements adjacents avec cette commande.

On propose donc ici une version modifiée de la commande *new* qui permet de créer ces n emplacement adjacents :

$$\{P \wedge no(E : l) \wedge no(E : l+1) \wedge \ldots \wedge no(E : l+n-1)\} new_{loc}(n, E) \{P * [E][l]e * \ldots * [E][l+n-1]e\}$$

et l'axiome arrière correspondant est :

$$\{\forall_{loc} x'.(no(E : x') \wedge no(E : x'+1)) \rightarrow (([E][l]e * \ldots * [E][x'+n-1]e) -\!\!* P\{^{x'}/_{x}\})\} new_{loc}(n, E) \{P\}$$

La correspondance entre la commande *cons* et le langage de manipulation d'arbre est alors donné par :

$$[\![x := cons(E_1, \ldots, E_n)]\!]_{point} = x := new_{loc}(n, nil); update(x, E_1); \ldots; update(x + n - 1, E_n);$$

Cette dernière relation met en avant une différence cruciale entre les deux modèles. Dans le cas des pointeurs, on doit nécessairement avoir une valeur pour créer un emplacement, on ne peut construire une cellule sans indiquer ce qu'elle va contenir. Les arbres de ressources permettent de créer des emplacements vides. On crée donc les n emplacements vides puis on les remplit avec les valeurs fournies en paramètre. Ainsi donc, en utilisant uniquement des commandes de manipulation d'arbres, qui correspondent à des manipulations de pointeurs, on ne peut pas créer d'emplacement vide.

On établit alors que l'exécution de programmes conserve la relation entre un tas de pointeurs et l'arbre de ressources correspondant :

Lemme 4.8. Soit une configuration de C, s, h (C une commande, s une pile et h un tas de pointeurs) non bloquée et une configuration s', h' telle que $C, s, h \rightsquigarrow s', h'$. Alors, $[\![C]\!]_{point}, s, [\![h]\!]_{RT}$ n'est pas bloquée et si $[\![C]\!]_{point}, s, [\![h]\!]_{RT} \rightsquigarrow s'', t$ alors on a $s' = s''$ et $[\![h']\!]_{RT} = t$.

Preuve. Le lemme est une conséquence directe de la sémantique des commandes. □

On a établit une correspondance entre le modèle des pointeurs et celui des arbres de ressources et également entre les commandes de ces deux modèles. On peut donc utiliser le modèle des arbres de ressources pour raisonner sur les pointeurs.

Théorème 4.1. *Soient deux assertions sur pointeurs P et Q et un programme de manipulation de tas de pointeurs C.*

1. *Si P et Q ne contiennent pas l'opérateur $-\!\!*$ alors $\{[\![P]\!]_{BI-Loc}\}[\![C]\!]_{point}\{[\![Q]\!]_{BI-Loc}\}$ ssi $\{P\}C\{Q\}$.*
2. *Si P et Q ne contiennent pas de sous-formules du type $P_1 \rightarrow P_2$ où P_1 contient l'opérateur $-\!\!*$ et si $\{[\![P]\!]_{BI-Loc}\}[\![C]\!]_{point}\{[\![Q]\!]_{BI-Loc}\}$ alors $\{P\}C\{Q\}$.*

Preuve. Le théorème est une conséquence directe des lemmes 4.6 et 4.8 pour le cas 1 et des lemmes 4.7 et 4.8 pour le cas 2. □

$\{P_0\}$	
$x := cons(a, a);$	
$\{P_1\}$	$P_0 \equiv \forall x'((x' \mapsto a) * ((x'+1) \mapsto a)) \mathrel{-\!\!*} P_1\{^{x'}/_x\}$
$y := cons(b, b);$	$P_1 \equiv \forall y'((x' \mapsto b) * ((y'+1) \mapsto b)) \mathrel{-\!\!*} P_2\{^{y'}/_y\}$
$\{P_2\}$	$P_2 \equiv \exists z((x+1) \mapsto z) * (((x+1) \mapsto t_x) \mathrel{-\!\!*} P_3)$
$[x+1] := t - x;$	$P_3 \equiv \exists z'.((y+1) \mapsto z') * (((y+1) \mapsto x-t) \mathrel{-\!\!*} P_f)$
$\{P_3\}$	$P_f \equiv (x \mapsto a) * ((x+1) \mapsto t-x) * (y \mapsto b) * ((y+1) \mapsto x-t)$
$[y+1] := x - t;$	
$\{P_f\}$	

FIG. 4.1 – Exemple de code manipulant des pointeurs

4.1.5 Exemple d'utilisation et intérêt

On présente ici un exemple simple de manipulation de pointeurs initialement présenté dans [75], repris en figure 4.1, qui illustre les procédures de transformations des commandes et des triplets.

Le programme et les assertions correspondantes sont donnés sur la gauche, la version longue des assertions est données sur la droite. Les assertions sont créées à partir des axiomes arrières présentés en section 4.1.4. Le programme de l'exemple est simple : il s'agit de créer deux couples de pointeurs. Les pointeurs du premier couple (x et $x+1$) contiendront tous deux la valeur a, ceux du second couple (y et $y+1$) contiendront quant à eux la valeur b. Ensuite, on remplace la valeur contenue dans $x+1$ par $t-x$ et celle contenue dans $y+1$ par $x-t$. La figure donne le code et les triplets de Hoare correspondants.

On construit à partir de ces données le programme et les triplets de Hoare équivalents dans le modèle des arbres partiel. Pour cela, on utilise les correspondances entre les commandes spécifiées en section 4.1.4. Commençons par déterminer les commandes équivalentes. La figure 4.2 place côte à côte les commandes sur les pointeurs (à gauche) et celle sur les arbres de ressources (à droite).

Manipulation de pointeurs :	**Manipulation d'arbre de ressources :**
$x := cons(a, a);$	$x := new_{loc}(2, nil); update(x, a); update(x+1, a);$
$y := cons(b, b);$	$y := new_{loc}(2, nil); update(y, b); update(y+1, b);$
$[x+1] := t - x;$	$update_{res}(x+1, t-x);$
$[y+1] := x - t;$	$update_{res}(y+1, x-t);$

FIG. 4.2 – Équivalences entre les commandes du programme exemple

On détermine les assertions pour les arbres de ressources. On commence par la formule de BI-Loc équivalente à la formule P_f. D'après la correspondance présenté dans la section 4.1.3, on a $[\![P_f]\!]_{BI-Loc} = [x]a * [x+1](t-x) * [y]b * [y+1](x-t)$. À partir de cette formule, on raisonne sur les axiomes arrières présentés dans le chapitre 2. Ainsi, commande par commande, on obtient les assertions présentées en figure 4.3.

Si l'on compare les axiomes de la version initiale (la figure 4.1) avec ceux de la version arbres de ressources (la figure 4.3), on remarque que les axiomes de mise à jour des ressources sont plus complexes dans cette dernière. Ceci s'explique par le fait que, dans le cas général des arbres de

$$\begin{array}{l}
\{P'_0\}\\
x := new_{loc}(2, nil);\\
\{P'_1\}\\
update_{res}(x, a);\\
\{P'_2\}\\
update_{res}(x+1, a);\\
\{P'_3\}\\
y := new_{loc}(2, nil);\\
\{P'_4\}\\
update_{res}(y, b);\\
\{P'_5\}\\
update_{res}(y+1, b);\\
\{P'_6\}\\
update_{res}(x+1, t-x);\\
\{P'_7\}\\
update_{res}(y+1, x-t);\\
\{[\![P_f]\!]_s\}
\end{array}$$

$$\begin{array}{lcl}
P'_0 & \equiv & \forall x'.((no(x') \wedge no(x'+1)) \rightarrow (([x']I * [x'+1]I) \mathrel{-\!\!*} P'_1\{^{x'}/_{x}\}))\\
P'_1 & \equiv & \exists_{res} z.[x]z * (\forall_{res} z'.((z'=e) \vee (([x]z' * \top) \rightarrow \bot)) \wedge ([x]a \mathrel{-\!\!*} P'_2))\\
P'_2 & \equiv & \exists_{res} z.[x+1]z * (\forall_{res} z'.((z'=e) \vee (([y+1]z' * \top) \rightarrow \bot))\\
 & & \qquad \wedge([x+1]a \mathrel{-\!\!*} P'_3))\\
P'_3 & \equiv & \forall y'.((no(y') \wedge no(y'+1)) \rightarrow (([y']I * [y'+1]I) \mathrel{-\!\!*} P'_4\{^{y'}/_{y}\}))\\
P'_4 & \equiv & \exists_{res} z.[y]z * (\forall_{res} z'.((z'=e) \vee (([y]z' * \top) \rightarrow \bot)) \wedge ([y]b \mathrel{-\!\!*} P'_5))\\
P'_5 & \equiv & \exists_{res} z.[y+1]z * (\forall_{res} z'.((z'=e) \vee (([y+1]z' * \top) \rightarrow \bot))\\
 & & \qquad \wedge([y+1]b \mathrel{-\!\!*} P'_6))\\
P'_6 & \equiv & \exists_{res} z.[x+1]z * (\forall_{res} z'.((z'=e) \vee (([x+1]z' * \top) \rightarrow \bot))\\
 & & \qquad \wedge([x+1](t-x) \mathrel{-\!\!*} P'_7))\\
P'_7 & \equiv & \exists_{res} z.[y+1]z * (\forall_{res} z'.((z'=e) \vee (([y+1]z' * \top) \rightarrow \bot))\\
 & & \qquad \wedge([y+1](x-t) \mathrel{-\!\!*} [\![P_f]\!]_s))\\
[\![P_f]\!]_s & \equiv & [x]a * [x+1](t-x) * [y]b * [y+1](x-t)
\end{array}$$

FIG. 4.3 – Assertions correspondantes avec le modèle d'arbres partiel

ressources, la séparation d'un arbre en deux sous-arbres n'assure pas que les sous-arbres soient disjoints. Cette précaution n'est pas utile ici puisque dans le cas des pointeurs, un emplacement ne peut contenir qu'une ressource indécomposable. Cependant, on verra dans la section 4.2 que, pour certaines applications, il est utile de considérer que les ressources présentes dans un emplacement peuvent se décomposer et mener à des arbres non disjoints.

L'exemple qui est décrit ici est mentionné dans [75] sous une autre forme, n'utilisant pas des axiomes arrières mais des axiomes *constructifs*, c'est-à-dire où la post-condition est donnée en fonction de la pré-condition. Dans la proposition originale, l'assertion de départ indique que le tas initial doit être vide. Normalement, l'assertion P_0 doit donc être une conséquence logique de la formule *emp*. Un des intérêts des arbres de ressources est que pour vérifier ces problèmes de conséquence logique, on peut se fonder sur les techniques de recherche de preuves développées dans le chapitre 3. La correspondance entre les deux modèles permet ainsi d'utiliser les outils du modèle d'arbres partiel pour le modèle des pointeurs.

4.1.6 La propriété de fenêtrage

On a vu que l'on peut exprimer les triplets des commandes sur les pointeurs comme des triplets de manipulation des arbres de ressources. De plus, il est établit que la propriété de fenêtrage *(frame property)* pour le modèle des pointeurs, fournie ci dessous, est vérifiée même si on inclut la commande qui crée des cellules.

$$\frac{\{P * P'\}C\{Q * P'\}}{\{P\}C\{Q\}}\;*$$

$$*: fv(P') \cap Modified(C) = \emptyset$$

On peut donc se demander pourquoi la propriété de fenêtrage pour les arbres de ressources exclut la commande new_{loc}. La différence, déjà signalée précédemment, vient du fait que l'on ne crée pas d'emplacement vide dans le cadre de la logique des pointeurs. On peut donc chercher à établir une propriété de fenêtrage équivalente pour les arbres de ressources si on restreint les formules des assertions à l'ensemble des formules correspondant à des formules de la logique des pointeurs et si on restreint les commandes à celles correspondant à des commandes sur les pointeurs.

En effet, la commande *cons* qui crée de nouvelles cellules ne crée jamais de cellule vide. Par conséquent son équivalent dans le langage de manipulation des arbres de ressources ne crée pas d'emplacement contenant uniquement la ressource e. De plus le lemme 4.2 démontre qu'une assertion de la logique des pointeurs ne nécessite jamais un emplacement vide.

Dans le cas précis des programmes de manipulations d'arbres de ressources représentant des programmes de manipulation de pointeurs, on peut donc proposée une propriété de fenêtrage intégrant la création d'emplacements, contrairement à celle du théorème 2.2.

Proposition 4.1. *La propriété de fenêtrage suivante est correcte pour* $\mathsf{BI\text{-}Loc}$ *:*

$$\frac{\{\llbracket P\rrbracket_{BI-Loc} * \llbracket P'\rrbracket_{BI-Loc}\}\llbracket C\rrbracket_{point}\{\llbracket Q\rrbracket_{BI-Loc} * \llbracket P'\rrbracket_{BI-Loc}\}}{\{\llbracket P\rrbracket_{BI-Loc}\}\llbracket C\rrbracket_{point}\{\llbracket Q\rrbracket_{BI-Loc}\}} \; *$$

$$* \; : fv(P') \cap Modified(C) = \emptyset$$

Preuve. Le point clé par rapport au théorème 2.2 concerne la commande de création d'emplacements. Supposons que l'on cherche à étendre le contexte d'un programme vérifiant le triplet $\{\llbracket P\rrbracket_{BI-Loc}\}\llbracket C\rrbracket_{point}\{\llbracket Q\rrbracket_{BI-Loc}\}$ par un arbre vérifiant $\llbracket P'\rrbracket_{BI-Loc}$. On s'intéresse essentiellement au cas où $C = cons(E_1, \ldots, E_n)$. On a alors trois possibilités pour chaque emplacement créé :

1. L'emplacement créé n'existe pas dans les arbres vérifiant $\llbracket P'\rrbracket_{BI-Loc}$. Par conséquent, on ne peut altérer l'arbre en faisant référence à ce nouvel emplacement dans la suite du programme.
2. L'emplacement créé existe dans les arbres vérifiant $\llbracket P'\rrbracket_{BI-Loc}$ et il contient une ressource autre que e. Par conséquent, l'ajout à cet emplacement de la ressource E_i va mener à un arbre indéfini. Et la configuration est donc bloquée pour ce choix de nom d'emplacement.
3. L'emplacement créé existe dans des arbres vérifiant $\llbracket P'\rrbracket_{BI-Loc}$ et il contient la ressource e. D'après le lemme 4.2, le même arbre sans cet emplacement vérifie aussi $\llbracket P'\rrbracket_{BI-Loc}$ et par conséquent les commandes suivantes n'altèrent pas les satisfaction de $\llbracket P'\rrbracket_{BI-Loc}$.

□

4.2 Arbres de ressources et permissions

Le modèle des permissions proposé par Bornat et al. [17] est une extension du modèle des pointeurs permettant de raisonner sur les accès concurrents à des variables. Il consiste à assigner un poids à chaque relation $\mapsto$ pour indiquer les droits d'accès associés à une variable. Par exemple $l \overset{1}{\mapsto} m$ indique qu'on peut accéder en lecture et en écriture à la vriable l alors que $l \overset{x}{\mapsto} m$ pour $0 < x < 1$ indique que l'on a accès à l uniquement en lecture. Ce principe permet de raisonner sur des programmes concurrents et d'analyser si le partage des données se fait correctement.

L'une des spécificités importante de ce modèle si on le compare au modèle des pointeurs est que deux tas de pointeurs ne doivent pas nécessairement être disjoints pour être composés. Ainsi, $(l \overset{0,5}{\mapsto} m) * (l \overset{0,5}{\mapsto} m)$ est défini. Cette nouveauté soulève des problèmes qui ne se posent pas dans le modèle des pointeurs, la formule $P * Q$ n'indiquant pas que P et Q sont disjoints. On montre ici comment l'utilisation d'un modèle plus général, comme celui des arbres de ressources, pour représenter les permissions permet d'apporter des réponses à certains de ces problèmes.

On donne en figure 4.4 un court exemple tiré de [17] qui montre une utilisation possible du modèle des permissions. Dans cet exemple, l'opérateur $\therefore$ permet d'indiquer que l'assertion de droite est une conséquence logique de celle de gauche.

Le code proposé crée donc une cellule x à laquelle il peut accéder en lecture et en écriture (elle a un poids de 1). Il y place ensuite la valeur 7. Le programme se sépare ensuite en 2 sous-programmes concurrents. Pour ce faire, il choisit donc de donner aux deux programmes l'accès à x en lecture (x a un poids de $0,5$ dans les deux sous-programmes). Chaque processus lit alors la valeur de x et l'utilise. Quand les deux sous-programmes se terminent, le programme principal peut de nouveau écrire dans la cellule x ou la supprimer. Le programme décide alors de supprimer la cellule x et se termine. Cet exemple montre comment le poids sur une ressource permet d'exprimer le partage possible de cette ressource entre divers processus.

$$
\begin{array}{l}
\{emp\} \\
x := new(); \\
\{x \overset{1}{\mapsto} \_\} \\
[x] := 7; \\
\{x \overset{1}{\mapsto} 7\} \therefore \{x \overset{0.5}{\mapsto} 7 * x \overset{0.5}{\mapsto} 7\} \\
\left(\begin{array}{l||l}
\{x \overset{1}{\mapsto} 7\} & \{x \overset{1}{\mapsto} 7\} \\
y := [x] - 1; & z := [x] + 1; \\
\{x \overset{1}{\mapsto} 7 \wedge y = 6\} & \{x \overset{1}{\mapsto} 7 \wedge z = 8\}
\end{array}\right) \\
\{x \overset{0.5}{\mapsto} 7 * x \overset{0.5}{\mapsto} 7 \wedge y = 6 \wedge z = 8\} \therefore \{x \overset{1}{\mapsto} 7 \wedge y = 6 \wedge z = 8\} \\
dispose(x) \\
\{emp \wedge y = 6 \wedge z = 8\}
\end{array}
$$

FIG. 4.4 – Exemple de code distribué utilisant les permissions

On montre ici que ce modèle peut être exprimé très naturellement comme une instance du modèle des arbres partiels. De plus, l'utilisation de BI-Loc comme logique d'assertions a permis de résoudre naturellement le problème de la spécification d'arbres avec permissions. Ce problème était un des problèmes ouverts mentionnés dans [17]. La principale difficulté étant de spécifier dans ce contexte que deux nœuds distincts ne partagent pas un fils.

4.2.1 Le modèle des permissions

Un modèle générique pour les permissions est défini comme suit :

$$Heap = Loc \rightharpoonup_{fin} (V \times M)$$

Ce modèle associe à un nombre fini d'emplacement de Loc un couple $(v, m) \in (V \times M)$ où v est la valeur contenue à cet emplacement et où m correspond aux permissions d'accès associées à un emplacement. Loc est donc un ensemble dénombrable d'emplacements, V est un ensemble de valeurs, M est un ensemble de ressources.

On considère un opérateur binaire $\bullet$ tel que $\mathcal{M} = (M, \bullet)$ est un semi-groupe partiel et commutatif et on définit la composition sur les couples de $V \times M$ par extension de l'opérateur

$\bullet$ sur l'ensemble M. Ainsi, pour $(v,m),(v',m') \in V \times M$ on a $(v,m) \bullet (v',m') = (v, m \bullet m')$ ssi $v = v'$ et $m \bullet m'$ est défini.

De même, la composition $h \bullet h'$ de deux tas $h, h' \in Heap$ est définie par :
$dom(h \bullet h') = dom(h) \cup dom(h')$ et pour tout $l \in dom(h \bullet h')$, on a :

$$(h \bullet h')(l) = \begin{array}{l} h(l) \text{ si } h'(l) \text{ n'est pas défini.} \\ h'(l) \text{ si } h(l) \text{ n'est pas défini.} \\ h(l) \bullet h'(l) \text{ sinon.} \end{array}$$

Contrairement aux tas de pointeurs que l'on manipulait dans la section 4.1, la composition de deux tas ne nécessite plus que leurs domaines soient disjoints. Cette propriété nouvelle pour les pointeurs ne l'est pas dans le cadre des arbres de ressources puisqu'on a prévu initialement de pouvoir composer des arbres ayant des chemins en commun.

4.2.2 Les arbres et les permissions

Notre préoccupation première est de déterminer le monoïde de ressources partiel qui sera utilisé pour définir les ressources contenues dans les nœuds. L'ensemble $V \times M$ et son opérateur de composition $\bullet$ pourrait convenir mais nous n'aurions pas alors d'élément neutre. On ajoute donc un élément neutre e à cet ensemble. L'ensemble des ressources que l'on manipule est donc $(V \times M) \cup \{e\}$ et l'opérateur de composition $\bullet$ est étendu à cet ensemble en posant pour tout $(v,m) \in V \times M, (v,m) \bullet e = e \bullet (v,m) = (v,m)$. Comme précédemment, on utilisera une relation de pré-ordre $\sqsubseteq$ complètement indéfinie.

Ainsi donc, on a un monoïde partiel $\mathcal{M} = \langle (V \times M) \cup \{e\}, \bullet, e, \sqsubseteq \rangle$ pouvant servir de base pour construire les arbres de ressources. Comme la composition des permissions correspond exactement à la composition des arbres, on peut établir une correspondance entre les deux domaines. Si une variable l est associée à la valeur v avec la permission m (ce qui est noté $l \overset{m}{\mapsto} v$), on lui associe l'arbre $[\![l \overset{m}{\mapsto} v]\!]_{perm} = (e, l \mapsto ((v,m), nil))$.

Mais peut-on trouver une correspondance pour tous les arbres de hauteur 1 dans le domaine des tas de permissions, en particulier, comment un arbre de la forme $(e, l \mapsto (e, nil))$ peut-il être interprété ?

Comme on a $(e, l \mapsto (e, nil)) | (e, l \mapsto ((v,m), nil)) = (e, l \mapsto ((v,m), nil))$, on peut alors considérer que $(e, l \mapsto (e, nil))$ assure que l'emplacement l existe mais indique que l'on ne peut pas ni modifier ni même lire sa valeur.

Comme précédemment, on peut donc utiliser les formules d'assertions sur les arbres pour raisonner sur les permissions.

4.2.3 Assertions pour représenter les arbres

Le problème de la représentation d'arbres avec les permissions, dans [17], consiste à trouver des assertions capables de définir qu'un tas donné représente un arbre. On doit également déterminer une assertion vérifiée par un arbre auquel on attache la permission z. La tentative suivante, utilisant une logique de séparation standard [62], avait été proposée et permet de mieux comprendre le problème de spécification qui était posé :

$$\begin{array}{lcl} ztree\ z\ nil\ Empty & \hat{=} & emp \\ ztree\ z\ t\ (Tip\ \alpha) & \hat{=} & t \overset{z}{\mapsto} (0, \alpha, 0) \\ ztree\ z\ t\ (Node\ \gamma\ \rho) & \hat{=} & \exists l, r. t \overset{z}{\mapsto} (1, l, r) * ztree\ z\ l\ \gamma * ztree\ z\ r\ \rho \end{array}$$

Dans la définition ci-dessus, $ztree\ z\ nil\ Empty$ correspond à l'assertion représentant l'arbre vide. Une feuille située en t contenant la valeur α correspond à l'assertion $ztree\ z\ t\ (Tip\ \alpha)$. Enfin, un nœud t menant à deux sous-arbres γ et ρ correspond lui à $ztree\ z\ t\ (Node\ \gamma\ \rho)$.

On utilise alors la représentation classique des arbres avec les pointeurs. L'arbre vide correspond au tas de pointeur vide. La feuille contenant α en t correspond à la cellule $t \overset{z}{\mapsto} (0, \alpha, 0)$ et le nœud t menant à deux sous-arbres γ et ρ correspond à la cellule $t \overset{z}{\mapsto} (1, l, r)$. Ainsi, les emplacements contiennent des triplets de valeurs qui ont la signification suivante :

- La première valeur peut être soit 0 pour indiquer que le contenu de la cellule est une feuille, soit 1 pour indiquer qu'il s'agit d'un nœud.
- La deuxième valeur correspond au contenu de la feuille si la première valeur est 0, elle correspond à l'emplacement où se trouve le fils gauche sinon.
- La troisième valeur n'a pas de signification si on est dans une feuille et vaut alors 0. Si on se trouve dans un nœud, elle correspond à l'emplacement du fils droit.

Le problème avec cette spécification naturelle est donc qu'elle n'assure pas que les sous arbres soient disjoints. En effet, si la permission z est telle que $z \bullet z$ est définie, la formule $ztree\ z\ t\ (Node\ \gamma\ \gamma)$ peut définir une structure qui est un graphe et non un arbre, puisque les fils gauches et droit de t peuvent pointer sur la même cellule. Concrètement, on peut avoir : $t \overset{z}{\mapsto} (1, u, u) * ztree\ z\ u\ \gamma * ztree\ z\ u\ \gamma$.

Pour garantir que la structure proposée est bien un arbre, il faut s'assurer que le fils gauche et le fils droit d'un nœud sont totalement disjoints. On doit donc s'assurer qu'aucun emplacement n'est commun aux deux sous arbres.

L'expression de telles propriétés a déjà été étudiée pour BI-Loc. On peut donc exprimer un arbre avec permissions dans BI-Loc. Et les assertions correspondantes sont les suivantes :

$$
\begin{array}{lcl}
ztree'\ z\ nil\ Empty & \hat{=} & e \\
ztree'\ z\ t\ (Tip\ \alpha) & \hat{=} & [t]((0, \alpha, 0), z) \\
ztree'\ z\ t\ (Node\ \gamma\rho) & \hat{=} & \exists_{loc} l, r.([t]((1, l, r), z) \wedge no(l) \wedge no(r)) * \\
 & & \quad disjoint((ztree\ z\ l\ \gamma, ztree\ z\ r\ \rho))
\end{array}
$$

où pour un emplacement l donné, $no(l) \equiv ([l]\top * \top) \rightarrow \bot$
et pour deux assertions P et Q, $disjoint(P, Q) \equiv (P * Q) \wedge \forall_{loc} l.((P * [l]e) \wedge (Q * [l]e) \rightarrow \bot)$.

La formule $no(l)$ indique que l'emplacement l n'est pas présent dans l'arbre qui satisfait la formule. Elle est utilisée ici pour éviter les liens cycliques dans notre arbre : un noeud ne peut être son propre fils. La formule $disjoint(P, Q)$ est satisfaite s'il existe un sous-arbre satisfaisant P et un autre satisfaisant Q et s'ils ne partagent aucun emplacement. Elle est donc utilisée ici pour indiquer que les deux sous-arbres sont bien disjoints et empêcher qu'un nœud se retrouve avec deux pères.

L'utilisation du modèle d'arbres partiels (arbres de ressources) pour raisonner permet donc d'établir une base commune pour raisonner sur le modèle des pointeurs tel que proposé dans [62] et sur le modèle des permissions qui est un raffinement de celui des pointeurs, puisque BI-Loc est utilisé dans les deux cas pour la spécification. Il est également intéressant de remarquer que le fait que la composition de deux arbres ne nécessite pas qu'ils soient disjoints permets de mieux appréhender des problèmes comme celui de la représentation des arbres dans le modèle des permissions.

Chapitre 5

Modèles d'arbres et données semi-structurées

L'un des enjeux du format XML est d'assurer l'interopérabilité entre différents programmes en manipulant les données reçues et envoyées sous forme de documents XML [41]. Un point crucial de l'interopérabilité est donc de pouvoir s'assurer du format d'un document dont on connaît la spécification initiale et subissant une transformation donnée. La représentation de documents XML sous forme d'arbres est naturelle [1] et l'utilisation de logiques reposant sur des modèles d'arbres a mené à différents travaux visant à représenter partiellement les document XML sous forme d'arbres et à utiliser la logique de ces modèles pour spécifier ces documents [23, 24, 26, 39].

Dans ce chapitre, on illustre les possibilités d'instanciation du modèle d'arbres partiel (arbres de ressources) en l'appliquant à la représentation des données XML. L'objectif est tout d'abord de représenter de manière complète non seulement la structure de l'arbre mais également son contenu. On propose pour cela un modèle qui permet de représenter les éléments ainsi que leur hiérarchie mais également la totalité des attributs de ses éléments et les zones de données. On définit ensuite une logique fondée sur BI-Loc permettant de raisonner plus spécifiquement sur ces arbres. Cette logique est notamment utilisable pour établir la définition du type d'un document (DTD), à l'instar de ce qui est proposé dans [85, 86] pour XML schéma. Une autre application potentielle non traitée ici, dans l'esprit des travaux de [38, 27], est d'utiliser la logique pour établir des requêtes sur un document XML

Ensuite, on établit un langage de manipulation et des triplets *à la Hoare* spécifiques à la manipulation de données XML, en suivant la démarche du chapitre 2. Cette approche logique constitue le point de départ à une vérification formelle des transformations de documents XML utilisant une logique fondée sur BI-Loc.

5.1 Arbres de ressources et documents XML

Pour pouvoir raisonner sur les documents XML on établit d'abord une relation entre les documents XML et les arbres de ressources. Pour cela, on commence par présenter le type de document XML que l'on cherche à manipuler et la terminologie que l'on utilise pour décrire ces documents, ainsi que les contraintes syntaxiques que doit respecter un document XML. On définit ensuite l'ensemble des arbres de ressources que l'on souhaite utiliser pour représenter ces documents, en précisant comment se fait la correspondance entre documents et arbre de ressources et comment le modèle d'arbres permet de représenter les contraintes syntaxiques des documents.

5.1.1 Document XML et terminologie

On détaille ici la construction d'un document XML du type de ceux que l'on souhaite manipuler, c'est à dire sans se soucier de l'ordre de nœuds frères et sans instruction exécutable. Un exemple est donné en figure 5.1. On fera référence à cet exemple pour détailler la structure du document et la terminologie associée.

```
<?xml version="1.0"?>
<!DOCTYPE map SYSTEM "map.dtd">
<map>
  <state id='s2'>
    <scode>NE</scode>
    <sname>Nevada</sname>
    <capital idref='c3' />
  </state>
  <city id='c3'>
    <ccode>CCN</ccode>
    <cname>Carson City</cname>
    <state_of idref='s2' />
  </city>
</map>
```

Fig. 5.1 – Exemple de fichier XML

Le document XML proposé est composé d'un prologue constitué d'une ligne indiquant qu'il s'agit bien d'un document XML et d'une seconde ligne indiquant la définition du type de document (DTD) utilisée pour définir le document. Cette DTD détaille la grammaire que doit respecter le document et fixe la sémantique de certains de ses éléments. Les propriétés qu'elle impose seront vues en détail dans la section 5.3 et seront associées à des assertions.

Le corps du document est composé d'un ou plusieurs *éléments*. Un *élément* est soit délimité par une *étiquette de début* et une *étiquette de fin* (comme **state, city** ou **scode** par exemple), soit délimité par une *étiquette unique* (comme **capital** ou **state_of**). Les étiquettes de début et les étiquettes uniques contiennent donc le nom de l'élément et éventuellement des couples associant des *valeurs* à des *attributs* (ainsi, l'étiquette *state* contient l'attribut *id* et lui associe la valeur *s2*). Enfin, les éléments délimités par une étiquette de début et une étiquette de fin contiennent d'autres éléments et/ou des *données* (l'élément **scode** contient la donnée **NE**, l'élément **cname** contient la donnée **Carson City**).

Les contraintes syntaxiques sont peu nombreuses (outre le fait de respecter la grammaire de base pour bien construire les éléments). On s'intéressera principalement aux restrictions suivantes : le même attribut ne peut être défini deux fois pour un même élément et une étiquette ne contient qu'un seul nom d'élément.

D'après la spécification XML, les éléments ne doivent pas nécessairement tous posséder un identifiant unique. La présence ou non d'attributs correspondant à des identifiants est définit dans la DTD attachée au document. Pour toute ces raisons, on choisit ici de traiter l'unicité des identifiants à un niveau logique. L'approche choisie ici diffère donc de celle proposée dans des modèles d'arbres comme celui de [24] où la présence d'un identifiant unique pour chaque élément est présentée comme une contrainte syntaxique.

5.1.2 Arbres représentant les documents

Comme nous l'avons vu dans le chapitre 4, la principale difficulté pour définir un modèle d'arbres dans ce contexte réside dans le choix du monoïde de ressources que l'on va utiliser. Une autre difficulté est de décider quelles informations des documents XML vont correspondre à des ressources de l'arbre et quelles autres informations vont correspondre à des nœuds de l'arbre.

La structure d'un arbre correspondant à un document va être définie par l'enchaînement des éléments et des données dans le document. On fixe alors la signification des noms d'emplacement des différents nœuds de l'arbre. Comme plusieurs éléments frères peuvent avoir le même nom, on ne peut choisir comme label d'un nœud le nom de l'élément ou de la donnée auquel il correspond. On choisit de donner un nom arbitraire aux emplacements. Les noms d'emplacements ne correspondent donc à aucun élément concret du document initial, ils peuvent être vus comme un moyen d'identifier l'emplacement *physique* d'un nœud donné.

On définit ensuite ce qui sera représenté comme des ressources de l'arbre. L'idée est donc que tous les types de contenu (nom d'éléments, attributs et données) correspondent à des ressources. On définit ainsi l'ensemble des élément $Elem$, l'ensemble des attributs $Attr$, celui des données Dat et celui des valeurs des attributs Val[5]. On définit donc le monoïde partiel permettant de manipuler ces ressources et permettant de représenter au mieux les contraintes syntaxiques imposées par un document XML.

On sait qu'un nœud va contenir soit une donnée (une ressource de Dat), soit un élément (une ressource de $Elem$) et des couples de valeurs et d'attributs (une ressource de $Attr \rightharpoonup_{fin} Val$). Par la suite, on souhaite pouvoir facilement rajouter des attributs dans un emplacement donné. Pour cela, on autorise également un emplacement à contenir uniquement une ressource appartenant à $Attr \rightharpoonup_{fin} Val$. L'ensemble des ressources M_{XML} que l'on utilise comme ensemble de base pour construire le monoïde de ressources partiel est donc le suivant :

$$M_{XML} = Elem \cup Attr \rightharpoonup_{fin} Val \cup (Elem \times Attr \rightharpoonup_{fin} Val) \cup Dat \cup \{e\}$$

où e est un élément neutre que l'on ajoute afin d'obtenir une structure monoïdale. On définit alors l'opérateur partiel de composition de ressources $\bullet$ comme suit :

- $m \bullet e = e \bullet m = m$ pour toute ressource $m \in M_{XML}$;
- Soit $m, m' \in Dat$, $[m \bullet m']\uparrow$ et $m \bullet m' \in Dat$;
- Soit $v \in Elem$ et $f \in Attr \rightharpoonup_{fin} Val$, $[v \bullet f]\uparrow$ et $v \bullet f = (v, f)$;
- Soit $f, f' \in Attr \rightharpoonup_{fin} Val$, $[f \bullet f']\uparrow$ ssi $dom(f) \cap dom(f') = \emptyset$ et $f \bullet f' = f \cup f'$.

Par extension, on a également pour tout $v \in Elem$, et pour tout $f, f' \in Attr \rightharpoonup_{fin} Val$, $[(v, f) \bullet f']\uparrow$ ssi $[f \bullet f']\uparrow$ et dans ce cas $(v, f) \bullet f' = (v, f \cap f')$.

Les choix qui ont été faits pour la définition de la composition permettent d'empêcher qu'un attribut ne soit défini deux fois dans un même nœud (et donc pour un unique élément). On utilise alors pour nos arbres le monoïde $\mathcal{M}_{XML} = \langle M_{XML}, e, \bullet, \sqsubseteq \rangle$ où $\sqsubseteq$ est réflexive ($m \sqsubseteq m'$ si et seulement si $m = m'$) et totalement indéfini ailleurs.

On utilise également le fait que la composition des arbres soit partielle pour imposer deux spécificités aux arbres que l'on manipule :

[5] Ces ensembles ne sont pas nécessairement disjoints.

1. Toutes les données présentes dans un élément donné doivent être regroupées dans un unique emplacement. Plus formellement, si l'on a $l \neq l'$ et si $m, m' \in Dat$, $(e, l \mapsto m)|(e, l' \mapsto m')$ n'est pas défini.

2. Un nom d'emplacement est unique dans l'arbre, c'est à dire qu'un nom d'emplacement ne peut désigner qu'un seul nœud dans tout l'arbre. Le nom d'un emplacement permet donc de retrouver sans ambiguïté le chemin qui mène à cet emplacement. Cette restriction s'exprime en indiquant que pour deux chemins L et L' tels que $L \neq L'$, $(e, (L : l) \mapsto m)|(e, (L : l) \mapsto m')$ n'est pas défini.

Comme indiqué précédemment, le seul écart que l'on se permet vis à vis des contraintes syntaxiques des documents XML est d'autoriser un nœud à contenir uniquement des couples attribut/valeur. La composition d'un tel nœud avec un nœud contenant un élément permettra de rajouter facilement du contenu dans un arbre donné, ce qui est très utile dans le contexte de la manipulation de données semi-structurées, comme nous le verrons plus loin dans ce chapitre.

On note alors T_{XML} le monoïde partiel où les arbres sont construits à partir du monoïde $\mathcal{M}_{XML}$ et où la composition respecte les deux contraintes ci-dessus. Ainsi, le document XML présenté dans la figure 5.1 correspond à l'arbre correspondant donné en figure 5.2.

```
<?xml version="1.0" ?>
<!DOCTYPE map SYSTEM "map.dtd">
<map>
  <state id='s2'>
    <scode>NE</scode>
    <sname>Nevada</sname>
    <capital idref='c3' />
  </state>
  <city id='c3'>
    <ccode>CCN</ccode>
    <cname>Carson City</cname>
    <state_of idref='s2' />
  </city>
</map>
```

$\leadsto$

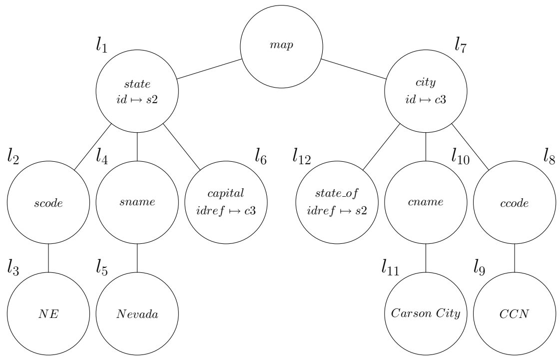


FIG. 5.2 – Document XML et arbre de ressource correspondant

On peut également spécifier un sous ensemble d'arbres tel que tout arbre lui appartenant correspond à au moins[6] un document XML.

Lemme 5.1. Un arbre partiel de ressources de T_{XML} correspond à un document XML si tous ses emplacements contiennent au moins un élément ou des données.

Preuve. Le résultat est immédiat et découle de la construction même de T_{XML}. □

5.1.3 Comparaisons avec d'autres représentations

L'utilisation de modèles logiques pour représenter les données semi-structurées qui est proposée ici présente des similitudes avec la démarche proposée par Calcagno et al. [24], ces travaux

[6] Un même arbre peut correspondre à plusieurs documents puisque l'ordre des éléments dans le document n'est pas pris en compte.

étant eux même une extension de ceux proposés initialement sur les arbres avec pointeurs [26]. Une différence majeure entre la représentation qui est définie dans cette thèse et ces travaux est que les arbres de ressources permettent de représenter toutes les informations contenues dans un document XML, alors que les arbres de [23, 24] se bornent à la représentation des éléments, et de deux types d'attributs particuliers : les identifiants et les pointeurs[7]. Cette différence est cruciale dans le cadre de la modification de documents XML qui nous intéresse ici, puisqu'elle permet une manipulation plus fine des données.

Une autre différence déjà mentionnée concerne la représentation des identifiants dans les documents XML. Dans le modèle présenté dans cette thèse, l'existence d'identifiants et la vérification de leur unicité dans un document donné est faite au niveau logique. Dans les travaux de [24], la présence d'identifiants uniques dans chaque nœud est une contrainte syntaxique. La gestion des identifiants au niveau logique que l'on a choisi présente une plus grande souplesse puisqu'elle ne nécessite pas que tous les éléments présents dans un document XML aient un identifiant, ce qui correspond à la spécification standard des documents XML.

5.2 Une logique d'assertions pour les arbres XML

On souhaite adapter le modèle d'arbre partiel pour pouvoir notamment proposer des quantificateurs sur les différents domaines sémantiques. Pour cela, on construit un modèle manipulant des couples constitués d'une pile de variables et d'un arbre de T_{XML}. Soit un ensemble de noms de variables Var, une pile est alors un élément de $Var \rightharpoonup_{fin} (M_{XML} \cup T_{XML})$.

On étend alors la logique BI-Loc présentée au chapitre 2 avec des quantificateurs sur les différents modèles sémantiques. La syntaxe de la logique pour les données XML est alors la suivante :

Définition 5.1 (Logique pour les données XML).

P	$::=$	$[l]P$	*modalité d'emplacement*
		$P \mathrel{-\!\!*} P \mid P * P \mid I$	*opérateurs et unité multiplicatifs.*
		$P \rightarrow P \mid P \wedge P \mid P \vee P \mid \top \mid \bot$	*opérateurs et unités additifs.*
		$\exists_{res} x.P \mid \forall_{res} x.P$	*quantificateurs sur les ressources.*
		$\exists_{loc} x.P \mid \forall_{loc} x.P$	*quantificateurs sur les emplacements.*
		$\exists_{path} x.P \mid \forall_{path} x.P$	*quantificateurs sur les chemins.*
		$\exists_{tree} x.P \mid \forall_{tree} x.P$	*quantificateurs sur les arbres.*
		E	*expressions.*

où une expression E est soit un arbre de ressources (pouvant être composé de variables) soit une proposition. On utilise donc une fonction d'interprétation $[\![\cdot]\!]_s$ qui associe à chaque expression E, qui n'est pas une proposition, l'arbre de ressources correspondant. Cette fonction s'étend naturellement à partir de la définition de la pile s. La sémantique des formules logiques est donnée par le modèle qui est une extension du modèle d'arbres partiel général qui est proposé au chapitre 2 :

Définition 5.2 (Modèle d'arbres partiel pour données XML). *Un* modèle d'arbres partiel pour données XML *est le monoïde d'arbres de données* XML $(T_{XML}, (e, nil), |, \sqsubseteq_T)_{\mathcal{M},Loc}$ *muni d'une relation de forcing* $\models_{\mathfrak{P}} \subseteq M_{XML} \times \Sigma$ *satisfaisant la condition suivante :*

[7] Ces deux types d'attributs sont détaillés dans la section 5.3.

$$\forall E \text{ si } [\![E]\!]_s \text{ est une proposition } \forall t, t' \in T_{XML}.(t \models_{\mathfrak{T}} p \text{ et } t' \sqsubseteq_T t \Rightarrow t' \models_{\mathfrak{T}} [\![E]\!]_s)$$
$$\text{et}$$
$$\text{si } [\![E]\!]_s \text{ est un arbre } s, t \models_{\mathfrak{P}} E \text{ si } t = [\![E]\!]_s$$

et s'étendant aux formules de la logique comme suit :

- $s, t \models_{\mathfrak{T}} P * Q$ *ssi* $\exists t', t''.[t \times t']\uparrow$, $t \sqsubseteq_T t'|t''$, $s, t' \models_{\mathfrak{T}} P$ *et* $s, t'' \models_{\mathfrak{T}} Q$;
- $s, t \models_{\mathfrak{T}} I$ *ssi* $t \sqsubseteq_T (e, nil)$;
- $s, t \models_{\mathfrak{T}} P \twoheadrightarrow Q$ *ssi* $\forall t' \in M$, $s, t' \models_{\mathfrak{T}} P$ *et* $[t \times t']\uparrow \Rightarrow s, t|t' \models_{\mathfrak{T}} Q$;
- $s, t \models_{\mathfrak{T}} P \wedge Q$ *ssi* $s, t \models_{\mathfrak{T}} P$ *et* $s, t \models_{\mathfrak{T}} Q$;
- $s, t \models_{\mathfrak{T}} \top$ *toujours ;*
- $s, t \models_{\mathfrak{T}} P \vee Q$ *ssi* $s, t \models_{\mathfrak{T}} P$ *ou* $s, t \models_{\mathfrak{T}} Q$;
- $s, t \models_{\mathfrak{T}} \bot$ *jamais ;*
- $s, t \models_{\mathfrak{T}} P \rightarrow Q$ *ssi* $\forall t'.t' \sqsubseteq_T t$ *et* $s, t' \models_{\mathfrak{T}} P \Rightarrow s, t \models_{\mathfrak{T}} Q$;
- $s, t \models_{\mathfrak{T}} [l]P$ *ssi* $t \sqsubseteq (m, l \mapsto t')$ *avec* $s, t' \models_{\mathfrak{T}} P$;
- $s, t \models_{\mathfrak{T}} \exists_{loc} x.P$ *ssi il existe* $l \in Loc$ *tel que* $s, t \models_{\mathfrak{T}} P\{^l/_x\}$;
- $s, t \models_{\mathfrak{T}} \forall_{loc} x.P$ *ssi pour tout* $l \in Loc$, *on ait* $s, t \models_{\mathfrak{T}} P\{^l/_x\}$;
- $s, t \models_{\mathfrak{T}} \exists_{path} X.P$ *ssi il existe* $L \in Loc^*$ *tel que* $s, t \models_{\mathfrak{T}} P\{^L/_X\}$;
- $s, t \models_{\mathfrak{T}} \forall_{path} X.P$ *ssi pour tout* $L \in Loc^*$, *on ait* $s, t \models_{\mathfrak{T}} P\{^L/_X\}$;
- $s, t \models \forall_{res} x.P$ *ssi pour tout* $m \in M_{XML}$, $[s|x \mapsto m], t \models P$;
- $s, t \models \exists_{res} x.P$ *ssi il existe* $m \in M_{XML}$, $[s|x \mapsto m], t \models P$;
- $s, t \models \forall_{tree} x.P$ *ssi pour tout* $t' \in T_{XML}$, $[s|x \mapsto t'], t \models P$;
- $s, t \models \exists_{tree} x.P$ *ssi il existe* $t' \in T_{XML}$, $[s|x \mapsto t'], t \models P$.

5.2.1 Exemple de spécifications

Avant de continuer, on détaille quelques formules, utilisées dans la suite de ce chapitre, qui correspondent à des spécifications.

- Une partie de l'arbre vérifie une formule

Pour assurer qu'une partie d'un arbre située au bout d'un chemin L vérifie une formule P, on définit la formule suivante :

$$contains(L, P) = \top * [L]P$$

qui indique qu'on peut décomposer l'arbre en deux parties : un sous-arbre situé au bout d'un chemin L qui vérifie la formule P et le reste de l'arbre.

- Existence d'un chemin dans un arbre

Pour s'assurer que le chemin L est défini dans un arbre, il suffit de s'assurer qu'il y a un sous-arbre vérifiant $\top$ au bout du chemin L, ce qui s'exprime par la formule suivante :

$$exists(L) = contains(L, \top)$$

- Non existence d'un chemin

Pour montrer qu'un arbre ne contient pas le chemin L, il s'agit simplement d'indiquer qu'il est impossible pour lui de vérifier qu'il existe, , ce qui correspond à la formule :

$$no(L) = exists(L) \rightarrow \bot$$

5.2.2 Spécifier l'ajout d'informations

Le but de ce chapitre étant de mettre en place un langage d'assertions pour la modification de données XML, il est utile de pouvoir spécifier qu'une propriété sera vérifiée si on ajoute certaines informations à un contexte initial.

L'une des spécificités de BI-Loc par rapport aux autres logiques d'arbres est que la séparation n'a pas lieu au niveau des noeuds mais au niveau des ressources des arbres. Grâce à cela, il est très simple d'indiquer que l'on ajoute de l'information à un emplacement donné. Ainsi, l'ajout d'informations vérifiant P à l'emplacement L pour prouver la propriété Q s'exprime par la formule :

$$[L]P \mathrel{-\!\!*} Q$$

Cette formule ne vérifie pas du tout si l'emplacement L existe ou non avant l'ajout. Il est toutefois facile de rajouter ce test. Si l'on veut s'assurer que l'emplacement L existe pour y ajouter des informations, on considère la formule :

$$exists(L) \wedge ([L]P \mathrel{-\!\!*} Q)$$

L'utilisation de $\wedge$ permet d'assurer à la fois que l'ajout aboutira à vérifier Q et que L existe initialement. De manière analogue, si l'on souhaite que l'emplacement n'existe pas avant d'y ajouter des informations, on considère la formule :

$$no(L) \wedge ([L]P \mathrel{-\!\!*} Q)$$

5.2.3 Mise à jour d'un arbre

La mise à jour d'un arbre consiste à remplacer tout ou parti du contenu d'un nœud. Le problème principal rencontré habituellement dans les logiques d'arbres pour spécifier ce type d'opération est de définir précisément le nœud à modifier [24], problème que l'on résout de la façon suivante.

Supposons qu'en remplaçant un sous-arbre vérifiant P par un sous-arbre vérifiant P' dans un chemin L, on obtient la propriété Q. Ce comportement se traduit dans la logique sur les données XML de la façon suivante :

$$[L]P * ([L]P' \mathrel{-\!\!*} Q)$$

Si l'on souhaite que tout un sous-arbre soit remplacé, car on doit s'assurer que l'on met bien de côté tout le sous-arbre concerné avant de faire la mise à jour, ce qui se traduit par la formule :

$$[L]\top * (([L]P' \mathrel{-\!\!*} Q) \wedge no(L))$$

Cette formule peut se lire comme suit : on peut décomposer l'arbre en deux parties, une vérifiant $[l]\top$ et le reste de l'arbre. Si on ajoute à ce reste un arbre vérifiant $[L]P'$, alors on obtient Q.

Ce problème de mise à jour de l'arbre illustre un des gros avantages de la gestion particulière de la composition dans les arbres de ressources. En effet, dans les modèles d'arbres fondés sur les ambients [23, 29], la composition d'arbres ne permet pas de modifier un nœud existant. C'est d'ailleurs ce problème qui a mené à la proposition de Calcagno et al. sur la logique d'arbres avec contextes [24], le but étant de permettre l'ajout d'informations dans des nœuds et non uniquement à la racine de l'arbre.

5.3 Définition logique d'un document

On a vu précédemment que, dans un document XML, l'enchaînement logique des différents éléments est fixé par une définition type du document (DTD). Notre but est d'exprimer sous forme logique les règles d'une DTD. Ainsi, d'après la correspondance entre arbres de ressources et documents XML exprimée dans la section 5.1.2, on peut vérifier si un document vérifie une DTD en vérifiant si l'arbre correspondant vérifie une formule donnée.

On trouve deux types de règles distincts dans une DTD. Les premières concernent les éléments, et indiquent comment ils se succèdent les uns aux autres. Les secondes concernent les attributs, indiquent dans quels éléments on peut les trouver, s'ils doivent avoir une valeur particulière et s'ils ont une sémantique particulière dans le document (comme les identifiants, par exemple, dont le rôle est de donner un moyen unique d'identifier un nœud).

On explique donc ici comment on définit les principales contraintes sur les éléments puis sur les attributs, et on s'intéresse ensuite aux cas particuliers des attributs ayant un rôle d'identifiant ou de pointeur, rôle que l'on détaille plus loin.

On utilisera dans cette partie les prédicats suivants pour tester le type des ressources contenues dans les nœuds de l'arbre :

$s, m \models_{\mathfrak{P}} cont$ ssi $m \in Dat$.
$s, m \models_{\mathfrak{P}} elem$ ssi $m \in Elem$.
$s, m \models_{\mathfrak{P}} elem(name)$ ssi $name \in Elem$ et $m = name$.
$s, m \models_{\mathfrak{P}} att(name)$ ssi $\exists f \in (Att \rightarrow_{fin} Val)$ tel que $m = f$ et $dom(f) = \{name\}$.
$s, m \models_{\mathfrak{P}} value(val)$ ssi $\exists f \in (Att \rightarrow_{fin} Val)$ tel que $m = f$, $\exists x.dom(f) = \{x\}$ et $f(x) = val$.

où les ensembles $Dat, Elem, Att$ et Val sont les ensembles de ressources définies dans la section 5.1.2.

5.3.1 Enchaînement des éléments

Pour définir un document XML, on doit donc indiquer pour chaque élément les fils possibles de cet élément. Un élément peut ne pas avoir de fils, avoir un fils unique d'un type d'élément précis qui peut être optionnel (au maximum un fils) ou répété (au moins un fils) ou avoir une séquence précise de fils. La solution peut enfin être un mélange des différents cas énumérés ci-dessus.

- Élément vide

Un nœud correspondant à un élément vide ne doit pas avoir de fils. Ainsi, pour s'assurer que les éléments « *seul* » d'un arbre n'ont pas de fils, on considère la formule :

$$\forall_{path} x.(contains(x, elem(seul)) \rightarrow \forall_{loc} z.no(x : z))$$

Cette formule indique donc que pour tout chemin x, si x contient un élément nommé *seul* alors il n'existe pas d'emplacement défini sous x, ce qui est traduit par $\forall_{loc} z.no(x : z)$).

- Élément avec un unique fils

Pour indiquer que les éléments de type « *personne* » doivent avoir un fils unique nommé « *nom* », on considère la formule :

$$\forall_{path} x.(contains(x, elem(personne)) \rightarrow ([x]\exists y.[y](elem(nom) * \top) * \forall_{loc} z.no(x:z)))$$

Chaque chemin menant à un élément nommé «$personne$» ($\forall_{path} x.contains(x, elem(personne))$) a donc un fils contenant un élément de type «nom» ($[x]\exists y.[y](elem(nom) * \top)$) et qu'aucun autre emplacement existe sous le chemin x ($\forall_{loc} z.no(x:z)$).

On suit un raisonnement similaire pour indiquer qu'un élément «nom» ne contient que des données :

$$\forall_{path} x.((\top * [x]elem(nom)) \rightarrow ([x]\exists y.[y]cont * \forall_{loc} z.no(x:z)))$$

- Élément avec au moins un fils

Pour s'assurer qu'un élément a au moins un fils d'un certain type, on considère l'exemple d'un élément nommé $fils$ qui apparaît au moins une fois sous un élément $parent$. La formule correspondante est alors la suivante :

$$\begin{aligned}\forall_{path} x.(contains(x, elem(parent)) \rightarrow \quad & ([x]\exists_{loc} y.[y](elem(fils) * \top) * \\ & \forall_{loc} z.(exists(z) \rightarrow [z]elem(fils) * \top)))\end{aligned}$$

Ici, pour chaque chemin menant à un élément «$parent$» ($\forall_{path} x.contains(x, elem(parent))$), il existe un emplacement sous x contenant un élément «$fils$» ($[x]\exists_{loc} y.[y](elem(fils) * \top)$) et tout autre emplacement défini sous le chemin x contient lui aussi un élément «$fils$» ($[x]\forall_{loc} z.(exists(z) \rightarrow [z]elem(fils) * \top)$).

- Élément avec au plus un fils

On veut spécifier qu'un élément possède au plus une occurrence d'un type de fils particulier, par exemple, on détermine que des éléments du type «$parent$» peuvent avoir comme fils une ou zéro occurrence d'un élément «$option$» par la formule :

$$\forall_{path} x.(contains(x, elem(parent)) \rightarrow [x]((\exists y.[y](elem(option) * \top)) \vee I) * \forall_{loc} z.no(x:z))$$

Si l'on trouve un élément «$parent$» au bout d'un chemin ($\forall_{path} x.contains(x, elem(parent))$), le sous-arbre de ce chemin est soit vide, soit composé d'un fils contenant un élément «$option$» ($[x](\exists_{loc} y.[y](elem(option) * \top)) \vee I)$) et aucun autre emplacement n'est défini sous x ($\forall_{loc} z.no(x:z)$).

- Composition complexe d'éléments

Nous montrons maintenant comment on peut associer les schémas logiques donnés ci-dessus pour spécifier des contenus complexes pour un élément donné. On suppose qu'un élément nommé $message$ doit avoir comme fils un élément $from$, un élément to, au minimum un élément $content$, un nombre indéfini (éventuellement zéro) d'élément $body$ et un élément optionnel $date$.

Pour exprimer ces conditions, on s'inspire des conditions exprimées ci-dessus.
On a alors la formule suivante :

$$\begin{aligned}&\forall_{path} x.(contains(x, elem(message)) \rightarrow \\ &\quad ([x](\exists_{loc} y_1, y_2, y_3, y_4.[y_1](elem(from) * \top) * \\ &\quad\ [y_2](elem(to) * \top) * [y_3](elem(content) * \top) * \\ &\quad\ [y_4](elem(date) * \top) \vee I) * \\ &\quad\ \forall_{loc} z.exists(x:z) \rightarrow ([x][z](elem(content) \vee elem(body)) * \top)))\end{aligned}$$

Cette formule indique que si on a un chemin x menant à un élément *message*, alors on a sous x :

- un élément *from* dans un emplacement y_1 ;
- un élément *to* dans un emplacement y_2 ;
- un élément *content* dans un emplacement y_3 ;
- éventuellement un élément *body* dans un emplacement y_4, le fait que sa présence soit éventuelle se traduit par l'utilisation de l'opérateur $\vee$;
- tout autre emplacement défini doit contenir soit un élément *body* soit un élément *content*.

5.3.2 Gestion des attributs

Le second type de déclaration dans une DTD concerne les attributs possibles pour un élément. Pour chaque élément, on définit le nom des attributs que l'on peut ou doit définir pour cet élément, si ces attributs sont requis ou optionnels et on indique éventuellement la valeur qu'ils doivent avoir. De plus, certains attributs peuvent avoir un rôle particulier. On s'intéresse ici aux identifiants *(id)* et aux pointeurs *(idref)*.

- Attributs requis

On spécifie qu'un attribut est requis pour un type d'élément particulier. On donne pour cela la spécification permettant d'indiquer que l'élément «*city*» doit contenir un attribut «*id*» :

$$\forall_{path} x.(contains(x, elem(city)) \rightarrow contains(x, attr(id))$$

qui exprime que, si un chemin x contient un élément *city*, le même chemin contient l'attribut *id*.

- Attribut optionnel

Pour indiquer qu'un attribut est optionnel pour un élément donné, il suffit de dire que l'on peut soit rencontrer l'attribut, soit ne rien rencontrer. Ainsi, si pour une ville l'attribut «*capital*» est optionnel, on obtient :

$$\forall_{path} x.(contains(x, elem(city)) \rightarrow contains(x, attr(capital) \vee I)$$

L'utilisation de l'opérateur $\vee$ et de la constante I permet d'exprimer la possibilité que la présence de l'attribut est optionnel.

- Valeur d'un attribut

Pour exprimer qu'un attribut *bar* d'un élément *foo* doit toujours avoir la valeur 42, on considère la formule :

$$\forall_{path} x.(contains(x, elem(foo) \wedge attr(bar)) \rightarrow contains(x, (attr(bar) \wedge value(42)))$$

Si un chemin x on a un élément *foo* avec un attribut *bar*, alors on a en x un attribut *bar* qui a également la valeur 42. Comme on ne peut pas avoir deux attributs ayant le même nom, l'attribut *bar* dont il est question dans les sous-formules à gauche et à droite de $\rightarrow$ est bien le même.

On peut également, sur le même principe, énumérer les valeurs possibles pour un attribut. Ainsi, si on veut autoriser les valeurs 42 et 21 pour l'attribut *bar* de l'exemple ci-dessus, on considère la formule :

$$\forall_{path} x.(contains(x, elem(foo) \wedge attr(bar)) \rightarrow contains(x, (attr(bar) \wedge (value(42) \vee value(21))))$$

5.3.3 Identifiants et pointeurs

Parmi les attributs, certains jouent un rôle particulier. C'est le cas des attributs pointeurs et identifiants que l'on utilise ici. Un *identifiant* permet de faire référence de manière unique à un nœud dans un document donné. On ne doit donc pas avoir deux identifiants possédant le même nom. Un *pointeur* est un attribut qui permet de faire référence à un nœud existant en indiquant comme valeur du pointeur la valeur de l'identifiant de ce nœud.

On montre ici comment on peut représenter les contraintes spécifiques à ces types d'attributs sous forme de formules logiques. On se fonde pour cela sur l'exemple initialement présenté en figure 5.2 et que l'on rappelle en figure 5.3 en y incluant sous forme de flèches en pointillé les liens des pointeurs. Dans l'exemple, les nœuds l_1 et l_7 contiennent des identifiants et ils sont référencés respectivement par les pointeurs situés en l_6 et l_{12}. On détaille maintenant comment s'expriment ces contraintes.

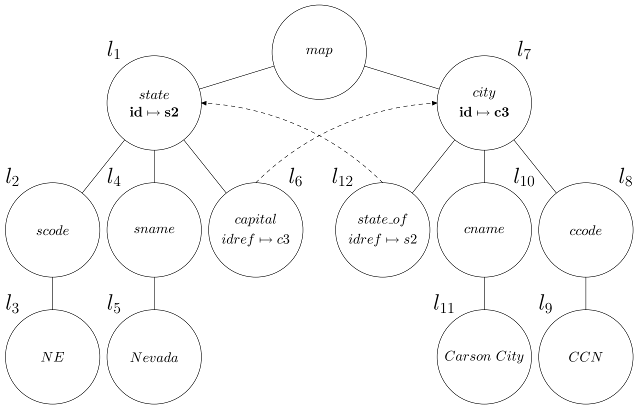


FIG. 5.3 – Représentation des pointeurs dans un arbre XML.

- Identifiants

Pour gérer les identifiants, on doit introduire un prédicat qui indique qu'un attribut est un identifiant pour un élément donné, c'est-à-dire un prédicat id qui est telle que $s, t \models id$ si $t = ((v, f), nil)$, $dom(f) = \{n\}$ et pour l'élément v, l'attribut n est de type identifiant.

Pour vérifier l'unicité des valeurs des identifiants dans un document donné, on doit alors vérifier la formule suivante :

$$\exists_{path} x, y.\exists_{res} v.([x]id \wedge (elem * value(v)) * [y]id \wedge (elem * value(v)) * \top) \rightarrow \bot$$

Cette formule assure en effet que l'on ne peut trouver deux chemins menant à des attributs qui sont des identifiants et qui ont la même valeur.

- Pointeurs

On s'intéresse maintenant à la représentation des spécificités des pointeurs et on exprime de manière logique que la valeur associée à un attribut de type pointeur correspond à une valeur présente dans un attribut de type identifiant.

On illustre ce point sur l'exemple de la figure 5.3. On veut montrer que l'attribut «*idref*» de l'élément «*state_of*» fait référence à un identifiant existant, ce qui correspond à la formule :

$$\forall_{path} x.\forall_{res} v.(contains(x, elem(state\_of) * (attr(idref) \land value(v))) \rightarrow \\ \exists_{path} y.(([y]id \land (elem * value(v))) * \top))$$

qui assure que pour tout attribut «*idref*» ayant comme valeur v et présent au coté d'un élément «*state_of*», on peut trouver au bout d'un chemin y un identifiant ayant comme valeur v.

Sur le point précis de la spécification des pointeurs, l'utilisation de BI-Loc peut même mener à des spécifications plus précises que ce qui peut se faire actuellement dans une DTD. Ainsi, il est possible d'indiquer facilement que la valeur du pointeur doit correspondre à un identifiant d'un type d'élément donné. Toujours sur l'exemple donné ci-dessus, on peut exprimer que l'élément pointé doit être un élément de type «*state*». Pour cela on doit vérifier la formule :

$$\forall_{path} x.\forall_{res} v.(contains(x, elem(state\_of) * (attr(idref) \land value(v))) \rightarrow \\ \exists_{path} y.(([y]id \land (elem(state) * value(v))) * \top))$$

Comparé à la formule précédente, on remarque que l'on précise ici le type de l'élément qui doit être présent dans l'emplacement où se trouve l'identifiant : un élément «*state*».

On peut même demander à ce que les liens entre la capitale et l'état soient réflexifs :

$$\begin{array}{ll} \forall_{path} x.\exists_{loc} y.\forall_{res} v, u.(contains & (x, elem(state) * \\ & \quad (attr(id) \land value(u)) * \\ & \quad [y](elem(capital) * (attr(idref) \land value(v)))) \rightarrow \\ & \qquad \exists_{path} z.\exists_{loc} t.contains(z, elem(city) * \\ & \qquad (attr(id) \land value(v)) * \\ & \qquad [y](elem(state\_of) * (attr(idref) \land value(u))))) \end{array}$$

La formule ci-dessus indique que pour chaque «*state*» avec un identifiant de valeur u et qui a un fils «*capital*» possédant une valeur de pointeur v, il existe un élément «*city*» avec un identifiant ayant comme valeur v et un attribut «*idref*» de type pointeur ayant comme valeur u.

5.3.4 Exemples de spécification de propriétés

On a montré dans la section 5.3.3 comment gérer les pointeurs dans l'exemple de la figure 5.3. On complète ici l'étude de la spécification du document de cet exemple avec des formules qui décrivent d'autres propriétés relatives à ce document.

On commence par exprimer le fait que chaque élément «*state*» doit être composé d'une séquence d'éléments «*scode*», «*sname*» et «*capital*» (et d'aucun autre élément). Ceci correspond à la formule suivante :

$$\begin{aligned}\forall_{path} x.(contains(x, elem(state)) \rightarrow \quad & [x]\exists_{loc} y_1, y_2, y_3.[y_1](elem(scode) * \top) * \\ & [y_2](elem(sname) * \top) * \\ & [y_3](elem(capital) * \top) * \\ & \forall_{loc} z.no(x : z))\end{aligned}$$

Si l'on souhaite de plus que chaque élément *city* possède un attribut optionnel *capital* qui peut prendre les valeurs *yes* ou *no*, on considère la formule :

$$\forall_{path} x.(contains(x, elem(city)) \rightarrow ([x]((attr(capital) \wedge (value(yes) \vee value(false))) \vee I) * \top))$$

Enfin, pour exprimer qu'un élément «*map*» ne doit contenir que des éléments «*state*» et des éléments «*city*», sans préciser leur nombre, on a la formule suivante :

$$\forall_{path} x.(contains(x, elem(map)) \rightarrow (\forall_{loc} z.(contains(x : z, (elem(state)) \vee elem(city)) \vee no(x : z))))$$

5.3.5 Expressivité et comparaisons avec les autres logiques

La possibilité d'utiliser la logique pour raisonner sur la totalité du contenu des nœuds est une nouveauté si l'on compare les travaux réalisés aux autres logiques d'arbres utilisées pour raisonner sur les données semi-structurées [23, 24]. Cette spécificité est une conséquence directe de la représentation fine du contenu des documents XML dans les arbres de ressources et de la présence de propositions dans la logique pour raisonner sur le type des ressources présentes dans l'arbre. La possibilité de représenter et de distinguer les données, les éléments et la totalité des attributs en est l'illustration. Dans les autres modèles d'arbre cités ici, les seules informations que l'on possède sur les nœuds sont attachés aux labels et correspondent donc à un format fixe. Cette spécificité rend notamment plus délicate la distinction de ces différents types d'informations.

De plus, la logique présentée ici contient des quantifications sur les emplacements et les chemins. Le rôle de ces quantifications peut être rapproché de celui des modalités spatiales de la logique des ambients [29]. Dans notre modèle, on peut capturer grâce aux quantificateurs l'emplacement ou le chemin qui vérifie une sous-formule et réutiliser cette information ailleurs dans la formule.

5.4 Un langage de manipulation des données XML

On souhaite permettre la transformation sûre de documents XML vérifiant une spécification donnée S_1 en des documents vérifiant une spécification S_2 en utilisant notre modèle fondé sur les arbres de ressources. On a vu jusqu'ici comment ces arbres pouvaient être utilisés pour représenter les documents XML et comment on peut spécifier ces documents grâce à la logique sur les arbres XML.

On propose maintenant un langage de commandes en vue de manipuler les arbres et donc les données XML qu'ils représentent. On définit également une sémantique sous forme de triplets de Hoare pour ces commandes, ce qui nous permet d'obtenir le lien souhaité entre deux spécifications.

Comme le modèle proposé pour représenter les données XML est une adaptation du modèle d'arbres partiel, le langage de manipulation proposé est une adaptation du langage de manipulation général présenté dans le chapitre 2. Les adaptations proposées ont pour but de manipuler de manière plus spécifique les ressources présentes dans les différents nœuds.

5.4.1 Définition des commandes

On ne discutera pas ici des commandes de contrôle qui sont les mêmes que dans le cas général du chapitre 2. Les commandes de transformation déjà introduites au chapitre 2 ont été modifiées de manière à ce que l'application d'une commande sur un document XML syntaxiquement correct conserve cette correction syntaxique. Par exemple, la commande permettant l'ajout d'un nœud vide dans l'arbre a été remplacée par des commandes permettant d'ajouter des éléments ou des données. De la même manière, pour ne pas obtenir d'arbres avec des nœuds ne contenant ni élément ni donnée et qui ne correspondrait donc pas à des documents XML, on ne peut pas supprimer uniquement les ressources représentant un élément ou des données dans un nœud.

Un autre changement par rapport au langage général est dû à l'unicité des noms d'emplacements. En effet, l'ensemble des arbres considéré est tel qu'on ne peut pas avoir deux nœuds différents avec le même nom. Pour atteindre un nœud, il est donc inutile de donner tout le chemin menant à lui, il suffit ici de donner directement le nom d'emplacement de ce nœud. On utilise cette spécificité dans nos commandes.

Définition 5.3 (Commandes de manipulation des arbres XML).

$C ::=$	$x := new_{cont} E_1 @ E_2$	*Ajouter des données E_1 sous l'emplacement E_2*	
$\mid$	$x := new_{elem} E_1 @ E_2$	*Création d'un nouvel élément E_1 sous l'emplacement E_2*	*Création de*
$\mid$	$x := new_{tree} E_1 @ E_2$	*Ajout d'un sous arbre E_1 sous l'emplacement E_2*	*ressources et de nœuds*
$\mid$	$new_{attr} E_1, E_2 @ E_3$	*Ajout d'un attribut E_1 avec une valeur E_2 à l'emplacement E_3*	
$\mid$	$delete E_1$	*Suppression du sous-arbre situé à l'emplacement E_1*	*Suppression de*
$\mid$	$delete E_2 @ E_1$	*Suppression d'un attribut E_2 à l'emplacement E_1*	*ressources et de nœuds*
$\mid$	$replace_{elem} E_1 @ E_2$	*Remplace l'élément de l'emplacement E_2 par l'élément E_1*	
$\mid$	$x := E$	*Affectation de variable*	
$\mid$	$x := tree@E$	*Observation de l'arbre situé en E*	
$\mid$	$x := elem@E$	*Observation de l'élement situé en E*	*Observation du*
$\mid$	$x := attr(E_1)@E_2$	*Observation de la valeur d'un attribut E_1 situé en E_2*	*contenu*
$\mid$	$x := content@E$	*Observation des données contenues lookup*	
$\mid$	$x := getLoc_{elem}(E_1)@E_2$	*Trouver l'emplacement d'un élément E_1 situé sous l'emplacement E_2*	
$\mid$	$x := getLoc_{attr}(E_1, E_2)@E_3$	*Trouver l'emplacement contenant un attribut E_1 avec la valeur E_2 sous l'emaplcement E_3*	*Récupération d'un nom d'emplacement*
$\mid$	$x := getLoc_{cont}@E$	*Trouver l'emplacement des données sous l'emplacement E*	

E_1, E_2 et E_3 sont des expressions.

Dans les commandes ci-dessus, E, E_i sont des expressions simples (qui ne contiennent pas de commandes). Le domaine sémantique de ces expressions dépend de la commande où elles sont introduites.

Les quatre premières commandes ajoutent des informations à l'arbre. La commande $x := new_{cont} E_1 @ E_2$, ajoute les données désignées par l'expression E_1 sous l'emplacement correspondant à l'expression E_2 et place en x le nom de l'emplacement créé. De la même manière

$x := new_{elem}E_1@E_2$ crée un emplacement sous celui correspondant à l'expression E_2, y place l'élément désigné par l'expression E_1 et affecte à x le nom de l'emplacement créé. La commande $x := new_{tree}E_1@E_2$ place l'arbre correspondant à l'expression E_1 dans un nouvel emplacement situé sous l'emplacement correspondant à l'expression E_2 et dont le nom est affecté à la variable x. La commande $new_{attr}E_1, E_2@E_3$ crée dans l'emplacement donné par l'expression E_3 un attribut dont le nom est donné par l'expression E_1 avec comme valeur celle correspondant à l'expression E_2.

Les deux commandes suivantes suppriment des ressources ou un sous-arbre de l'arbre initial. La commande $deleteE_1$ supprime le sous-arbre situé à l'emplacement désigné par l'expression E_1 et $deleteE_2@E_1$ supprime l'attribut correspondant à l'expression E_2 de l'emplacement désigné par l'expression E_1.

On a ensuite la commande $replace_{elem}E_1@E_2$ qui permet de changer le nom d'un élément situé à l'emplacement correspondant à l'emplacement E_2 en le remplaçant par l'élément désigné par l'expression E_1.

On a ensuite quatre commandes qui permettent d'accéder au contenu du document, une commande par type de contenu. La commande $x := tree@E$ permet de récupérer dans la variable x un sous-arbre situé à l'emplacement donné par l'expression E. $x := elem@E$ récupère l'élément situé à cet emplacement et $x := content@E$ récupère les données de cet emplacement. Enfin, $x := attr(E_1)@E_2$ donne à x la valeur de l'attribut désigné par l'expression E_1 situé à l'emplacement correspondant à l'emplacement E_2.

Les trois dernières commandes permettent de retrouver des noms d'emplacement à l'intérieur d'un arbre. La commande $x := getLoc_{elem}(E_1)@E_2$ met dans la variable x le nom de l'emplacement situé sous l'emplacement désigné par l'expression E_2 qui contient l'élément donné par l'expression E_1 à condition qu'il n'y ait qu'un élément de ce type sous E_2. La commande $x := getLoc_{attr}(E_1, E_2)@E_3$ place en x le nom de l'emplacement qui contient un attribut correspondant à l'expression E_1 ayant comme valeur celle de l'expression E_2 situé sous l'emplacement désigné par l'expression E_3. La commande $x := getLoc_{cont}@E$ place en x le nom de l'emplacement situé sous l'emplacement désigné par l'expression E qui contient des données.

5.4.2 Sémantique des commandes

Les commandes sont interprétées selon une relation $\rightsquigarrow$ entre des configurations. Nous ne reprenons pas ces définitions qui ont déjà été données dans la section 2.4. La sémantique des commandes est donc donnée à travers la définition de la relation $\rightsquigarrow$.

$$\frac{[\![E_2]\!]_s = l \in Loc, [\![E_1]\!]_s = c \in Dat, \exists L \in Loc^* \text{ t.q. } [t(L:l)]\uparrow, l' \in Loc \text{ t.q. } \forall L' \in Loc^*.t(L':l') \text{ n'est pas défini}}{x := new_{cont}E_1@E_2, s, t \rightsquigarrow [s|x \mapsto l'], t|(e, L:l:l' \mapsto c)}$$

$$\frac{[\![E_2]\!]_s = l \in Loc, [\![E_1]\!]_s = c \in Elem, \exists L \in Loc^* \text{ t.q. } [t(L:l)]\uparrow, l' \in Loc \text{ t.q. } \forall L' \in Loc^*.t(L':l') \text{ n'est pas défini}}{x := new_{elem}E_1@E_2, s, P \rightsquigarrow [s|x \mapsto l'], t|(e, L:l:l' \mapsto c}$$

$$\frac{[\![E_2]\!]_s = l \in Loc, [\![E_1]\!]_s = t' \in \mathcal{T}_{M_{XML}}, \exists L \in Loc^* \text{ t.q. } [t(L:l)]\uparrow, l' \in Loc \text{ t.q. } \forall L' \in Loc^*.t(L':l') \text{ n'est pas défini}}{x := new_{tree}E_1@E_2, s, P \rightsquigarrow [s|x \mapsto l'], t|(e, L:l:l' \mapsto t'}$$

Comme indiqué dans la section 5.4.1, les trois règles ci-dessus introduisent un nouvel emplacement dans l'arbre de ressources. On indique que l'emplacement l' doit être un nouvel emplacement car sinon l'arbre ainsi modifié ne serait pas défini.

$$\frac{[\![E_3]\!]_s = l \in Loc,\ [\![E_1]\!]_s = m \in Attr,\ [\![E_1]\!]_s = n \in Val, \exists L \in Loc^* \text{ t.q. } [t(L:l)]\uparrow}{new_{attr}E_1, E_2@E_3, s, t \rightsquigarrow s, t|(e, L:l \mapsto (m \mapsto n))}$$

$$\frac{[\![E_1]\!]_s = l \in Loc, \exists L \in Loc^* \text{ t.q. } t = t'|(e, L:l \mapsto t'') \bigwedge t'(L:l) \text{ n'est pas défini}}{delete E_1, s, t \rightsquigarrow s, t'}$$

La règle précédente décompose t en deux sous-arbres t' et $(e, L : l \mapsto t'')$, t'' étant exactement le sous-arbre de t situé au chemin $L : l$. Pour s'assurer de cela, on indique que t' ne contient aucune information sous ce chemin.

$$\frac{[\![E_1]\!]_s = l \in Loc,\ [\![E_2]\!]_s = m \in Attr, \exists n \in Val, \exists L \in Loc^* \text{ t.q. } t = t'|(e, L:l \mapsto ((m \mapsto n), nil))}{delete E_2@E_1, s, t \rightsquigarrow s, t'}$$

$$\frac{[\![E_1]\!]_s = l \in Loc,\ [\![E_2]\!]_s = m \in Elem, \exists n \in Elem, \exists L \in Loc^* \text{ t.q. } t = t'|(e, L:l \mapsto (n, nil))}{replace_{elem}E_2@E_1, s, t \rightsquigarrow s, t'|(e, L:l \mapsto (m, nil))}$$

On a ensuite les règles classiques d'affectations et de récupération d'informations dans l'arbre :

$$\frac{[\![E]\!]_s = m \in (Elem \cup Attr \cup Val \cup Dat \cup Loc \cup \mathcal{T}_{M_{XML}})}{x := E, s, t \rightsquigarrow [s, x \mapsto m], t}$$

$$\frac{[\![E]\!]_s = l \in Loc, \exists L \in Loc^*, t' \in T_{XML} \text{ t.q. } t(L:l) = t'}{x := tree@E, s, t \rightsquigarrow [s, x \mapsto t'], t}$$

$$\frac{[\![E]\!]_s = l \in Loc, \exists L \in Loc^*; \exists m \in Elem, t' \in T_{XML} \text{ t.q. } t(L:l) = (m, t')}{x := elem@E, s, t \rightsquigarrow [s, x \mapsto m], t}$$

$$\frac{[\![E]\!]_s = l \in Loc, \exists L \in Loc^*; \exists m \in Dat, t' \in T_{XML} \text{ t.q. } t(L:l) = (m, t')}{x := content@E, s, t \rightsquigarrow [s, x \mapsto m], t}$$

$$\frac{[\![E]\!]_s = l \in Loc,\ [\![E_1]\!]_s = n \in Attr, \exists L \in Loc^*, \exists m \in Val, t' \in T_{XML} \text{ t.q. } t(L:l) = ((n \mapsto m), t')}{x := attr(E_1)@E, s, t \rightsquigarrow [s, x \mapsto m], t}$$

Les règles suivantes présentent les commandes permettant de récupérer des noms d'emplacement, commandes qui n'étaient pas présentes dans le langage de manipulation de base. La première de ces commandes est plus simple que les deux autres car pour un élément donné, l'unicité de son fils contenant des données est assurée par la définition même de l'ensemble des arbres.

Pour les deux dernières règles, on doit s'assurer que l'élément ou l'attribut recherché est bien défini de manière unique dans l'arbre. Si ce n'est pas le cas, la configuration est bloquée : on ne peut résoudre la commande.

$$\frac{[\![E]\!]_s = l \in Loc,\ [\![E_1]\!]_s = m \in Dat, \exists L \in Loc^*, \exists l' \in Loc, t' \in T_{XML} \text{ t.q. } [t(L:l:l')]\uparrow \text{ et } t(L:l:l') = (m, t')}{x := getLoc_{cont}(E_1)@E, s, t \rightsquigarrow [s, x \mapsto l'], t}$$

$$\frac{\begin{array}{c}[\![E]\!]_s = l \in Loc,\ [\![E_1]\!]_s = m \in Elem, \exists L \in Loc^*, \exists l' \in Loc, t' \in T_{XML} \text{ t.q.} \\ t = t'|(e, L:l':l \mapsto (m, nil)) \text{ et } \forall l'' \in loc, t'' \in T_{XML}.t'(L:l:l'') \neq (m, t'')\end{array}}{x := getLoc_{elem}(E_1)@E, s, t \rightsquigarrow [s, x \mapsto l'], t}$$

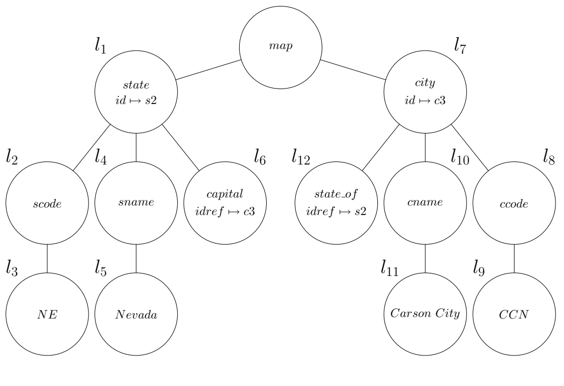
$$\frac{[\![E_3]\!]_s = l \in Loc,\ [\![E_2]\!]_s = n \in Val,\ [\![E_1]\!]_s = m \in Attr, \exists L \in Loc^*, \exists l' \in Loc \text{ t.q. } t = t'|(e, L : l : l' \mapsto ((m \mapsto n), nil) \text{ et } \forall l'' \in loc, t'' \in T_{XML}.t'(L : l : l'') \neq t''|(e, L : l : l'' \mapsto ((m \mapsto n), nil)}{x := getLoc_{attr}(E_1, E_2)@E_3, s, t \rightsquigarrow [s, x \mapsto l'], t}$$

Les conditions des commandes *getLoc* permettent de s'assurer de l'unicité de l'emplacement correspondant aux critères de recherche définis. Ainsi, les commandes de recherche de nom d'emplacement ne peuvent être exécutées que si elle désigne sans ambiguïté un emplacement.

Les règles ainsi proposées fixent formellement la sémantique des commandes de manipulations.

5.4.3 Un exemple de programme

On présente maintenant un exemple permettant d'illustrer les possibilités offertes par le langage de transformation. On suppose que l'évolution de notre DTD implique que le nom de la capitale doit être indiqué sous forme de données sous l'élément *capital* et que tout élément *city* doit contenir un attribut capital dont la valeur est soit *yes*, soit *no*. On propose alors un programme qui modifie l'arbre représentant les données XML de l'exemple initialement proposé en figure 5.1 pour y apporter les changement nécessaires. L'arbre initial et le programme sont donnés en figure 5.4 ; le résultat de la transformation est donné en figure 5.5.



$\rightsquigarrow$

1. $x := getLoc_{elem}(state)@nil$
2. $x := getLoc_{elem}(capital)@x$
3. $y := getAttr(idref)@x$
4. $z := getLoc_{attr}(id, y)@nil$
5. $addAttr(capital, yes)@z$
6. $z := getLoc_{elem}(cname)@z$
7. $z = getLoc_{cont}@z$
8. $y = getContent@z$
9. $y = new_{cont}(y)@x$

FIG. 5.4 – Données initiales et programme de modification de données XML

Le programme de la figure 5.4 démarre par la recherche de l'élément capital (code des lignes 1 et 2). On récupère alors la valeur de l'attribut *idref* de cet élément (ligne 3). On utilise alors cette valeur pour chercher l'élément *city* correspondant à cette capitale (ligne 4). Une fois dans cet élément, on lui ajoute un attribut *capital* avec la valeur *yes* (ligne 5). On cherche alors le nœud correspondant au contenu de l'élément *cname* (lignes 6 et 7), on récupère le contenu de ce nœud (ligne 8) et on le rajoute comme donnée de l'élément capital récupéré à la ligne deux (ligne 9).

Cet exemple illustre, entre autres, le comportement des trois commandes *getLoc* et montre comment elles peuvent être utilisées pour naviguer entre les différents nœuds. Il montre également comment diverses informations peuvent être ajoutées à différents emplacements de l'arbre.

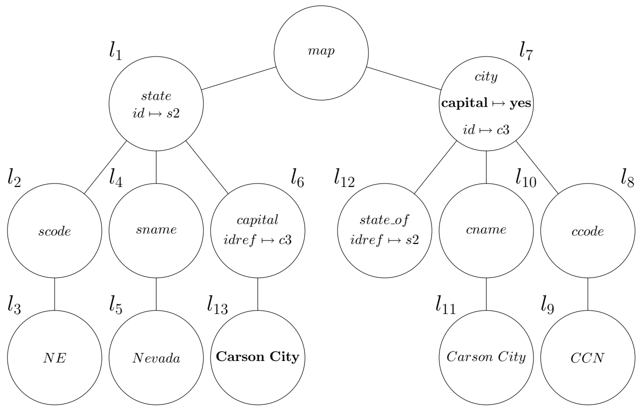


FIG. 5.5 – Résultat après exécution du programme de modification

5.5 Assertions et propriété de fenêtrage

On utilise maintenant la logique de spécification pour XML présentée plus tôt dans ce chapitre pour construire des triplets de Hoare pour les commandes de manipulation des données XML. Ainsi, on pourra prouver les propriétés que l'on peut obtenir en appliquant un programme à une spécification initiale donnée.

Ce travail est un raffinement de celui présenté dans le chapitre 2 à propos du langage générique de manipulation des arbres de ressources. On présente ici les plus faibles pré-conditions pour les commandes de manipulation des données XML et on discute ensuite du problème de la propriété de fenêtrage : étant donné une pré-condition, une post-condition et un programme donnés, quelle formule peut-on ajouter aux pré et post-conditions sans risquer de fausser le résultat du programme ?

5.5.1 Triplets de Hoare et plus faibles pré-conditions

On rappelle qu'un triplet de Hoare est une expression de la forme $\{P\}C\{Q\}$ où P et Q sont des formules logiques et C est une commande. L'interprétation d'un tel triplet est la suivante :

$$\begin{gathered}\{P\}C\{Q\} \text{ est vrai si pour chaque état } s,t \text{ tel que } s,t \models_{\mathfrak{P}} P, \\ C,s,t \text{ est une configuration sûre et si } C,s,t \rightsquigarrow^* s',t' \text{ alors } s',t' \models_{\mathfrak{P}} Q.\end{gathered}$$

On ne donne pas les règles pour les commandes de contrôle, la conséquence et l'enchaînement des commandes qui sont exactement les mêmes que celles du cas général (voir la section 2.4).

- Observation du contenu

Les axiomes des commandes qui accèdent au contenu de l'arbre ne présentent pas de difficulté ni d'évolution majeure par rapport aux commandes accédant au contenu dans le langage générique du chapitre 2.

$$\{\exists_{path} y.(exists(y:E) \wedge \exists_{tree} t.(([y][E]t * no(y:E)) \wedge P\{^t/_x\}))\} \quad x := tree@E \quad \{P\}$$
$$\{\exists_{path} y.(exists(y:E) \wedge \exists_{res} c.(([y][E]elem(c) * \top) \wedge P\{^c/_x\}))\} \quad x := elem@E \quad \{P\}$$
$$\{\exists_{path} y.(exists(y:E_2) \wedge \exists_{res} c.(([y][E_2](attr(E_1) \wedge val(c)) * \top) \wedge P\{^c/_x\})) \quad x := attr(E_1)@E_2 \quad \{P\}$$
$$\{\exists_{path} y.(exists(y:E) \wedge \exists_{res} c.(([y][E]data(c) * \top) \wedge P\{^c/_x\})) \quad x := content@E \quad \{P\}$$

- Ajout d'information

On poursuit avec les triplets des commandes nécessitant la création d'un emplacement :

$$\begin{array}{lll} \{\exists_{path} y.(exists(y:E_2)\wedge & & \\ \forall_{path} y'.\forall_{loc} z.(no(y':z) \rightarrow ([y][E_2][z]cont(E_1) \mathrel{-\!\!*} P\{^z/_x\})))\} & x := new_{cont} E_1@E_2 & \{P\} \\ \{\exists_{path} y.(exists(y:E_2)\wedge & & \\ \forall_{path} y'.\forall_{loc} z.(no(y':z) \rightarrow ([y][E_2][z]elem(E_1) \mathrel{-\!\!*} P\{^z/_x\})))\} & x := new_{elem} E_1@E_2 & \{P\} \\ \{\exists_{path} y.(exists(y:E_2)\wedge & & \\ \forall_{path} y'.\forall_{loc} z.(no(y':z) \rightarrow ([y][E_2][z]E_1 \mathrel{-\!\!*} P\{^z/_x\})))\} & x := new_{tree} E_1@E_2 & \{P\} \end{array}$$

L'utilisation de $no(y' : E_2 : z)$ où y' et z représentent respectivement n'importe quel chemin et n'importe quel emplacement permet de s'assurer que l'emplacement qui va être créé sera un emplacement qui n'existe pas ailleurs dans l'arbre et que quelque soit le nom d'emplacement choisi, la propriété P sera satisfaite. L'utilisation de $\mathrel{-\!\!*}$ indique ce que l'on doit ajouter à l'arbre de ressource pour obtenir la propriété P.

$$\begin{array}{lll} \{\exists_{path} y.(exists(y:E_3)\wedge & & \\ ([y][E_3](attr(E_1) \wedge value(E_2)) \mathrel{-\!\!*} P\{^z/_x\}))\} & new_{attr} E_1, E2@E_3 & \{P\} \end{array}$$

- Suppression et remplacement d'information

$$\begin{array}{lll} \{\exists_{path} y.((P \wedge no(y:E)) * [y][E]\top)\} & delete E_1 & \{P\} \\ \{\exists_{path} y.(P * [y][E_1]attr(E_2))\} & delete E_2@E_1 & \{P\} \\ \{\exists_{path} y.([y][E_1]elem * ([y][E_1]elem(E_2) \mathrel{-\!\!*} P))\} & replace_{elem} E_2@E_1 & \{P\} \end{array}$$

Pour ces règles, on doit juste identifier ce qui doit être supprimé de l'arbre et assurer sa séparation grâce à l'opérateur $*$.

- Recherche d'emplacements

$$\begin{array}{lll} \{\exists_{path} y.\exists_{loc} z.([y][E_2][z]elem(E_1)* & & \\ \forall_{loc} z'.((\top * [y][E_2][z']elem(E_1)) \rightarrow \bot)) \wedge P\{^z/_x\}\} & x := getLoc_{elem}(E_1)@E_2 & \{P\} \\ \{\exists_{path} y.\exists_{loc} z.([y][E_2][z]cont(E_1)* & & \\ \forall_{loc} t.((\top * [y][E_2][t]cont(E_1)) \rightarrow \bot)) \wedge P\{^z/_x\}\} & x := getLoc_{cont}(E_1)@E_2 & \{P\} \\ \{\exists_{path} y.\exists_{loc} z.([y][E_3][z](attr(E_1) \wedge value(E_2))* & & \\ \forall_{loc} t.((\top * [y][E_3][t](attr(E_1) \wedge value(E_2)) \rightarrow \bot)) \wedge P\{^z/_x\}\} & x := getLoc_{attr}(E_1, E_2)@E_3 & \{P\} \end{array}$$

Les axiomes pour les commandes *getLoc* doivent permettre de vérifier que ce que l'on cherche existe et permet d'identifier un emplacement unique. On doit donc à la fois s'assurer de l'existence d'un emplacement correspondant à ce qu'on cherche et s'assurer qu'il n'y a pas d'ambiguïté possible.

Nos axiomes sont tous exprimés sous la forme d'axiomes arrières (*backward axioms* [62]), on s'est donc efforcé d'exprimer des contraintes sur la pré-condition alors que la post-condition peut avoir une forme générale. Ces contraintes indiquent comment on doit étendre ou modifier l'arbre pour aboutir à la post-condition souhaitée. Prenons l'exemple caractéristique de l'axiome arrière de remplacement d'un élément donné :

$$\{\exists_{path} y.([y][E_2]elem * ([y][E_2]elem(E_1) \mathrel{-\!\!*} P)\} replace_{elem} E_1 @ E_2 \{P\}$$

Un arbre satisfait la pré-condition si on peut le décomposer en deux sous-arbres : un qui ne contient que l'élément contenu dans l'emplacement E_2 et un second qui contient le reste de l'information. Ce dernier sous-arbre est alors tel que si on lui ajoute un élément E_1, il vérifiera P.

Lemme 5.2. Les axiomes arrières sont corrects d'après la sémantique des commandes.

Preuve. La preuve est une preuve par cas sur les différents axiomes. Soit un couple s, t et une commande C :

- Cas $x := new_{cont} E_1 @ E_2$: Supposons que $s, t \models_{\mathfrak{P}} \exists_{path} y.(exists(y : E_2) \wedge \forall_{path} y'.\forall_{loc} z.(no(y' : z) \rightarrow ([y][E_2][z]cont(E_1) \mathrel{-\!\!*} P\{^z/_x\})))$. Il existe donc un chemin L tel que (1) $s, t \models_{\mathfrak{P}} exists(L : E_2)$ et (2) $s, t \models_{\mathfrak{P}} \forall_{path} y'.\forall_{loc} z.(no(y' : z) \rightarrow ([L][E_2][z]cont(E_1) \mathrel{-\!\!*} P\{^z/_x\})))$. D'après (1) l'emplacement $L : E_2$ existe donc la configuration n'est pas bloquée, on a donc un emplacement l' tel que $C, s, t \leadsto [s|x \mapsto l'], t|(e, L : E_2 : l' \mapsto E_1)$ où l' est un nouvel emplacement. Comme l' est un nouvel emplacement, on a donc $s, t \models_{\mathfrak{P}} \forall_{path} y'.\forall_{loc} z.no(y' : z)$. (2) nous mène alors à $s, t \models_{\mathfrak{P}} [L][E_2][z]cont(E_1) \mathrel{-\!\!*} P\{^z/_x\}$, ce qui implique $[s|x \mapsto l'], t|(e, L : E_2 : l' \mapsto E_1) \models_{\mathfrak{P}} P$.
- Cas $x := new_{elem} E_1 @ E_2$: Similaire à $x := new_{cont} E_1 @ E_2$.
- Cas $x := new_{tree} E_1 @ E_2$: Similaire à $x := new_{cont} E_1 @ E_2$.
- Cas $new_{attr} E_1, E_2 @ E_3$:
  Supposons que $s, t \models_{\mathfrak{P}} \exists_{path} y.(exists(y : E_3) \wedge ([y][E_3](attr(E_1) \wedge value(E_2)) \mathrel{-\!\!*} P\{^z/_x\}))$. Il existe donc un chemin L tel que (1) $s, t \models_{\mathfrak{P}} exists(L : E_3)$ et (2) $s, t \models_{\mathfrak{P}} [L][E_3](attr(E_1) \wedge value(E_2)) \mathrel{-\!\!*} P\{^z/_x\}$. D'après (1) l'emplacement $L : E_2$ existe donc la configuration n'est pas bloquée, on a donc $C, s, t \leadsto s, t|(e, L : l \mapsto (m \mapsto n))$. D'après (2), on a $s, t|(e, L : l \mapsto (m \mapsto n)) \models_{\mathfrak{P}} P\{^z/_x\}$.
- Cas $delete E_1$: Supposons que $s, t \models_{\mathfrak{P}} \exists_{path} y.((P \wedge no(y : E)) * [y][E]\top)$. Il existe donc un chemin L tel que $s, t \models_{\mathfrak{P}} ((P \wedge no(y : E)) * [L][E]\top)$. Il existe donc t_1, t_2 tels que (1) $t = t_1|t_2$, (2) $s, t_1 \models_{\mathfrak{P}} P \wedge no(L : E)$ et (3) $s, t_2 \models_{\mathfrak{P}} [L][E]\top$. D'après (1) et (3), le chemin $L : E$ existe en t et la configuration n'est pas bloquée. On a donc $s, t \leadsto s, t'$ où il existe t'' tel que $t = t'|(e, L : l \mapsto t'')$ et $t'(L : l)$ n'est pas défini. On peut facilement établir que $t' = t_1$ et $t'' = t_2$ et conclure.
- Cas $delete E_2 @ E_1$: Supposons que $s, t \models_{\mathfrak{P}} \exists_{path} y.(P * [y][E_1]attr(E_2))$. Il existe donc un chemin L tel que $s, t \models_{\mathfrak{P}} (P * [L][E_1]attr(E_2))$. Il existe donc t_1, t_2 tels que (1) $t = t_1|t_2$, (2) $s, t_1 \models_{\mathfrak{P}} P$ et (3) $s, t_2 \models_{\mathfrak{P}} [L][E_1]attr(E_2)$. D'après (1) et (3), le chemin $L : E_1$ existe en t et la configuration n'est pas bloquée. On a donc $s, t \leadsto s, t'$ avec $t = t'|(e, L : l \mapsto ((m \mapsto n), nil))$. On établit facilement que $t_1 = t'$ et que $t_2 = (e, L : l \mapsto ((m \mapsto n), nil))$.
- Cas $replace_{elem} E_2 @ E_1$: Supposons que $s, t \models_{\mathfrak{P}} \exists_{path} y.([y][E_1]elem * ([y][E_1]elem(E_2) \mathrel{-\!\!*} P))$. Il existe donc un chemin L tel que $s, t \models_{\mathfrak{P}} [L][E_1]elem * ([L][E_1]elem(E_2) \mathrel{-\!\!*} P)$. Il existe donc t_1, t_2 tels que (1) $t = t_1|t_2$, (2) $s, t_1 \models_{\mathfrak{P}} [L][E_1]elem$ et (3) $s, t_2 \models_{\mathfrak{P}} [L][E_1]elem(E_2) \mathrel{-\!\!*} P$.

D'après (1) et (2), le chemin $L : E_1$ existe en t et la configuration n'est pas bloquée. On a donc $s, t \rightsquigarrow s, t'|(e, L : l \mapsto (m, nil))$ et n un élément tel que $t = t'|(e, L : l \mapsto (n, nil))$. On établit facilement que $t_1 = (e, L : l \mapsto (n, nil))$ et que $t_2 = t'$. Comme $s, t_2 \models_{\mathfrak{P}} [L][E_1]elem(E_2) \mathrel{-\!\!*} P$, on a bien $s, t'|(e, L : l \mapsto (m, nil)) \models_{\mathfrak{P}} P$.

- Cas $x := getLoc_{elem}(E_1)@E_2$: Supposons que $s, t \models_{\mathfrak{P}} \exists_{path} y.\exists_{loc} z.([y][E_2][z]elem(E_1) * \forall_{loc} z'.((\top * [y][E_2][t]elem(E_1)) \rightarrow \bot)) \wedge P\{^z/_x\}$. Il existe donc un chemin L et un emplacement l tel que $s, t \models_{\mathfrak{P}} ([L][E_2][l]elem(E_1) * \forall_{loc} z'.((\top * [L][E_2][z']elem(E_1)) \rightarrow \bot)) \wedge P\{^z/_x\}$. donc il existe t_1, t_2 tels que (1) $t = t_1|t_2$, (2) $s, t_1 \models_{\mathfrak{P}} [L][E_2][l]elem(E_1)$ et (3) $s, t_2 \models_{\mathfrak{P}} \forall_{loc} z'.((\top * [L][E_2][t]elem(E_1)) \rightarrow \bot))$ et on a également $s, t \models_{\mathfrak{P}} P\{^z/_x\}$. D'après (2), l'élément recherché est bien présent, et d'après (3), il est défini de façon unique. Donc la configuration n'est pas bloquée et on peut conclure.
- Cas $x := getLoc_{cont}(E_1)@E_2$: Similaire à $x := getLoc_{elem}(E_1)@E_2$.
- Cas $x := getLoc_{attr}(E_1, E_2)@E_3$: Similaire à $x := getLoc_{elem}(E_1)@E_2$.

□

Rappelons maintenant la définition d'une plus faible pré-condition, telle qu'énoncée dans le chapitre 2 :

Définition 5.4 (Plus faible pré-condition). *Soit une commande C et une formule Q la* plus faible pré-condition *$wp(C, Q)$ du couple C, Q est l'ensemble des configurations telles que :*
$s, t \in wp(C, Q)$ si C, s, t est sure et si $C, s, t \rightsquigarrow s', t'$ implique $s, t' \models_{\mathfrak{A}} Q$.

On peut finalement prouver le résultat suivant :

Théorème 5.1. *Les plus faibles pré-conditions des commandes axiomatiques sont données par l'axiome arrière du correspondant.*

Preuve. Le lemme 5.2 a montré que les axiomes sont corrects d'après la sémantique des commandes. Il reste donc a prouver que pour chaque triplet $\{P\}C\{Q\}$ et configuration $s, t \in wp(C, Q)$ vérifie $s, t \models_{\mathfrak{A}} P$:

- Cas $x := new_{cont} E_1@E_2$: Soit $s, t \in wp(C, Q)$. Donc $C, s, t \rightsquigarrow [s|x \mapsto l'], t|(e, L : l : l' \mapsto c) \models_{\mathfrak{P}} Q$ où l' n'est pas présent dans t et L est le chemin menant à l. On a donc $s, t|(e, L : l : l' \mapsto c) \models_{\mathfrak{P}} Q\{^x/_l\}$. Comme on a trivialement $s, (e, L : l : l' \mapsto c) \models_{\mathfrak{P}} [L : l : l']cont(E_1)$, d'après la sémantique de $\mathrel{-\!\!*}$, on obtient : $s, t \models_{\mathfrak{P}} [L : l : l']cont(E_1) \mathrel{-\!\!*} Q\{^x/_l\}$. Comme l est choisi arbitrairement, ceci est vrai pour tout emplacement l' qui n'est pas dans t on peut donc affirmer que l'on a $s, t \models_{\mathfrak{P}} \forall_{path} y'.\forall_{loc} z.(no(y' : z) \rightarrow ([y][E_2][z]cont(E_1) \mathrel{-\!\!*} P\{^z/_x\}))$. Enfin, comme la configuration initiale n'est pas bloquée, on sait que E_2 existe en t et on a bien : $s, t \models_{\mathfrak{P}} \exists_{path} y.(exists(y : E_2) \wedge \forall_{path} y'.\forall_{loc} z.(no(y' : z) \rightarrow ([y][E_2][z]cont(E_1) \mathrel{-\!\!*} P\{^z/_x\})))$.
- Cas $deleteE_2@E_1$: Soit $s, t \in wp(C, Q)$. Donc on a $C, s, t \rightsquigarrow s, t'$ où $t = t'|(e, L : l \mapsto ((m \mapsto n), nil))$ avec L le chemin menant à l'emplacement l, $[\![E_2]\!]_s = m$ et $s, t' \models_{\mathfrak{P}} Q$. Il est évident que $s, (e, L : l \mapsto ((m \mapsto n), nil)) \models_{\mathfrak{P}} [L : l]attr(E_2)$. On peut donc conclure que $s, t \models_{\mathfrak{P}} Q * [L : l]attr(E_2)$ et donc que $s, t \models_{\mathfrak{P}} \exists_{path} y.(P * [y][E_1]attr(E_2))$.
- Cas $x := getLoc_{elem}(E_1)@E_2$: Soit $s, t \in wp(C, Q)$. Donc on a $C, s, t \rightsquigarrow [s, x \mapsto l'], t$ où l' est l'emplacement de l'unique élément E_1 situé sous l'emplacement l. L'existence de l' implique $s, t \models_{\mathfrak{P}} \exists_{path} y.\exists_{loc} z.[y][E_2][z]elem(E_1)$. L'unicité de l' assure que tout autre emplacement sous l ne contient pas l'élément E_2, ce qui implique $s, t \models_{\mathfrak{P}} \exists_{path} y.\exists_{loc} z.([y][E_2][z]elem(E_1) * \forall_{loc} z'.((\top * [y][E_2][z']elem(E_1)) \rightarrow \bot))$. De plus, comme l'arbre n'est pas modifié par la commande, on a également $s, t \models_{\mathfrak{P}} Q\{^z/_x\}$. On peut alors conclure.

Les autres cas se traitent de façon similaire. □

5.5.2 Propriété de fenêtrage et de localité

Le principe de localité (ou propriété de fenêtrage) *(frame property)* permet de vérifier dans quel contexte un programme donné peut être exécuté sans que le contexte ne fausse le résultat ou soit modifié par le programme. Il s'agit donc de savoir ce qu'on peut ajouter à un document donné sans corrompre le comportement de notre programme ni modifier ce qu'on a ajouté.

Les problèmes rencontrés englobent ceux rencontrés dans le langage général pour la manipulation d'arbres du chapitre 2 et pour d'autres langages [24, 62] concernant la modification des variables et la création de nouveaux emplacements (voir la section 2.7).

Concernant le problème de la création de noms d'emplacements, la situation est toutefois légèrement différente ici. S'il est difficile dans le cas général de s'assurer qu'un nom d'emplacement qui est nouveau dans un arbre le restera si on élargit le contexte, on sait ici qu'un arbre qui représente un document XML contient dans chaque nœud une ressource correspondant à un nom élément ou à des données. Supposons alors que l'on crée un emplacement (qui est donc censé ne pas exister dans l'arbre d'origine) mais que le contexte ajouté contienne déjà ce nom emplacement. Si le chemin de l'emplacement créé est différent du chemin où se trouve ce nom d'emplacement dans le contexte, l'arbre ainsi créé est indéfini puisqu'on ne peut avoir le même emplacement à deux chemins différents. Si les chemins sont identiques et que le contexte correspond à un document XML, le contexte contient donc, à cet emplacement, soit un élément, soit des données. Comme les commandes créant des emplacements y ajoutent obligatoirement elles aussi un élément ou des données, on a donc dans un même emplacement soit deux ressources *« données »*, soit deux ressources *« élément »*, soit une de chaque type, ce qui par définition mène à un arbre indéfini. De plus, on sait qu'un arbre indéfini valide trivialement la post-condition. Contrairement au cas général donc, la création d'emplacement ne pose pas de problème si on se limite aux arbres représentant des données XML.

On a également le problème relatif aux commandes qui manipulent tout un sous-arbre (pour en lire ou en supprimer le contenu), problème lui aussi déjà soulevé dans le cas général. En effet, rien ne permet d'assurer que le contexte ne va pas modifier le sous-arbre en question. Dans un tel cas, l'exécution du reste du programme ou le contexte peut être modifiée.

L'introduction dans les commandes spécifiques aux manipulations de données XML des commandes *getLoc* pose également des problèmes. Ces commandes doivent rechercher un emplacement et s'assurer qu'aucun autre ne correspond à leur critère. Si l'on élargit le contexte, ce dernier point ne peut être assuré. Par conséquent, on ne peut pas utiliser ces commandes, sauf en imposant des conditions sur l'arbre obtenu après ajout du contexte.

Si l'on considère un ensemble de commandes restreint duquel on exclut les commandes *getLoc*, la commande $deleteE_1$ et la commande $tree@E$, on a alors la propriété de fenêtrage suivante :

Proposition 5.1. *La règle suivante est correcte :*

$$\frac{\{(P * P') \land \forall_{path} x.(exists(x) \rightarrow ([x](elem \lor cont) * \top))\}C\{Q * P'\}}{\{P\}C\{Q\}} \, *$$

$$* \ : \ fv(Q) \cap Modified(C) = \emptyset$$

Preuve. La preuve de cette proposition suit le même raisonnement que celle du théorème 2.2.

La condition $\forall_{path} x.(exists(x) \rightarrow ([x](elem \lor cont) * \top))$ permet de s'assurer que chaque chemin contient bien un élément ou des données. Cette condition permet de s'assurer que les commandes devant créer des emplacements ne posent pas de problème. En effet, soit l'emplacement créé porte un nouveau nom, soit l'arbre sera indéfini si on tente d'y ajouter du contenu. □

5.6 Un exemple de preuve

Pour illustrer l'utilisation qui peut être faite du langage d'assertion, on reprend l'exemple de code proposé en figure 5.4. Le but de ce programme était de rajouter les données contenant le nom de la capitale sous l'emplacement l_6 et de rajouter un attribut *capital* de valeur *yes* en l_7.

Admettons que l'on veuille uniquement vérifier ces deux points après exécution du programme de transformation. La post-condition à vérifier est donc :

$$P_{final} = \top * [l_7](attr(capital) \wedge value(yes)) * \exists_{loc}x.\exists_{res}c.([l_7][l_1 0][l_1 1]dat(c) * [l1][l_6][x]dat(c)).$$

On utilise commande par commande les axiomes arrières présentés précédemment, selon le schéma présenté en figure 5.6. On obtient alors une pré-condition initiale P_0 à vérifier.

$$\begin{array}{lll}
\{P_0\} & x := getLoc_{elem}(state)@nil & \{P_1\} \\
\{P_1\} & x := getLoc_{elem}(capital)@x & \{P_2\} \\
\{P_2\} & y := getAttr(idref)@x & \{P_3\} \\
\{P_3\} & z := getLoc_{attr}(idref, y)@nil & \{P_4\} \\
\{P_4\} & addAttr(capital, yes)@z & \{P_5\} \\
\{P_5\} & z := getLoc_{elem}(cname)@z & \{P_6\} \\
\{P_6\} & z = getLoc_{cont}@z & \{P_7\} \\
\{P_7\} & y = getContent@z & \{P_8\} \\
\{P_8\} & y = new_{cont}(y)@x & \{P_{final}\}
\end{array}$$

FIG. 5.6 – Triplets de Hoare et preuve de programmes

Supposons que l'on sache que les données initiales valident une proposition P_{init}. Le problème est alors de s'assurer que P_{init} permet de déduire que P_0. On doit donc trouver une *preuve* assurant que P_0 est la conséquence logique de P_{init}. Les travaux présentés au chapitre 3 ouvrent la voie pour la réalisation d'une telle preuve. Ils doivent cependant être étendus pour prendre en compte les quantificateurs et pour être adaptés au modèle d'arbres partiel particulier que l'on manipule ici.

On a donc présenté dans ce chapitre une première approche pour établir une vérification formelle des transformations de documents XML en utilisant les arbres de ressources. Cette application illustre les possibilités de spécifications offertes par les arbres de ressources, dont elle utilise de nombreuses spécificités (emplacements, composition partielle, ressources,...), et la logique BI-Loc. De plus, elle traite un problème clé dans le domaine des données semi-structurées, celui de l'interopérabilité et de la transformation de documents.

Enfin, cette application, comme celles sur les pointeurs et les permissions présentées au chapitre 4, illustre la nécessité de pouvoir adapter la méthode des tableaux présentée au chapitre 3 à un modèle particulier et de l'élargir aux quantificateurs.

Conclusions et perspectives

Le travaux développés au cours de cette thèse permettent de faire un premier bilan sur l'extension de logiques par des modalités d'emplacement et/ou de déplacement. Dans les deux cas étudiés dans cette thèse, ceux de DMLL et de BI-Loc, on montre que ces modalités peuvent être suffisamment bien maîtrisées pour améliorer la spécification de systèmes où les notions de distribution et de mobilité sont importantes et également pour pouvoir étendre les résultats existants dans les logiques sous-jacentes. Pour DMLL, on a montré que l'on pouvait conserver l'élimination des coupures et que l'on pouvait proposer une sémantique des ressources pour la version propositionnelle. Pour BI-Loc, l'ajout des emplacements dans BI a été suffisamment bien maîtrisé pour que les résultats de décidabilité pour BI établis au cours des travaux puissent être étendus à BI-Loc. Cela nous a mené à établir que la satisfaction et la validité d'une formule propositionnelle de BI-Loc est décidable et que la satisfaction est décidable pour le fragment de BI-Loc avec quantificateurs n'utilisant pas l'opérateur $-\!\!*$. De même, les techniques de recherche de preuves et de génération de contre-modèles établies par [72] ont pu servir de guide pour définir une procédure de construction de preuves ou de contre-modèles pour BI-Loc. On a donc obtenu une méthode permettant de déterminer si une formule de BI-Loc est la conséquence logique d'une autre formule donnée. La proximité entre les méthodes de recherche de preuves pour BI et BI-Loc est telle que l'on peut espérer s'inspirer de BILL, le prouveur développé pour BI [48] pour développer les algorithmes de recherche de preuves et de contre-modèles présentés pour BI-Loc.

Un langage de transformation des arbres de ressources a également été proposé. Ce langage générique, censé être utilisable pour tous les modèles d'arbres partiels, a pour vocation de servir de base à des langages utilisés pour des instances plus spécifiques du modèle d'arbre. On a déterminé une sémantique à base de triplets de Hoare pour ce langage en déterminant les plus faibles pré-conditions pour chacune des commandes. Enfin, on a expliqué sous quelles conditions le contexte d'un programme pouvait être agrandi sans modifier son exécution, cette propriété étant plus difficile à mettre en œuvre que dans des travaux comme ceux de [62] du fait de la composition particulière des arbres qui n'assure pas la séparation totale des nœuds.

D'un point de vue applicatif, on a montré que l'on pouvait utiliser le modèle d'arbres partiels pour représenter des modèles mémoires comme ceux des pointeurs [62] ou encore des document semi-structurés. L'application aux pointeurs puis à son extension avec permission [17] a montré l'intérêt d'avoir un modèle plus général pouvant être instancié de différentes manières. En effet, alors que le modèle des permissions semblait introduire des problèmes de partage nouveaux par rapport au modèle des pointeurs, ces problèmes sont déjà pris en compte dans BI-Loc. Ainsi, l'étude du modèle des arbres partiels et de BI-Loc permet de répondre au problème de la spécification des arbres dans le modèles des permissions.

L'application de BI-Loc aux données semi-structurées est certainement celle qui illustre le mieux les possibilités offertes par le modèle d'arbre partiel. On a tout d'abord établi une relation entre un document XML et un arbre de ressources. Cette relation permet de représenter intégralement les données du document. La différence majeure avec le document initial est de ne pas fixer l'ordre des nœuds frères puisqu'aucun ordre n'est disponible sur les nœuds des arbres partiels. On a établit également une relation entre les modèles de document de type DTD (Document Type Definition) et une variante de BI-Loc. Ceci nous permet entre autre de pouvoir utiliser le model-checking pour vérifier qu'un document vérifie une certaine définition de document. On a ensuite proposé un langage spécifique à la manipulation des arbres représentant des données semi-structurées, ainsi qu'une sémantique à base de triplets de Hoare pour ce langage. On utilise alors la même logique pour spécifier les document et pour établir les conséquences logiques d'un programme de transformation de document. L'idée sous-jacente est de pouvoir s'assurer qu'un programme donné permet de passer d'une spécification à une autre en raisonnant sur la représentation logique de ces spécifications, les travaux réalisés constituent un premier pas dans cette direction.

À partir de ces résulttas, plusieurs pistes sont à explorer. Le développement des algorithmes et des techniques de recherche de preuves présentées dans cette thèse est un point important. Cela nécessitera en particulier de développer des stratégies efficaces de construction des tableaux. Il sera également nécessaire de prendre en considération les aspects de complexité algorithmique qui n'étaient pas jusqu'à présent notre priorité. À ce sujet, il sera intéressant d'étudier la possibilité d'utiliser les emplacements pour améliorer l'efficacité de la recherche de preuves. Concernant la recherche de preuve toujours, il sera également intéressant d'étudier comment les travaux présentés dans ce document peuvent s'adapter aux différentes instanciations du modèle d'arbres partiel. En effet, dans le modèle présenté, on raisonne sur l'ensemble des modèles. Quid alors de la prouvabilité pour un modèle particulier ? Il semble intéressant d'étudier si les contraintes générées pour la construction de contre-modèle peuvent être exploitées lorsqu'on raisonne sur un modèle particulier et si l'on peut trouver une méthode générale permettant de raisonner sur différents modèles.

D'un point de vue applicatif, les résultats obtenus sur les modèles à base de pointeurs laissent à penser qu'il sera également intéressant d'étudier comment les travaux réalisés peuvent s'appliquer à des modèles de langage utilisant une représentation plus complexe de la mémoire, comme le modèle de stockage hiérarchique [4]. Pour cette application et également pour celle présentée ici pour les données XML, il sera également important de chercher à enrichir le modèle d'arbres pour y inclure une notion d'ordre entre les fils, comme dans [3]. Dans cette optique, il est également envisagé d'étudier s'il est possible d'inclure des opérateurs non-commutatifs [82, 50] à la logique et si cela est possible, d'étudier les conséquences sur la prouvabilité.

Concernant les applications des arbres de ressources, les travaux sur XML nous ont permis d'obtenir des résultats encourageants et laissent à penser que cette voie est à poursuivre. Il serait notamment intéressant d'établir un lien plus clair entre le langage de manipulation présenté et des langages de manipulation standard dans le domaine des données semi-structurées, comme DOM ou XSL.

Bibliographie

[1] Serge Abiteboul, Peter Buneman, and Dan Suciu. *Data on the Web : from relations to semi-structured data and XML*. Morgan Kaufmann Publishers Inc., 1999.

[2] S. Abramsky. Computational interpretations of linear logic. *Theoretical Computer Science*, 111(1-2) :3–58, 1993.

[3] L. Afanasiev, P. Blackburn, I. Dimitriou, B. Gaiffe, E. Goris, M. Marx, and M. de Rijke. Pdl for ordered trees. *Journal of Applied Non-Classical Logics*, 15(2) :115–135, 2005.

[4] A. Ahmed, L. Jia, and D. Walker. Reasoning about hierarchical storage. In *18th Symposium on Logic in Computer Science*, Ottawa, Canada, 2003.

[5] V. Alexiev. Applications of Linear Logic to Computation : An Overview. *Logic Journal of the IGPL*, 2(1) :77–107, 1994.

[6] G. Allwein and J.M. Dunn. Kripke models for linear logic. *Journal of Symbolic Logic*, 58(2) :514–545, 1993.

[7] J.M. Andreoli. Logic Programming with Focusing Proofs in Linear Logic. *Journal of Logic and Computation*, 2(3) :297–347, 1992.

[8] J.M. Andreoli and R. Maieli. Focusing and proof-nets in linear and non-commutative logic. In *Int. Conf. on Logic for Programming and Automated Reasoning, LPAR'99, LNCS 1705*, pages 320–333, Tbilisi, Georgia, September 1999.

[9] A. Avron. Gentzen-type systems, resolution and tableaux. *Journal of Automated Reasoning*, 10 :265–281, 1993.

[10] H. Bal, J. Steiner, and A. Tanenbaum. Programming languages for distributed computing systems. *ACM Comput. Surv.*, 21(3) :261–322, 1989.

[11] G. Bellin. Subnets of proof-nets in multiplicative linear logic with mix. *Mathematical Structure in Computer Science*, 7 :663–699, 1997.

[12] B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and P. Schnoebelen. *Systems and Software Verification. Model-Checking Techniques and Tools.* Springer, 2001.

[13] N. Biri and D. Galmiche. A Separation Logic for Resource Distribution. In *23rd Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS'03, LNCS 2914*, pages 23–37, Mumbai, India, December 2003.

[14] P. Blackburn, M. de R., and Y. Venema. *Modal logic.* Cambridge University Press, New York, NY, USA, 2001.

[15] R. Bornat. Proving pointer programs in Hoare Logic. In *MPC '00 : Proceedings of the 5th International Conference on Mathematics of Program Construction*, pages 102–126, London, UK, 2000. Springer-Verlag.

[16] R. Bornat, C. Calcagno, and P. O'Hearn. Local reasoning, separation, and aliasing. In *Proceedings of the Second workshop on Semantics, Program Analysis and Computing Environments for Memory Management (SPACE 2004)*, 2004.

[17] R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission accounting in separation logic. In *The 32nd Annual Symposium on Principles of Programming Languages, POPL'05*, Long Beach, California, January 2005.

[18] E. Boudinet and D. Galmiche. Proofs, concurrent objects and computations in a FILL framework. In *Workshop on Object-based Parallel and Distributed Computation, OBPDC'95, LNCS 1107*, pages 148–167, Tokyo, Japan, 1996.

[19] M. Bozga, R. Iosif, and Y. Lakhnech. Storeless semantics and alias logic. In *PEPM'03 : 2003 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation*, 2003.

[20] M. Bozzano and G. Delzanno. Protocol verification via a bottom-up evaluation strategy for linear logic programs. In *FLOC'02 Workshop on Linear Logic*, July 2002. Copenhagen, Danemark.

[21] L. Caires and L. Cardelli. A spatial logic for concurrency (part I). In *4th Int. Symposium on Theoretical Aspects of Computer Software, TACS 2001, LNCS 2215*, pages 1–37, Sendai, Japan, October 2001.

[22] L. Caires and L. Cardelli. A spatial logic for concurrency (part II). In *Int. Conference on Concurrency Theory, CONCUR 2002, LNCS 2421*, pages 209–225, 2002.

[23] C. Calcagno, L. Cardelli, and A. Gordon. Deciding validity in a spatial logic for trees. In *ACM Sigplan Workshop on Types in Language Design and Implementation, TLDI'03*, New Orleans, USA, 2003.

[24] C. Calcagno, P. Gardner, and U. Zarfaty. Context logic and tree update. In *The 32nd Annual Symposium on Principles of Programming Languages, PoPL'05*, Long Beach, California, January 2005.

[25] C. Calcagno, H. Yang, and P. O'Hearn. Computability and complexity results for a spatial assertion language for data structures. In *21st Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS'01, LNCS 2245*, pages 108–119, Bangalore, India, 2001.

[26] L. Cardelli, P. Gardner, and G. Ghelli. Querying trees with pointers, 2003. Draft.

[27] L. Cardelli and G. Ghelli. TQL : a query language for semistructured data based on the ambient logic. *Mathematical Structures in Computer Science*, 14(3) :285–327, 2004.

[28] L. Cardelli and A.D. Gordon. Mobile ambients. In *Foundations of Software Science and Computation Structures, FoSSaCS'98, LNCS 1378*, pages 140–155, Lisbon, Portugal, March/April 1998.

[29] L. Cardelli and A.D. Gordon. Anytime, anywhere - modal logics for mobile ambients. In *27th ACM Symposium on Principles of Programming Languages, PoPL 2000*, pages 365–377, Boston, USA, 2000.

[30] I. Cervesato, N. Durgin, M. Kanovich, and A. Scedrov. Interpreting strands in Linear Logic. In *Workshop on Formal Methods and Computer Security (satellite workshop of CAV 2000)*, Chicago, US, July 2000.

[31] I. Cervesato, N. Durgin, P. Lincoln, J. Mitchell, and A. Scedrov. A meta-notation for protocol analysis. In *12th IEEE Computer Security Foundations Workshop, CSFW'99*, Mordano, Italy, June 1999.

[32] I. Cervesato, N. Durgin, P. Lincoln, J. Mitchell, and A. Scedrov. Relating strands and multiset rewriting for security protocol analysis. In *13th IEEE Computer Security Foundations Workshop, CSFW'00*, Cambridge, UK, July 2000.

[33] I. Cervesato, J. Hodas, and F. Pfenning. Efficient resource management for linear logic proof search. *Theoretical Computer Science*, 232(1-2) :133–163, 2000.

[34] I. Cervesato and F. Pfenning. A linear logical framework. In *11th IEEE Symposium on Logic in Computer Science*, pages 264–275, New Brunswick, New Jersey, July 1996.

[35] W. Charatonik and J.M. Talbot. The decidability of model checking mobile ambients. In *15th Int. Workshop on Computer Science Logic, CSL 2001, LNCS 2142*, pages 339–354, Paris, France, 2001.

[36] W. Charatonik, S. Dal Zilio, A. Gordon, S. Mukhopadhyay, and J.M. Talbot. Model checking mobile ambients. *Theoretical Computer Science*, 308(1-3) :277–331, 2003.

[37] J. Clark and J. Jacob. *A survey of authentication protocol literature : Version 1.0.* 1997.

[38] D. Colazzo, C. Sartiani, A. Albano, P. Manghi, G. Ghelli, L. Lini, and M. Paoli. A typed text retrieval query language for XML documents. *Journal of the American Society for Information Science and Technology*, 53(6) :467–488, 2002.

[39] G. Conforti, D. Macedonio, and V. Sassone. Bigraphical logics for XML. In *Sistemi Evoluti per Basi di Dati (SEBD) 2000*, pages 392–399, L'Aquila, Italy, 2005.

[40] M. D'Agostino and D.M. Gabbay. A Generalization of Analytic Deduction via Labelled Deductive Systems. Part I : Basic Substructural Logics. *Journal of Automated Reasoning*, 13(2) :243–281, 1994.

[41] J. Euzenat. An infrastructure for formally ensuring interoperability in a heterogeneous semantic web. In *1st international semantic web working symposium (SWWS)*, pages 345–360, Stanford, Californy, United States, 2001.

[42] M. Fitting. *First-Order Logic and Automated Theorem Proving.* Texts and Monographs in Computer Science. Springer Verlag, 1990.

[43] D.M. Gabbay. *Labelled Deductive Systems, Volume I - Foundations.* Oxford University Press, 1996.

[44] D. Galmiche. Connection Methods in Linear Logic and Proof nets Construction. *Theoretical Computer Science*, 232(1-2) :231–272, 2000.

[45] D. Galmiche and D. Méry. Proof-search and countermodel generation in propositional BI logic - extended abstract -. In *4th Int. Symposium on Theoretical Aspects of Computer Software, TACS 2001, LNCS 2215*, pages 263–282, Sendai, Japan, 2001.

[46] D. Galmiche and D. Méry. Connection-based proof search in propositional BI logic. In *18th Int. Conference on Automated Deduction, CADE-18, LNAI 2392*, pages 111–128, Copenhagen, Danemark, 2002.

[47] D. Galmiche and D. Méry. Semantic labelled tableaux for propositional BI without bottom. *Journal of Logic and Computation*, 13(5) :707–753, 2003.

[48] D. Galmiche and D. Méry. Resource graphs and countermodels in resource logics. *Electronic Notes in Theoretical Computer Science*, 125(3) :117–135, 2005.

[49] D. Galmiche, D. Méry, and D. Pym. Resource Tableaux (extended abstract). In *16th Int. Workshop on Computer Science Logic, CSL 2002, LNCS 2471*, pages 183–199, Edinburgh, Scotland, September 2002.

[50] D. Galmiche and J.M. Notin. Calculi with Dependency Relations for Mixed Linear Logic. In *International Workshop on Logic and Complexity in Computer Science, LCCS'2001*, pages 81–102, Créteil, France, 2001.

[51] D. Galmiche and G. Perrier. A procedure for automatic proof nets construction. In *LPAR'92, International Conference on Logic Programming and Automated Reasoning, LNAI 624*, pages 42–53, St. Petersburg, Russia, July 1992.

[52] D. Galmiche and G. Perrier. Foundations of proof search strategies design in linear logic. In *Logic at St Petersburg '94, Symposium on Logical Foundations of Computer Science, LNCS 813*, pages 101–113, St Petersburg, Russia, July 1994.

[53] D. Galmiche and G. Perrier. On Proof Normalization in Linear Logic. *Theoretical Computer Science*, 135(1) :67–110, 1994.

[54] P. Gardner and S. Maffeis. Modelling dynamic web data. In *DBPL*, pages 130–146, Potsdam, Germany, September 2003.

[55] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50(1) :1–102, 1987.

[56] J. Harland and D. Pym. The uniform proof-theoretic foundation of linear logic programming. In *International Symposium on Logic Programming, MIT Press*, pages 304–318, San Diego, October 1991.

[57] J. Harland and D. Pym. Resource-distribution via boolean constraints (extended abstract). In *14th Int. Conference on Automated Deduction, CADE-12, LNAI 814*, pages 222–236, Townsville, North Queensland, Australia, July 1997.

[58] D. Hirschkoff, É. Lozes, and D. Sangiorgi. Separability, expressiveness and decidability of the ambient logic. In *17th Annual IEEE Symposium on Logic in Computer Science, LICS'02*, pages 423–432, Copenhagen, Denmark, July 2002.

[59] D. Hirschkoff, É. Lozes, and D. Sangiorgi. Minimality results for spatial logics. In *23rd Conference on Fundations of Software Technology and Theoretical Computer Science,FSTTCS'03, LNCS 2914*, pages 252–264, Mumbai, India, December 2003.

[60] C.A.R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10) :576–580, 1969.

[61] J. Hodas and D. Miller. Logic programming in a fragment of intuitionistic linear logic. *Journal of Information and Computation*, 110 :327–365, 1994.

[62] S. Ishtiaq and P. O'Hearn. BI as an assertion language for mutable data structures. In *28th ACM Symposium on Principles of Programming Languages, POPL 2001*, pages 14–26, London, UK, 2001.

[63] J. Jenson, M. Jorgensen, N. Klarkund, and M. Schwartzback. Automatic verification of pointer programs using monadic second-order logic. In *Conf. on Programming Language Design and Implementation, PLDI'97*, pages 225–236, 1997.

[64] M. Kanovich and T. Ito. Temporal linear logic specifications for concurrent processes. In *12th Annual IEEE Symposium on Logic in Computer Science (LICS'97)*, pages 48–57, Washington - Brussels - Tokyo, June 1997. IEEE.

[65] N. Kobayashi, T. Shimizu, and A. Yonezawa. Distributed concurrent linear logic programming. *Theoretical Computer Science*, 227(1-2) :185–220, 1999.

[66] N. Kobayashi and A. Yonezawa. Asynchronous communication model based on linear logic. *Formal Aspects of Computing*, 3 :279–294, 1994.

[67] N. Kobayashi and A. Yonezawa. Higher-order concurrent linear logic programming. In *Int. Workshop on Theory and Practice of Parallel Programming, LNCS 907*, pages 137–166, Sendai, Japan, November 1994.

[68] C. Kreitz and J. Otten. Connection-based theorem proving in classical and non-classical logics. *Journal of Universal Computer Science*, 5(3) :88–112, 1999.

[69] G. Lowe. An attack on the Needham-Schroeder public-key authentication protocol. *Information Processing Letters*, 56 :131–133, 1995.

[70] H. Mantel and J. Otten. linTAP : A Tableau Prover for Linear Logic. In *Int. Conference on Analytic Tableaux and Related Methods, TABLEAUX'99, LNCS 1617*, pages 216–231, Saratoga Springs, NY, USA, 1999.

[71] F. Mehta and T. Nipkow. Proving pointer programs in higher-order logic. In *Int. Conference on Automated Deduction, CADE-19, LNCS 2741*, pages 121–135, Miami, USA, July 2003.

[72] D. Méry. *Preuves et sémantiques dans des logiques de ressources.* PhD thesis, Université Henri Poincaré, 2004.

[73] D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51 :125–157, 1991.

[74] R. Needham and M. Schroeder. Using encryption for authentication in large networks of computers. *Commun. ACM*, 21(12) :993–999, 1978.

[75] P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *15th Int. Workshop on Computer Science Logic, CSL 2001, LNCS 2142*, pages 1–19, Paris, France, 2001.

[76] P.W. O'Hearn and D. Pym. The Logic of Bunched Implications. *Bulletin of Symbolic Logic*, 5(2) :215–244, 1999.

[77] M. Parkinson and G. Bierman. Separation logic and abstraction. In *PoPL '05 : Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 247–258, New York, NY, USA, 2005.

[78] D. Pym and J. Harland. A uniform proof-theoretic investigation of linear logic programming. *Journal of Logic and Computation*, 4(2) :175–207, 1994.

[79] D.J. Pym. *The Semantics and Proof Theory of the Logic of Bunched Implications*, volume 26 of *Applied Logic Series*. Kluwer Academic Publishers, 2002.

[80] S. Ranise and D. Deharbe. Applying light-weight theorem proving to debugging and verifying pointer programs. *Electronic Notes in Theoretical Computer Science*, 86(1), 2003.

[81] J.C. Reynolds. Intuitionistic reasoning about shared mutable data structure. In *Millenial Perspectives in Computer Science*. Palgrove, 2000.

[82] P. Ruet and F. Fages. Concurrent constraint programming and non-commutative logic. In *11th Int. Workshop on Computer Science Logic, CSL'97, LNCS 1414*, pages 406–423, Aarhus, Denmark, August 1997.

[83] B. A. Trakhtenbrot. The impossibilty of an algorithm for the decision problem for finite models. *Doklady Akademii Nauk SSR*, 70 :569–572, 1950.

[84] A. Voronkov. Proof-search in intuitionistic logic based on constraint satisfaction. In *5th Int. Workshop on Theorem Proving with Analytic Tableaux and Related Methods, LNAI 1071*, pages 312–327, Terrasini, Italy, May 1996.

[85] S. Dal Zilio and D. Lugiez. XML schema, tree logic and sheaves automata. In *Rewriting Techniques and Applications, RTA'03, LNCS 2076*, pages 246–263, 2003.

[86] S. Dal Zilio, D. Lugiez, and C. Meyssonnier. A logic you can count on. In *31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '04*, pages 135–146, New York, NY, USA, 2004. ACM Press.

Annexe A

Preuves de l'attaque du protocole de Needham-Schroeder

On présente ici la preuve correpondant à l'attaque du protocole de Needham-Schroeder proposée en figure 1.5. Cette preuve fait intervenir deux agents, l'agent a, situé en l_1, jouant le rôle d'Alice, l'agent b, situé en l_2, jouant celui de Bob et l'intrus qui est situé à l'emplacement z. Pour que la preuve soit la plus concise possible on ne mentionnera que les formules intervenant dans l'attaque. Au démarrage de l'attaque, a contacte z et b attend d'être contacté.

La preuve étant de taille conséquente, elle est ici découpée étape par étape. On commence par introduire les notations que l'on utilise afin que les jugements soient plus concis.

A.1 Configuration initiale et notation

On propose ici plusieurs définitions de formules dont le but est de rendre plus concise l'écriture des séquents. On commence par définir une écriture plus concise pour les messages intervenants dans le protocole :

$$\begin{array}{lcl}
\forall x, y.m2(x,y) & \stackrel{def}{=} & msg(msg(x), msg(y)) \\
\forall x, y, t.m2_t(x,y) & \stackrel{def}{=} & msg(pubk(t, m2(x,y))) \\
\forall x, t.m_t(x) & \stackrel{def}{=} & msg(pubk(t, msg(x)))
\end{array}$$

De même, on utilisera des définitions pour définir chacune des étapes du protocole pour Alice et Bob Pour rappel, lors de l'attaque, Alice démarre une authentification auprès de l'intrus et Bob attend une demande d'authentification de l'intrus. Les définitions sont légèrement différente de celles données aux chapitre 1 car pour alléger la preuve la création des *nonces* et le choix des destinataires a déjà été fait. Ainsi par exemple, la partie correspondant à Alice utilise le *nonce* Na et doit contacter z :

$$\begin{array}{lcl}
Alice3 & \stackrel{def}{=} & \langle l_1\rangle send(z, l_1, m_z(Nx)) \\
Alice2 & \stackrel{def}{=} & \forall Nx.(from(z, (m2_a(Na, Nx))) \multimap (Alice3 \otimes ct(z))) \\
Alice1 & \stackrel{def}{=} & [l_1](\langle l_1\rangle send(z, l_1, m2_a(Na, A)) \otimes Alice2) \\[1ex]
Bob3 & \stackrel{def}{=} & from(z, m_b(Nb)) \multimap ct_by(a) \\
Bob2 & \stackrel{def}{=} & \langle l_2\rangle send(z, l_2, m2_a(Nx, Nb)) \otimes Bob3 \\
Bob1 & \stackrel{def}{=} & [l_2]\forall Nx.(from(z, m2_b(Nx, a)) \multimap Bob2)
\end{array}$$

Dans cette preuve, l'intrus a uniquement besoin de rediriger des messages et de d'en recréer à partir de ceux qu'il reçoit. Seules les règles suivantes seront donc utilisées :

$$
\begin{array}{lcl}
\forall x.decomp(x) & \stackrel{def}{=} & [z]((\exists y.from(y,x)) \multimap x) \\
\forall x,y.decomp(x,y) & \stackrel{def}{=} & [z](m2(x,y) \multimap msg(x) \otimes msg(y)) \\
\forall x.decrypt(x) & \stackrel{def}{=} & [z](m_z(x) \multimap msg(x)) \\
\forall x.decrypt(x,y) & \stackrel{def}{=} & [z](m2_z(x,y) \multimap m2(x,y)) \\
\forall t,x.encrypt_t(x) & \stackrel{def}{=} & [z](msg(x) \multimap m_z(x)) \\
\forall t,x,y.encrypt_t(x,y) & \stackrel{def}{=} & [z](m2(x,y) \multimap m2_z(x,y)) \\
\forall t,x,y.sender_t(x,y) & \stackrel{def}{=} & [z](y \multimap \langle z \rangle send(x,t,y)) \\
Intrus & \stackrel{def}{=} & !\forall x,y.decomp(x,y), !\forall x.decomp(x), \\
 & & !\forall x,t.encrypt_t(x), !\forall x,y,t.encrypt_t(x,y) \\
 & & !\forall x.decrypt(x), !\forall x,y.decrypt(x,y) \\
 & & !\forall x,y,t.sender_t(x,y)
\end{array}
$$

À la définition des trois acteurs, il faut rajouter la formule permettant de *router* les messages :

$$\forall x,y,t.route(x,y,t) \stackrel{def}{=} \forall x,y,t.send(x,y,t) \multimap \{x\}from(y,t)$$

La configuration initale pour l'attaque est donc :

$$Alice1, Bob1, Intrus, !\forall x,y.route(x,y)$$

et l'exécution doit mener à la confirmation pour Alice que tout c'est bien passé et à l'impression pour Bob qu'Alice l'a contacté, ce qui correspond à : $[l_1]contacted(z) \otimes [l_2]ct\_by(a)$.

A.2 La preuve

On présente maintenant la preuve correspondant à l'attaque. On la découpe en plusieurs partie, chaque partie correspondant à un message de l'attaque décrite en figure 1.5. Pour chaque règle d'inférence, on indique en gras la proposition à la quelle est appliquée la règle. De plus, par soucis de consision, les règles de congruence permettant d'internaliser les emplacements (du type $[l](A \otimes B) \cong [l]A \otimes [l]B$) ne seront pas indiquées dans la preuve.

A.2.1 Première étape : $A \rightarrow Z : \{Na \| A\}_{pk(z)}$

On commence par présenter la partie de la preuve qui correspond à la première étape du protocole : l'envoi par Alice du premier message d'authentification à l'intrus. Comme tous les envois de message, celui ci commence par la sortie d'un prédicat de la forme $send(x, y, z)$ de l'emplacement qui envoie le message. Ensuite, le message est analysé à la racine et est envoyé à son destinaire (c'est le rôle assuré par la formule *route*).

$$\dfrac{\dfrac{\begin{matrix}(1)\\ \vdots\\ [l_1]Alice2, Bob1, Intrus,\ !\forall x,y,t.route(x,y,t),\ [z]from(l_1, m2_z(Na,A)) \vdash [l_1]ct(z)\otimes[l_2]ct\_by(a)\end{matrix}}{[l_1]Alice2, Bob1, Intrus,\ !\forall x,y,t.route(x,y,t),\ \mathbf{\{z\}from(l_1, m2_z(Na,A))} \vdash [l_1]ct(z)\otimes[l_2]ct\_by(a)}\,(\text{In : L}) \qquad \dfrac{}{\mathbf{send(z,l_1,m2_z(Na,A))} \vdash \mathbf{send(z,l_1,m2_z(Na,A))}}\,(\text{Id})}{send(z,l_1,m2_z(Na,A)), [l_1]Alice2, Bob1, Intrus, !\forall x,y,t.route(x,y,t), \mathbf{route(z,y,m2_z(Na,A))} \vdash [l_1]ct(z)\otimes[l_2]ct\_by(a)}\,(\multimap : \text{L})$$

$$\dfrac{send(z,l_1,m2_z(Na,A)), [l_1]Alice2, Bob1, Intrus, !\forall x,y,t.route(x,y,t), \mathbf{route(z,y,m2_z(Na,A))} \vdash [l_1]ct(z)\otimes[l_2]ct\_by(a)}{send(z,l_1,m2_z(Na,A)), [l_1]Alice2, Bob1, Intrus, !\forall x,y,t.route(x,y,t), \forall\mathbf{t.route(z,l_1,t)} \vdash [l_1]ct(z)\otimes[l_2]ct\_by(a)}\,(\forall : \text{L})$$

$$\dfrac{}{send(z,l_1,m2_z(Na,A)), [l_1]Alice2, Bob1, Intrus, !\forall x,y,t.route(x,y,t), \forall\mathbf{y},\mathbf{t.route(z,y,t)} \vdash [l_1]ct(z)\otimes[l_2]ct\_by(a)}\,(\forall : \text{L})$$

$$\dfrac{}{send(z,l_1,m2_z(Na,A)), [l_1]Alice2, Bob1, Intrus, !\forall x,y,t.route(x,y,t), \forall\mathbf{x},\mathbf{y},\mathbf{t.route(x,y,t)} \vdash [l_1]ct(z)\otimes[l_2]ct\_by(a)}\,(\forall : \text{L})$$

$$\dfrac{}{send(z,l_1,m2_z(Na,A)), [l_1]Alice2, Bob1, Intrus, !\forall\mathbf{x},\mathbf{y},\mathbf{t.route(x,y,t)} \vdash [l_1]ct(z)\otimes[l_2]ct\_by(a)}\,(\text{Con})$$

$$\dfrac{}{\mathbf{[l_1]\langle l_1\rangle send(z,l_1,m2_z(Na,A))}, [l_1]Alice2, Bob1, Intrus, !\forall x,y,t.route(x,y,t) \vdash [l_1]ct(z)\otimes[l_2]ct\_by(a)}\,(\text{Out : L})$$

$$\dfrac{}{\mathbf{[l_1](\langle l_1\rangle send(z,l_1,m2_z(Na,A))\otimes Alice2)}, Bob1, Intrus, !\forall x,y,t.route(x,y,t) \vdash [l_1]ct(z)\otimes[l_2]ct\_by(a)}\,(\otimes : \text{L})$$

$$\dfrac{}{\mathbf{Alice1}, Bob1, Intrus, !\forall x,y,t.route(x,y,t) \vdash [l_1]ct(z)\otimes[l_2]ct\_by(a)}\,(\cong : \text{L})$$

A.2.2 Deuxième étape : $Z(A) \to B : \{Na\|A\}_{pk(b)}$

On passe maintenant à l'étape de l'attaque 2. Il s'agit pour l'intrus de récupérer le message de l'étape 1 et d'envoyer le message $\{Na\|A\}_{pK(b)}$ à l'emplacement de Bob, en l_2. Il doit pour cela commencer par décomposer le message qu'il a reçu lors de l'étape précédente. On remarquera ici que comme on l'a expliqué au chapitre 1, toutes les formules intervenant sont situées en z, ce qui permet d'identifier instantanément que ces parties de la preuve correspondent à des actions de l'intrus.

$$\dfrac{\dfrac{\begin{matrix}(2)\\ \vdots\end{matrix}}{[l_1]Alice2, Bob1, Intrus, !\forall x,y,t.route(x,y,t), [z]m2_z(Na,A) \vdash [l_1]ct(z)\otimes[l_2]ct\_by(a)} \qquad \dfrac{\dfrac{}{\mathbf{[z]from(l_1,m2_z(Na,A))} \vdash \mathbf{[z]from(l_1,m2_z(Na,A))}}\,(\text{Id})}{[z]from(l_1,m2_z(Na,A)) \vdash \mathbf{[z]\exists y.from(y,m2_z(Na,A))}}\,(\exists : \text{R})}{[l_1]Alice2, Bob1, Intrus, !\forall x,y,t.route(x,y,t), [z]from(l_1,m2_z(Na,A)), \mathbf{decomp(m2_z(Na,A))} \vdash [l_1]ct(z)\otimes[l_2]ct\_by(a)}\,(\multimap : \text{L})$$

$$\dfrac{}{[l_1]Alice2, Bob1, Intrus, !\forall x,y,t.route(x,y,t), [z]from(l_1,m2_z(Na,A)), \forall\mathbf{x.decomp(x)} \vdash [l_1]ct(z)\otimes[l_2]ct\_by(a)}\,(\forall : \text{L})$$

$$\dfrac{}{\begin{matrix}[l_1]Alice2, Bob1, \mathbf{Intrus}, !\forall x,y,t.route(x,y,t), [z]from(l_1,m2_z(Na,A)) \vdash [l_1]ct(z)\otimes[l_2]ct\_by(a)\\ (1)\end{matrix}}\,(\text{Con})$$

L'étape que l'on vient de passer permet de *décapsuler* le message (on enlève l'information indiquant sa provenance), on peut maintenant le décrypter puisqu'il est encodé avec la clé publique de l'intrus :

$$\dfrac{\begin{matrix}(3)\\ \vdots\\ [l_1]Alice2, Bob1, Intrus, !\forall x,y,t.route(x,y,t), [z]m2(Na,A \vdash [l_1]ct(z)\otimes[l_2]ct\_by(a)\end{matrix} \qquad \dfrac{}{\mathbf{[z]m2_z(Na,A)} \vdash \mathbf{[z]m2_z(Na,A)}}\,(\text{Id})}{[l_1]Alice2, Bob1, Intrus, !\forall x,y,t.route(x,y,t), [z]m2_z(Na,A), \mathbf{decrypt(Na,A)} \vdash [l_1]ct(z)\otimes[l_2]ct\_by(a)}\,(\multimap : \text{L})$$

$$\dfrac{}{[l_1]Alice2, Bob1, Intrus, !\forall x,y,t.route(x,y,t), [z]m2_z(Na,A), \forall\mathbf{y.decrypt(Na,y)} \vdash [l_1]ct(z)\otimes[l_2]ct\_by(a)}\,(\forall : \text{L})$$

$$\dfrac{}{[l_1]Alice2, Bob1, Intrus, !\forall x,y,t.route(x,y,t), [z]m2_z(Na,A), \forall\mathbf{x},\mathbf{y.decrypt(x,y)} \vdash [l_1]ct(z)\otimes[l_2]ct\_by(a)}\,(\forall : \text{L})$$

$$\dfrac{}{\begin{matrix}[l_1]Alice2, Bob1, \mathbf{Intrus}, !\forall x,y,t.route(x,y,t), [z]m2_z(Na,A) \vdash [l_1]ct(z)\otimes[l_2]ct\_by(a)\\ (2)\end{matrix}}\,(\text{Con})$$

On encode ensuite le message avec la clé publique de Bob pour obtenir le message qui doit être envoyé à l'étape 2 de l'attaque.

$$
\begin{array}{c}
(4) \\
\vdots \\
\dfrac{
\begin{array}{c}[l_1]Alice2, Bob1, Intrus, \\ [z]m2_b(Na,A) \forall x,y,t.route(x,y,t)\end{array} \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)
\qquad
\dfrac{}{[\mathbf{z}]\mathbf{m2(Na,A)} \vdash [\mathbf{z}]\mathbf{m2(Na,A)}}\ (\mathrm{Id})
}{
\dfrac{
\begin{array}{c}[l_1]Alice2, Bob1, Intrus, \mathbf{encrypt_b(Na,A)} \\ \forall x,y,t.route(x,y,t), [z]m2(Na,A)\end{array} \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)
}{
\dfrac{
\begin{array}{c}[l_1]Alice2, Bob1, Intrus, \forall \mathbf{x}.\mathbf{encrypt_b(x,A)} \\ \forall x,y,t.route(x,y,t), [z]m2(Na,A)\end{array} \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)
}{
\dfrac{
\begin{array}{c}[l_1]Alice2, Bob1, Intrus, \forall \mathbf{x}, \mathbf{y}.\mathbf{encrypt_b(x,y)} \\ \forall x,y,t.route(x,y,t), [z]m2(Na,A)\end{array} \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)
}{
\dfrac{
\begin{array}{c}[l_1]Alice2, Bob1, Intrus, \forall \mathbf{x}, \mathbf{y}, \mathbf{t}.\mathbf{encrypt_t(x,y)} \\ \forall x,y,t.route(x,y,t), [z]m2(Na,A)\end{array} \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)
}{
\begin{array}{c}[l_1]Alice2, Bob1, \mathbf{Intrus}, \\ !\forall x,y,t.route(x,y,t), [z]m2(Na,A)\end{array} \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)
}\ (\mathrm{Con:L})
}\ (\forall:\mathrm{L})
}\ (\forall:\mathrm{L})
}\ (\forall:\mathrm{L})
}\ (\multimap:\mathrm{L}) \\
(3)
\end{array}
$$

Une fois le message prêt, on peut l'envoyer à Bob, on doit pour cela l'encapsuler et le faire sortir de z :

$$
\begin{array}{c}
(5) \\
\vdots \\
\dfrac{
\dfrac{
\begin{array}{c}[l_1]Alice2, Bob1, !\forall x,y,t.route(x,y,t), \\ Intrus, send(l_2, z, m2(Na,A))\end{array} \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)
}{
\begin{array}{c}[l_1]Alice2, Bob1, !\forall x,y,t.route(x,y,t), \\ Intrus, [\mathbf{z}]\langle\mathbf{z}\rangle\mathbf{send(l_2, z, m2(Na,A))}\end{array} \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)
}\ (\mathrm{Out:L})
\qquad
\dfrac{}{[\mathbf{z}]\mathbf{m2(Na,A)} \vdash [\mathbf{z}]\mathbf{m2(Na,A)}}\ (\mathrm{Id})
}{
\dfrac{
\begin{array}{c}[l_1]Alice2, Bob1, Intrus, \\ !\forall x,y,t.route(x,y,t), [z]m2(Na,A), \mathbf{sender_z(l_2, m2(Na,A))}\end{array} \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)
}{
\dfrac{
\begin{array}{c}[l_1]Alice2, Bob1, Intrus, \\ !\forall x,y,t.route(x,y,t), [z]m2(Na,A), \forall \mathbf{y}.\mathbf{sender_z(l_2, y)}\end{array} \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)
}{
\dfrac{
\begin{array}{c}[l_1]Alice2, Bob1, Intrus, \\ !\forall x,y,t.route(x,y,t), [z]m2(Na,A), \forall \mathbf{x}, \mathbf{y}.\mathbf{sender_z(x,y)}\end{array} \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)
}{
\dfrac{
\begin{array}{c}[l_1]Alice2, Bob1, Intrus, \\ !\forall x,y,t.route(x,y,t), [z]m2(Na,A), \forall \mathbf{t}, \mathbf{x}, \mathbf{y}.\mathbf{sender_t(x,y)}\end{array} \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)
}{
\begin{array}{c}[l_1]Alice2, Bob1, \mathbf{Intrus}, \\ !\forall x,y,t.route(x,y,t), [z]m2(Na,A)\end{array} \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)
}\ (\mathrm{Con})
}\ (\forall:\mathrm{L})
}\ (\forall:\mathrm{L})
}\ (\forall:\mathrm{L})
}\ (\multimap:\mathrm{L}) \\
(4)
\end{array}
$$

Le message est ensuite envoyé à Bob :

$$
\begin{prooftree}
\AxiomC{(6)}
\noLine
\UnaryInfC{$\vdots$}
\noLine
\UnaryInfC{$[l_1]Alice2, Bob1, !\forall x,y,t.route(x,y,t), Intrus, [l_2]from(z, m2(Na, A)) \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)$}
\RightLabel{(In : L)}
\UnaryInfC{$[l_1]Alice2, Bob1, !\forall x,y,t.route(x,y,t), Intrus, \{\mathbf{l_2}\}\mathbf{from}(\mathbf{z}, \mathbf{m2}(\mathbf{Na}, \mathbf{A})) \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)$}
\AxiomC{}
\RightLabel{(Id)}
\UnaryInfC{$\mathbf{send}(\mathbf{l_2}, \mathbf{z}, \mathbf{m2}(\mathbf{Na}, \mathbf{A})) \vdash \mathbf{send}(\mathbf{l_2}, \mathbf{z}, \mathbf{m2}(\mathbf{Na}, \mathbf{A}))$}
\RightLabel{($\multimap$: L)}
\BinaryInfC{$[l_1]Alice2, Bob1, !\forall x,y,t.route(x,y,t), \mathbf{route}(\mathbf{l_2}, \mathbf{z}, \mathbf{m2}(\mathbf{Na}, \mathbf{A})), Intrus, send(l_2, z, m2(Na, A)) \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)$}
\RightLabel{($\forall$: L)}
\UnaryInfC{$[l_1]Alice2, Bob1, !\forall x,y,t.route(x,y,t), \forall \mathbf{t}.\mathbf{route}(\mathbf{l_2}, \mathbf{z}, \mathbf{t}), Intrus, send(l_2, z, m2(Na, A)) \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)$}
\RightLabel{($\forall$: L)}
\UnaryInfC{$[l_1]Alice2, Bob1, !\forall x,y,t.route(x,y,t), \forall \mathbf{y}, \mathbf{t}.\mathbf{route}(\mathbf{l_2}, \mathbf{y}, \mathbf{t}), Intrus, send(l_2, z, m2(Na, A)) \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)$}
\RightLabel{($\forall$: L)}
\UnaryInfC{$[l_1]Alice2, Bob1, !\forall x,y,t.route(x,y,t), \forall \mathbf{x}, \mathbf{y}, \mathbf{t}.\mathbf{route}(\mathbf{x}, \mathbf{y}, \mathbf{t}), Intrus, send(l_2, z, m2(Na, A)) \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)$}
\RightLabel{(Con)}
\UnaryInfC{$[l_1]Alice2, Bob1, !\forall \mathbf{x}, \mathbf{y}, \mathbf{t}.\mathbf{route}(\mathbf{x}, \mathbf{y}, \mathbf{t}), Intrus, send(l_2, z, m2(Na, A)) \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)$}
\noLine
\UnaryInfC{(5)}
\end{prooftree}
$$

A.2.3 Troisième étape : $B \rightarrow Z(A) : \{Na \| Nb\}_{pk(a)}$

Bob reçoit le premier message du protocole, il doit donc répondre à l'intrus qui se fait passer pour Alice. Il commence par analyser le message reçu. Ici, on remarque que les formules *actives* sont situées en l_2, l'emplacement de Bob.

$$
\begin{prooftree}
\AxiomC{(7)}
\noLine
\UnaryInfC{$\vdots$}
\noLine
\UnaryInfC{$[l_1]Alice2, Intrus, !\forall x,y,t.route(x,y,t)\; send(z, l_2, m2_a(Na, Nb)), [l_2]Bob3 \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)$}
\RightLabel{(Out : L)}
\UnaryInfC{$[l_1]Alice2, Intrus, !\forall x,y,t.route(x,y,t)\; [\mathbf{l_2}]\langle \mathbf{l_2} \rangle \mathbf{send}(\mathbf{z}, \mathbf{l_2}, \mathbf{m2_a}(\mathbf{Na}, \mathbf{Nb})), [l_2]Bob3 \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)$}
\RightLabel{($\otimes$: L)}
\UnaryInfC{$[l_1]Alice2, Intrus, !\forall x,y,t.route(x,y,t)\; [\mathbf{l_2}](\langle \mathbf{l_2} \rangle \mathbf{send}(\mathbf{z}, \mathbf{l_2}, \mathbf{m2_a}(\mathbf{Na}, \mathbf{Nb})) \otimes \mathbf{Bob3}) \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)$}
\RightLabel{($\cong$: L)}
\UnaryInfC{$[l_1]Alice2, [\mathbf{l_2}]\mathbf{Bob2}, Intrus, !\forall x,y,t.route(x,y,t) \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)$}
\AxiomC{}
\RightLabel{(Id)}
\UnaryInfC{$[\mathbf{l_2}]\mathbf{from}(\mathbf{z}, \mathbf{m2_b}(\mathbf{Na}, \mathbf{a})) \vdash [\mathbf{l_2}]\mathbf{from}(\mathbf{z}, \mathbf{m2_b}(\mathbf{Na}, \mathbf{a}))$}
\RightLabel{($\multimap$: L)}
\BinaryInfC{$[l_1]Alice2, [\mathbf{l_2}](\mathbf{from}(\mathbf{z}, \mathbf{m2_b}(\mathbf{Na}, \mathbf{a})) \multimap \mathbf{Bob2}), Intrus, !\forall x,y,t.route(x,y,t), [l_2]from(z, m2(Na, A)) \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)$}
\RightLabel{($\forall$: L)}
\UnaryInfC{$[l_1]Alice2, [\mathbf{l_2}]\forall \mathbf{Nx}.(\mathbf{from}(\mathbf{z}, \mathbf{m2_b}(\mathbf{Nx}, \mathbf{a})) \multimap \mathbf{Bob2}), Intrus, !\forall x,y,t.route(x,y,t), [l_2]from(z, m2(Na, A)) \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)$}
\RightLabel{($\cong$: L)}
\UnaryInfC{$[l_1]Alice2, \mathbf{Bob1}, !\forall x,y,t.route(x,y,t), Intrus, [l_2]from(z, m2(Na, A)) \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)$}
\noLine
\UnaryInfC{(6)}
\end{prooftree}
$$

La réponse de Bob est alors dirigé vers z, ce qui marque la fin de l'étape 3 :

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\begin{array}{c}(8)\\ \vdots \\ [l_1]Alice2, [l_2]Bob3, !\forall x,y,t.route(x,y,t) \quad \\ Intrus, [z]from(l_2, m2_a(Na,Nb)) \end{array} \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)}{\begin{array}{c}[l_1]Alice2, [l_2]Bob3, !\forall x,y,t.route(x,y,t)\\ Intrus, \{\mathbf{z}\}\mathbf{from}(\mathbf{l_2}, \mathbf{m2_a}(\mathbf{Na},\mathbf{Nb}))\end{array} \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)}\,(\mathrm{In:l}) \quad \dfrac{}{\begin{array}{c}send(z,l_2,\\ m2_a(Na,Nb))\end{array} \vdash \begin{array}{c}send(z,l_2,\\ m2_a(Na,Nb))\end{array}}\,(\mathrm{Id})}{\begin{array}{c}[l_1]Alice2, [l_2]Bob3, Intrus, !\forall x,y,t.route(x,y,t)\\ \mathbf{route}(\mathbf{z}, \mathbf{l_2}, \mathbf{m2_a}(\mathbf{Na},\mathbf{Nb})), send(z,l_2,m2_a(Na,Nb))\end{array} \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)}\,(\multimap:\mathrm{l})}{\begin{array}{c}[l_1]Alice2, [l_2]Bob3, Intrus, !\forall x,y,t.route(x,y,t)\\ \forall \mathbf{t}.\mathbf{route}(\mathbf{z}, \mathbf{l_2}, \mathbf{t}), send(z,l_2,m2_a(Na,Nb))\end{array} \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)}\,(\forall:\mathrm{l})}{\begin{array}{c}[l_1]Alice2, [l_2]Bob3, Intrus, !\forall x,y,t.route(x,y,t)\\ \forall \mathbf{y}, \mathbf{t}.\mathbf{route}(\mathbf{z}, \mathbf{y}, \mathbf{t}), send(z,l_2,m2_a(Na,Nb))\end{array} \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)}\,(\forall:\mathrm{l})}{\begin{array}{c}[l_1]Alice2, [l_2]Bob3, Intrus, !\forall x,y,t.route(x,y,t)\\ \forall \mathbf{x}, \mathbf{y}, \mathbf{t}.\mathbf{route}(\mathbf{x}, \mathbf{y}, \mathbf{t}), send(z,l_2,m2_a(Na,Nb))\end{array} \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)}\,(\forall:\mathrm{l})}{\begin{array}{c}[l_1]Alice2, Intrus, !\forall \mathbf{x}, \mathbf{y}, \mathbf{t}.\mathbf{route}(\mathbf{x}, \mathbf{y}, \mathbf{t})\\ send(z,l_2,m2_a(Na,Nb)), [l_2]Bob3\end{array} \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)}\,(\mathrm{Con})
$$

$$(7)$$

A.2.4 Quatrième étape : $Z \rightarrow A : \{Na \| Nb\}_{pk(a)}$

Pour démarrer l'étape 4, z doit *décapsuler* le message reçu :

$$
\dfrac{\dfrac{\dfrac{\dfrac{\begin{array}{c}(9)\\ \vdots\end{array}}{\begin{array}{c}[l_1]Alice2, [l_2]Bob3, Intrus,\\ !\forall x,y,t.route(x,y,t), [z]m2_a(Na,Nb)\end{array} \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)} \quad \dfrac{}{\begin{array}{c}[\mathbf{z}]\mathbf{from}(\mathbf{l_2},\\ \mathbf{m2_a}(\mathbf{Na},\mathbf{Nb}))\end{array} \vdash \begin{array}{c}[\mathbf{z}]\mathbf{from}(\mathbf{l_2},\\ \mathbf{m2_a}(\mathbf{Na},\mathbf{Nb}))\end{array}}\,(\mathrm{Id})}{\begin{array}{c}[l_1]Alice2, [l_2]Bob3, !\forall x,y,t.route(x,y,t)\\ Intrus, \mathbf{decomp}(\mathbf{m2_a}(\mathbf{Na},\mathbf{Nb})), [z]from(l_2, m2_a(Na,Nb))\end{array} \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)}\,(\multimap:\mathrm{L})}{\begin{array}{c}[l_1]Alice2, [l_2]Bob3, !\forall x,y,t.route(x,y,t)\\ Intrus, \forall \mathbf{x}.\mathbf{decomp}(\mathbf{x}), [z]from(l_2, m2_a(Na,Nb))\end{array} \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)}\,(\forall:\mathrm{L})}{\begin{array}{c}[l_1]Alice2, [l_2]Bob3, !\forall x,y,t.route(x,y,t)\\ \mathbf{Intrus}, [z]from(l_2, m2_a(Na,Nb))\end{array} \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)}\,(\mathrm{Con})
$$

$$(8)$$

L'intrus doit ensuite envoyer le message qu'il a reçu directement vers Alice sans pouvoir l'ouvrir :

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\begin{array}{c}(10)\\ \vdots\\ [l_1]Alice2, [l_2]Bob3, Intrus,\\ !\forall x,y,t.route(x,y,t), send(l_1,z,m2_a(Na,Nb))\end{array} \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)}{\begin{array}{c}[l_1]Alice2, [l_2]Bob3, Intrus,\\ !\forall x,y,t.route(x,y,t), [\mathbf{z}]\langle \mathbf{z}\rangle \mathbf{send}(\mathbf{l_1}, \mathbf{z}, \mathbf{m2_a}(\mathbf{Na},\mathbf{Nb}))\end{array} \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)}\,(\mathrm{Out:L}) \quad \dfrac{}{[\mathbf{z}]\mathbf{m2_a}(\mathbf{Na},\mathbf{Nb}) \vdash [\mathbf{z}]\mathbf{m2_a}(\mathbf{Na},\mathbf{Nb})}\,(\mathrm{Id})}{\begin{array}{c}[l_1]Alice2, [l_2]Bob3, Intrus,\\ !\forall x,y,t.route(x,y,t), \mathbf{sender_z}(\mathbf{l_1}, \mathbf{m2_a}(\mathbf{Na},\mathbf{Nb})), [z]m2_a(Na,Nb)\end{array} \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)}\,(\multimap:\mathrm{L})}{\begin{array}{c}[l_1]Alice2, [l_2]Bob3, !\forall x,y,t.route(x,y,t),\\ Intrus, \forall \mathbf{t}.\mathbf{sender_t}(\mathbf{l_1}, \mathbf{m2_a}(\mathbf{Na},\mathbf{Nb})), [z]m2_a(Na,Nb)\end{array} \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)}\,(\forall:\mathrm{L})}{\begin{array}{c}[l_1]Alice2, [l_2]Bob3, Intrus,\\ !\forall x,y,t.route(x,y,t), \forall \mathbf{y}, \mathbf{t}.\mathbf{sender_t}(\mathbf{l_1}, \mathbf{y}), [z]m2_a(Na,Nb)\end{array} \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)}\,(\forall:\mathrm{L})}{\begin{array}{c}[l_1]Alice2, [l_2]Bob3, Intrus,\\ !\forall x,y,t.route(x,y,t), \forall \mathbf{x}, \mathbf{y}, \mathbf{t}.\mathbf{sender_t}(\mathbf{x}, \mathbf{y}), [z]m2_a(Na,Nb)\end{array} \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)}\,(\forall:\mathrm{L})}{\begin{array}{c}[l_1]Alice2, [l_2]Bob3, \mathbf{Intrus},\\ !\forall x,y,t.route(x,y,t), [z]m2_a(Na,Nb)\end{array} \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)}\,(\mathrm{Con})
$$

$$(9)$$

Alice reçoit ensuite le message, ce qui correspond à la fin de l'étape 3 :

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\overset{(11)}{\vdots}}{[l_1]Alice2, [l_2]Bob3, !\forall x,y,t.route(x,y,t), Intrus, [l_1]from(z, m2_a(Na,Nb)) \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)}}{[l_1]Alice2, [l_2]Bob3, !\forall x,y,t.route(x,y,t), Intrus, \{\mathbf{l_1}\}\mathbf{from}(\mathbf{z}, \mathbf{m2_a}(\mathbf{Na},\mathbf{Nb})) \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)}\,(\text{In : L}) \quad \dfrac{}{\mathbf{send}(\mathbf{l_1},\mathbf{z},\mathbf{m2_a}(\mathbf{Na},\mathbf{Nb})) \vdash \mathbf{send}(\mathbf{l_1},\mathbf{z},\mathbf{m2_a}(\mathbf{Na},\mathbf{Nb}))}\,(\text{Id})}{[l_1]Alice2, [l_2]Bob3, !\forall x,y,t.route(x,y,t), Intrus, \mathbf{route}(\mathbf{l_1},\mathbf{z},\mathbf{m2_a}(\mathbf{Na},\mathbf{Nb})), send(l_1,z,m2_a(Na,Nb)) \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)}\,(\multimap : \text{L})}{[l_1]Alice2, [l_2]Bob3, !\forall x,y,t.route(x,y,t), Intrus, \forall \mathbf{t}.\mathbf{route}(\mathbf{l_1},\mathbf{z},\mathbf{t}), send(l_1,z,m2_a(Na,Nb)) \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)}\,(\forall : \text{L})}{[l_1]Alice2, [l_2]Bob3, !\forall x,y,t.route(x,y,t), Intrus, \forall \mathbf{y},\mathbf{t}.\mathbf{route}(\mathbf{l_1},\mathbf{y},\mathbf{t}), send(l_1,z,m2_a(Na,Nb)) \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)}\,(\forall : \text{L})}{[l_1]Alice2, [l_2]Bob3, !\forall x,y,t.route(x,y,t), Intrus, \forall \mathbf{x},\mathbf{y},\mathbf{t}.\mathbf{route}(\mathbf{x},\mathbf{y},\mathbf{t}), send(l_1,z,m2_a(Na,Nb)) \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)}\,(\forall : \text{L})}{\underset{(10)}{[l_1]Alice2, [l_2]Bob3, Intrus, !\forall \mathbf{x},\mathbf{y},\mathbf{t}.\mathbf{route}(\mathbf{x},\mathbf{y},\mathbf{t}), \mathbf{send}(\mathbf{l_1},\mathbf{z},\mathbf{m2_a}(\mathbf{Na},\mathbf{Nb})) \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)}}\,(\text{Con})
$$

A.2.5 Cinquième étape : $A \rightarrow Z : \{Nb\}_{pk(z)}$

Cette quatrième commence par la récupération et le traitement du message par Alice :

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\overset{(12)}{\vdots}}{send(z,l_1,m_z(Nb)), [l_2]Bob3, Intrus, [l_1]ct(z), !\forall x,y,t.route(x,y,t) \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)}}{[\mathbf{l_1}]\langle \mathbf{l_1}\rangle \mathbf{send}(\mathbf{z},\mathbf{l_1},\mathbf{m_z}(\mathbf{Nb})), [l_2]Bob3, Intrus, [l_1]ct(z), !\forall x,y,t.route(x,y,t) \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)}\,(\text{Out : L})}{[\mathbf{l_1}]\mathbf{Alice3}, [l_1]ct(z), [l_2]Bob3, Intrus, !\forall x,y,t.route(x,y,t) \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)}\,(\cong : \text{L})}{[\mathbf{l_1}](\mathbf{Alice3} \otimes \mathbf{ct}(\mathbf{z})), [l_2]Bob3, Intrus, !\forall x,y,t.route(x,y,t) \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)}\,(\otimes : \text{L}) \quad \dfrac{}{[\mathbf{l_1}](\mathbf{from}(\mathbf{z},(\mathbf{m2_a}(\mathbf{Na},\mathbf{Nb})))) \vdash [\mathbf{l_1}](\mathbf{from}(\mathbf{z},(\mathbf{m2_a}(\mathbf{Na},\mathbf{Nb}))))}\,(\text{Id : L})}{[\mathbf{l_1}](\mathbf{from}(\mathbf{z},(\mathbf{m2_a}(\mathbf{Na},\mathbf{Nb}))) \multimap (\mathbf{Alice3} \otimes \mathbf{ct}(\mathbf{z}))), [l_2]Bob3, !\forall x,y,t.route(x,y,t), Intrus, [l_1]from(z,m2_a(Na,Nb)) \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)}\,(\multimap : \text{L})}{[\mathbf{l_1}]\forall \mathbf{Nx}.(\mathbf{from}(\mathbf{z},(\mathbf{m2_a}(\mathbf{Na},\mathbf{Nx}))) \multimap (\mathbf{Alice3} \otimes \mathbf{ct}(\mathbf{z}))), [l_2]Bob3, !\forall x,y,t.route(x,y,t), Intrus, [l_1]from(z,m2_a(Na,Nb)) \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)}\,(\forall : \text{L})}{\underset{(11)}{[\mathbf{l_1}]\mathbf{Alice2}, [l_2]Bob3, !\forall x,y,t.route(x,y,t), Intrus, [l_1]from(z,m2_a(Na,Nb)) \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)}}\,(\cong : \text{L})
$$

Le message est alors routé vers son destinataire : z.

$$\dfrac{\dfrac{\begin{array}{c}(13)\\ \vdots \\ [z]from(l_1, m_z(Nb)), [l_2]Bob3, Intrus, [l_1]ct(z), !\forall x,y,t.route(x,y,t) \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)\end{array}}{\{\mathbf{z}\}\mathbf{from}(\mathbf{l_1}, \mathbf{m_z}(\mathbf{Nb})), [l_2]Bob3, Intrus, [l_1]ct(z), !\forall x,y,t.route(x,y,t) \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)}\,(\mathrm{In}:\mathrm{L}) \qquad \dfrac{}{\mathbf{send}(\mathbf{z}, \mathbf{l_1}, \mathbf{m_z}(\mathbf{Nb})) \vdash \mathbf{send}(\mathbf{z}, \mathbf{l_1}, \mathbf{m_z}(\mathbf{Nb}))}\,(\mathrm{Id})}{send(z, l_1, m_z(Nb)), [l_2]Bob3, Intrus, [l_1]ct(z), !\forall x,y,t.route(x,y,t), \mathbf{route}(\mathbf{z}, \mathbf{l_1}, \mathbf{m_z}(\mathbf{Nb})) \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)}\,(\multimap:\mathrm{L})$$

$$\dfrac{send(z, l_1, m_z(Nb)), [l_2]Bob3, Intrus, [l_1]ct(z), !\forall x,y,t.route(x,y,t), \mathbf{route}(\mathbf{z}, \mathbf{l_1}, \mathbf{m_z}(\mathbf{Nb})) \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)}{send(z, l_1, m_z(Nb)), [l_2]Bob3, Intrus, [l_1]ct(z), !\forall x,y,t.route(x,y,t), \forall \mathbf{t}.\mathbf{route}(\mathbf{z}, \mathbf{l_1}, \mathbf{t}) \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)}\,(\forall:\mathrm{L})$$

$$\dfrac{}{send(z, l_1, m_z(Nb)), [l_2]Bob3, Intrus, [l_1]ct(z), !\forall x,y,t.route(x,y,t), \forall \mathbf{y}, \mathbf{t}.\mathbf{route}(\mathbf{z}, \mathbf{y}, \mathbf{t}) \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)}\,(\forall:\mathrm{L})$$

$$\dfrac{}{send(z, l_1, m_z(Nb)), [l_2]Bob3, Intrus, [l_1]ct(z), !\forall x,y,t.route(x,y,t), \forall \mathbf{x}, \mathbf{y}, \mathbf{t}.\mathbf{route}(\mathbf{x}, \mathbf{y}, \mathbf{t}) \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)}\,(\forall:\mathrm{L})$$

$$\dfrac{}{\begin{array}{c}send(z, l_1, m_z(Nb)), [l_2]Bob3, Intrus, [l_1]ct(z), !\forall \mathbf{x}, \mathbf{y}, \mathbf{t}.\mathbf{route}(\mathbf{x}, \mathbf{y}, \mathbf{t}) \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)\\ (12)\end{array}}\,(\mathrm{Con})$$

A.2.6 Sixième étape : $Z(A) \to B\{N_b\}_{pk(B)}$

L'intrus (z) commence par *décapsuler* le message reçu.

$$\dfrac{\dfrac{\begin{array}{c}(14)\\ \vdots\end{array}}{[z]msg(Nb), [l_2]Bob3, Intrus, [l_1]ct(z), !\forall x,y,t.route(x,y,t) \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)} \qquad \dfrac{}{[\mathbf{z}]\mathbf{m_z}(\mathbf{Nb}) \vdash [\mathbf{z}]\mathbf{m_z}(\mathbf{Nb})}\,(\mathrm{Id})}{[z]m_z(Nb), [l_2]Bob3, Intrus, [l_1]ct(z), !\forall x,y,t.route(x,y,t), \mathbf{decrypt}(\mathbf{Nb}) \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)}\,(\multimap:\mathrm{L})$$

$$\dfrac{}{[z]m_z(Nb), [l_2]Bob3, Intrus, [l_1]ct(z), !\forall x,y,t.route(x,y,t), \forall \mathbf{x}.\mathbf{decrypt}(\mathbf{x}) \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)}\,(\forall:\mathrm{L})$$

$$\dfrac{\dfrac{}{[z]m_z(Nb), [l_2]Bob3, \mathbf{Intrus}, [l_1]ct(z), !\forall x,y,t.route(x,y,t) \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)}\,(\mathrm{Con}) \qquad \dfrac{\dfrac{}{[\mathbf{z}]\mathbf{from}(\mathbf{l_1}, \mathbf{m_z}(\mathbf{Nb})) \vdash [\mathbf{z}]\mathbf{from}(\mathbf{l_1}, \mathbf{m_z}(\mathbf{Nb}))}\,(\mathrm{Id})}{[z]from(l_1, m_z(Nb)) \vdash [\mathbf{z}]\exists \mathbf{y}.\mathbf{from}(\mathbf{l_1}, \mathbf{m_z}(\mathbf{y}))}\,(\exists:\mathrm{L})}{[z]from(l_1, m_z(Nb)), [l_2]Bob3, Intrus, [l_1]ct(z), !\forall x,y,t.route(x,y,t), \mathbf{decomp}(\mathbf{m_z}(\mathbf{Nb})) \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)}\,(\multimap:\mathrm{L})$$

$$\dfrac{}{[z]from(l_1, m_z(Nb)), [l_2]Bob3, Intrus, [l_1]ct(z), !\forall x,y,t.route(x,y,t), \forall \mathbf{x}.\mathbf{decomp}(\mathbf{x}) \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)}\,(\forall:\mathrm{L})$$

$$\dfrac{}{\begin{array}{c}[z]from(l_1, m_z(Nb)), [l_2]Bob3, \mathbf{Intrus}, [l_1]ct(z), !\forall x,y,t.route(x,y,t) \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)\\ (13)\end{array}}\,(\mathrm{Con})$$

On compose ensuite le message à envoyer :

$$
\dfrac{\dfrac{\dfrac{\dfrac{\begin{array}{c}(15)\\ \vdots\\ \hline [z]m_b(Nb), [l_2]Bob3, Intrus, [l_1]ct(z), \\ !\forall x,y,t.route(x,y,t) \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)\end{array} \qquad \dfrac{}{\mathbf{[z]msg(Nb) \vdash [z]msg(Nb)}}\,(\mathrm{Id})}{\begin{array}{c}[z]msg(Nb), [l_2]Bob3, Intrus, [l_1]ct(z), \\ !\forall x,y,t.route(x,y,t), \mathbf{encrypt_b(Nb)}\end{array} \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)}\,(\multimap : \mathrm{L})}{\begin{array}{c}[z]msg(Nb), [l_2]Bob3, Intrus, [l_1]ct(z), \\ !\forall x,y,t.route(x,y,t), \forall \mathbf{t.encrypt_t(Nb)}\end{array} \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)}\,(\forall : \mathrm{L})}{\begin{array}{c}[z]msg(Nb), [l_2]Bob3, Intrus, [l_1]ct(z), \\ !\forall x,y,t.route(x,y,t), \forall \mathbf{x}, \mathbf{t.encrypt_t(x)}\end{array} \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)}\,(\forall : \mathrm{L})}{\begin{array}{c}[z]msg(Nb), [l_2]Bob3, \mathbf{Intrus}, \\ [l_1]ct(z), !\forall x,y,t.route(x,y,t)\end{array} \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)}\,(\mathrm{Con})
$$

$$(14)$$

Le message doit ensuite sortir de z avec comme instruction d'être envoyé Bob :

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\begin{array}{c}(16)\\ \vdots\\ send(l_2, z, m_b(Nb), [l_2]Bob3, \\ Intrus, [l_1]ct(z), !\forall x,y,t.route(x,y,t)\end{array} \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)}{\begin{array}{c}\mathbf{[z]\langle z\rangle send(l_2, z, m_b(Nb))}, [l_2]Bob3, \\ Intrus, [l_1]ct(z), !\forall x,y,t.route(x,y,t)\end{array} \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)}\,(\mathrm{Out} : \mathrm{L}) \qquad \dfrac{}{\mathbf{[z]m_b(Nb) \vdash [z]m_b(Nb)}}\,((\mathrm{Id})}{\begin{array}{c}[z]m_b(Nb), [l_2]Bob3, Intrus, [l_1]ct(z), \\ !\forall x,y,t.route(x,y,t), \mathbf{sender_z(l_2, m_b(Nb))}\end{array} \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)}\,(\multimap : \mathrm{L})}{\begin{array}{c}[z]m_b(Nb), [l_2]Bob3, Intrus, [l_1]ct(z), \\ !\forall x,y,t.route(x,y,t), \forall \mathbf{y.sender_z(l_2, y)}\end{array} \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)}\,(\forall : \mathrm{L})}{\begin{array}{c}[z]m_b(Nb), [l_2]Bob3, Intrus, [l_1]ct(z), \\ !\forall x,y,t.route(x,y,t), \forall \mathbf{x}, \mathbf{y.sender_z(x, y)}\end{array} \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)}\,(\forall : \mathrm{L})}{\begin{array}{c}[z]m_b(Nb), [l_2]Bob3, Intrus, [l_1]ct(z), \\ !\forall x,y,t.route(x,y,t), \forall \mathbf{t}, \mathbf{x}, \mathbf{y.sender_t(x, y)}\end{array} \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)}\,(\forall : \mathrm{L})}{\begin{array}{c}[z]m_b(Nb), [l_2]Bob3, Intrus, \\ [l_1]ct(z), !\forall \mathbf{x}, \mathbf{y}, \mathbf{t.route(x, y, t)}\end{array} \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)}\,(\mathrm{Con})
$$

$$(15)$$

Le message est donc *routé* vers l'emplacement de Bob :

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\begin{array}{c}(17)\\ \vdots\\ [l_2]from(z, m_b(Nb)), Intrus, [l_1]ct(z), \\ [l_2](from(z, m_b(Nb)) \multimap ct\_by(a))\end{array} \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)}{\begin{array}{c}[l_2]from(z, m_b(Nb)), \\ \mathbf{[l_2]Bob3}, Intrus, [l_1]ct(z)\end{array} \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)}\,(\cong : \mathrm{L})}{\begin{array}{c}\mathbf{\{l_2\}from(z, m_b(Nb))}, \\ [l_2]Bob3, Intrus, [l_1]ct(z)\end{array} \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)}\,(\mathrm{In} : \mathrm{L}) \qquad \dfrac{}{\begin{array}{c}\mathbf{send(l_2,} \\ \mathbf{z, m_b(Nb))}\end{array} \vdash \begin{array}{c}\mathbf{send(l_2,} \\ \mathbf{z, m_b(Nb))}\end{array}}\,(\mathrm{Id})}{\begin{array}{c}send(l_2, z, m_b(Nb)), [l_2]Bob3, \\ Intrus, [l_1]ct(z), \mathbf{route(l_2, z, m_b(Nb))}\end{array} \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)}\,(\multimap : \mathrm{L})}{\begin{array}{c}send(l_2, z, m_b(Nb)), [l_2]Bob3, \\ Intrus, [l_1]ct(z), \forall \mathbf{t.route(l_2, z, t)}\end{array} \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)}\,(\forall : \mathrm{L})}{\begin{array}{c}send(l_2, z, m_b(Nb)), [l_2]Bob3, \\ Intrus, [l_1]ct(z), \forall \mathbf{y}, \mathbf{t.route(l_2, y, t)}\end{array} \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)}\,(\forall : \mathrm{L})}{\begin{array}{c}send(l_2, z, m_b(Nb)), [l_2]Bob3, \\ Intrus, [l_1]ct(z), \forall \mathbf{x}, \mathbf{y}, \mathbf{t.route(x, y, t)}\end{array} \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)}\,(\forall : \mathrm{L})}{\begin{array}{c}send(l_2, z, m_b(Nb)), [l_2]Bob3, \\ Intrus, [l_1]ct(z), !\forall \mathbf{x}, \mathbf{y}, \mathbf{t.route(x, y, t)}\end{array} \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)}\,(! : \mathrm{L})
$$

$$(16)$$

A.2.7 Épilogue

Il s'agit maintenant de s'assurer que l'envoi des messages tel qu'il a été effectué permet bien d'aboutir au but recherché et donc de prouver que l'intrus (z) s'est bien fait passer pour Alice auprès de Bob :

$$
\dfrac{
\dfrac{
\dfrac{
\dfrac{}{\mathbf{[l_1]ct(z)} \vdash \mathbf{[l_1]ct(z)}}\,(\text{Id})
\quad
\dfrac{}{\mathbf{[l_2]ct\_by(a)} \vdash \mathbf{[l_2]ct\_by(a)}}\,(\text{Id})
}{[l_1]ct(z), [l_2]ct\_by(a) \vdash \mathbf{[l_1]ct(z)} \otimes \mathbf{[l_2]ct\_by(a)}}\,(\otimes : \text{R})
}{\mathbf{Intrus}, [l_1]ct(z), [l_2]ct\_by(a) \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)}\,(\text{Weak : L})
\quad
\dfrac{}{\mathbf{[l_2]from(z, m_b(Nb))} \vdash \mathbf{[l_2]from(z, m_b(Nb))}}\,(\text{Id})
}{
\begin{array}{c}[l_2]from(z, m_b(Nb)), Intrus, [l_1]ct(z), \\ \mathbf{[l_2](from(z, m_b(Nb)) \multimap ct\_by(a))} \\ (17)\end{array} \vdash [l_1]ct(z) \otimes [l_2]ct\_by(a)
}\,(\multimap : \text{L})
$$

On a donc une preuve montrant que l'intrus peut se faire passer pour Alice auprès de Bob.

Cette preuve met en évidence les spécificités dues à l'utilisation des modalités d'emplacement et de déplacement. Dans toutes les étapes, on remarque bien que l'on peut isoler l'acteur en action grâce à l'emplacement des formules manipulées. Ainsi par exemple, à l'étape A.2.1, les formules utilisées sont situées à l'emplacement d'Alice puisque c'est elle qui doit émettre un message.

De même, on remarque bien que le cheminement des messages peut être suivi facilement grâce aux modalités de déplacement. Ainsi, chaque fin d'étape est marqué par la sortie du message qui doit être envoyé de l'emplacement correspondant à son agent, puis de l'entrée du message vers son destinataire. Il est donc possible de retrouver les étapes de l'attaque juste en analysant les mouvements des messages.

Résumé

Le développement de plus en plus important de services mobiles et d'applications distribuées et la volonté de développer des applications sûres nécessite une adaptation des outils de spécifications et de preuves pour mieux prendre en compte la distribution et la mobilité des ressources dans les systèmes informatiques.

Dans cette optique, cette thèse propose tout d'abord une logique nommée logique linéaire distribuée et mobile (DMLL), fondée sur la logique linéaire et intégrant les notions d'emplacements et de déplacements. On propose un calcul des séquents pour cette logique où les preuves intégrent le déplacement des formules. On démontre l'élimination des coupures et on propose une sémantique *à la Kripke* correcte et complète pour cette logique. Enfin, on montre à travers une application pour la validation des protocoles d'authentification comment les modalités d'emplacement et de déplacement peuvent aider à l'analyse des attaques de protocoles. Pour cela, on représente les acteurs d'un protocole et leurs connaissances sous forme de formules logiques et on montre comment extraire des informations d'une preuve correspondant à une attaque.

Cependant, DMLL est une logique définie pour être utilisée selon le paradigme *proof-search-as-computation*, selon lequel les formules correspondent à des progammes et les preuves correspondent à leur exécution. Ce choix est mal adapté pour modéliser et spécifier les propriétés de systèmes distribués. On propose donc un nouveau modèle logique fondé sur une structure adaptée à la représentation de systèmes distribués. Cette structure, l'*arbre de ressources*, consiste en un arbre dont les nœuds, qui sont étiquetés par des labels, contiennent des ressources appartenant à un monoïde d'arbres partiels. On présente ensuite une nouvelle logique, BI-Loc permettant de raisonner sur cette structure. Cette logique est fondée sur la logique BI et intègre une modalité d'emplacement permettant de raisonner sur la structure spatiale de arbres. On propose un langage de manipulation des arbres et une axiomatisation logique de ce langage sous forme de *triplets de Hoare*. On montre que la satisfaction et la décidabilité du *model-checking* pour la version propositionnelle de cette logique sont décidables. Pour la version avec quantificateurs, on démontre que le *model-checking* est indécidable et on determine des fragments décidables. On propose également une méthode de recherche de preuves avec tableaux sémantiques pour la version propositionnelle.

On montre ensuite comment ce modèle peut être utilisé à travers plusieurs types d'applications. Concernant les modèles des pointeurs et des permissions, on montre que l'on peut utiliser une logique adaptée de BI-Loc pour exprimer des propriétés de ce modèle. Cette possibilité permet notamment d'exprimer sous forme d'une formule logique le typage des arbres dans le modèles des permissions, problème qui était sans solution avec des logiques de séparation classique. On détaille ensuite comment les arbres de ressources permettent de représenter la manipulation de documents XML et de raisonnner sur leur transformation. On présente en détails comment certaines règles de spécifications des documents XML se traduisent dans BI-Loc et on montre comment le modèle peut être utilisé pour valider les propriétés de certains programmes de transformation.